THIRTY-FIRST ANNUAL
PACIFIC NORTHWEST

SOFTWARE QUALITY CONFERENCE

P N S Q C ™

October 14-16, 2013

World Trade Center Portland
Portland, Oregon

PNSQC™ 2013 President's Welcome

This year's theme, "The Many Faces of Quality," came about as we discussed who owns quality. We all own quality and we should all feel empowered to do what is needed to meet and exceed our customers' expectations. Recently, I was sitting down with a software quality leader who was rightfully frustrated that a software team was ignoring all the basic best practices of software engineering. This team was relying on the software quality organization to test the product and determine its quality. How backwards and fitting, as we look forward to this year's conference theme, for that engineering team! You can't test quality in – it must be built in from the very foundation.

This year we are pleased to have Capers Jones who will kick us off with a presentation on "Software Quality in 2013: A Survey of the State of the Art". Capers is a well-known author and international public speaker who explains the state of the art of defect prevention and removal, pre-test removal such as static analysis and inspections, the economic value in defect removal efficiency, and the ROI of quality excellence. He will cover known factors that influence software quality, including methodologies, tools, and staffing levels. He will also provide empirical data on the impact of major quality approaches, as well as other topics that can impact overall quality levels.

On Tuesday, Michael Mah will kick-off the second day of the technical program. Michael teaches, writes, and consults to technology companies on measuring, estimating, and managing software projects. He will be discussing, "When Agile Becomes a Quality Game Changer; What Data Says from Recent Agile Benchmark Research". He will talk about factors that can make a meaningful difference and will describe findings that can help accelerate your success. Join Michael and find out how you can assess your own patterns and apply them to your development in an effort to improve.

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

Monday night this year we have partnered with the International Institute of Business Analysis (IIBA) to include a reception with poster papers, hors d'oeuvres and beverages. Karl Wiegers will discuss *"Requirements Reuse: Fantasy or Feasible"* later that evening. On Tuesday night we will have the pleasure of collaborating with Rose City Software Process Improvement Network (SPIN) when Diana Larsen and James Shore will be discussing "*Your Path Through Agile Fluency: Making Agile Work For You."*

As we have done in the past during Monday's and Tuesday's lunch, we will be hosting Deep Dive Birds of a Feather. This is an excellent opportunity to discuss quality topics with your peers while being moderated by an industry activist.

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

Doug Reynolds, President, PNSQC 2013

TABLE OF CONTENTS

Keynote Addresses

Invited Speakers

Agile Track

Automation Track

Business Analysis Track

Collaboration Track

Management Track

Process Improvement Track

Project Management Track

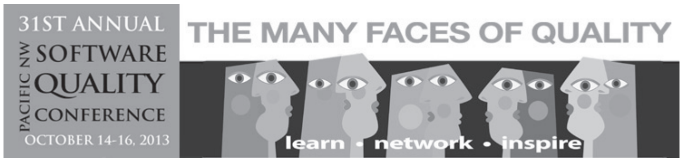
Quality Assurance Track

Quality in Cloud Computing Track

Quality Reviews Track

Testing Track

Topics on Quality Track



PNSQC 2013 – Conference At A Glance

Monday Oct 14, 2013 • Day One • Technical Program				
7:00-8:00 AM	**Registration and Exhibits Open**			
8:00-8:15 AM	**Welcome and Opening Remarks**			
8:15-9:45 AM	**KEYNOTE: Software Quality in 2013: A Survey of the State of the Art** Capers Jones, Namcook Analytics LLC			
9:45-10:15 AM	**Break in Exhibit Hall**			
	INVITED SPEAKER	**QUALITY REVIEWS**	**AUTOMATION**	**QUALITY IN CLOUD COMPUTING**
10:15-11:00 AM	**Lean in the Test Lab: The Potential for Big Improvements** Kathy Iberle, Iberle Consulting Group, Inc.	Effective Peer Reviews: Role in Quality Anil Chakravarthy, Sudeep Das, McAfee	Software Analytics by Storage (SAS) Sundaresan Nagarajan, McAfee	High Availability Testing – Stories from Enterprise and Consumer Services Srinivas Samprathi, Microsoft
11:10-11:55 AM		How to Review Technical Documentation Moss Drake, Dentists Management Corp. (DMC)	Challenges with Test Automation for Virtualization Sharookh Daruwalla, Intel	Are You In It For The Long Haul? Brian Rogers, Microsoft
12:00-1:25 PM	**Lunch Program – Deep-Dive Discussions to Choose From**			
	INVITED SPEAKER	**TOPICS ON QUALITY**	**AUTOMATION**	**AGILE**
1:30-2:15 PM	**The Myths Behind Software Metrics** Douglas Hoffman, Software Quality Methods	Towards Understanding and Improving Mobile User Experience Philip Lew, XBOSoft, Luis Olsina, Nat. Univ. La Pampa	How Can I Automate Stuff? A Guide for the Beginner On Making Their Testing Life Easier Alan Ark, Eid Passport	QA Role in Scrum Karen Wysopal, HP
2:25-3:10 PM		Using FMEA to Improve Software Reliability Kraig Strong, Tektronix	Considerations Before Starting Test Automation Steven Vore, Telerik	Scrum in a Land Far, Far Away (or Using Scrum Across Dispersed Sites) Rick Anderson, Wind River Systems
3:15-3:45 PM	**Break in Exhibit Hall**			
	INVITED SPEAKER	**BUSINESS ANALYSIS**	**MANAGEMENT**	**AGILE**
3:45-4:30 PM	**Today's Testing Innovations** Lee Copeland, Consultant	The Perfect Couple: Domain Models & Behavior-Driven Development Dan Elbaum, Con-Way Enterprise Services, Carlin Scott, Intel	Administering Quality Assurance for Large IT Programs Sreeram Gopalakrishnan, Cognizant Technology Solutions	Colocation: A Case Study in the Rewards and Perils Mary Panza, Parametric Portfolio Associate
4:40-5:20 PM		The Far Away BA: Business Analysis Tips For Pulling It All Together Without Being There At All Jourdan Arenson, Business Analyst	Evolution of the Tester Ben Williams, Janrain	Agile Test Automation: Transition Challenges and Ways to Overcome Them Venkat Moncompu, Cognizant Technology Solutions
5:30-7:30 PM	**Conference Social Kickoff – Exhibits & Poster Papers** Complimentary hors d'oeuvres & beverages; open to the public			
6:30-7:30 PM	IIBA (International Institute of Business Analysis) Presentation **Requirements Reuse: Fantasy or Feasible?** Karl Wiegers, Process Impact			

Current as of 9/27/13

PNSQC 2013 – Conference At A Glance

Tuesday Oct 15, 2013 • Day Two • Technical Program				
7:30-8:00 AM	**Registration Open**			
8:00-9:30 AM	**KEYNOTE: When Agile Becomes a Quality Game Changer; What Data Says from Recent Agile Benchmark Research** Michael Mah, QSM Associates Inc.			
9:30-10:15 AM	**Break with Poster Papers**			
	INVITED SPEAKER	**PROJECT MANAGEMENT**	**TOPICS ON QUALITY**	**COLLABORATION**
10:15-11:00 AM	**How to Estimate ANYTHING** Payson Hall, Catalysis Group, Inc.	Test Automation: A Project Management Perspective Amith Pulla, Intel	Alternatives: A New Dimension In Software Quality Murugan Sundararaj, McAfee	The Many Faces Of A Software Engineer In A Research Community Cristina Marinovici, Harold Kirkham,PNNL
11:10-11:55 AM		Nine Low- Tech Tips for Project Management Moss Drake, Dentists Management Corp. (DMC)	Performance Benchmarking an Enterprise Message Bus Anurag Sharma, et.al., McAfee	Games In The Workplace: Revolutionary Or Run-Of-The-Mill? Chermaine Li, et.al., Microsoft
12:00-1:25 PM	**Lunch Program – Deep-Dive Discussions to Choose From**			
	INVITED SPEAKER	**PROCESS IMPROVEMENT**	**TESTING**	**AGILE**
1:30-2:15 PM	**Agile Across Cultures: When National Culture Trumps Strategy** Valerie Berset-Price, Consultant	Process Fitness - Process Improvement Lessons from Personal Trainers F. Michael Dedolph, Levi Deal Consulting	A Risk-Based Testing (RBT) Approach for the Masses Bruce Kovalsky, Capgemini	Achieving Right Automation Balance in Agile Projects Vijayagopal Narayanan, Cognizant Technology Solutions
2:25-3:10 PM		The Tail that Wags the Dogma Rhea Stadick, Intel	How Did I Miss That Bug? Peter Varhol, Seapine Software, Gerie Owen	From Dinosaur to Cutting Edge: 5 Practical Keys to Avoiding Extinction in An Agile QA World Robert Gormley, SWAT Solutions
3:15-3:45 PM	**Break with Poster Papers**			
	QUALITY ASSURANCE	**PROCESS IMPROVEMENT**	**AUTOMATION**	**AGILE**
3:45-4:30 PM	Preparing the Next Testers: An Undergraduate Course in Quality Assurance Peter Tucker, Whitworth University	Value Oriented IT Process Design and Definition: Integrating Customer Value into Process Quality Shivanand Parappa, Mangesh Khunte, Cognizant Technology Solutions	Agile Testing at Scale Mark Fink	Bridging the Gap Between Acceptance Criteria and Definition of Done Amith Pulla, Intel Sowmya Purushotham, Clinicient Inc.
4:40-5:20 PM	Quality Begins With Design Jeyasekar Marimuthu, Mcafee	Turning Projects into Products (and Back Again) Adam Light, SoTech Advisors	Championing Test Automation At A New Team: The Challenges And Benefits Alan Leung, Sierra Systems	Turning a Marathon Runner into a Sprinter: Adopting Agile Testing Strategies and Practices at Microsoft Jean Hartmann, Microsoft
5:30-6:30 PM	**Networking co-sponsored with the Rose City SPIN** Complimentary hors d'oeuvres & beverages; open to the public			
6:30-7:30 PM	**SPIN Presentation – Your Path Through Agile Fluency: Making Agile Work for You** Diana Larsen, FutureWorks Consulting and James Shore, Consultant			

Current as of 9/27/13

PNSQC™ BOARD MEMBERS, OFFICERS, and COMMITTEE CHAIRS

Doug Reynolds – Board Member & President
Tektronix, Inc.

Les Grove – Board Member, Vice President & Volunteer Chair
Catalyst IT Services

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

Ian Dees – Board Member & Secretary
Tektronix, Inc.

Brian Gaudreau – Board Member, Audit & Marketing Chairs

Michael Dedolph – Board Member & Invited Speaker Chair
Levi Deal Consulting

Kal Toth – Board Member, Conference Chair & Program Co-Chair
Persaunix Consulting

Shauna Gonzales – Conference Format Chair
Nike, Inc.

Dwayne Thomas – Networking Chair
Rentrak

PNSQC™ VOLUNTEERS

Laura Bright

Sadie Brundage

David Butt

Rick Clements

Moss Drake

Josh Eisenberg

Moji Friedhoff

Ruchir Garg

Bob Goetz

Brenda Green

Bhushan Gupta

Leesa Hicks

Aaron Hockley

Michael Hoffman

Bill Opsal

Philip Lew

Sumit Lohani

Aniket Malatpure

Cristina Marinovici

Launi Mead

Dev Panajkar

Shivanand Parappa

Dave Patterson

Amith Pulla

Emily Ren

Tim Rodani

Rajesh Sachdeva

Jeanette Schadler

Pronil Sengupta

Keith Stobie

Vadiraj Thayur

Patt Thomasson

Timothy Thomasson

Tho Tran

Heather Wilcox

PNSQC Call for Volunteers

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

Benefits of Volunteering: Professional Development, Contribution & Recognition

Opportunities to Get Involved:

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

Contact Us: by submitting your name in the conference survey which is a link sent to all PNSQC attendees or complete the contact form at www.pnsqc.org>About>Contact and address it to the PNSQC Volunteer Coordinator.

PNSQC 2014 Call for Abstracts — Technical and Poster Papers

Inspired? Got an idea you want to tell us about? Consider submitting your ideas for PNSQC 2014. All it takes is a paragraph or two and selected authors receive waived fees.

Get more information at www.pnsqc.org, link to the Call for Abstracts page for more details. Become part of the program by submitting an abstract.

SOFTWARE DEFECT ORIGINS AND REMOVAL METHODS

Capers Jones, Vice President and Chief Technology Officer

Namcook Analytics LLC www.Namcook.com

Draft 7.0 July 21, 2013

Abstract

The cost of finding and fixing bugs or defects is the largest single expense element in the history of software. Bug repairs start with requirements and continue through development. After release bug repairs and related customer support costs continue until the last user signs off. Over a 25 year life expectancy of a large software system in the 10,000 function point size range almost 50 cents out of every dollar will go to finding and fixing bugs.

Given the fact that bug repairs are the most expensive element in the history of software, it might be expected that these costs would be measured carefully and accurately. They are not. Most companies do not measure defect repair costs, and when they do they often use metrics that violate standard economic assumptions such as "lines of code" and "cost per defect" neither of which measure the value of software quality. Both of these measures distort quality economics. Lines of code penalize high-level languages. Cost per defect penalizes quality. A new metric, "technical debt" is a good metaphor but incomplete. Technical debt does not include projects with such bad quality they are canceled and never delivered so there is no downstream debt.

Poor measurement practices have led to the fact that a majority of companies do not know that achieving high levels of software quality will shorten schedules and lower costs at the same time. But testing alone is insufficient. A synergistic combination of defect prevention, pre-test defect removal, and formal testing using mathematical methods all need to be part of the quality technology stack.

SOFTWARE DEFECT ORIGINS AND REMOVAL METHODS

Introduction

The software industry spends about $0.50 out of every $1.00 expended for development and maintenance on finding and fixing bugs. Most forms of testing are below 35% in defect removal efficiency or remove only about one bug out of three. All tests together seldom top 85% in defect removal efficiency. About 7% of bug repairs include new bugs. About 6% of test cases have bugs of their own. These topics need to be measured, controlled, and improved. Security flaws are leading to major new costs for recovery after attacks. Better security is a major subset of software quality.

A synergistic combination of defect prevention, pre-test defect removal, and formal testing by certified personnel can top 99% in defect removal efficiency while simultaneously lowering costs and shortening schedules.

For companies that know how to achieve it, high quality software is faster and cheaper than low quality software. This article lists of all of the major factors that influence software quality as of year-end 2013.

These quality-related topics can all be measured and predicted using the author's Software Risk Master ™ tool. Additional data is available from the author's books, The Economics of Software Quality, Addison Wesley 2011 and Software Engineering Best Practices, McGraw Hill 2010.

Software Defect Origins

Software defects originate in multiple origins. The approximate U.S. total for defects in requirements, design, code, documents, and bad fixes is 5.00 per function point. Best in class projects are below 2.00 per function point. Projects in litigation for poor quality can top 7.00 defects per function point.

Defect potentials circa 2013 for the United States average:

• Requirements	1.00 defects per function point
• Design	1.25 defects per function point
• Code	1.75 defects per function point
• Documents	0.60 defects per function point
• Bad fixes	0.40 defects per function point
• Totals	5.00 defects per function point

Note that these values are not constants and vary from < 2.00 per function point to > 7.00 per function point based on team experience, methodology, programming language or languages, certified reusable materials, and application size.

The best results would come from small projects < 500 function points in size developed by expert teams using quality-strong methods and very high-level programming languages with substantial volumes of certified reusable materals.

The worst results would come from large systems > 50,000 function points in size developed by novice teams using ineffective methodologies and low-level programming languages with little or no use of certified reusable materials.

The major defect origins include:

1. Functional requirements
2. Non-functional requirements
3. Architecture
4. Design
5. New source code
6. Uncertified reused code from external sources
7. Uncertified reused code from legacy applications
8. Uncertified reused designs, architecture, etc.
9. Uncertified reused test cases
10. Documents (user manuals, HELP text, etc.)
11. Bad fixes or secondary defects in defect repairs (7% is U.S. average)
12. Defects due to creeping requirements that bypass full quality controls
13. Bad test cases with defects in them (6% is U.S. average)
14. Data defects in data bases and web sites
15.Security flaws that are invisible until exploited

Far too much of the software literature concentrates on code defects and ignores the more numerous defects found in requirements and design. It is also interesting that many of the companies selling quality tools such as static analysis tools and test tools focus only on code defects.

Unless requirement and design defects are prevented or removed before coding starts, they will eventually find their way into the code where it may be difficult to remove them. It should not be forgotten that the famous “Y2K” problem ended up in code, but originated as a corporate requirement to save storage space.

Some of the more annoying Windows 8 problems, such as the hidden and arcane method needed to shut down Windows 8, did not originate in the code, but rather in questionable upstream requirements and design decisions.

Why the second most common operating system command is hidden from view and requires three mouse clicks to execute it is a prime example of why applications need requirement inspections, design inspections, and usability studies as well as ordinary code testing.

Proven Methods for Preventing and Removing Software Defects

Defect Prevention

The set of defect prevention methods can lower defect potentials from U.S. averages of about 5.00 per function point down below 2.00 per function point. Certified reusable materials are the most effective known method of defect prevention. A number of Japanese quality methods are beginning to spread to other countries and are producing good results. Defect prevention methods include:

1. Joint Application Design (JAD)
2. Quality function deployment (QFD)
3. Certified reusable requirements, architecture, and design segments
4. Certified reusable code
5. Certified reusable test plans and test cases (regression tests)
6. Kanban for software (mainly in Japan)
7. Kaizen for software (mainly in Japan)
8. Poka-yoke for software (mainly in Japan)
9. Quality circles for software (mainly in Japan)
10. Six Sigma for Software
11. Achieving CMMI levels => 3 for critical projects
12. Using quality-strong methodologies such as RUP and TSP
13. Embedded users for small projects < 500 function points
14. Formal estimates of defect potentials and defect removal before starting projects

15. Formal estimates of cost of quality (COQ) and technical debt (TD) before starting
16. Quality targets such as > 97% defect removal efficiency (DRE) in all contracts
17. Function points for normalizing quality data
18. Analysis of user-group requests or customer suggestions for improvement
19. Formal inspections (leads to long-term defect reductions)
20. Pair programming (leads to long-term defect reductions)

Analysis of software defect prevention requires measurement of similar projects that use and do not use specific approaches such as JAD or QFD. Of necessity studying defect prevention needs large numbers of projects and full measures of their methods and results.

Note that two common metrics for quality analysis, "lines of code" and "cost per defect" have serious flaws and violate standard economic assumptions. These two measures conceal, rather than reveal, the true economic value of high software quality.

Another newer measure is that of "technical debt" or the downstream costs of repairing defects that are in software released to customers. Although technical debt is an interesting metaphor it is not complete and ignores several major quality costs.

The most serious omission from technical debt are projects where quality is so bad that the application is canceled and never delivered at all. Canceled projects have a huge cost of quality but zero technical debt since they never reach customers. Another omission from technical debt is the cost of financial awards by courts where vendors are sued and lose lawsuits for poor quality.

Function point metrics are the best choice for quality economic studies. The new SNAP non-functional size metric has recently been released, but little quality data is available because that metric is too new.

The new metric concept of "technical debt" is discussed again later in this article, but needs expansion and standardization.

Pre-Test Defect Removal

What happens before testing is even more important than testing itself. The most effective known methods of eliminating defects circa 2013 include requirements models, automated proofs, formal inspections of requirements, design, and code; and static analysis of code and text.

(A popular method for Agile and extreme programming (XP) projects is that of "pair programming." This method is expensive and does not have better quality than individual programmers who use static analysis and inspections. Pair programming would not exist if companies had better measures for costs and quality. Unlike inspection pair programming was not validated prior to introduction, and still has little available benchmark data whereas inspections have thousands of measured projects.)

Formal inspections have been measured to top 85% in defect removal efficiency and have more than 40 years of empirical data from thousands of projects. Methods such as inspections also raise testing defect removal efficiency by more than 5% for each major test stage. The major forms of pre-test defect removal include:

1. Desk checking by developers
2. Debugging tools (automated)
3. Pair programming (with caution)
4. Quality Assurance (QA) reviews of major documents and plans
5. Formal inspections of requirements, design, code, UML, and other deliverables
6. Formal inspections of requirements changes
7. Informal peer reviews of requirements, design, code
8. Editing and proof reading critical requirements and documents
9. Text static analysis of requirements, design
10. Code static analysis of new, reused, and repaired code
11. Running FOG and FLESCH readability tools on text documents
12. Requirements modeling (automated)
13. Automated correctness proofs
14. Refactoring
15. Independent verification and validation (IV&V)

Pre-test inspections have more than 40 years of empirical data available and rank as the top method of removing software defects, consistently topping 85% in defect removal efficiency (DRE).

Static analysis is a newer method that is also high in DRE, frequently toping 65%. Requirements modeling is another new and effective method that has proved itself on complex software such as that operating the Mars Rover. Requirements modeling and inspections can both top 85% in defect removal efficiency (DRE).

As mentioned earlier, one of the more unusual off shoots of some of the Agile methods such as extreme programming (XP) is “pair programming.” The pair programming approach is included in the set of pre-test defect removal activities.

With pair programming two individuals share an office and work station and take turns coding while the other observes.

This should have been an interesting experiment, but due to poor measurement practices it has started into actual use, with expensive results. Individual programmers who use static analysis and inspections have better quality at about half the cost and 75% of the schedule of a pair.

If two top guns are paired the results will be good, but the costs about 40% higher than either one working alone. Since there is a severe shortage of top-gun software engineers, it is not cost effective to have two of them working on the same project. It would be better for each of them to tackle a separate important project. Top-guns only comprise about 5% of the overall software engineering population.

If a top gun is paired with an average programmer, the results will be better than the average team member might product, but about 50% more expensive. The quality is no better than the more experienced pair member working alone. If pairs are considered a form of mentoring there is some value for improving the performance of the weaker team member.

If two average programmers are paired the results will still be average, and the costs will be about 80% higher than either one alone.

If a marginal or unqualified person is paired with anyone, the results will be suboptimal and the costs about 115% higher than the work of the better team member working alone. This is because the unqualified person is a drag on the performance of the pair.

Since there are not enough qualified top-gun programmers to handle all of the normal work in many companies, doubling them up adds costs but subtracts from the available work force.

There are also 116 occupation groups associated with software. If programmers are to be paired, why not pair architects, designers, testers, and project managers?

Military history is not software but it does provide hundreds of examples that shared commands often lead to military disaster. The Battle of Cannae is one such example, since the Roman commanders alternated command days.

Some pairs of authors can write good books, such as Douglas Preston and Lincoln Child. But their paired books are no better than the books each author wrote individually

Pairing should have been measured and studied prior to becoming an accepted methodology, but instead it was put into production with little or no empirical data. This phenomenon of rushing to use the latest fad without any proof that it works is far too common for software.

Most of the studies of pair programming do not include the use of inspections or static analysis. They merely take a pair of programmers and compare the results against one unaided programmer who does not use modern pre-test removal methods such as static analysis and peer reviews.

Two carpenters using hammers and hand saws can certainly build a shed faster than one carpenter using a hammer and a hand saw. But what about one carpenter using a nail gun and an electric circular saw? In this case the single carpenter might well win if the pair is only using hammers and hand saws.

By excluding other forms of pre-test defect removal such as inspections and static analysis the studies of pair programming are biased and incomplete.

Test Defect Removal

Testing has been the primary software defect removal method for more than 50 years. Unfortunately most forms of testing are only about 35% efficient or find only one bug out of three.

Defects in test cases themselves and duplicate test cases lower test defect removal efficiency. About 6% of test cases have bugs in the test cases themselves. In some large companies as many as 20% of regression test libraries are duplicates which add to testing costs but not to testing rigor.

Due to low defect removal efficiency at least eight forms of testing are needed to achieve reasonably efficient defect removal efficiency. Pre-test inspections and static analysis are synergistic with testing and raise testing efficiency.

Tests by certified test personnel using test cases designed with formal mathematical methods have the highest levels of test defect removal efficiency and can top 65%. The major forms of test-related factors for defect removal include:

1. Certified test personnel
2. Formal test plans published and reviewed prior to testing
3. Certified reusable test plans and test cases for regression testing
4. Mathematically based test case design such as using design of experiments
5. Test coverage tools for requirements, code, data, etc.
6. Automated test tools
7. Cyclomatic complexity tools for all new and changed code segments
8. Test library control tools
9. Capture-recapture testing
10. Defect tracking and routing tools
11. Inspections of major code changes prior to testing
12. Inspections of test libraries to remove bad and duplicate test cases
13. Special tests for special defects; performance, security, etc.
14. Full suite of test stages including:
    a. Subroutine test
    b. Unit test
    c. New function test
    d. Regression test
    e. Component test
    f. Performance test
    g. Usability test
    h. Security test

i. System test
j. Supply-chain test
k. Cloud test
l. Data migration test
m. ERP link test
n. External beta test
o. Customer acceptance test
p. Independent test (primarily military projects)

Testing by itself without any pre-test inspections or static analysis is not sufficient to achieve high quality levels. The poor estimation and measurement practices of the software industry have long slowed progress on achieving high quality in a cost-effective fashion.

However modern risk-based testing by certified test personnel with automated test tools who also use mathematically-derived test case designs and also tools for measuring test coverage and cyclomatic complexity can do a very good job and top 65% in defect removal efficiency for the test stages of new function test, component test, and system test.

Untrained amateur personnel such as developers themselves seldom top 35% for any form of testing. Also the "bad fix injection" rate or new bugs added while fixing older bugs tops 7% for repairs by ordinary development personnel.

Bad fixes are inversely proportional to cyclomatic complexity, and also inversely proportional to experience. Bad fixes by a top-gun software engineer working with software with low cyclomatic complexity can be only a fraction of 1%.

At the other end of the spectrum, bad fixes by a novice trying to fix a bug in an error-prone module with high cyclomatic complexity can top 25%.

In one lawsuit where the author was an expert witness the vendor tried four times over nine months to fix a bug in a financial package. Each of the first four fixes failed, and each added new bugs. Finally after nine months the fifth bug repair fixed the original bug and did not cause regressions. By then the client had restated prior year financials, and that was the reason for the lawsuit.

Another issue that is seldom discussed in the literature is that of bugs or errors in the test cases themselves. On average about 6% of test cases contain errors. Running defective test cases adds costs to testing but nothing to defect removal efficiency. In fact defective test cases lower DRE.

National Defect Removal Efficiency and Software Quality

Some countries such as Japan, India, and South Korea place a strong emphasis on quality in all manufactured products including software. Other countries, such a China and Russia, apparently have less interest and less understanding of quality economics and seem to lag in quality estimation and measurement. Among the quality-strongcountries Japan, for example, had more than 93% DRE on the projects examined by the author.

From a comparatively small number of international studies the approximate national rankings for defect removal efficiency levels of the top 20 countries in terms of DRE are:

Quality Ranks of 20 Countries:

1. Japan
2. India
3. Denmark
4. South Korea
5. Switzerland
6. Israel
7. Canada
8. United Kingdom
9. Sweden
10. Norway
11. Netherlands
12. Hungary
13. Ireland
14. United States
15. Brazil
16. France
17. Australia
18. Austria
19. Belgium
20. Finland

All of the countries in the top 20 can produce excellent software and often do. Countries with significant amounts of systems and embedded software and defense software are more likely to have good quality control than countries producing mainly information technology packages.

Almost all countries in 2013 produce software in significant volumes. More than 150 countries produce millions of function points per year. The preliminary ranks shown here indicate that more studies are needed on international software quality initiatives.

Countries that might be poised to join the top-quality set in the future include Malaysia, Mexico, the Philippines, Singapore, Taiwan, Thailand, and Viet Nam. Russia and China should be top-ranked but are not, other than Hong Kong.

Quality measures and predictive quality estimation are necessary precursors to achieving top quality status. Defect prevention and pre-test defect removal must be added to testing to achieve top-rank status.

One might think that the wealthier countries of the Middle East such as Dubai, Saudi Arabia, Kuwait, and Jordan would be in the top 20 due to regional wealth and the number of major software-producing companies located there. Although the Middle East ranks near the top in requesting benchmark data, little or no data has been published from the region about software quality.

Industry Defect Removal Efficiency and Software Quality

In general the industries that produce complex physical devices such as airplanes, computers, medical devices, and telephone switching systems have the highest levels of defect removal efficiency, the best quality measures, and the best quality estimation capabilities.

This is a necessity because these complex devices won't operate unless quality approaches zero defects. Also, the manufacturers of such devices have major liabilities in case of failures including possible criminal charges. The top 20 industries in terms of defect removal efficiency are:

Quality Ranks of 20 U.S. Industries:

1. Government – intelligence agencies
2. Manufacturing – medical devices
3. Manufacturing – aircraft
4. Manufacturing – mainframe computers
5. Manufacturing – telecommunications switching systems
6. Telecommunications – operations
7. Manufacturing – defense weapons systems
8. Manufacturing – electronic devices and smart appliances
9. Government – military services
10. Entertainment – film and television production
11. Manufacturing – pharmaceuticals
12. Transportation - airlines
13. Manufacturing – tablets and personal computers
14. Software – commercial
15. Manufacturing – chemicals and process control
16. Banks – commercial
17. Banks – investment
18. Health care – medical records
19. Software – open source
20. Finance – credit unions

As of 2013 more than 500 industries produce software in significant volumes. Some of the lagging industries that are near the bottom in terms of software defect removal efficiency levels are those of state governments, municipal governments, wholesale chains, retail chains, public utilities, cable television billing and finance, and some but not all insurance companies in the areas of billing and accounting software. Even worse are some of the companies that do programmed stock market trading, which all by themselves have triggered major financial disruptions due to software bugs.

For example in Rhode Island one of the author's insurance companies seems to need more than a week to post payments and often loses track of payments. Once the author's insurance company even sent back a payment check with a note that it had been "paid too early." Apparently the company was unable to add early payments to accounts. (The author was leaving on an international trip and wanted to pay bills prior to departure.)

Another more recent issue involved the insurance company unilaterally changing all account numbers. Unfortunately they seemed not to develop a good method for mapping clients' old account numbers to the new account numbers.

A payment made just after the cut over but using the old account number required 18 days from when the check reached the insurance company until it was credited to the right account. Before that the company had sent out a cancellation notice for the policy. From discussions with other clients, apparently the company loses surprisingly many payments. This is a large company with a large IT staff and hundreds of clerical workers.

Companies that are considering outsourcing may be curious as to the placement of software outsource vendors. From the author's studies of various industries outsource vendors rank as number 25 out of 75 industries for ordinary information technology outsourcing. For embedded and systems software outsourcing the outsource vendors are approximately equal to industry averages for aircraft, medical device, and electronic software packages.

Another interesting question is how good are the defect removal methods practiced by quality companies themselves such as static analysis companies, automated test tool companies, independent testing companies, and defect tracking tool companies?

Interestingly, these companies publish no data about their own results and seem to avoid having outside consulting studies done that would identify their own defect removal efficiency levels. No doubt the static analysis companies use their own tools on their own software, but they do not publish accurate data on the measured effectiveness of these tools.

All of the test and static analysis companies should publish annual reports that show ranges of defect removal efficiency (DRE) results using their tools, but none are known to do this.

Software Development Methods

Some software development methods such as IBM's Rational Unified Process (RUP) and Watts Humphrey's Team Software Process (TSP) can be termed "quality strong" because they lower defect potentials and elevate defect removal efficiency levels.

Other methods such as Waterfall and Cowboy development can be termed "quality weak" because they raise defect potentials and have low levels of defect removal efficiency. The 30 methods shown here are ranked in approximate order of quality strength. The list is not absolute and some methods are better than others for specific sizes and types of projects. Development methods in rank order of defect prevention include:

1. Mashup (construction from certified reusable components)
2. Hybrid
3. IntegraNova
4. TSP/PSP
5. RUP
6. T-VEC
7. Extreme Programming (XP)
8. Agile/Scrum
9. Data state design (DSD)
10. Information Engineering (IE)
11. Object-Oriented (OO)
12. Rapid Application Development (RAD)
13. Evolutionary Development (EVO)
14. Jackson development
15. Structured Analysis and Design Technique (SADT)
16. Spiral development
17. Structured systems analysis and design method (SSADM)
18. Iterative development
19. Flow-based development
20. V-Model development
21. Prince2
22. Merise
23. Data state design method (DSDM)
24. Clean-room development
25. ISO/IEC
26. Waterfall
27. Pair programming
28. DoD 2167A
29. Proofs of correctness (manual)
30. Cowboy

Once again this list is not absolute and situations change. Since Agile development is so popular, it should be noted that Agile is fairly strong in quality but not the best in quality. Agile projects frequently achieve DRE in the low 90% range, which is better than average but not top-ranked.

Agile lags many leading methods in having very poor quality measurement practices. The poor measurement practices associated with Agile for both quality and productivity will eventually lead CIO's, CTO's, CFO's, and CEO's to ask if actual Agile results are as good as being claimed.

Until Agile projects publish productivity data using function point metrics and quality data using function points and defect removal efficiency (DRE) the effectiveness of Agile remains ambiguous and uncertain.

Studies by the author found Agile to be superior in both quality and productivity to waterfall development, but not as good for quality as either RUP or TSP.

Also, a Google search using phrases such as “Agile failures” and “Agile successes” turns up about as many discussions of failure as success. A new occupation of “Agile coach” has emerged to help reduce the instances of getting off track when implementing Agile.

Overall Quality Control

Successful quality control stems from a synergistic combination of defect prevention, pre-test defect removal, and test stages. The best projects in the industry circa 2013 combined defect potentials in the range of 2.0 defects per function point with cumulative defect removal efficiency levels that top 99%. The U.S. average circa 2013 is about 5.0 bugs per function point and only about 85% defect removal efficiency. The major forms of overall quality control include:

1. Formal software quality assurance (SQA) teams for critical projects
2. Measuring defect detection efficiency (DDE)
3. Measuring defect removal efficiency (DRE)
4. Targets for topping 97% in DRE for all projects
5. Targets for topping 99% in DRE for critical projects
6. Inclusion of DRE criteria in all outsource contracts (> 97% is suggested)
7. Formal measurement of cost of quality (COQ)
8. Measures of “technical debt” but augmented to fill major gaps
9. Measures of total cost of ownership (TCO) for critical projects
10. Monthly quality reports to executives for on-going and released software
11. Production of an annual corporate software status and quality report
12. Achieving > CMMI level 3

IBM started to measure defect origins, defect potentials, and defect removal efficiency (DRE) levels in the early 1970’s. These measures were among the reasons for IBM’s market success in both hardware and software. High quality products are usually cheaper to produce, are much cheaper to maintain, and bring high levels of customer loyalty.

The original IBM DRE studies used six months after release for calculating DRE, but due to updates that occur before six months that interval was difficult to use and control. The switch from six month to 90-day DRE intervals occurred in 1984.

Defect removal efficiency is measured by accumulating data on all bugs found prior to release and also on bugs reported by clients in the first 90 days of use. If developers found 90 bugs and users reported 10 bugs then DRE is clearly 90%.

The International Software Benchmark Standards Group (ISBSG) uses only a 30 day interval after release for measuring DRE. The author measures both 30-day and 90-day intervals.

Unfortunately the 90-day defect counts average about four to five times larger than the 30-day defect counts, due to installation and learning curves of software which delay normal usage until late in the first month.

A typical 30-day ISBSG count of DRE might show 90 bugs found internally and 2 bugs found in 30 days, for a DRE of 97.82%.

A full 90-day count of DRE would still show 90 bugs found internally but 10 bugs found in three months for a lower DRE of only 90.00%.

Although a fixed time interval is needed to calculate DRE, that does not mean that all bugs are found in only 90 days. In fact the 90-day DRE window usually finds less than 50% of the bugs reported by clients in one calendar year.

Bug reports correlate strongly with numbers of production users of software applications. Unless a software package is something like Windows 8 with more than 1,000,000 users on the first day, it usually takes at least a month to install complex applications, train users, and get them started on production use.

If there are less than 10 users the first month, there will be very few bug reports. Therefore in addition to measuring DRE, it is also significant to record the numbers of users for the first three months of the application's production runs.

If we assume an ordinary information technology application the following table shows the probable numbers of reported bugs after one, two, and three months for 10, 100, and 1000 users:

Defects by Users for Three Months

Month	10 Users	100 Users	1000 Users
1	1	3	6
2	3	9	18
3	6	12	24

As it happens the central column of 100 users for three months is a relatively common pattern.

Note that for purposes of measuring defect removal efficiency a single month of usage tends to yield artificially high levels of DRE due to a normal lack of early users.

Companies such as IBM with continuous quality data are able to find out many interesting and useful facts about defects that escape and are delivered to clients. For example for financial software there will be extra bug reports at the end of standard fiscal years, due to exercising annual routines.

Also of interest is the fact that about 15% of bug reports are “invalid” and not true bugs at all. Some are user errors, some are hardware errors, and some are bugs against other software packages that were mistakenly reported to the wrong place. It is very common to confuse bugs in operating systems with bugs in applications.

As an example of an invalid defect report, the author's company once received a bug report against a competitive product, sent to us by mistake. Even though this was not a bug against our software, we routed it to the correct company and sent a note back to the originator as a courtesy.

It took about an hour to handle a bug against a competitive software package. Needless to say invalid defects such as this do not count as technical debt or cost of quality (COQ). However they do count as overhead costs.

An interesting new metaphor called “technical debt” was created by Ward Cunningham and is now widely deployed, although it is not deployed the same way by most companies. Several software quality companies such as OptiMyth in Spain, CAST Software, and SmartBear feature technical debt discussions on their web sites.

The concept of technical debt is intuitively appealing. Shortcuts made during development that lead to complex code structures or to delivered defects will have to be fixed at some point in the future. When the time comes to fix these problems downstream, the costs will be higher and the schedules longer than if they had been avoided in the first place.

The essential concept of technical debt is that questionable design and code decisions have increasing repair costs over time. As a metaphor or interesting concept technical debt has much to recommend it.

But the software industry is far from sophisticated in understanding finance and economic topics. In fact for more than 50 years the software industry has tried to measure quality costs with "lines of code" and "cost per defect" which are so inaccurate as to be viewed as professional malpractice for quality economics.

Also, many companies only measure about 37% of software project effort and 38% of software defects. Omitting unpaid overtime, managers, and specialists are common gaps. Omitting bugs found in requirements, design, and by unit testing are common quality omissions.

Until the software industry adopts standard charts of accounts and begins to use generally acceptable accounting principles (GAAP) measures of technical debt will vary widely from company to company and not be comparable.

Technical debt runs head on into the general ineptness of the software world in understanding and measuring the older cost of quality (COQ) in a fashion that matches standard economic assumptions. Cost per defect penalizes quality. Lines of code penalize modern high-level languages and of course make requirements and design defects invisible. Defect repair costs per function point provide the best economic indicator. However the new SNAP metric for non-functional requirements needs to be incorporated.

The main issues with technical debt as widely deployed by the author's clients are that it does not include or measure some of the largest quality costs in all of software history.

About 35% of large software systems are cancelled and never delivered at all. The most common reason for cancellation is poor quality. But since the cancelled projects don't get delivered, there are no downstream costs and hence no technical debt either. The costs of cancelled projects are much too large to ignore and just leave out of technical debt.

The second issue involves software that does get delivered and indeed accumulates technical debt in the form of changes that need to be repaired. But some software applications have such bad quality that clients sue the developers for damages. The costs of litigation and the costs of any damages that the court orders software vendors to pay should be part of technical debt.

What about the consequential damages that poor software quality brings to the clients who have been harmed by the upstream errors and omissions? Currently technical debt as used by most companies is limited to internal costs borne by the development organization.

For example suppose a bug in a financial application caused by rushing through development costs the software vendor $100,000 to fix a year after release, and it could have been avoided for only $10,000. The expensive repair is certainly technical debt that might have been avoided.

Now suppose this same bug damaged 10 companies and caused each of them to lose $3,000,000 due to having to restate prior year financial statements. What about the $30,000,000 in consequential damages to users of the software? These damages are currently not considered to be part of technical debt.

If the court orders the vendor to pay for the damages and the vendor is charged $30,000,000 that probably should be part of technical debt. Litigation costs and damages are not currently included in the calculations most companies use for technical debt.

For financial debt there is a standard set of principles and practices called the "Generally Accepted Accounting Principles" or GAAP. The software industry in general, and technical debt in particular, need a similar set of "Software Generally Accepted Accounting Principles" or SGAAP that would allow software projects and software costs to be compared in a uniform fashion.

As this article is being written the United States GAAP rules are being phased out in favor of a newer set of international financial rules called International Financial Reporting Standards (IFRS). Here too software needs a set of "Software International Financial Reporting Standards" or SIFRS to ensure accurate software accounting across all countries.

Software engineers interested in technical debt are urged to read the GAAP and IFRS accounting standards and familiarize themselves with normal cost accounting as a precursor to applying technical debt.

The major GAAP principles are relevant to software measures and also to technical debt:

1. Principle of regularity
2. Principle of consistency
3. Principle of sincerity
4. Principle of permanence of methods
5. Principle of non-compensation or not replacing a debt with an asset
6. Principle of prudence
7. Principle of continuity
8. Principle of periodicity
9. Principle of full disclosure
10. Principle of utmost good faith

The major software metric associations such as the International Function Point User Group (IFPUG) and the Common Software Metric International Consortium (COSMIC) should both be participating in establishing common financial principles for measuring software costs, including cost of quality and technical debt. However neither group has done much outside of basic sizing of applications. Financial reporting is still ambiguous for the software industry as a whole.

Interestingly the countries of Brazil and South Korea, which require function point metrics for government software contracts, appear to be somewhat ahead of the United States and Europe in melding financial accounting standards with software projects. Even in Brazil and South Korea costs of quality and technical debt remain ambiguous.

Many companies are trying to use technical debt because it is an intriguing metaphor that is appealing to CFO's and CEO's. However without some form of SIFRS or standardized accounting principles every company in every country is likely to use technical debt with random rules that would not allow cross-country, cross-company, or cross-project comparisons.

Harmful Practices to be Avoided

Some of the observations of harmful practices stem from lawsuits where the author has worked as an expert witness. Discovery documents and depositions reveal quality flaws that are not ordinarily visible or accessible to standard measurements.

In every case where poor quality was alleged by the plaintiff and proven in court, there was evidence that defect prevention was lax, pre-test defect removal such as inspections and static analysis were bypassed, and testing was either perfunctory or truncated to meet arbitrary schedule targets.

These poor practices were unfortunate because a synergistic combination of defect prevention, pre-test defect removal, and formal testing leads to short schedules, low costs, and high quality at the same time.

The most severe forms of schedule slips are due to starting testing with excessive numbers of latent defects, which stretch out testing intervals by several hundred percent compared to original plans. Harmful and dangerous practices to be avoided are:

1. Bypassing pre-test inspections
2. Bypassing static analysis
3. Testing by untrained, uncertified amateurs
4. Truncating testing for arbitrary reasons of schedule
5. The "good enough" quality fallacy
6. Using "lines of code" for data normalization (professional malpractice)
7. Using "cost per defect" for data normalization (professional malpractice)
8. Failure to measure bugs at all
9. Failure to measure bugs before release
10. Failure to measure defect removal efficiency (DRE)
11. Error-prone modules (EPM) with high defect densities
12. High cyclomatic complexity of critical modules
13. Low test coverage of critical modules
14. Bad-fix injections or new bugs in bug repairs themselves
15. Outsource contracts that do not include quality criteria and DRE
16. Duplicate test cases that add costs but not test thoroughness
17. Defective test cases with bugs of their own

It is an unfortunate fact that poor measurement practices, failure to use effective quality predictions before starting key projects, and bypassing defect prevention and pre-test defect removal methods have been endemic problems of the software industry for more than 40 years.

Poor software quality is like the medical condition of whooping cough. That condition can be prevented via vaccination and in today's world treated effectively.

Poor software quality can be eliminated by the "vaccination" of early estimation and effective defect prevention. Pre-test defect removal such as inspections and static analysis are effective therapies. Poor software quality is a completely treatable and curable condition.

It is technically possible to lower defect potential from around 5.00 per function point to below 2.00 per function point. It is also technically possible to raise defect removal efficiency (DRE) from today's average of about 85% to at least 99%. These changes would also shorten schedules and reduce costs.

Illustrating Software Defect Potentials and Defect Removal Efficiency (DRE)

Figure one shows overall software industry results in terms of two dimensions. The vertical dimension shows defect potentials or the probable total number of bugs that will occur in requirements, design, code, documents, and bad fixes:

Note that large systems have much higher defect potentials than small applications. It is also harder to remove defects from large systems.

Figure 1: U.S. Ranges of Software Defect Potentials and Defect Removal Efficiency (DRE)

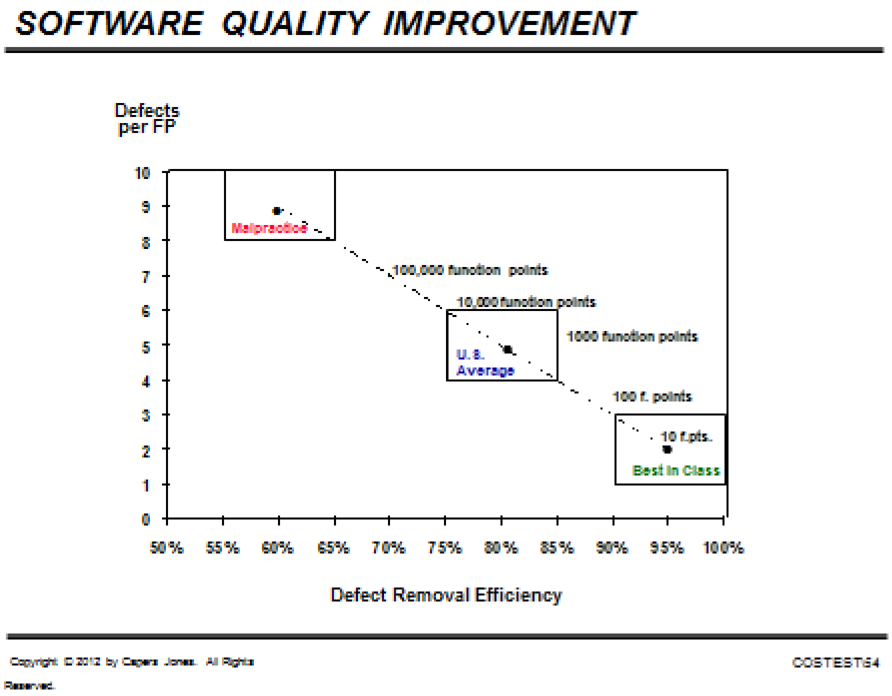
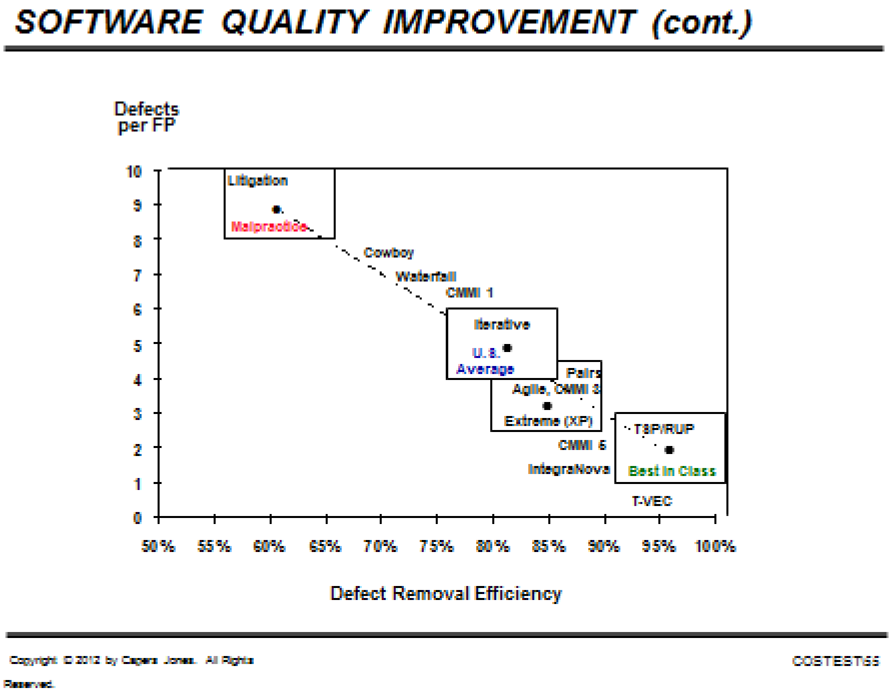


Figure 2 shows the relationship between software methodologies and software defect potentials and defect removal:

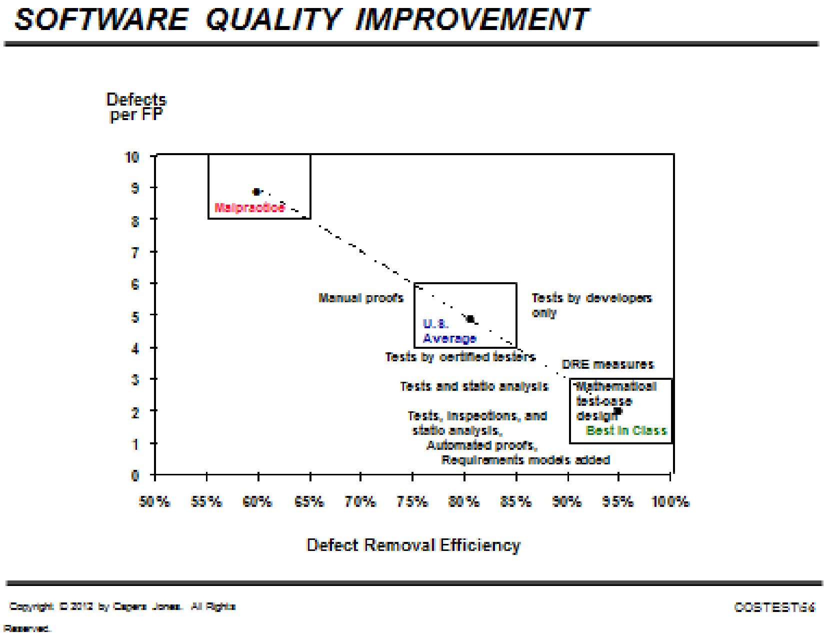
Figure 2: Methodologies and Software Defect Potentials and Removal



Note that “quality strong” methods have fewer defects to remove and remove a much higher percentage of software defects than the “quality weak’ methods.

Figure 3 shows the relationship between various forms of defect removal and effective quality results:

Figure 3: Software Defect Removal Methods and Removal Efficiency



Note that testing by untrained amateur developers is neither efficient in finding bugs nor cost effective. A synergistic combination of defect prevention, pre-test defect removal, and formal testing gives the best quality results at the lowest costs and the shortest schedules.

These three illustrations show basic facts about software defects and defect removal efficiency:

- Defect volumes increase with application size
- Quality-strong methods can reduce defect potentials
- Synergistic combinations of defect prevention, pre-test removal and testing are needed.

Achieving a high level of software quality is the end product of an entire chain of methods that start with 1) defect measurements, 2) defect estimation before starting projects, 3) careful defect prevention, 4) pre-test inspections and static analysis, and ends with 5) formal testing using mathematically designed test cases.

All five links of this chain are needed
er costs, and longer schedules than including all five links.

Quantifying a Best-Case Scenario for Defect Removal Efficiency

To illustrate the principles of optimal defect prevention, pre-test removal, and test defect removal table 1 shows sample outputs from Software Risk Master ™ for a best-case scenario. This scenario assumes 1,000 function points, a top-gun team, CMMI level 5, hybrid methodology, the Objective-C programming language, and a monthly burdened compensation rate of $10,000:

Table 1: Best-Case Scenario	**OUTPUT DATA**
Requirements defect potential	**134**
Design defect potential	**561**
Code defect potential	**887**
Document defect potential	**135**
Total Defect Potential	**1,717**
Per function point	**1.72**
Per KLOC	**32.20**

Defect Prevention	**Efficiency**	**Remainder**	**Bad Fixes**	**Costs**
JAD	27%	**1,262**	**5**	$28,052
QFD	30%	**88 8**	**4**	$39,633
Prototype	20%	**71 3**	**2**	$17,045
Models	68%	**22 9**	**5**	$42,684
Subtotal	**86%**	**23 4**	**15**	$127,415

Pre-Test Removal	**Efficiency**	**Remainder**	**Bad Fixes**	**Costs**
Desk check	27%	**17 1**	**2**	$13,225
Static analysis	55%	**7 8**	**1**	$7,823
Inspections	93%	**5**	**0**	$73,791
Subtotal	**98%**	**6**	**3**	$94,839

Test Removal	Efficiency	Remainder	BadFixes	Costs
Unit	32%	4	0	$22,390
Function	35%	2	0	$39,835
Regression	14%	2	0	$51,578
Component	32%	1	0	$57,704
Performance	14%	1	0	$33,366
System	36%	1	0	$63,747
Acceptance	17%	1	0	$15,225
Subtotal	87%	1	0	$283,845

				Costs
PRE-RELEASE COSTS		1,734	3	$506,099
POST-RELEASE REPAIRS	(TECHNICAL DEBT)	1	0	$658
MAINTENANCE OVERHEAD				$46,545
COST OF QUALITY (COQ)				$553,302

Defects delivered	**1**
High severity	**0**
Security flaws	**0**
High severity %	**11.58%**
Delivered Per FP	**0.001**
High severity per FP	**0.000**
Security flaws per FP	**0.000**
Delivered Per KLOC	**0.014**
High severity per KLOC	**0.002**
Security flaws per KLOC	**0.001**
Cumulative Removal Efficiency	**99.96%**

This scenario utilizes a sophisticated combination of defect prevention and pre-test defect removal as well as formal testing by certified test personnel. Note that cumulative DRE is 99.96%, which is about as good as ever achieved.

Quantifying a Worst-Case Scenario for Defect Removal Efficiency

To illustrate the principles of inadequate defect prevention, pre-test removal, and test defect removal table 2 shows sample outputs from Software Risk Master ™ for a worst-case scenario. This scenario assumes 1,000 function points, a novice team, CMMI level 1, waterfall methodology, the Java programming language, and a monthly burdened compensation rate of $10,000:

Table 2: Worst-Case Scenario	**OUTPUT DATA**
Requirements defect potential	**327**
Design defect potential	**584**
Code defect potential	**1,109**
Document defect potential	**140**
Total Defect Potential	**2,160**
Per function point	**2.16**
Per KLOC	**40.50**

Defect Prevention	**Efficiency**	**Remainder**	**Bad Fixes**	**Costs**
JAD - not used	0%	**2,158**	**0**	$0
QFD - not used	0%	**2,156**	**0**	$0
Prototype	20%	**1,725**	**22**	$17,045
Models - not used	0%	**1,744**	**0**	$0
Subtotal	**19%**	**1,743**	**21**	$17,045

Pre-Test Removal	**Efficiency**	**Remainder**	**Bad Fixes**	**Costs**
Desk check	23%	**1,342**	**67**	$19,734
Static analysis - not used	0%	**1,408**	**0**	$0
Inspections - not used	0%	**1,407**	**0**	$0
Subtotal	**19%**	**1,407**	**67**	$19,734

Test Removal	**Efficiency**	**Remainder**	**Bad Fixes**	**Costs**
Unit	28%	**1,013**	**51**	$59,521
Function	31%	**734**	**73**	$134,519
Regression	10%	**726**	**36**	$53,044
Component	28%	**549**	**55**	$91,147
Performance	6%	**568**	**28**	$30,009
System	32%	**405**	**41**	$102,394
Acceptance	13%	**388**	**19**	$25,058
Subtotal	**72%**	**388**	**304**	$495,691

				Costs
PRE-RELEASE COSTS		**2,144**	**371**	**$532,470**
POST-RELEASE REPAIRS	**(TECHNICAL DEBT)**	**388**	**19**	**$506,248**
MAINTENANCE OVERHEAD				**$298,667**
COST OF QUALITY	**(COQ)**			**$1,337,385**

Defects delivered	**407**
High severity	**103**
Security flaws	**46**
High severity %	**25.25%**
Delivered Per FP	**0.407**
High severity per FP	**0.103**
Security flaws per FP	**0.046**
Delivered Per KLOC	**7.639**
High severity per KLOC	**1.929**
Security flaws per KLOC	**0.868**
Cumulative Removal Efficiency	**81.14%**

Note that with the worst-case scenario defect prevention was sparse and pre-test defect removal omitted the two powerful methods of inspections and static analysis. Cumulative DRE was only 81.14% which is below average and below minimum acceptable quality levels. If this had been an outsource project then litigation would probably have occurred.

It is interesting that the best-case and worst-case scenarios both used exactly the same testing stages. With the best-case scenario test DRE was 87% while the test DRE for the worst-case scenario was only 72%. The bottom line is that testing needs the support of good defect prevention and pre-test defect removal.

Note the major differences in costs of quality (COQ) between the best-case scenarios. The best-case COQ was only $553,302 while the COQ for the worst-case scenario was more than double, or $1,337,385.

The technical debt differences are even more striking. The best-case scenario had only 1 delivered defect and a technical debt of only $658. For the worst-case scenario there were 388 delivered defects and repair costs of $506,248.

For software, not only is quality free but it leads to lower costs and shorter schedules at the same time.

Summary and Conclusions on Software Quality

The software industry spends more money on finding and fixing bugs than for any other known cost driver. This should not be the case. A synergistic combination of defect prevention, pre-test defect removal, and formal testing can lower software defect removal costs by more than 50% compared to 2012 averages. These same synergistic combinations can raise defect removal efficiency (DRE) from the current average of about 85% to more than 99%.

Any company or government group that averages below 95% in cumulative defect removal efficiency (DRE) is not adequate in software quality methods and needs immediate improvements.

Any company or government group that does not measure DRE and does not know how efficient they are in finding software bugs prior to release is in urgent need of remedial quality improvements.

When companies that do not measure DRE are studied by the author during on-site benchmarks, they are almost always below 85% in DRE and usually lack adequate software quality methodologies. Inadequate defect prevention and inadequate pre-test defect removal are strongly correlated with failure to measure defect removal efficiency.

Phil Crosby, the vice president of quality for ITT, became famous for the aphorism “quality is free.” For software quality is not only free but leads to shorter development schedules, lower development costs, and greatly reduced costs for maintenance and total costs of ownership (TCO).

References and Readings on Software Quality

Beck, Kent; Test-Driven Development; Addison Wesley, Boston, MA; 2002; ISBN 10: 0321146530; 240 pages.

Black, Rex; Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing; Wiley; 2009; ISBN-10 0470404159; 672 pages.

Chelf, Ben and Jetley, Raoul; "*Diagnosing Medical Device Software Defects Using Static Analysis*"; Coverity Technical Report, San Francisco, CA; 2008.

Chess, Brian and West, Jacob; Secure Programming with Static Analysis; Addison Wesley, Boston, MA; 20007; ISBN 13: 978-0321424778; 624 pages.

Cohen, Lou; Quality Function Deployment – How to Make QFD Work for You; Prentice Hall, Upper Saddle River, NJ; 1995; ISBN 10: 0201633302; 368 pages.

Crosby, Philip B.; Quality is Free; New American Library, Mentor Books, New York, NY; 1979; 270 pages.

Everett, Gerald D. And McLeod, Raymond; Software Testing; John Wiley & Sons, Hoboken, NJ; 2007; ISBN 978-0-471-79371-7; 261 pages.

Gack, Gary; Managing the Black Hole: The Executives Guide to Software Project Risk; Business Expert Publishing, Thomson, GA; 2010; ISBN10: 1-935602-01-9.

Gack, Gary; *Applying Six Sigma to Software Implementation Projects*; http://software.isixsigma.com/library/content/c040915b.asp.

Gilb, Tom and Graham, Dorothy; Software Inspections; Addison Wesley, Reading, MA; 1993; ISBN 10: 0201631814.

Hallowell, David L.; *Six Sigma Software Metrics, Part 1.*; http://software.isixsigma.com/library/content/03910a.asp.

International Organization for Standards; ISO 9000 / ISO 14000; http://www.iso.org/iso/en/iso9000-14000/index.html.

Jones, Capers and Bonsignour, Olivier; The Economics of Software Quality; Addison Wesley, Boston, MA; 2011; ISBN 978-0-13-258220-9; 587 pages.

Jones, Capers; Software Engineering Best Practices; McGraw Hill, New York; 2010; ISBN 978-0-07-162161-8;660 pages.

Jones, Capers; "Measuring Programming Quality and Productivity; IBM Systems Journal; Vol. 17, No. 1; 1978; pp. 39-63.

Jones, Capers; Programming Productivity - Issues for the Eighties; IEEE Computer Society Press, Los Alamitos, CA; First edition 1981; Second edition 1986; ISBN 0-8186—0681-9; IEEE Computer Society Catalog 681; 489 pages.

Jones, Capers; “A Ten-Year Retrospective of the ITT Programming Technology Center”; Software Productivity Research, Burlington, MA; 1988.

Jones, Capers; Applied Software Measurement; McGraw Hill, 3rd edition 2008; ISBN 978=0-07-150244-3; 662 pages.

Jones, Capers; Critical Problems in Software Measurement; Information Systems Management Group, 1993; ISBN 1-56909-000-9; 195 pages.

Jones, Capers; Software Productivity and Quality Today -- The Worldwide Perspective; Information Systems Management Group, 1993; ISBN -156909-001-7; 200 pages.

Jones, Capers; Assessment and Control of Software Risks; Prentice Hall, 1994; ISBN 0-13-741406-4; 711 pages.

Jones, Capers; New Directions in Software Management; Information Systems Management Group; ISBN 1-56909-009-2; 150 pages.

Jones, Capers; Patterns of Software System Failure and Success; International Thomson Computer Press, Boston, MA; December 1995; 250 pages; ISBN 1-850-32804-8; 292 pages.

Jones, Capers; Software Quality – Analysis and Guidelines for Success; International Thomson Computer Press, Boston, MA; ISBN 1-85032-876-6; 1997; 492 pages.

Jones, Capers; Estimating Software Costs; 2nd edition; McGraw Hill, New York; 2007; 700 pages..

Jones, Capers; “The Economics of Object-Oriented Software”; SPR Technical Report; Software Productivity Research, Burlington, MA; April 1997; 22 pages.

Jones, Capers; “Becoming Best in Class”; SPR Technical Report; Software Productivity Research, Burlington, MA; January 1998; 40 pages.

Jones, Capers; “Software Project Management Practices: Failure Versus Success”; Crosstalk, October 2004.

Jones, Capers; “Software Estimating Methods for Large Projects”; Crosstalk, April 2005.

Kan, Stephen H.; Metrics and Models in Software Quality Engineering, 2nd edition; Addison Wesley Longman, Boston, MA; ISBN 0-201-72915-6; 2003; 528 pages.

Land, Susan K; Smith, Douglas B; Walz, John Z; Practical Support for Lean Six Sigma Software Process Definition: Using IEEE Software Engineering Standards; WileyBlackwell; 2008; ISBN 10: 0470170808; 312 pages.

Mosley, Daniel J.; The Handbook of MIS Application Software Testing; Yourdon Press, Prentice Hall; Englewood Cliffs, NJ; 1993; ISBN 0-13-907007-9; 354 pages.

Myers, Glenford; The Art of Software Testing; John Wiley & Sons, New York; 1979; ISBN 0-471-04328-1; 177 pages.

Nandyal; Raghav; Making Sense of Software Quality Assurance; Tata McGraw Hill Publishing, New Delhi, India; 2007; ISBN 0-07-063378-9; 350 pages.

Radice, Ronald A.; High Qualitiy Low Cost Software Inspections; Paradoxicon Publishingl Andover, MA; ISBN 0-9645913-1-6; 2002; 479 pages.

Royce, Walker E.; Software Project Management: A Unified Framework; Addison Wesley Longman, Reading, MA; 1998; ISBN 0-201-30958-0.

Wiegers, Karl E.; Peer Reviews in Software – A Practical Guide; Addison Wesley Longman, Boston, MA; ISBN 0-201-73485-0; 2002; 232 pages.

Namcook Analytics LLC

SOFTWARE QUALITY IN 2013:

A SURVEY OF THE STATE OF THE ART

Capers Jones, VP and CTO

Blog: **http://Namcookanalytics.com**
Email: **Capers.Jones3@GMAILcom**

August 18, 2013

SOURCES OF QUALITY DATA

Data collected from 1984 through 2013

- **About 675 companies (150 clients in Fortune 500 set)**
- **About 35 government/military groups**
- **About 13,500 total projects**
- **New data = about 50-75 projects per month**
- **Data collected from 24 countries**
- **Observations during more than 15 lawsuits**

BASIC DEFINITIONS OF SOFTWARE QUALITY

- **Functional Software Quality**
  Software that combines low defect rates and high levels Of user satisfaction. The software should also meet all user requirements and adhere to international standards.

- **Structural Software Quality**
  Software that exhibits a robust architecture and can operate In a multi-tier environment without failures or degraded performance. Software has low cyclomatic complexity levels.

- **Aesthetic Software Quality**
  Software with elegant and easy to use commands and Interfaces, attractive screens, and well formatted outputs.

ECONOMIC DEFINITIONS OF SOFTWARE QUALITY

- **"Technical debt"**
  The assertion (by Ward Cunningham in 1992) that quick and careless development with poor quality leads to many years of expensive maintenance and enhancements.

- **Cost of Quality (COQ)**
  The overall costs of prevention, appraisal, internal failures, and external failures. For software these mean defect prevention, pre-test defect removal, testing, and post-release defect repairs. (Consequential damages are usually not counted.)

- **Total Cost of Ownership (TCO)**
  The sum of development + enhancement + maintenance + support from day 1 until application is retired. (Recalculation at 5 year intervals is recommended.)

SOFTWARE QUALITY HAZARDS IN TEN INDUSTRIES

INDUSTRY	HAZARD
Airlines	Safety hazards

Air traffic control problems

Flight schedule confusion

Navigation equipment failures

Maintenance schedules thrown off

Delay in opening Denver airport

Passengers booked into non-existent seats

Passengers misidentified as terror suspects

SOFTWARE QUALITY HAZARDS IN TEN INDUSTRIES

INDUSTRY	HAZARD
Defense	Security hazards

Base security compromised

Computer security compromised

Strategic weapons malfunction

Command, communication network problems

Aircraft maintenance records thrown off

Logistics and supply systems thrown off

Satellites malfunction

SOFTWARE QUALITY HAZARDS IN TEN INDUSTRIES

INDUSTRY HAZARD

Finance Financial transaction

hazards

Interest calculations in error

Account balances thrown off

Credit card charges in error

Funds transfer thrown off

Mortgage/loan interest payments in error

Hacking and identity theft due to software security flaws

Denial of service attacks due to software security flaws

SOFTWARE QUALITY HAZARDS IN TEN INDUSTRIES

INDUSTRY HAZARD

Health Care Safety hazards

Patient monitoring devices malfunction

Operating room schedules thrown off

Medical instruments malfunction

Prescription refill problems

Hazardous drug interactions

Billing problems

Medical records stolen or released by accident

SOFTWARE QUALITY HAZARDS IN TEN INDUSTRIES

INDUSTRY	HAZARD
Insurance	Liability, benefit hazards

- Policy due dates in error
- Policies cancelled in error
- Benefits and interest calculation errors
- Annuities miscalculated
- Errors in actuarial studies
- Payment records in error

SOFTWARE QUALITY HAZARDS IN TEN INDUSTRIES

INDUSTRY	HAZARD
State, Local Governments	Local economic hazards

- School taxes miscalculated
- Jury records thrown off
- Real-estate transactions misfiled
- Divorce, marriage records misfiled
- Alimony, child support payment records lost
- Death records filed for wrong people
- Traffic light synchronization thrown off
- Errors in property tax assessments

SOFTWARE QUALITY HAZARDS IN TEN INDUSTRIES

INDUSTRY	HAZARD
Manufacturing	Operational hazards

Subcontract parts fail to arrive

Purchases of more or less than economic order quantities

Just-in-time arrivals thrown off

Assembly lines shut down

Aging errors for accounts receivable and cash flow

Aging errors for accounts payable and cash flow

Pension payments miscalculated

SOFTWARE QUALITY HAZARDS IN TEN INDUSTRIES

INDUSTRY	HAZARD
National Government	Citizen record hazards

Tax records in error

Annuities and entitlements miscalculated

Social Security payments miscalculated or cancelled

Disbursements miscalculated

Retirement benefits miscalculated

Personal data stolen or released by accident

SOFTWARE QUALITY HAZARDS IN TEN INDUSTRIES

INDUSTRY	HAZARD
Public Utilities	**Safety hazards**

- **Electric meters malfunction**
- **Gas meters malfunction**
- **Distribution of electric power thrown off**
- **Billing records in error**
- **Nuclear power plants malfunction**

SOFTWARE QUALITY HAZARDS IN TEN INDUSTRIES

INDUSTRY	HAZARD
Telecommunications	**Service disruption**

hazards

- **Intercontinental switching disrupted**
- **Domestic call switching disrupted**
- **Billing records in error**

SOFTWARE QUALITY HAZARDS ALL INDUSTRIES

1. **Software is blamed for more major business problems than any other man-made product.**
2. **Poor software quality has become one of the most expensive topics in human history: > $150 billion per year in U.S.; > $500 billion per year world wide.**
3. **Projects cancelled due to poor quality >15% more costly than successful projects of the same size and type.**
4. **Software executives, managers, and technical personnel are regarded by many CEO's as a painful necessity rather than top professionals.**
5. **Improving software quality is a key topic for all industries.**

FUNDAMENTAL SOFTWARE QUALITY METRICS

- **Defect Potentials**
  - **Sum of requirements errors, design errors, code errors, document errors, bad fix errors, test plan errors, and test case errors**
- **Defect Discovery Efficiency (DDE)**
  - **Percent of defects discovered before release**
- **Defect Removal Efficiency (DRE)**
  - **Percent of defects removed before release**
- **Defect Severity Levels (Valid unique defects)**

  Severity 1 = Total stoppage
  Severity 2 = Major error
  Severity 3 = Minor error
  Severity 4 = Cosmetic error

FUNDAMENTAL SOFTWARE QUALITY METRICS (cont.)

- **Standard Cost of Quality**
  - **Prevention**
  - **Appraisal**
  - **Internal failures**
  - **External failures**

- **Revised Software Cost of Quality**
  - **Defect Prevention**
  - **Pre-Test Defect Removal (inspections, static analysis)**
  - **Testing Defect Removal**
  - **Post-Release Defect Removal**

- **Error-Prone Module Effort**
  - **Identification**
  - **Removal or redevelopment**
  - **repairs and rework**

QUALITY MEASUREMENT PROBLEMS

- **Cost per defect penalizes quality!**
- **(Buggiest software has lowest cost per defect!)**
- **Technical debt ignores canceled projects and litigation.**
- **Technical debt covers less than 30% of true costs.**
- **Lines of code penalize high-level languages!**
- **Lines of code ignore non-coding defects!**
- **Most companies don’t measure all defects!**
- **Missing bugs are > 70% of total bugs!**

COST PER DEFECT PENALIZES QUALITY

	Case A High quality	Case B Low quality
Defects found	50	500
Test case creation	$10,000	$10,000
Test case execution	$10,000	$10,000
Defect repairs	$10,000	$70,000
TOTAL	$30,000	$90,000
Cost per Defect	$600	$180
$ Cost savings	$60,000	$0.00

A BASIC LAW OF MANUFACTURING ECONOMICS

"If a manufacturing cycle has a high proportion of fixed costs
and there is a decline in the number of units produced
the cost per unit will go up."

1. As quality improves the number of defects goes down.
2. Test preparation and test execution act like fixed costs.
3. Therefore the "cost per defect" must go up.
4. Late defects must cost more than early defects.
5. Defects in high quality software cost more than in bad quality software.

LINES OF CODE HARM HIGH-LEVEL LANGUGES

	Case A	Case B
	JAVA	C
KLOC	50	125
Function points	1,000	1,000
Code defects found	500	1,250
Defects per KLOC	10.00	10.00
Defects per FP	0.5	1.25
Defect repairs	$70,000	$175,000
$ per KLOC	$1,400	$1,400
$ per Defect	$140	$140
$ per Function Point	$70	$175
$ cost savings	$105,000	$0.00

A BASIC LAW OF MANUFACTURING ECONOMICS

"If a manufacturing cycle has a high proportion of fixed costs and there is a decline in the number of units produced the cost per unit will go up."

1) As language levels go up the number of lines of code produced comes down.

2) The costs of requirements, architecture, design, and documentation act as fixed costs.

3) Therefore the "cost per line of code" must go up.

4) Cost per line of code penalizes languages in direct proportion to their level.

U.S. AVERAGES FOR SOFTWARE QUALITY

(Data expressed in terms of defects per function point)

Defect Origins	Defect Potential	Removal Efficiency	Delivered Defects
Requirements	1.00	77%	0.23
Design	1.25	85%	0.19
Coding	1.75	95%	0.09
Documents	0.60	80%	0.12
Bad Fixes	0.40	70%	0.12
TOTAL	5.00	85%	0.75

(Function points show all defect sources - not just coding defects)
(Code defects = 35% of total defects)

BEST IN CLASS SOFTWARE QUALITY

(Data expressed in terms of defects per function point)

Defect Origins	Defect Potential	Removal Efficiency	Delivered Defects
Requirements	0.40	85%	0.08
Design	0.60	97%	0.02
Coding	1.00	99%	0.01
Documents	0.40	98%	0.01
Bad Fixes	0.10	95%	0.01
TOTAL	2.50	96%	0.13

OBSERVATIONS

(Most often found in systems software > SEI CMM Level 3 or in TSP projects)

POOR SOFTWARE QUALITY - MALPRACTICE

(Data expressed in terms of defects per function point)

Defect Origins	Defect Potential	Removal Efficiency	Delivered Defects
Requirements	1.50	50%	0.75
Design	2.20	50%	1.10
Coding	2.50	80%	0.50
Documents	1.00	70%	0.30
Bad Fixes	0.80	50%	0.40
TOTAL	8.00	62%	3.05

OBSERVATIONS

(Most often found in large water fall projects > 10,000 Function Points).

GOOD QUALITY RESULTS > 90% SUCCESS RATE

- Formal Inspections (Requirements, Design, and Code)
- Text static analysis
- Code static analysis (for about 25 languages out of 2,500 in all)
- Joint Application Design (JAD)
- Requirements modeling
- Functional quality metrics using function points
- Structural quality metrics such as cyclomatic complexity
- Defect Detection Efficiency (DDE) measurements
- Defect Removal Efficiency (DRE) measurements
- Automated defect tracking tools
- Active quality Assurance (> 3% SQA staff)
- Mathematical test case design based on design of experiments
- Quality estimation tools
- Testing specialists (certified)
- Root-Cause Analysis

MIXED QUALITY RESULTS: < 50% SUCCESS RATE

- CMMI level 3 or higher (some overlap among CMMI levels:
  Best CMMI 1 groups better than worst CMMI 3 groups)
- ISO and IEEE quality standards (Prevent low quality;
  Little benefit for high-quality teams)
- Six-Sigma methods (unless tailored for software projects)
- Quality function deployment (QFD)
- Independent Verification & Validation (IV & V)
- Quality circles in the United States (more success in Japan)
- Clean-room methods for rapidly changing requirements
- Kaizan (moving from Japan to U.S. and elsewhere)
- Cost of quality without software modifications
- Pair programming

POOR QUALITY RESULTS: < 25% SUCCESS RATE

- Testing as only form of defect removal
- Informal Testing and uncertified test personnel
- Testing only by developers; no test specialists
- Passive Quality Assurance (< 3% QA staff)
- Token Quality Assurance (< 1% QA staff)
- LOC Metrics for quality (omits non-code defects)
- Cost per defect metric (penalizes quality)
- Failure to estimate quality or risks early
- Quality measurement “leakage” such as unit test bugs

A PRACTICAL DEFINITION OF SOFTWARE QUALITY (PREDICTABLE AND MEASURABLE)

- **Low Defect Potentials (< 2.5 per Function Point)**
- **High Defect Removal Efficiency (> 95%)**
- **Unambiguous, Stable Requirements (< 2.5% change)**
- **Explicit Requirements Achieved (> 97.5% achieved)**
- **High User Satisfaction Ratings (> 90% "excellent")**

  - Installation
  - Ease of learning
  - Ease of use
  - Functionality
  - Compatibility
  - Error handling
  - User information (screens, manuals, tutorials)
  - Customer support
  - Defect repairs

SOFTWARE QUALITY OBSERVATIONS

Quality Measurements Have Found:

- **Individual programmers -- Less than 50% efficient in finding bugs in their own software**

- **Normal test steps -- often less than 75% efficient (1 of 4 bugs remain)**

- **Design Reviews and Code Inspections -- often more than 65% efficient; have topped 90%**

- **Static analysis –often more than 65% efficient; has topped 95%**

- **Inspections, static analysis, and testing combined lower costs and schedules by > 20%; lower total cost of ownership (TCO) by > 45%.**

SOFTWARE DEFECT ORIGINS

1. Requirements	Hardest to prevent and repair
2. Requirements creep	Very troublesome source of bugs
3. Architecture	Key to structural quality
4. Design	Most severe and pervasive
5. Source code	Most numerous; easiest to fix
6. Security flaws	Hard to find and hard to fix
7. Documentation	Can be serious if ignored
8. Bad fixes	Very difficult to find
9. Bad test cases	Numerous but seldom measured
10. Data errors	Very common but not measured
11. Web content	Very common but not measured
12. Structure	Hard to find by testing; inspections and static analysis can identify multi-tier platform defects

TOTAL SOFTWARE DEFECTS IN RANK ORDER

Defect Origins	Defects per Function Point
1. Data defects	2.50 *
2. Code defects	1.75
3. Test case defects	1.65 *
4. Web site defects	1.40 *
5. Design defects	1.25 **
6. Requirement Defects	1.00 **
7. Structural defects	0.70 **
8. Document defects	0.60 **
9. Bad-fix defects	0.40 **
10. Requirement creep defects	0.30 **
11. Security defects	0.25 **
12. Architecture Defects	0.20 *
TOTAL DEFECTS	12.00

* NOTE 1: Usually not measured due to lack of size metrics

** NOTE 2: Often omitted from defect measurements

ORIGINS OF HIGH-SEVERITY SOFTWARE DEFECTS

Defect Origins	Percent of Severity 1 and 2 Defects
1. Design defects	17.00%
2. Code defects	15.00%
3. Structural defects	13.00%
4. Data defects	11.00%
5. Requirements creep defects	10.00%
6. Requirements defects	9.00%
7. Web site defects	8.00%
8. Security defects	7.00%
9. Bad fix defects	4.00%
10. Test case defects	2.00%
11. Document defects	2.00%
12. Architecture Defects	2.00%
TOTAL DEFECTS	100.00%

Severity 1 = total stoppage; Severity 2 = major defects

ORIGINS OF LOW-SEVERITY SOFTWARE DEFECTS

Defect Origins	Percent of Severity 3 and 4 Defects
1. Code defects	35.00%
2. Data defects	20.00%
3. Web site defects	10.00%
4. Design defects	7.00%
5. Structural defects	6.00%
6. Requirements defects	4.00%
7. Requirements creep defects	4.00%
8. Security defects	4.00%
9. Bad fix defects	4.00%
10. Test case defects	3.00%
11. Document defects	2.00%
12. Architecture Defects	1.00%
TOTAL DEFECTS	100.00%

Severity 3 = minor defects; Severity 4 = cosmetic defects

ORIGINS OF DUPLICATE DEFECTS

	Defect Origins	Percent of Duplicate Defects (Many reports of the same bugs)
1.	Code defects	30.00%
2.	Structural defects	20.00%
3.	Data defects	20.00%
4.	Web site defects	10.00%
5.	Security defects	4.00%
6.	Requirements defects	3.00%
7.	Design defects	3.00%
8.	Bad fix defects	3.00%
9.	Requirements creep defects	2.00%
10.	Test case defects	2.00%
11.	Document defects	2.00%
12.	Architecture Defects	1.00%
TOTAL DEFECTS		100.00%

Duplicate = Multiple reports for the same bug (> 10,000 can occur)

ORIGINS OF INVALID DEFECTS

	Defect Origins	Percent of Invalid Defects (Defects not caused by software itself)
1.	Data defects	25.00%
2.	Structural defects	20.00%
3.	Web site defects	13.00%
4.	User errors	12.00%
5.	Document defects	10.00%
6.	External software	10.00%
7.	Requirements creep defects	3.00%
8.	Requirements defects	1.00%
9.	Code defects	1.00%
10.	Test case defects	1.00%
11.	Security defects	1.00%
12.	Design defects	1.00%
13.	Bad fix defects	1.00%
14.	Architecture Defects	1.00%
TOTAL DEFECTS		100.00%

Invalid = Defects caused by platforms or external software applications

WORK HOURS AND COSTS FOR DEFECT REPAIRS

	Defect Origins	Work Hours	Costs ($75 per hour)
1.	Security defects	10.00	$750.00
2.	Design defects	8.50	$637.50
3.	Requirements creep defects	8.00	$600.00
4.	Requirements defects	7.50	$562.50
5.	Structural defects	7.25	$543.75
6.	Architecture defects	7.00	$525.00
7.	Data defects	6.50	$487.50
8.	Bad fix defects	6.00	$450.00
9.	Web site defects	5.50	$412.50
10.	Invalid defects	4.75	$356.25
11.	Test case defects	4.00	$300.00
12.	Code defects	3.00	$225.00
13.	Document defects	1.75	$131.50
14.	Duplicate defects	1.00	$75.00
	AVERAGES	5.77	$432.69

Maximum can be > 10 times greater

DEFECT DAMAGES AND RECOVERY COSTS

	Defect Origins	
1.	Security defects	$200,000,000
2.	Design defects	$175,000,000
3.	Requirements defects	$150,000,000
4.	Data defects	$125,000,000
5.	Code defects	$100,000,000
6.	Structural defects	$95,000,000
7.	Requirements creep defects	$90,000,000
8.	Web site defects	$80,000,000
9.	Architecture defects	$80,000,000
10.	Bad fix defects	$60,000,000
11.	Test case defects	$50,000,000
12.	Document Defects	$25,000,000
	AVERAGES	$102,500,000

Defect recovery costs for major applications in large companies and government agencies

WORK HOURS AND COSTS BY SEVERITY

Defect Severity	Work Hours	Costs ($75 per hour)
Severity 1 (total stoppage)	6.00	$450.00
Severity 2 (major errors)	9.00	$675.00
Severity 3 (minor errors)	3.00	$225.00
Severity 4 (cosmetic errors)	1.00	$75.00
Abeyant defects (special case)	40.00	$3,000.00
Invalid defects	4.75	$355.25
Duplicate defects	1.00	$75.00

Maximum can be > 10 times greater

DEFECT REPAIRS BY APPLICATION SIZE

Function. Points	Sev 1 Hours	Sev 2 Hours	Sev 3 Hours	Sev 4 Hours	AVERAGE HOURS
10	2.00	3.00	1.50	0.50	1.75
100	4.00	6.00	2.00	0.50	3.13
1000	6.00	9.00	3.00	1.00	4.75
10000	8.00	12.00	4.00	1.50	6.38
100000	18.00	24.00	6.00	2.00	12.50

Function Points	Sev 1 $	Sev 2 $	Sev 3 $	Sev 4 $	AVERAGE COSTS
10	$150	$220	$112	$38	$132
100	$300	$450	$150	$38	$234
1000	$450	$675	$225	$75	$356
10000	$600	$900	$300	$113	$478
100000	$1350	$1800	$450	$150	$938

DEFECT REPORTS IN FIRST YEAR OF USAGE

Function Points	10	100	1000	10,000	100,000
Users					
1	55%	27%	12%	3%	1%
10	65%	35%	17%	7%%	3%
100	75%	42%	20%	10%	7%
1000	85%	50%	*27%*	12%	10%
10,000	95%	75%	35%	20%	12%
100,000	99%	87%	45%	35%	20%
1,000,000	100%	96%	77%	45%	32%
10,000,000	100%	100%	90%	65%	45%

ELAPSED TIME IN DAYS FOR DEFECT RESOLUTION

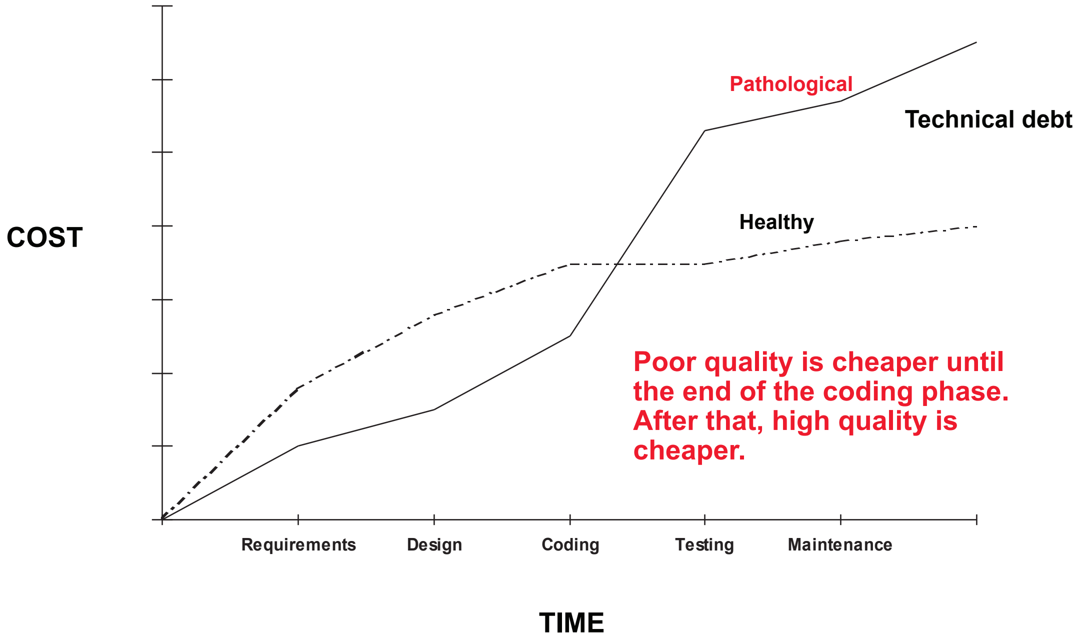
Removal method	Stat. Analy.	Unit Test	Inspect.	Funct. Test	Sys. Test	Maint.
Preparation	1	2	5	6	8	7
Execution	1	1	2	4	6	3
Repair	1	1	1	1	2	2
Validate	1	1	1	2	4	5
Integrate	1	1	1	2	4	6
Distribute	1	1	1	2	3	7
TOTAL DAYS	6	7	11	17	27	30

Defect repairs take < 12% of elapsed time

SOFTWARE DEFECT SEVERITY CATEGORIES

Severity 1:	TOTAL FAILURE S	1% at release
Severity 2:	MAJOR PROBLEMS	20% at release
Severity 3:	MINOR PROBLEMS	35% at release
Severity 4:	COSMETIC ERRORS	44% at release
STRUCTURAL	MULTI-TIER DEFECTS	15% of reports
INVALID	USER OR SYSTEM ERRORS	15% of reports
DUPLICATE	MULTIPLE REPORTS	30% of reports
ABEYANT	CAN'T RECREATE ERROR	5% of reports

HOW QUALITY AFFECTS SOFTWARE COSTS



Poor quality is cheaper until the end of the coding phase. After that, high quality is cheaper.

U. S. SOFTWARE QUALITY AVERAGES CIRCA 2013

(Defects per Function Point)

	System Software	Commercial Software	Information Software	Military Software	Outsource Software
Defect Potentials	6.0	5.0	4.5	7.0	5.2
Defect Removal Efficiency	94%	90%	73%	96%	92%
Delivered Defects	0.36	0.50	1.22	0.28	0.42
First Year Discovery Rate	65%	70%	30%	75%	60%
First Year Reported Defects	0.23	0.35	0.36	0.21	0.25

U. S. SOFTWARE QUALITY AVERAGES CIRCA 2013

(Defects per Function Point)

	Web Software	Embedded Software	SEI-CMM 3 Software	SEI-CMM 1 Software	Overall Average
Defect Potentials	4.0	5.5	5.0	5.75	5.1
Defect Removal Efficiency	72%	95%	95%	83%	86.7%
Delivered Defects	1.12	0.3	0.25	0.90	0.68
First Year Discovery Rate	95%	90%	60%	35%	64.4%
First Year Reported Defects	1.06	0.25	0.15	0.34	0.42

SOFTWARE SIZE VS DEFECT REMOVAL EFFICIENCY

(Data Expressed in terms of Defects per Function Point)

Size	Defect Potential	Defect Removal Efficiency	Delivered Defects	1st Year Discovery Rate	1st Year Reported Defects
1	1.85	95.00%	0.09	90.00%	0.08
10	2.45	92.00%	0.20	80.00%	0.16
100	3.68	90.00%	0.37	70.00%	0.26
1000	5.00	85.00%	0.75	50.00%	0.38
10000	7.60	78.00%	1.67	40.00%	0.67
100000	9.55	75.00%	2.39	30.00%	0.72
AVERAGE	*5.02*	*85.83%*	*0.91*	*60.00%*	*0.38*

SOFTWARE DEFECT POTENTIALS AND DEFECT REMOVAL EFFICIENCY FOR EACH LEVEL OF SEI CMM

(Data Expressed in Terms of Defects per Function Point
For projects nominally 1000 function points in size)

SEI CMM Levels	Defect Potentials	Removal Efficiency	Delivered Defects
SEI CMMI 1	5.25	80%	1.05
SEI CMMI 2	5.00	85%	0.75
SEI CMMI 3	4.75	90%	0.48
SEI CMMI 4	4.50	93%	0.32
SEI CMMI 5	4.25	96%	0.17

SOFTWARE DEFECT POTENTIALS AND DEFECT REMOVAL EFFICIENCY FOR EACH LEVEL OF SEI CMM

(Data Expressed in Terms of Defects per Function Point
For projects 10,000 function points in size)

SEI CMM Levels	Defect Potentials	Removal Efficiency	Delivered Defects
SEI CMMI 1	6.50	75%	1.63
SEI CMMI 2	6.25	82%	1.13
SEI CMMI 3	5.50	87%	0.71
SEI CMMI 4	5.25	90%	0.53
SEI CMMI 5	4.75	94%	0.29

DEFECTS AND SOFTWARE METHODOLGOIES

(Data Expressed in Terms of Defects per Function Point
For projects nominally 1000 function points in size)

Software methods	Defect Potential	Removal Efficiency	Delivered Defects
Waterfall		5.50	80%
1.10			
Iterative	4.75	87%	0.62
Object-Oriented	4.50	88%	0.54
Agile with scrum	4.00	90%	0.40
Rational Unified Process (RUP)	4.25	94%	0.26
PSP and TSP	3.50	96%	0.14
Model-based	3.00	98%	0.06
85% Certified reuse	1.75	99%	0.02

DEFECTS AND SOFTWARE METHODOLGOIES

(Data Expressed in Terms of Defects per Function Point
For projects nominally 10,000 function points in size)

Software methods	Defect Potential	Removal Efficiency	Delivered Defects
Waterfall	7.00	75%	1.75
Iterative	6.25	82%	1.13
Object-Oriented	5.75	85%	0.86
Agile with scrum	5.50	87%	0.72
Rational Unified Process (RUP)	5.50	90%	0.55
PSP and TSP	5.00	94%	0.30
Model-based	4.00	96%	0.15
85% Certified reuse	2.25	96%	0.09

GLOBAL SOFTWARE QUALITY SAMPLES

Data Expressed in Terms of Defects per Function Point
Top countries for software quality out of 66

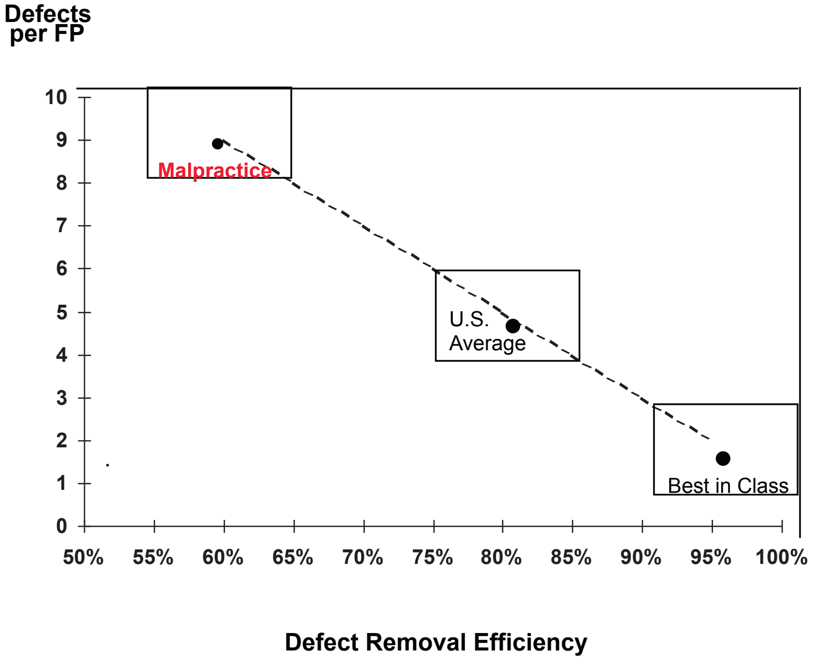
Country	Defect Potential	Removal Efficiency	Delivered Defects
Japan	4.50	93.50%	0.29
India	4.90	93.00%	0.34
Denmark	4.80	92.00%	0.38
Canada	4.75	91.75%	0.39
South Korea	4.90	92.00%	0.39
Switzerland	5.00	92.00%	0.40
United Kingdom	5.10	91.50%	0.40
Israel	5.10	92.00%	0.41

GLOBAL SOFTWARE QUALITY SAMPLES

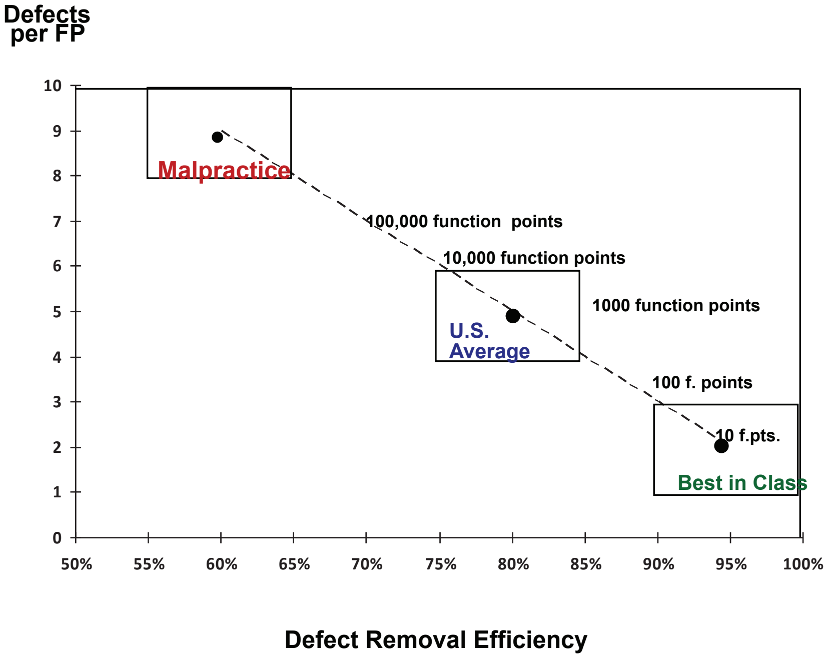
Data Expressed in Terms of Defects per Function Point
Selected Countries out of 66 compared

Country	Defect Potential	Removal Efficiency	Delivered Defects
United States	4.82	90.15%	0.47
France	4.85	90.00%	0.49
Germany	4.95	88.00%	0.59
Italy	4.95	87.50%	0.62
Spain	4.90	86.50%	0.66
Russia	5.15	86.50%	0.70
China	5.20	86.50%	0.70
Ukraine	4.95	85.00%	0.74

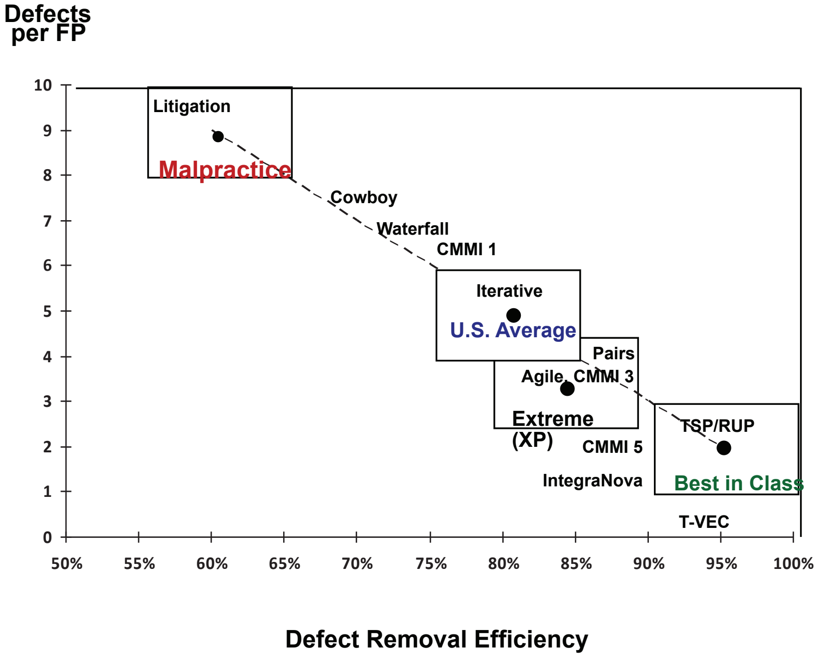
MAJOR SOFTWARE QUALITY ZONES



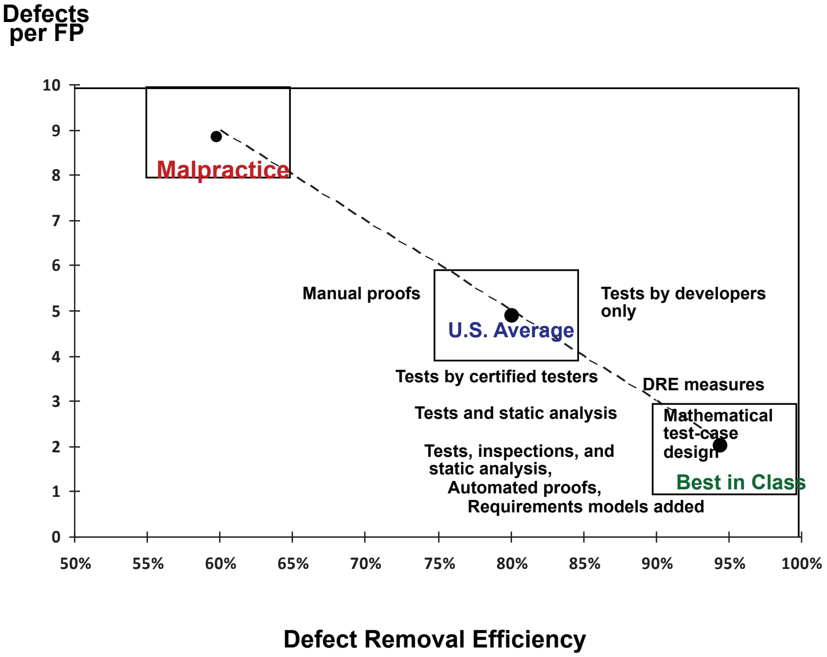
SOFTWARE QUALITY IMPROVEMENT



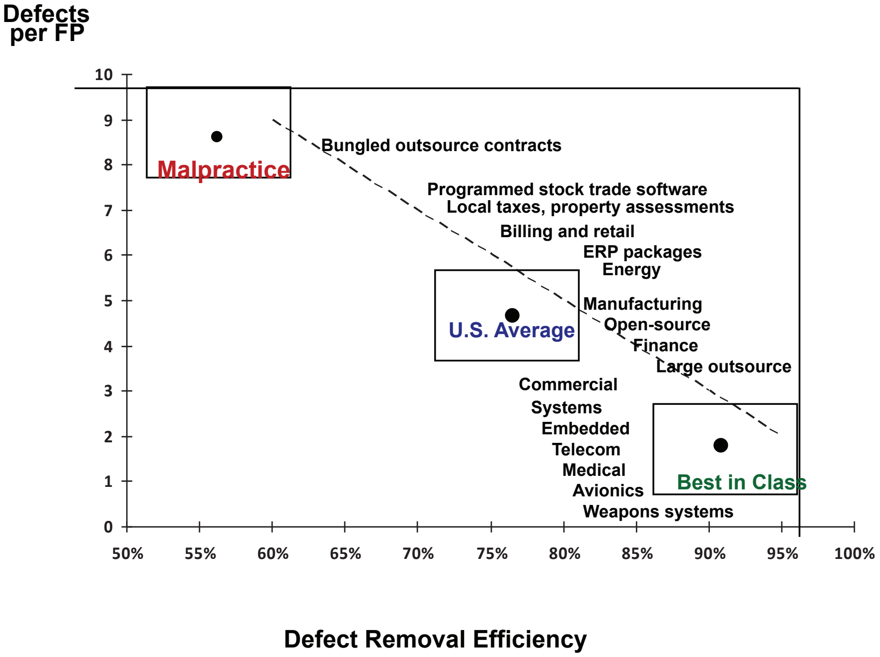
SOFTWARE QUALITY IMPROVEMENT (cont.)



SOFTWARE QUALITY IMPROVEMENT



SOFTWARE QUALITY IMPROVEMENT



DEFECT REMOVAL EFFICIENCY EXAMPLE 1

Inspections + static analysis + testing

DEVELOPMENT DEFECTS REMOVED

Inspections	**390**
Static analysis	**350**
Testing	**250**
Subtotal	**990**

USER-REPORTED DEFECTS IN FIRST 90 DAYS

Valid unique defects	**10**

TOTAL DEFECT VOLUME

Defect totals	**1,000**

DEFECT REMOVAL EFFICIENCY

Dev. (990) / Total (1,000) = 99%

DEFECT REMOVAL EFFICIENCY EXAMPLE 2

No static analysis. Inspections + testing

DEVELOPMENT DEFECTS REMOVED

Inspections	**560**
Static analysis	**0**
Testing	**400**
Subtotal	**960**

USER-REPORTED DEFECTS IN FIRST 90 DAYS

Valid unique defects	**40**

TOTAL DEFECT VOLUME

Defect totals	**1,000**

DEFECT REMOVAL EFFICIENCY

Dev. (960) / Total (1,000) = 96%

DEFECT REMOVAL EFFICIENCY EXAMPLE 3

No inspections. Static analysis + testing

DEVELOPMENT DEFECTS REMOVED	
Inspections	**0**
Static analysis	**525**
Testing	**400**
Subtotal	**925**
USER-REPORTED DEFECTS IN FIRST 90 DAYS	
Valid unique defects	**75**
TOTAL DEFECT VOLUME	
Defect totals	**1,000**
DEFECT REMOVAL EFFICIENCY	
Dev. (925) / Total (1,000) =	**92.5%**

DEFECT REMOVAL EFFICIENCY EXAMPLE 4

No inspections; no static analysis. Testing only.

DEVELOPMENT DEFECTS REMOVED	
Inspections	**0**
Static analysis	**0**
Testing	**850**
Subtotal	**850**
USER-REPORTED DEFECTS IN FIRST 90 DAYS	
Valid unique defects	**150**
TOTAL DEFECT VOLUME	
Defect totals	**1,000**
DEFECT REMOVAL EFFICIENCY	
Dev. (850) / Total (1,000) =	**85%**

INDUSTRY-WIDE DEFECT CAUSES

Ranked in order of effort required to fix the defects:

1. **Requirements problems (omissions; changes, errors)**
2. **Design problems (omissions; changes; errors)**
3. **Security flaws and vulnerabilities**
4. **Interface problems between modules**
5. **Logic, branching, and structural problems**
6. **Memory allocation problems**
7. **Testing omissions and poor coverage**
8. **Test case errors**
9. **Stress/performance problems**
10. **Bad fixes/Regressions**

OPTIMIZING QUALITY AND PRODUCTIVITY

Projects that achieve 95% cumulative Defect Removal Efficiency will find:

1) **Minimum schedules**

2) **Maximum productivity**

3) **High levels of user and team satisfaction**

4) **Low levels of delivered defects**

5) **Low levels of maintenance costs**

6) **Low risk of litigation**

INDUSTRY DATA ON DEFECT ORIGINS

Because defect removal is such a major cost element, studying defect origins is a valuable undertaking.

IBM Corporation (MVS)	
45%	Design errors
25%	Coding errors
20%	Bad fixes
5%	Documentation errors
5%	Administrative errors
100%	

SPR Corporation (client studies)	
20%	Requirements errors
30%	Design errors
35%	Coding errors
10%	Bad fixes
5%	Documentation errors
100%	

TRW Corporation	
60%	Design errors
40%	Coding errors
100%	

MITRE Corporation	
64%	Design errors
36%	Coding errors
100%	

Nippon Electric Corp.	
60%	Design errors
40%	Coding errors
100%	

SOFTWARE QUALITY AND PRODUCTIVITY

- The most effective way of improving software productivity and shortening project schedules is to reduce defect levels.

- Defect reduction can occur through:

1. Defect prevention technologies
   - Structured design and JAD
   - Structured code
   - Use of inspections, static analysis
   - Reuse of certified components

2. Defect removal technologies
   - Design inspections
   - Code inspections, static analysis
   - Formal Testing using mathematical test case design

DEFECT PREVENTION METHODS

DEFECT PREVENTION

- Joint Application Design (JAD)
- Quality function deployment (QFD)
- Software reuse (high-quality components)
- Root cause analysis
- Six-Sigma quality programs for software
- Model-based requirements, design
- Climbing > Level 3 on the SEI CMMI
- Static analysis, inspections

DEFECT PREVENTION - Continued

DEFECT PREVENTION

- Life-cycle quality measurements
- Kaizen, Poka Yoke, Kanban, Quality Circles (from Japan)
- Prototypes of final application (disposable are best)
- Pair programming
- Formal design inspections
- Formal code inspections
- Embedding users with development team (Agile methods)
- SCRUM (issue-oriented team meetings)

DEFECT REMOVAL METHODS

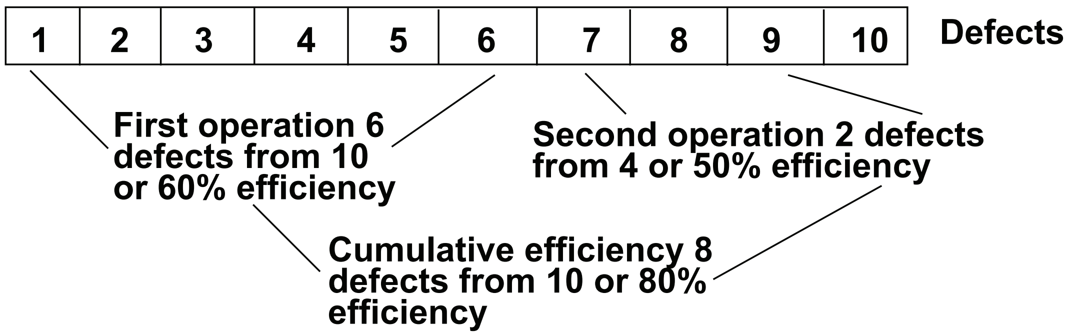
DEFECT REMOVAL

- **Requirements inspections**
- **Design inspections**
- **Test plan inspections**
- **Test case inspections**
- **Text static analysis; FOG and Flesch analysis**
- **Code static analysis (C, Java, COBOL, SQL etc.)**
- **Code inspections**
- **Automated testing (unit, performance)**
- **All forms of manual testing (more than 40 kinds of test)**

DEFECT REMOVAL EFFICIENCY

- **Defect removal efficiency is a key quality measure**
- **Removal efficiency =** $\frac{\textbf{Defects found}}{\textbf{Defects present}}$
- **“Defects present” is the critical parameter**

DEFECT REMOVAL EFFICIENCY - continued



First operation 6 defects from 10 or 60% efficiency

Second operation 2 defects from 4 or 50% efficiency

Cumulative efficiency 8 defects from 10 or 80% efficiency

Defect removal efficiency = Percentage of defects removed by a single level of review, inspection or test

Cumulative defect removal efficiency = Percentage of defects removed by a series of reviews, inspections or tests

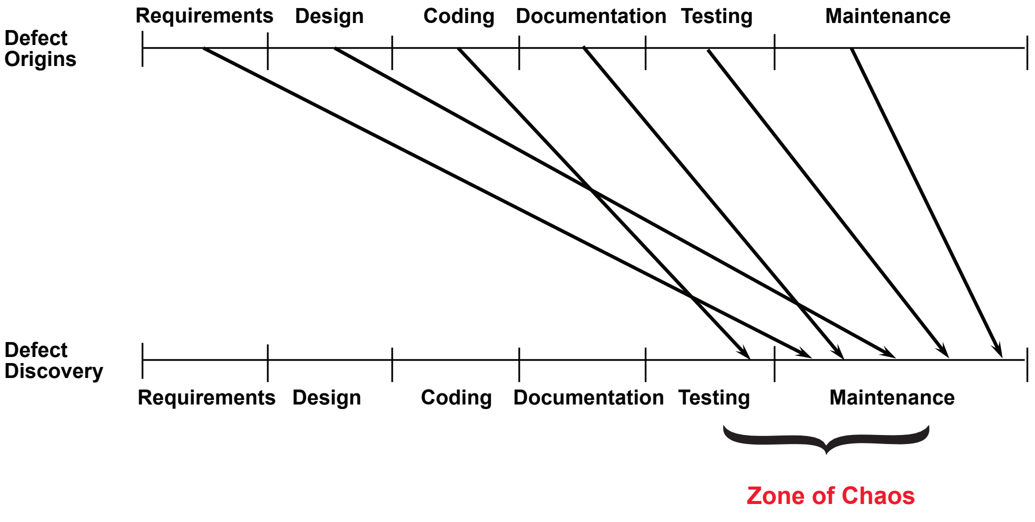
DEFECT REMOVAL EFFICIENCY EXAMPLE

DEVELOPMENT DEFECTS REMOVED	
Inspections	350
Static analysis	350
Testing	250
Subtotal	950
USER-REPORTED DEFECTS IN FIRST 90 DAYS	
Valid unique defects	50
TOTAL DEFECT VOLUME	
Defect totals	1000
REMOVAL EFFICIENCY	
Dev. (900) / Total (1000) =	95%

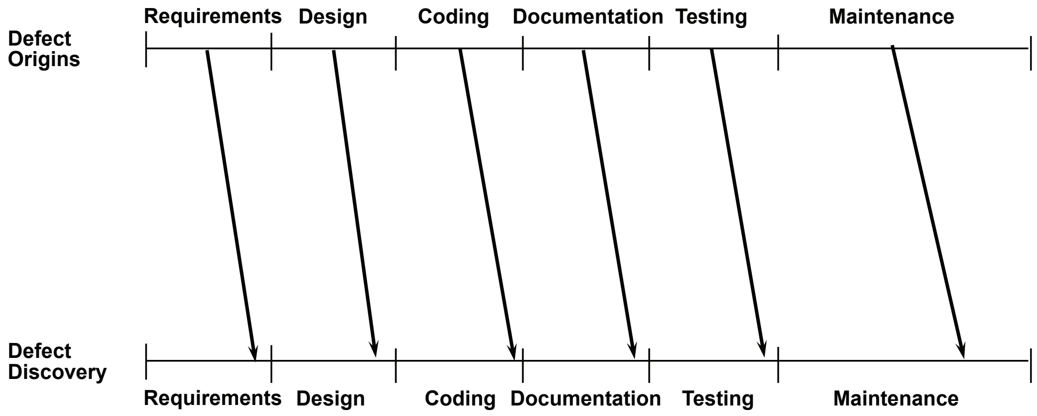
RANGES OF DEFECT REMOVAL EFFICIENCY

	Lowest	Median	Highest
1 Requirements review (informal)	20%	30%	50%
2 Top-level design reviews (informal)	30%	40%	60%
3 Detailed functional design inspection	30%	65%	85%
4 Detailed logic design inspection	35%	65%	75%
5 Code inspection or static analysis	35%	60%	90%
6 Unit tests	10%	25%	50%
7 New Function tests	20%	35%	65%
8 Integration tests	25%	45%	60%
9 System test	25%	50%	65%
10 External Beta tests	15%	40%	75%
CUMULATIVE EFFICIENCY	75%	98%	99.99%

NORMAL DEFECT ORIGIN/DISCOVERY GAPS



DEFECT ORIGINS/DISCOVERY WITH INSPECTIONS



Defect Origins

Requirements Design Coding Documentation Testing Maintenance

Defect Discovery

Requirements Design Coding Documentation Testing Maintenance

SOFTWARE DEFECT REMOVAL RANGES

WORST CASE RANGE

TECHNOLOGY COMBINATIONS	DEFECT REMOVAL EFFICIENCY		
	Lowest	Median	Highest
1. No Design Inspections No Code Inspections or static analysis No Quality Assurance No Formal Testing	30%	40%	50%

SOFTWARE DEFECT REMOVAL RANGES (cont.)

SINGLE TECHNOLOGY CHANGES

	TECHNOLOGY COMBINATIONS	DEFECT REMOVAL EFFICIENCY		
		Lowest	Median	Highest
2.	No design inspections No code inspections or static analysis FORMAL QUALITY ASSURANCE No formal testing	32%	45%	55%
3.	No design inspections No code inspections or static analysis No quality assurance FORMAL TESTING	37%	53%	60%
4.	No design inspections CODE INSPECTIONS/STATIC ANALYSIS No quality assurance No formal testing	43%	57%	65%
5.	FORMAL DESIGN INSPECTIONS No code inspections or static analysis No quality assurance No formal testing	45%	60%	68%

SOFTWARE DEFECT REMOVAL RANGES (cont.)

TWO TECHNOLOGY CHANGES

	TECHNOLOGY COMBINATIONS	DEFECT REMOVAL EFFICIENCY		
		Lowest	Median	Highest
6.	No design inspections No code inspections or static analysis FORMAL QUALITY ASSURANCE FORMAL TESTING	50%	65%	75%
7.	No design inspections FORMAL CODE INSPECTIONS/STAT. AN. FORMAL QUALITY ASSURANCE No formal testing	53%	68%	78%
8.	No design inspections FORMAL CODE INSPECTIONS/STAT.AN. No quality assurance FORMAL TESTING	55%	70%	80%

SOFTWARE DEFECT REMOVAL RANGES (cont.)

TWO TECHNOLOGY CHANGES - continued

TECHNOLOGY COMBINATIONS		DEFECT REMOVAL EFFICIENCY		
		Lowest	Median	Highest
9.	FORMAL DESIGN INSPECTIONS No code inspections or static analysis FORMAL QUALITY ASSURANCE No formal testing	60%	75%	85%
10.	FORMAL DESIGN INSPECTIONS No code inspections or static analysis No quality assurance FORMAL TESTING	65%	80%	87%
11.	FORMAL DESIGN INSPECTIONS FORMAL CODE INSPECTIONS/STAT.AN. No quality assurance No formal testing	70%	85%	90%

SOFTWARE DEFECT REMOVAL RANGES (cont.)

THREE TECHNOLOGY CHANGES

TECHNOLOGY COMBINATIONS		DEFECT REMOVAL EFFICIENCY		
		Lowest	Median	Highest
12.	No design inspections FORMAL CODE INSPECTIONS/STAT.AN. FORMAL QUALITY ASSURANCE FORMAL TESTING	75%	87%	93%
13.	FORMAL DESIGN INSPECTIONS No code inspections or static analysis FORMAL QUALITY ASSURANCE FORMAL TESTING	77%	90%	95%
14.	FORMAL DESIGN INSPECTIONS FORMAL CODE INSPECTIONS/STAT. AN. FORMAL QUALITY ASSURANCE No formal testing	83%	95%	97%
15.	FORMAL DESIGN INSPECTIONS FORMAL CODE INSPECTIONS/STAT.AN. No quality assurance FORMAL TESTING	85%	97%	99%

SOFTWARE DEFECT REMOVAL RANGES (cont.)

BEST CASE RANGE

TECHNOLOGY COMBINATIONS	DEFECT REMOVAL EFFICIENCY		
	Lowest	Median	Highest
16. FORMAL DESIGN INSPECTIONS STATIC ANALYSIS FORMAL CODE INSPECTIONS FORMAL QUALITY ASSURANCE FORMAL TESTING	95%	99%	99.99%

DISTRIBUTION OF 1500 SOFTWARE PROJECTS BY DEFECT REMOVAL EFFICIENCY LEVEL

Defect Removal Efficiency Level (Percent)	Number of Projects	Percent of Projects
> 99	6	0.40%
95 - 99	104	6.93%
90 - 95	263	17.53%
85 - 90	559	37.26%
80 - 85	408	27.20%
< 80	161	10.73%
Total	1,500	100.00%

SOFTWARE QUALITY UNKNOWNS IN 2013

SOFTWARE QUALITY TOPICS NEEDING RESEARCH:

Errors in software test plans and test cases

Errors in web content such as graphics and sound

Correlations between security flaws and quality flaws

Errors in data and creation of a “data point” metric

Error content of data bases, repositories, warehouses

Causes of bad-fix injection rates

Impact of complexity on quality and defect removal

Impact of creeping requirements on quality

CONCLUSIONS ON SOFTWARE QUALITY

- No single quality method is adequate by itself.
- Formal inspections, static analysis, models are effective
- Inspections + static analysis + testing > 97% efficient.
- Defect prevention + removal best overall
- QFD, models, inspections, & six-sigma prevent defects
- Higher CMMI levels, TSP, RUP, Agile, XP are effective
- Quality excellence has ROI > $15 for each $1 spent
- High quality benefits schedules, productivity, users

REFERENCES ON SOFTWARE QUALITY

Black, Rex; Managing the Testing Process; Microsoft Press, 1999.

Crosby, Phiip B.; Quality is Free; New American Library, Mentor Books, 1979.

Gack Gary, Managing the Black Hole; Business Expert Publishing, 2009

Gilb, Tom & Graham, Dorothy; Software Inspections; Addison Wesley, 1983.

Jones, Capers & Bonsignour, Olivier; The Economics of Software Quality; Addison Wesley, 2011 (summer)

Jones, Capers; Software Engineering Best Practices; McGraw Hill, 2010

Jones, Capers; Applied Software Measurement; McGraw Hill, 2008.

Jones, Capers; Estimating Software Costs, McGraw Hill, 2007.

Jones, Capers; Assessments, Benchmarks, and Best Practices, Addison Wesley, 2000.

REFERENCES ON SOFTWARE QUALITY

Kan, Steve; Metrics and Models in Software Quality Engineering, Addison Wesley, 2003.

McConnell; Steve; Code Complete 2; Microsoft Press, 2004

Pirsig, Robert; Zen and the Art of Motorcycle Maintenance; Bantam; 1984

Radice, Ron; High-quality, Low-cost Software Inspections, Paradoxican Publishing, 2002.

Wiegers, Karl; Peer Reviews in Software, Addison Wesley, 2002.

REFERENCES ON SOFTWARE QUALITY

www.ASQ.org	(American Society for Quality)
www.IFPUG.org	(Int. Func. Pt. Users Group)
www.ISBSG.org	(Int. Software Bench. Standards Group)
www.ISO.org	(International Organization for Standards)
www.ITMPI.org	(Infor. Tech. Metrics and Productivity Institute)
www.PMI.org	(Project Management Institute)
www.processfusion.net	(Process Fusion)
www.SEI.CMU.edu	(Software Engineering Institute)
www.SPR.com	(Software Productivity Research LLC)
www.SSQ.org	(Society for Software Quality)

REFERENCES ON SOFTWARE QUALITY

www.SEMAT.org	(Software Engineering Methods and Theory)
www.CISQ.org	(Consortium for IT software quality)
www.SANS.org	Sans Institute listing of software defects
www.eoqsg.org	European Institute for Software Qualiy
www,galorath.com	Galorath Associates
www.associationforsoftwaretesting.org	Association for Software Testing
www.qualityassuranceinstitute.com	Quality Assurance Institute (QAI)

When Agile Becomes a Quality Game Changer: What Recent Benchmark Data Says About Agile's Development Advantage

Michael Mah

QSM Associates

Abstract

With Agile now becoming mainstream, what's happening on the topic of "Clean Code?" What patterns are being revealed, and what does this mean to teams responsible for that final lap (the testing one!)? Industry research from QSM Associates reveals varying degrees of success. Some of the best teams – whether they be XP, SCRUM, Lean, etc. – are finding significant quality implications that are literally redefining the economics of software. Others are not. What factors can make a meaningful difference? With the latest industry analysis of velocity, burndown, and quality data, we discuss productivity, time-to-market, quality, and cost patterns as this community matures. Serving as a comparison framework is the QSM SLIM industry database, with more than 10,000 completed projects (waterfall, agile, offshore, onshore) collected worldwide. This talk will describe findings that can help accelerate your success. Join us for an overview of this approach, and find out how you can assess your own patterns that could be applied to your development, and informing your executive teams.

Biography

Michael Mah (Twitter: @MichaelCMah), is Managing Partner of QSM Associates, Inc., which helps organizations measure, plan, estimate and control software projects. He is also the Benchmark Practice Director at the Cutter Consortium, a Boston-area IT think-tank. QSM Associates uses (and offers to its clients) the SLIM (Software Lifecycle Management) Suite of tools, so managers can benchmark and forecast Agile, waterfall, in-house, offshore/multi-shore or ERP/package implementation projects.

There's something in a name. Take Agile: does Agile development really make companies' IT projects succeed? And, more quickly? Are Agile software teams really more agile?

It wasn't so long ago that many developers, CTOs and project executives –plus a few pundits-- adopted a "wait and see" attitude about Agile. Some waited, some adopted early. And now, over the past 20 years, all have seen: as Agile development has become mainstream, interest in quality and productivity trends is escalating.

For example, the topics of high-value code and "clean code" – software that is poorly written but can nonetheless function-- are making their way into the foreground. What patterns are being revealed, and what does this mean to teams responsible for that final lap (the testing one!)?

Industry research from QSM Associates reveals varying degrees of success. Some of the best teams – whether they be XP, SCRUM, Lean, etc. – are finding significant quality implications that are literally redefining the economics of software. Others are not.

The findings can help accelerate the success of software development projects. Serving as a comparison framework is the QSM SLIM industry database, with more than 12,000 completed projects (waterfall, agile, offshore, onshore) collected worldwide.

What factors can make a meaningful difference? As this community matures –as documented by our latest industry analysis of velocity, burndown, and quality data-- productivity, time-to-market, quality, and cost patterns are all impacted.

One of the factors is cultural: where the work gets done. Following a wave of offshore outsourcing, arguably spawned –or at least encouraged—by Thomas Freidman's *The World Is Flat*, enterprises have had time to consider the impact of offshoring on quality, cost, and time to market. They are asking: "Can intellectual processes be outsourced as easily and effectively as a repetitive manufacturing one?" At one time, the answers seemed to range from "Yes" to "Maybe." Now, it seems to be "Maybe" or "No."

Outsourcing functions like application design, or even coding, is more difficult in an era when knowledge workers are trying to solve design challenges. The Agile development community in Columbus, Ohio was the first to discover data to prove that. And, while they were at it, the Agile community in mid-America learned a few things about itself, as well.

A first-of-its-kind study analyzing the Agile development practices of one programming community –in this case, Columbus— reveals a level of achievement that far exceeded expectations of both the analysts and the participants. The study allows participants to objectively benchmark the organization's performance to create an initial productivity baseline. This enables companies to identify strategic directions and goals, and to focus their improvement efforts with optimal efficiency. These early results of the Columbus Agile Benchmark Study are consistent with the experience at Nationwide, a marquee insurance company based in Columbus.

Guru Vasudeva, Senior VP and Enterprise CTO, says that the data from participating companies shows a strong pattern of productivity, fast time-to-market and low cost, along with an impressively low level of defects. "The fact that our own results are consistent with a larger community, in both quality and productivity, adds credibility to the claims of Agile's benefits," he remarked. With the success of the Columbus study becoming more widely known and understood, similar studies are now being launched in Munich, Chicago, and Boston.

Despite some initial hesitation from a small number of participants who were concerned about favorable *vs.* unfavorable industry comparisons, the defect data in Columbus was significantly better than industry averages. Most installations can expect an improvement in schedule when adopting new technologies like Agile, but the improvement in quality was striking.

The study, being conducted by QSM Associates in tandem with the Central Ohio Agile Association (COHAA) and the Columbus Executive Agile SIG, bodes well for adopters of Agile software development. As for offshoring, not so much.

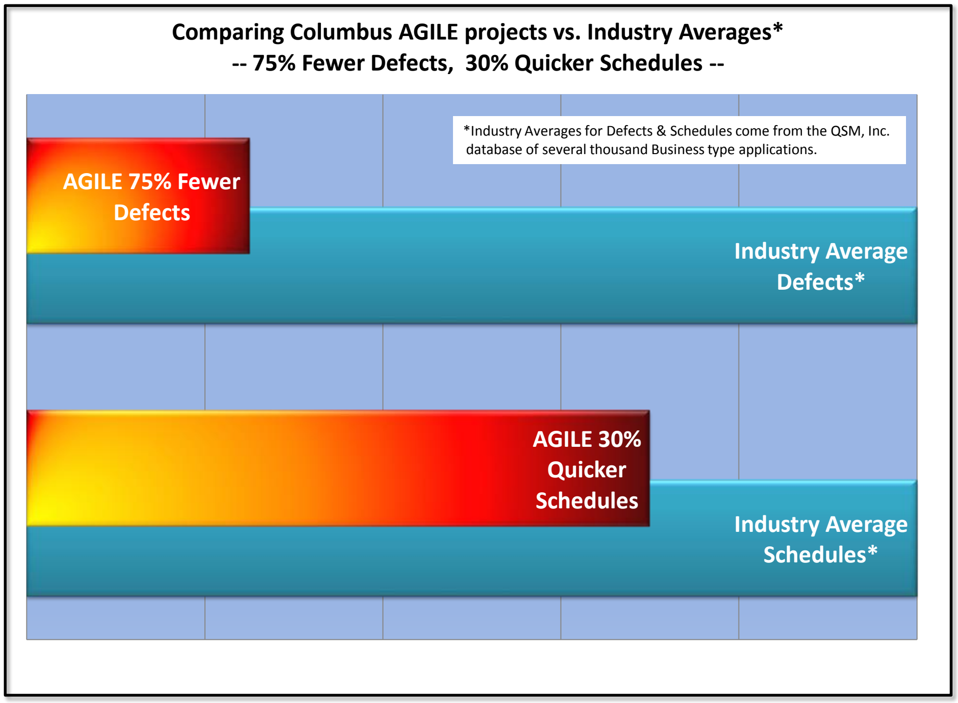
Why Columbus? There is a strong and cohesive Agile community there, and they were the first to decide that they wanted to see how good they are. And, that turns out to be pretty good: Early results from the Columbus-area participants show that a typical business system comprising 50,000 lines of code is completed 31% faster than the industry average in the QSM industry database of completed projects (4.4 months vs. 6.4 industry average). Even more remarkable is the defect rate, which is 75% lower than the industry norm.

It is important to point out that not all participants in studies like this –even if they are committed to Agile development-- may achieve such extreme results, because not all participants will have adopted all of the best practices that lead to success. Early results do show that concepts embraced by Agile deliver remarkable results in areas of compressing a schedule and reducing defects. Some of these approaches include acceptance-test-driven development (TDD), pair programming, and co-location.

But even participants that had not adopted all of these techniques achieved better-than- average results. Survey participants are able to see their own results contrasted with the industry at large; and, the Columbus community also compared the regional results in the aggregate with worldwide data.

However, participants like the idea that other companies cannot identify the individual results –other than their own-- thus protecting the confidentiality of divisions or companies involved in the study. For this reason, we have been able to confirm information that did not seem intuitive at first: for example, as will be seen below, some successful Agile development shops are reversing the trend of outsourcing or "offshoring" some of their software development efforts.

The study –which is ongoing and still attracting participants -- is providing the Columbus Agile community with valuable information on factual patterns on productivity and quality, instead of just anecdotal claims. Moreover, the data helps answer questions about addressing development projects schedules and budgets.



Aside from the value inherent in knowing the current status, the Columbus experience will be analyzed on current and future projects, says Ben Blanquera, Application Delivery Executive at Pillar Technologies. At the time of the study, Ben was Curator of the Columbus Executive Agile SIG, and he says that that experience will be reflected in a continuous improvement process. "Participants never stop learning about their own productivity gains when compared with industry practices," he says. "The benchmark, as important as it is, is just a starting point that will help establish the Columbus community as a hotbed for knowledge and experience in all manner of Agile development."

Agile is maturing. And, as more companies increase their emphasis on Test Driven Development and perhaps Acceptance Test Driven Development, we can anticipate even greater improvements, says Vasudeva. "Within Nationwide, we are seeing significant improvement in productivity and quality thru Acceptance Test Driven Development."

The agile-vs.-offshore debate is familiar territory, and is fast becoming the central issue. The SEI model for software process maturity opened the floodgates for American development jobs to go overseas. With agile, we are reclaiming the software quality and innovation back where it started, here in North America. We know more today than we used to, about creative knowledge work (software).

Bart Murphy, Treasurer of COHAA, explains one reason why this reconsideration is taking place: "Outsourcing is proving to be an old-fashioned concept that might have worked well in old-line industries, such as manufacturing, but it is coming back to haunt some new-age industries."

I agree. But recognizing one of the main reasons that outsourcing can be so risky – the lack of assurance or common expectations on quality and productivity-- can help make outsourcing more successful. That is, measurement/benchmarking helps both sides set –and thus agree on-- more realistic expectations. Estimating is difficult at best in an outsourcing environment, and downright daunting without the proper tools, which is why studies like this are so important. They document actual practice.

The results from Columbus were aggregated into the SLIM software lifecycle management solution, which includes an industry-wide database of thousands of completed projects. SLIM allows "normal humans" to accomplish sophisticated analysis with ease. Chief among the results is the fact that programming teams that are co-located tend to be more effective than those where expertise is geographically divided. This is one of the facts that have lead to the reassessment of outsourcing software development.

Sure, outsourcing or offshoring may make sense in an Industrial Economy based in cost-effective manufacturing. And yes, as Bart says, it is harder in an Information Economy when knowledge workers are trying to solve design challenges. And, true, we have data to prove all that.

That being said, some companies may still choose to outsource. For those that do, benchmarking techniques serve a vital purpose in negotiation and relationship management. Knowing industry success factors and best practices will help the customer get the right price from the vendor. Software project history can help establish reasonable productivity targets and service levels for the client. As a vendor, measurement helps establish fair and reasonable business terms for both parties, and document fairness of goals.

Based on fact and not hyperbole, measurement helps create reasonable business terms and thus a healthy relationship, contributing to solid, proven relationship management practices.

If you do choose to outsource and it's working, you want to be able to demonstrate this with facts and figures, not just anecdotes.

Some of the best results in the Columbus study were achieved by enterprises working with Pillar Technology, developer of enterprise software solutions and consulting services. Agile has enabled Pillar to develop software solutions at a productivity and quality rate that can be up to 10x better than the industry norms.

Reflecting on the survey, the process, and on Agile itself, Bart Murphy of COHAA believes Agile development has proven itself. "In fact, we are now seeing some medium-to-large software shops actually repatriating programming resources that had been shipped to Asia or other countries, in what were considered strategic cost saving initiatives." And why is this? Although overseas development can sometimes bring dramatic schedule improvement, this often comes at a price: more defects, he says. "The cost to remediate and maintain the delivered work product outweighs the cost savings gained in the outsourced model, especially as offshore operating costs continue to rise."

Enrollments are still being accepted for companies that want to join the study, or any of the other surveys being planned for Munich, Chicago, Boston, and other locales (obtain information packet from info@qsma.com). As an incentive to participate, and a tool to facilitate the process, companies receive a temporary use license of SLIM, a data capture template that captures key metrics for Agile projects, including stories, story points, time, effort, defects, velocity, and backlog. Participants will receive a private, confidential analysis of their patterns along with the study results for the group.

Lean in the Software Test Lab

Kathy Iberle

kiberle@kiberle.com

Abstract

This paper applies the methods of Lean Science to the management of software test labs. The fundamentals of Lean Science are introduced, and the effects of batch size and Work-in-Process limits are explored. Lean Science can dramatically increase productivity and throughput in most test labs. You really can work smarter, not harder!

This paper will give you a gut feel for why these methods work and why they produce dramatic improvement. It also introduces enough of the basic concepts and tools to allow you to assess the potential for your own work.

Biography

***Kathy Iberle** has over fifteen years of experience in developing processes to plan and execute efficient development of high-quality products in a variety of fields. She has been working in agile software development and Lean development for many years, and has developed unusual expertise in the challenging area where hardware and software meet. Kathy recently retired from Hewlett-Packard after a rewarding career and is now the principal consultant and owner of the Iberle Consulting Group. Kathy served as co-chair of the Program Committee of the Pacific Northwest Software Quality Conference (PNSQC) in 2009, participated in the invitation-only Software Test Managers Roundtable for five years, and has published regularly since 1997. Kathy has an M.S. in Computer Science from the University of Washington, and an excessive collection of degrees in Chemistry from the University of Washington and the University of Michigan. Visit her website: www.kiberle.com.*

1 What Is Lean Science and Why Should I Care?

The term “Lean” was originally coined in the 1980s to denote a set of management practices and principles used very successfully at Toyota in the preceding decades. Like most management practices, Lean deals with both technical and people-oriented areas. More recently, Alan Shalloway described Lean as consisting of three bodies of knowledge (Shalloway 2010):

- Lean Science: Batch size, WIP limits, Pull management, Cadence, Flow
- Lean Management: Visual Controls, Leadership, Education
- Lean Knowledge Stewardship: A3, 5 Whys, Continuous Improvement, Kaizens

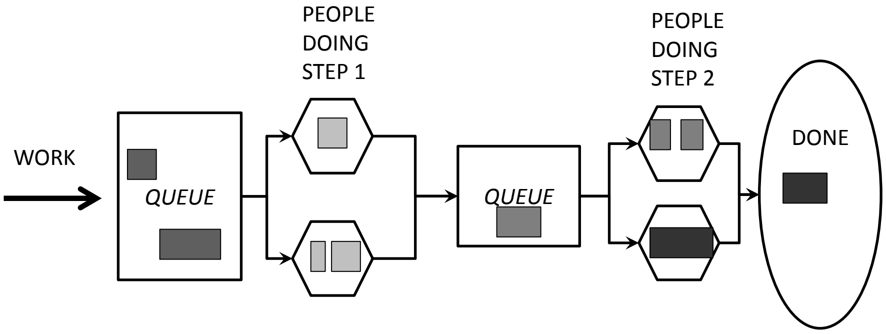
In this paper, we’ll talk mostly about Lean Science, sometimes known as Second-Generation Lean (Reinertsen 2009). The intent of Lean Science is to optimize an organization so it will produce more output over time. Lean Science is grounded in a body of mathematics known as *queuing theory*, which predicts how work flowing through an organization will behave. Queuing theory is used to design phone systems, Internet networking, traffic control systems, and other types of systems, so we know a lot about it and we know that it works.

Luckily, it’s not necessary to learn a lot of math in order to benefit from Lean Science and queuing theory. You do need to understand the basic model involved – how to see your work flowing through your organization. I’ll then describe the steps I use to get the work under control, using principles derived from queuing theory. We’ll look at how to apply those steps in a software test lab to achieve a more effective, less overloaded organization.

1 Lean Science – a Systems View

The key to Lean Science is the *systems view* - looking at the organization as if it were a system or machine for turning work requests into finished, saleable work. Lean Science is all about understanding how work flows through your system.

A systems view of a typical organization is shown below. The little boxes represent chunks of work flowing through the system. These chunks are known as *batches*. They are different sizes and arrive somewhat randomly.



At any given time, a chunk of work is either being worked on (hexagons), or waiting in a *queue* for somebody to work on it (rectangles).

The goal of Lean Science is to make the batches reach “Done” faster, without increasing the number of people doing the activities. What other parameters can be changed, and what effect will changes have? The consequence of a change is often not intuitively obvious. We don’t seem to be wired very well to reason about systems like this. However, queuing theory identifies the parameters for us and provides ways to predict what will happen if a parameter is changed.

2 Visibility First, Then Improvement

The first step in applying Lean Science is to make our work visible. We'll change the way we look at, talk about, and measure our work. This will enable us to clearly see how much value is created each week or month. The basic steps are these:

- Split our work into small, clearly defined *batches,* each of which delivers value.
- Map our system by identifying the activities and the wait states (*queues*).
- Make the progress of each batch visible.

Just making the work visible often creates improvement by making duplication, unnecessary work, and poorly defined handoffs obvious.

Once we can see our work, and we can see how much value is (or isn't) being produced, we use a few concepts from queuing theory to look for potential improvements. Typical improvements include (not in any particular order).

- Manage the batches on a cadence.
- Limit the number of batches under development at any given time.
- Reduce batch size.

This is a general approach that works for all types of "knowledge work" - software development, mechanical engineering, pharmaceutical research, etc. However, in this paper, we'll assume that the organization is a software test lab.

3 Make the Work Visible

3.1 Split the Work into Batches Which Deliver Value

The work must first be split into discrete batches. In manufacturing, this is easy to do since the work consists of physical items flowing through a manufacturing line. In software testing, we've got to decide what will constitute a batch or chunk of work for our particular lab.

Applying Lean Science will maximize the output of batches, regardless of what those batches consist of. If we simply list tasks as our batches, we will maximize the output of tasks. This is fine if your lab is recompensed by the hour. If your value is assessed any other way (which is commonly the case), each of your batches needs to deliver some value ***as perceived by your customers***. Otherwise, you'll simply maximize the amount of effort you put in, not the amount of value you produce.

Here are some typical batches which deliver value:

- Writing or finding the appropriate tests to adequately cover a newly delivered feature, running the tests, and reporting the defects in a clear, reproducible manner.
- Designing and running in-depth performance tests and reporting what performs adequately and what doesn't.
- Writing and running localization tests and reporting the defects.

- Running a regression test and delivering an assessment of readiness to release or a prediction of the likelihood of unfound defects.
- Preparing a status report on progress to date, trouble spots, and perhaps a prediction of when the software will be ready to release.

All these batches are delivering information which is valued by the development team. The developers value well-written defect reports, because those help them quickly identify the root cause of the defect. The managers value lists and totals of unfixed defects or tests still to be done, because those help them assess how much work remains to be done. Advocates for the end-user appreciate defect characterization and severity ratings, because they want to know what the end-user will see if a particular defect is not fixed. Management also values any good prediction you can make regarding how many ***unfound*** defects are still in the product. The point of the batch is not to run tests *per se*, but to acquire and deliver information.

Most test labs also have batches whose value is to reduce the lab's overhead, rather than directly provide anything to outside customers. The customer in this case is the test lab's own management. Some examples:

- Design and write a set of tests for our reusable test library.
- Automate a set of tests.

And then there's always the stuff that we just have to do:

- Move equipment due to a space reassignment.
- Hire a new employee.

So, there'll be several different types of batches, and the batches will be different sizes. That's all fine. The important points are these:

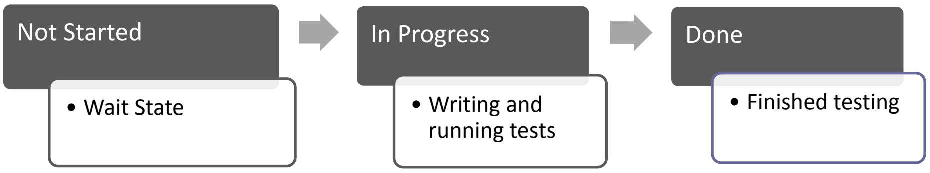
- All of your work is captured in batches.
- The value of each batch is clearly stated.
- Each batch's exit criteria are clearly defined, so we can tell when each batch is done.

3.2 Map Your System

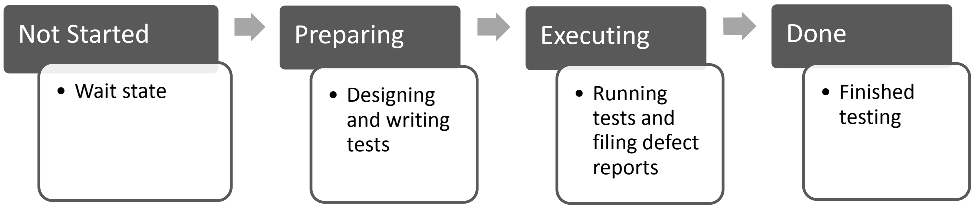
3.2.1 A Basic Map

Now you've got a list of the batches of work which your organization performs. Let's think about what happens to each batch as it moves through your organization. Who decides when a batch will be started? What are the major steps which must be performed? Will the batch be handed off between individuals or teams specializing in different areas?

Since Lean Science is meant to optimize your lab as whole, the same map will be used for all the types of work you do. This means it needs to be quite general. The simplest map has just three states, as shown to the right.



The three-state map is often sufficient for smaller teams. It might also work for a larger, more complex organization, or the larger organization may want another state or two to show the handoffs between specialized teams, as shown below.


Not Started
• Wait state
Preparing
• Designing and writing tests
Executing
• Running tests and filing defect reports
Done
• Finished testing

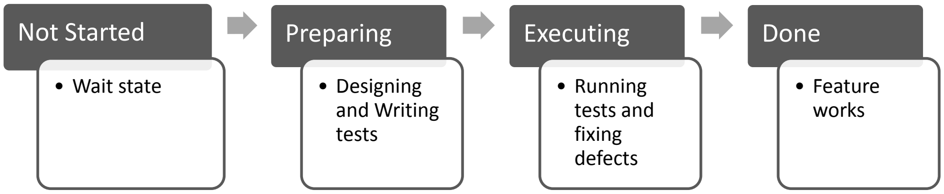
Some people call this type of map a process map and others call it a value-stream map. However, this map usually is less detailed than either a formal value-stream map or a formal process map. It's better to start simple, and add detail only as you need it. It's not necessary to capture the total reality of the process – you just need enough to give you useful insights.

3.2.2 Thoughts on System Optimization

You may notice that all this is going to optimize *the test lab*, not the overall organization. Will this help the overall organization make more money or be more effective? If the organization sells testing services, it will. If the overall organization sells software, or sells something else and is using the software to assist in the something else, then making the test lab more productive may not help the bottom line. It can even make things worse! This depends on whether testing is or is not a bottleneck in the production of the software.

Most organizations will argue that testing is a bottleneck, at least some of the time. In reality, changes in the development process might help more than changes in the test lab, but the test lab manager normally doesn't make decisions at that level. The bottom line: use Lean Science to optimize the area of the organization which you control, but keep in mind that you are only a piece of the whole. If you get the opportunity to participate in optimizing a wider area, take it!

One way to make sure that the whole system is being optimized is to focus your batches on feature completion, not on testing. Change the definition of “done” from “finished running the tests” to “this feature works”, as shown in this process map:


Not Started
• Wait state
Preparing
• Designing and Writing tests
Executing
• Running tests and fixing defects
Done
• Feature works

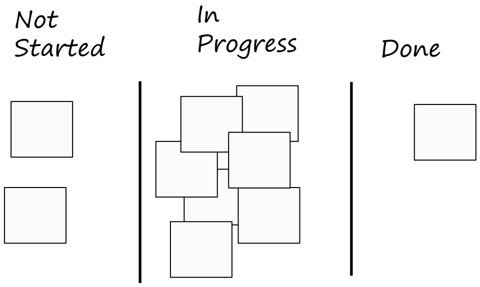
In a completely integrated agile development system, this is already the way things work. In a waterfall situation, or where the test lab is integrating the work of several unsynchronized agile teams, this method can be applied. (Iberle 2010) As we'll see later, this approach also allows the test lab to predict the end date of the project fairly accurately.

3.3 Make Each Batch and Its Progress Visible

Once all your work is expressed in batches, it must be made visible in a concrete way.

The simplest way to make work visible is to create a list of the batches. I've seen this cause improvement all by itself, particularly in large multi-site organizations. Once all the work is listed in one place, it's easy to see duplications and work that was supposed to have been discontinued. Since the batches are stated in terms of value, it's also easier to see why the work is being done.

The next level up is the Visual Planning Board, which tracks not only the existence of the work but also its current status. Your process map is depicted as columns on a wall or whiteboard, and each batch is represented by a sticky note in the appropriate column. It's much easier to grasp the "big picture" with this visual format, which makes day-to-day problems more visible.

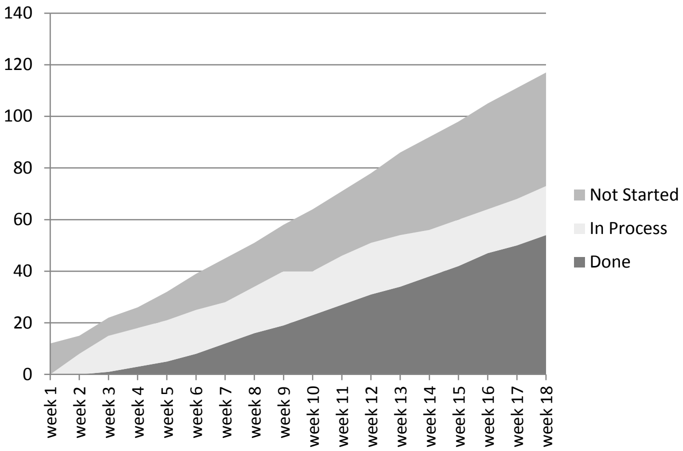


The third tool, and the most powerful, is the *Cumulative Flow Diagram* or CFD. The Visual Planning Board is a snapshot in time. The CFD tracks progress trends over time.

Each week, count how many batches are in each state. Graph the totals as a stacked area chart, with "Done" on the bottom and "Not Started" on the top, as shown to the right.

For all you testers: Have you used a chart like this to track defect trends over time? If so, you already know quite a bit about reading this chart.

For all you agilists – yes, this is a "burn-up" chart. Notice that it's burning up *batches*, not tasks. This keeps us focused on improving throughput of batches, not throughput of effort.



3.4 What Can the Cumulative Flow Diagram Tell Me?

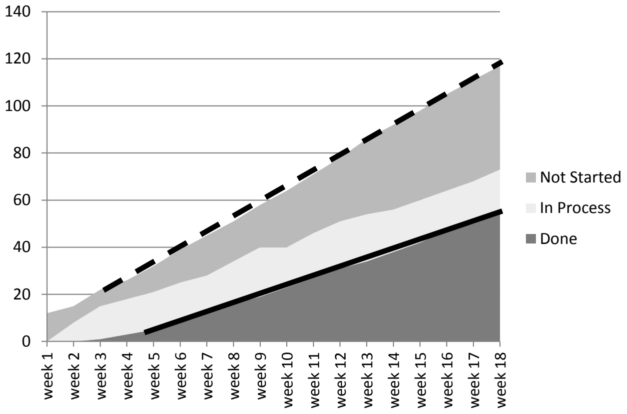
Let's see what we can learn from our CFD.

3.4.1 Are We Getting Behind?

The solid line on the CFD to the right shows how many batches are delivered each week. This is known as the *throughput*.

The dashed line shows how many batches are requested each week. This is the *demand*.

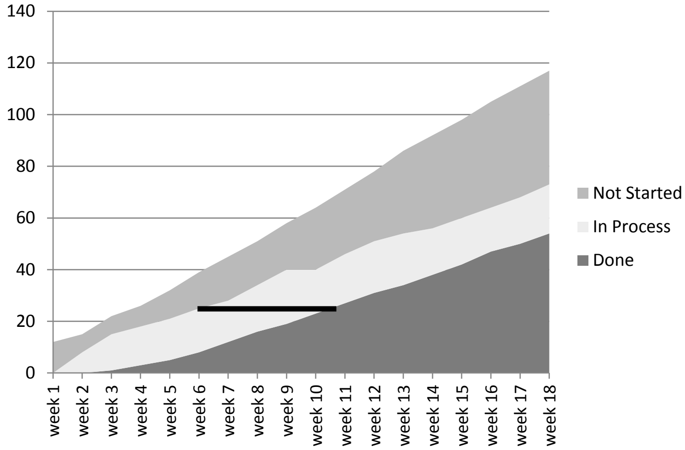
If work is requested faster than it can be done, the work is going to pile up. The CFD makes the problem visible, in a way that is compelling in management meetings.



3.4.2 How long does it take for the average item to finish?

A team can have good throughput but still be criticized for the length of time it takes to address a newly submitted item. The time from the start of active work to completion is known as the *cycle time*.

The average cycle time can be read right off the CFD by drawing a horizontal line across the "In Process" area. In this example, the average item spends four and a half weeks "In Process". The average cycle time can change over time. We'll discuss that in a later section.



3.4.3 When will we be done?

If your batches are designed around features or capabilities of the product, your CFD might be able to predict when the product will be ready to release.

You *can* predict the future if these criteria are met:

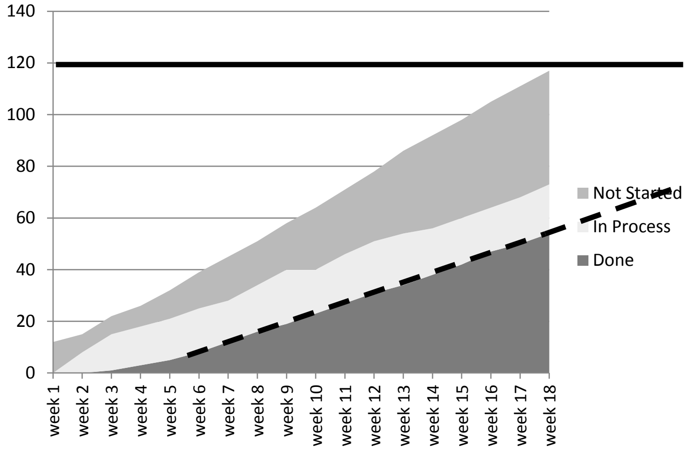
- Each batch represents a useful feature and/or capability.
- The batch is not considered "done" until the feature or capability is *passing* the tests.
- Only the batches representing testing of a given product or release are shown. The CFD for this product doesn't include batches for ongoing lab maintenance, testing of other projects, etc.
- There is a list of the probable total features and/or capabilities that need to be finished before the product can release. Your test plan probably contains such a list.
- The batches are small enough that the total project contains "a lot". I like to see at least fifty batches.

In this CFD,

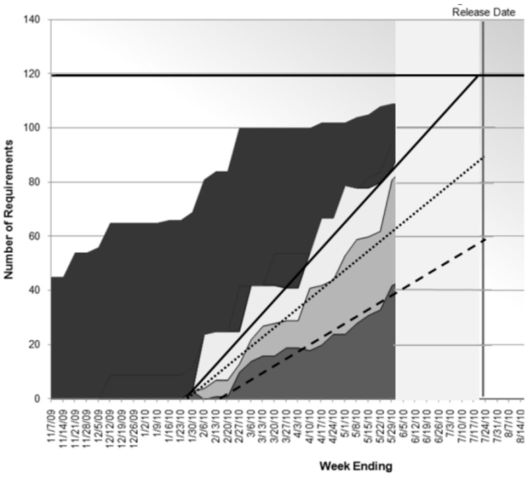
- The dashed throughput line shows how fast the batches are getting done.
- The solid line represents the approximate number of batches which are planned for this project.

The approximate end date of the project can be estimated by extending the throughput line until it meets the planned-batches line. Agile teams often do this type of prediction using their burn-up or burn-down charts.

Of course, real data is much messier than this "toy" example.



This CFD contains data from a real project (Iberle 2010).



- The solid throughput line represents features entering system test.
- The middle dotted line represents features which have been tested but have not yet passed the tests. They are waiting for a fix, or being fixed, or being re-tested.
- The dashed line represents features completed.

All three throughput lines are fitted to the actual data. It's possible to use least-squares regression but most people simply draw a line that looks like it fits, as was done in this case. The distance between the line and the actual data points on either side of it gives you an idea of the accuracy of the prediction.

The prediction is more accurate if many batches are involved, just like the total of many coin-flips will be very close to 50% heads and 50% tails. Very small teams won't get much help from a CFD because they just don't have enough data points to form a pattern. It also helps if the average size of the batches doesn't radically change between one time point and the next.

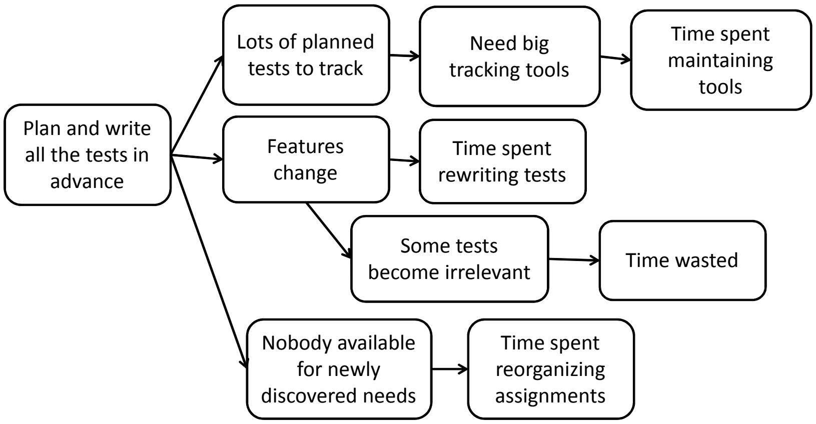
The CFD provides an unparalleled view of the long-term performance of an organization, regardless of the exact nature of the work performed from day to day. We'll see some more ways to use the CFD as we proceed.

4 Improvements in Your Test Lab

Now that you've got your tools in place, you can make improvements in the throughput of your test lab. As our batches move through our system, we want to reduce waste (unneeded work) and delay (batches waiting instead of moving along).

Software development and software testing both consist of **variably sized** chunks of work arriving at a **variable and unpredictable** rate. Queuing theory tells us that the most common and nefarious sources of waste and delay in such a system are

- excessively large batch size
- too much work-in-process


Plan and write all the tests in advance
Lots of planned tests to track
Need big tracking tools
Time spent maintaining tools
Features change
Time spent rewriting tests
Some tests become irrelevant
Time wasted
Nobody available for newly discovered needs
Time spent reorganizing assignments

In his 2009 book *Principles of Product Development Flow*, Reinertsen lists ten major negative effects of excessive batch size. We're often so used to the effects of large batches that we don't even see them. For instance, in some organizations the standard process calls for all the tests to be written before the first build of the product arrives for testing. On the surface, this sounds like good practice. But when you look at it carefully, there is a lot of waste, as shown above.

The other major, and perhaps more insidious, source of waste is *Work-in-Process* or WIP. The waste here is caused by the cost of task-switching. A human being cannot switch from one task or topic to another for free. The simplest switches take a few seconds. In a switch between complex tasks, the worker may take as much as ten minutes to come fully up to speed on the second task. The worker can get the most work done, with the least stress, when he or she works on only two or three different things in a given day.

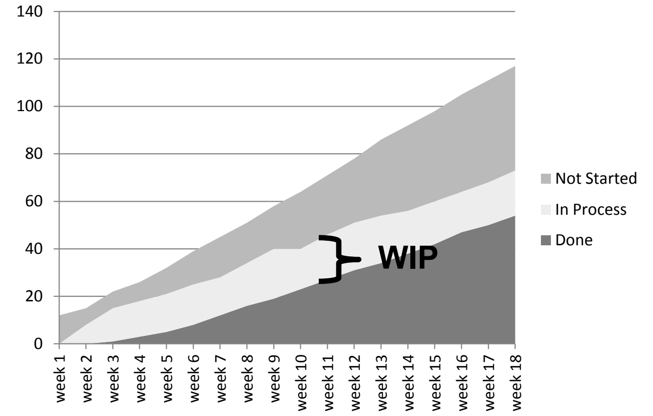
In addition, when a number of batches are in progress at the same time, the organization has to keep track of them. This means creating and maintaining lists, tracking systems, status reports, and so forth. Most organizations are investing a lot of time and money in keeping track of half-finished work. If you limit the Work-in-Process – finish items before starting new ones – both the task-switching time and the tracking work can be cut dramatically.

I'm going to talk about WIP first and then batch size, because simply defining the batches as we did in the earlier sections often corrects the batch size enough to get started. You may have different results.

By the way, if you're familiar with Lean, you may be wondering why I'm not talking about the "seven wastes" or "one-piece flow". These rules are derived from applying queuing theory to a system of **consistently sized** chunks of work arriving at a **steady and predictable** rate, such as you find in manufacturing or health care or office management. Systems of variably sized work behave somewhat differently, so the rules derived from queuing theory for variable systems are not exact analogues of the rules for consistently sized systems. In addition, product development is meant to discover new knowledge, so variation is sometimes valuable rather than always being detrimental.

4.1 Limit the Number of Batches in Progress

The current work-in-process or WIP can be read off the CFD as shown to the right. This shows the WIP of the whole department. Ideally, no individual in the department is working on more than two or three batches at the same time. The ideal WIP limit for the department thus is usually twice or three times the number of people.

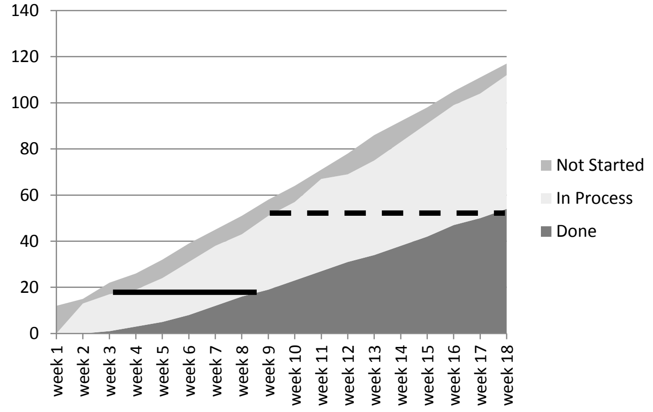


This seems counter-intuitive to many people. If there are more requests coming in, we should start more items. Everyone will see that we're working hard and using all our people to full capacity, right?

Wrong!

Let's see why…

Here's a CFD for an organization which keeps taking on new batches before the older ones are finished. Its WIP is gradually increasing.



The two lines show the average time elapsed between starting an item and finishing an item. In week 3, the average elapsed time is about 5 weeks. By week 9, the average elapsed time has grown to 9 weeks. As the WIP grows, the average elapsed time grows, which means that responsiveness drops.

In addition, the overhead caused by the increased WIP makes the throughput decrease. Now you're taking longer to get less done. This doesn't impress your departments' customers.

This effect is particularly troublesome for batches representing internal improvement initiatives, which are usually getting only your "left-over" time in the first place. In a large lab, it's frighteningly easy to start so many worthwhile initiatives that almost nothing gets done.

It may seem impossible at first to limit your Work-in-Process. The key is to "Stop Starting, Start Finishing". You've got to stop starting new batches until some of the old batches are finished. Start with the work that is under your direct control, such as improvement initiatives. Stop starting new ones until you've finished some of the older ones. You will gradually be able to expand this to include testing batches. The discipline is easier if batch size is kept small. When batches are small, more batches will finish each week, providing more opportunities to start something new or change priorities.

4.2 Reduce Batch Size

Now that you can see and control your WIP, it's time to talk again about batch size.

What is the right size for a batch? The ideal batch size is a trade-off between the positive aspects of getting value more frequently, and the negative aspects of managing lots of small batches. When the management costs get too high, your batches are too small. Don Reinertsen gives much more detailed explanation, with the math, in *The Principles of Product Development Flow* (Reinertsen 2009 Ch. 5).

You can approximate the math by listing both the costs and the benefits of the batch size, so you can compare the effects of changing the batch size.

Typical benefits of small batches:

- Earlier delivery of valuable results to your customers.
- Faster feedback on how long this batch really took. (As we observed earlier, cumulative information on actual elapsed time improves schedule predictions.)
- More frequent opportunities to respond to newly discovered needs without leaving half-finished work behind.
- Reduced risk. You'll find out sooner if your test equipment isn't adequate, or the network keeps failing, or the build system isn't reliable.
- Earlier opportunities to realize that further testing on this build will just waste time and money.

Typical costs of small batches:

- Keeping track of all the batches – maintenance of tools, routine meetings to review the batches.
- The effort needed to define each batch.

One rule of thumb is to make the batch sizes small enough to change people's assignments reasonably often. For instance, if you'd like to change people's assignments every two weeks, then the average batch size must be around two weeks or less. Anything larger will cause your system to "clog up" and your responsiveness will go down. Another rule of thumb is to make the batches correspond to specific questions to be answered. For instance, a week's delivery from a development team of ten probably corresponds to a couple of test batches, not just one.

It takes some practice to get good at defining batches that are both useful and small, particularly for improvement initiatives. At first, people have trouble imagining that anything useful can be delivered without doing the entire project. Agile development discussions of “splitting user stories” can be helpful, particularly when you’re looking at batches which are improvement projects. Try using the Extreme Programming mantra “What is the simplest thing that could possibly work?” (Beck 2000).

4.3 Manage the Batches on a Cadence

A cadence is a regular, predictable rhythm within a process. For instance, a particular meeting happens on the same day of every week, or your website is refreshed on the last day of the month every month.

A cadence saves time by reducing waste. When an activity happens at a predictable time, people plan their work to take advantage of the cadence. A cadence also saves time by reducing overhead. Instead of coordinating multiple people’s availability and finding a meeting room each time the meeting is needed, the calendar coordination and meeting room reservation is done just once.

Batches are managed on a cadence by making decisions about which batches to do, or not do, on evenly spaced dates. Ideally these dates are quite close together – every week, every two weeks, or every four weeks. This reduces the need to discuss priorities repeatedly and maintain lists in priority order. Instead, every two weeks, the group decides which are the most important things to do *in the next two weeks*. Since only a limited number of things can be done in the next two weeks, the team focuses on just those and decisions can be made quickly. Little or no time is spent debating the relative priorities of items in the far future.

Of course, if you’re working within an agile method already, there are at least two cadences already in place – the sprint cadence and the standup meeting cadence. If you’re not, establishing a regular cadence to review the planned testing (preferably with the developers!) will probably cut your overhead. Your internal improvement projects can be managed on a similar cadence, although probably in a different meeting. David Anderson’s book *Kanban: Successful Evolutionary Change for Your Technology Business* (Anderson 2010) describes Kanban in fairly general terms, providing a structure which is usable by groups who aren’t doing software development.

Think about other ways a cadence might benefit you. For instance, if your senior people review the work of more junior people, regularly scheduled “office hours” are likely to be more efficient than interrupting every time someone is ready for a review. Any time someone has a part-time activity working with or helping someone else, applying a cadence might help.

5 Case Studies

Most of the methods in this paper are based on Don Reinertsen’s work in *The Principles of Product Development Flow: Second-Generation Lean Product Development* (Reinertsen 2009). This book is a general treatment of the principles and how they can be applied in a variety of fields, with a few examples involving software testing. The usefulness of the methods is demonstrated by (among other examples) the success of agile software development, although I suspect that many agile practitioners don’t realize that a large part of their success is due to controlling the WIP and batch size. In his book *Agile Software Requirements*, Dean Leffingwell describes how the theory behind Second-Generation Lean applies to software development in general. (Leffingwell 2011).

However, there is very little published on software testing per se. My personal experience with this method is captured in three publications:

- “Introducing Fast Flexible Flow at Hewlett-Packard”. (Iberle 2012). This paper describes how these methods were initially used to get better control of the work flowing through a large system test lab.

- “Kanban: What is it and why should I care?” (Reese 2011). Landon Reese and Kathy Iberle describe the next step taken by the same group, which was implementation of a Kanban system to handle tooling work. This system was later expanded to manage test writing and test execution requests.

- “Lean System Integration at Hewlett-Packard” (Iberle 2010). This paper describes the first year or two of managing the integration of a large system, including the prediction of project end dates. More progress was made since the publication of that paper, but has not been published.

Matt Heusser shares some thoughts on a Lean approach to software testing in his blog “Applying Lean Concepts to Software Testing” (Heusser 2010).

6 Bottlenecks, Value Stream Maps, and Other Topics

This paper has focused on software testing as an activity that consists of **variably sized** chunks of work arriving at a **variable and unpredictable** rate. However, most test labs do have activities where the work is both **consistently sized** and arrives at **a predictable rate**. These tend to be repetitive activities, such as installation of operating systems, build-and-smoke-test procedures, or regression testing. In this case, the methods used in Lean Manufacturing can be very helpful.

The Lean Manufacturing approach typically starts by creating a detailed value-stream map, and then using the map to look for bottlenecks and wasted effort. Descriptions of this method are readily available from many sources, such as the Lean Enterprise Institute website. (Lean Enterprise Institute 2013). If the you find that the waste is not readily visible or the bottleneck moves around, I suggest trying the approach in this paper instead.

7 Conclusion

Applying Lean Science to a software test lab is not a trivial activity, but it can be very rewarding. The first, and often the most difficult part, is to re-define the work into small, discrete batches which have clear end points and independently deliver value. A simple system map is also necessary. Simply defining the work in batches and making the batches visible often creates improvement in several ways:

- The increased visibility can expose unnecessary work – accidental duplication, failure to communicate a cancellation, etc.

- Defining the value of the information to be delivered by each batch can reveal testing that is scheduled too early, excessively thorough, or simply not needed.

- Establishing the criteria for “done” can resolve long-standing mismatches between subteams.

Once your work can be viewed as batches flowing through your system, you have a large number of tools at your disposal to manage that work more efficiently. The Visual Planning Board provides a quick and easy way to communicate status. The Cumulative Flow Diagram provides a view of the long-term performance of longer projects and of the lab itself.

Once these tools are available, the organization can improve its performance by going after the most likely candidates for waste in a variable system – excessive WIP and excessive batch size. Other Lean methods can also be applied, such as cadence.

Good luck with your journey into Lean Science!

8 References

Anderson, David J. 2010. *Kanban: Successful Evolutionary Change for Your Technology Business*. Sequim, WA: Blue Hole Press.

Beck, Kent. 2000. *Extreme Programming Explained: Embrace Change*. Boston: Addison-Wesley.

Heusser, Matt; “Applying Lean Concepts to Software Testing” (posted November 2010). http://searchsoftwarequality.techtarget.com/tip/Applying-lean-concepts-to-software-testing?asrc=EM_NLN_12859187&utm_medium=EM&utm_source=NLN&utm_campaign=20101116_Today's%20News:%20Applying%20lean%20concepts%20to%20software%20testing_mewebb&track=NL-498&ad=798129 (accessed July 10th, 2013).

Iberle, Kathleen A. 2010. “Lean System Integration at Hewlett-Packard”. Proceedings of the Pacific Northwest Software Quality Conference 2010.

Iberle, Kathleen A. 2012. “Introducing Fast Flexible Flow at Hewlett-Packard”. Lean Product and Process Development Exchange, 2012.

Iberle, Kathy. 2013. “Finding the Best Frequency for a Recurring Activity”. http://www.kiberle.com/2013KB/CadenceMath.pdf (accessed July 10th, 2013).

Leffingwell, Dean. 2011. *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*. Boston: Addison-Wesley Professional.

Lean Enterprise Institute. http://www.lean.org/WhatsLean/Principles.cfm. (accessed August 09, 2013).

Reese, Landon. Iberle, Kathleen A. 2011. “Kanban: What is it and why should I care?”. Proceedings of the Pacific Northwest Software Quality Conference.

Reinertsen, Donald G. 2009. *The Principles of Product Development Flow: Second Generation Lean Product Development*. Redondo Beach, CA: Celeritas Publishing.

Shalloway, Alan. Beaver, Guy. Trott, James R. 2010. *Lean-Agile Software Development: Achieving Enterprise Agility*. Boston, MA: Pearson Education. pp. 217-218.

Kathy Iberle’s publications can be found at http://www.kiberle.com/articles.html

The Myths Behind Software Metrics

Douglas Hoffman, Software Quality Methods

Measurement, metrics, and statistics can be powerful tools for understanding the world we live in. Measures and metrics abound in software engineering, QA, and especially testing. Coming up with numbers is the easiest part. Making some computations using the numbers is straightforward. Getting meaningful, useful metrics is a whole different story. Computing metrics is easy to do poorly. It is much more difficult to take measurements and generate truly meaningful and useful metrics or statistics.

Douglas Hoffman, *Software Quality Methods, has been teaching software testing, software engineering, and quality assurance for over 20 years. He does training and management consulting in strategic and tactical planning for quality software across varied applications and industries. His technical emphasis is on test automation and test oracles. He is Past President of the Association for Software Testing (AST), a Fellow of the ASQ (American Society for Quality), and holds degrees including MBA, MSEE, and BACS. He holds certificates from ASQ in Software Quality Engineering and as a Manager of Quality/Organizational Excellence. Douglas is a founding member and current Chair of SSQA (Silicon Valley Software Quality Association), past Chair of the Silicon Valley Section of ASQ, a founding member for AST, and a Senior Member of ACM and IEEE. He is author of dozens of papers, has spoken at dozens of conferences, and has been a leader for several national and international conferences on software quality, including PNSQC.*

Today's Testing Innovations

Lee Copeland, Consultant

As a consultant, Lee Copeland has spoken with thousands of software testers in hundreds of different organizations. Generally, he comes away from these discussions depressed with the state of testing. Many organizations neither know about nor have adopted recent important innovations in our field. Lee will discuss nine of the important innovations in testing—the context-driven school, test-first development, really good books, open source tools, session-based test management, testing workshops, freedom of the press, virtualization, and testing in the cloud. Join Lee for his list, and propose others if you'd like. Discuss the keys to innovation and take a test evaluating your organization's innovation quota.

With more than thirty years of experience as an information systems professional at commercial and nonprofit organizations, ***Lee Copeland*** *has held technical and managerial positions in applications development, software testing, and software process improvement. Lee has developed and taught numerous training courses on software development and testing issues, and is a well-known speaker with Software Quality Engineering. Lee presents at software conferences in the United States and abroad. He is the author of the popular reference book, A Practitioner's Guide to Software Test Design.*

How to Estimate ANYTHING

Payson Hall, Catalysis Group, Inc.

Given the choice between estimating a task and getting a root canal, most people gladly choose the dentist. It’s not that they enjoy pain; it’s just that the pain of a root canal is short lived… while bad estimates can cause grief for months or even years. This session offers attendees a practical estimation process that can be applied to estimate ANYTHING – and hands on experience applying the process during the presentation.

Payson Hall *is a Consulting Systems Engineer and Project Manager employed by Catalysis Group, Inc. in Sacramento. He has consulted on projects throughout North America and Europe during his 30-year career and published numerous articles on project management and systems engineering.*

How to Estimate ANYTHING

Payson Hall
Catalysis Group, Inc.
payson@catalysisgroup.com

What's an "Estimate"?

A rough calculation/prediction of the quantity or size of something

Not

☑ Promise
☑ Target

Demonstration

Increased Error…

- No time
- Can't see all
- Distractions
- No experience
- Highly complex
- Moving target

Less Error…

- Smaller chunks
- Familiar
- Well defined
- Comprehensible

Estimation Basics

- Clear written definition
- Divide & conquer
- Manageable hunks
- Extra credit if hunks resemble familiar things
- Expect change as more information becomes available

Another Demonstration

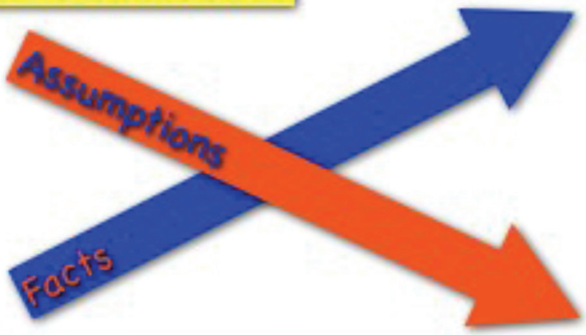
Estimation Process

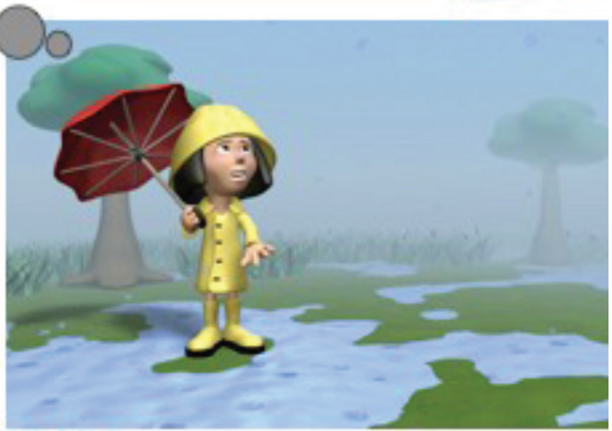
0. Document what is being estimated & what's excluded
1. Identify relevant factors – What do you wish you knew?
2. Gather data/make assumptions – Write them down
3. Consult – Problem definition, relevant metrics & assumptions
4. Do math – Show your work
5. Compare estimate & actual – Seek process improvement

Challenging Assumptions

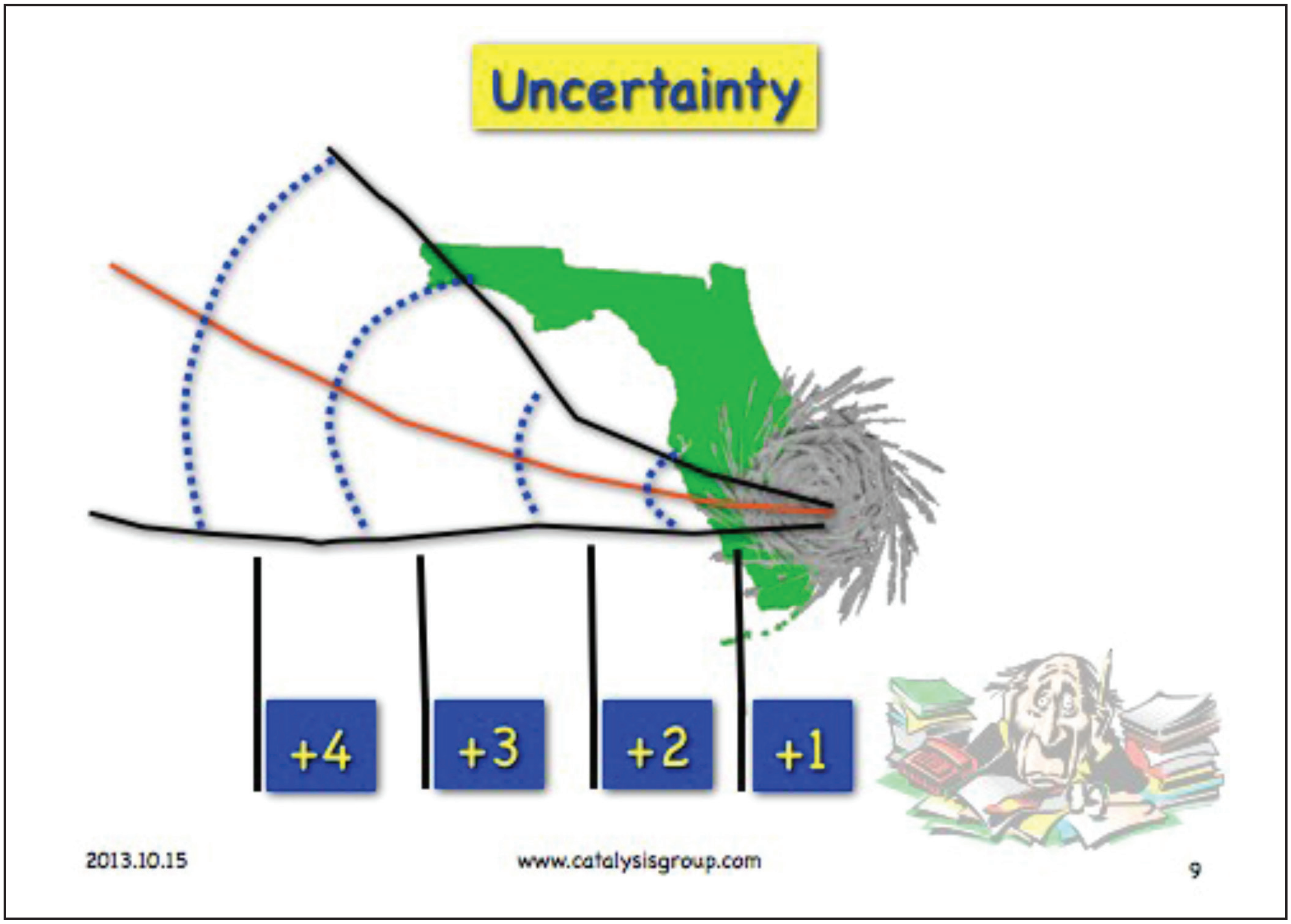
- Assumptions are place-holders for uncertain things
- The future is uncertain
- There are NO facts about the future
- Assumptions essential to estimation
- Easy to mistake assumptions for truths

Presenting Estimates

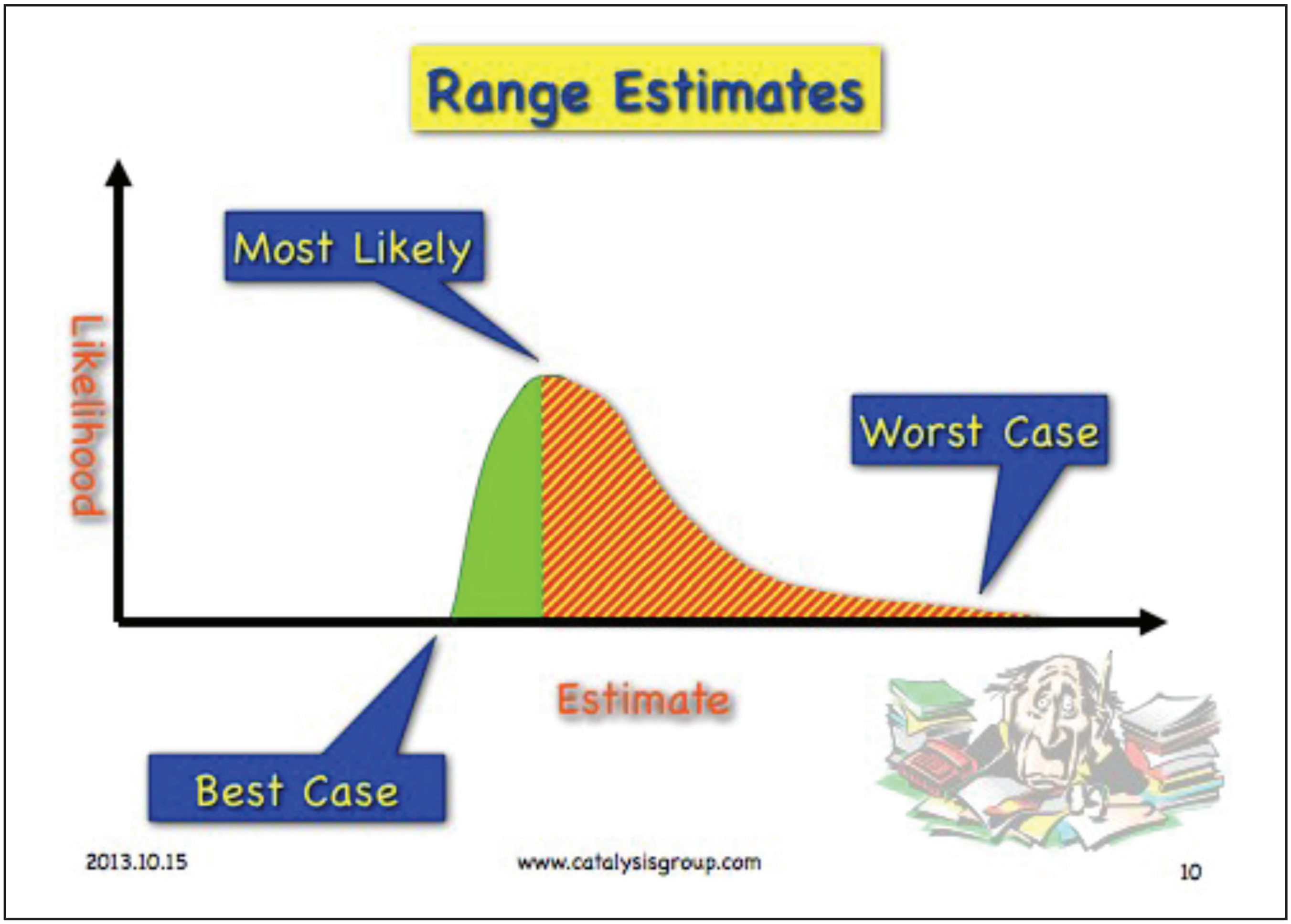




- Best estimate for now
- What could improve
- When we'll be smarter



Uncertainty



Range Estimates

Practice: How Many… ?

1. ID Relevant factors
2. Gather data/make assumptions
3. Consult
4. Do Math (Extra credit: Range)

Wrap

- Questions?
- Thank you!

Payson Hall
Catalysis Group, Inc.
payson@catalysisgroup.com

Agile Across Cultures: When National Culture Trumps Strategy

Valerie Berset-Price, Consultant

The search for global talent coupled with cost reduction through outsourcing is nowadays perceived by most multinationals as necessary to remain competitive. As a result, software engineering teams tend to be culturally diverse and spread around multiple time zones. In this environment, Agile is often considered the solution to achieve transparency, maneuverability and effective communication within the team. While Agile is a powerful project management method, it focuses on corporate culture while downplaying the national culture that team members bring to the equation. Moreover, Agile pushes a philosophy rooted in the linear culture defining U.S. nationals, assuming that everybody on the team has been brought up to think, solve and act in a linear manner.

In this session, we will look at the tenets of the Agile principles, such as team ownership, self-organization, trust and respect, macro-management, easy-access communication, prioritization and continuous realignment. We will examine how those concepts are inherently interpreted in multi-active and reactive cultures, placing an emphasis on how the cultures of India, the Philippines and Brazil compare with the United States. Armed with that knowledge, we will look at the essential tools multicultural distributed teams need to excel within the Agile framework.

With more than 20 years of international business development experience as a dual citizen of Switzerland and the United States, ***Valerie Berset-Price*** *is an experienced consultant, cross-cultural troubleshooter, international speaker and trainer. With an extensive entrepreneurial background in international business development, she offers practical insights about the tools necessary to achieve meaningful and profitable results. During the last 15 years, Valerie has worked for Swiss, Taiwanese, South African, Brazilian, and U.S. companies providing training that brings the business world into focus, bridging cultures to succeed in today's world marketplace. Valerie is a graduate of the Swiss International Business School and the Monterey Institute of International Studies. She has presented her training program to the World Customs Organization and at the U.S. Embassy of Santiago, Chile. She has lectured at numerous universities and is an international business expert for the Huffington Post.*

QA Role in Scrum

Leveraging Agile for Defect Prevention

Karen Ascheim Wysopal

karen@karenwysopal.com

Abstract

The key to a successful adoption of any software development methodology is a clear understanding of the roles and responsibilities of each team member within that framework. As Agile continues its rapid adoption in organizations across the globe, it's essential to define the role of QA in Scrum as concretely as we've defined the other Scrum Team roles.

During our team's recent Agile transition, I was approached from all corners of the organization, with questions like, "What does QA do every day in the sprint? What are their responsibilities beyond testing and reporting defects? What should they do in the early part of the sprint, when there's no software to test yet?" Despite being a recent "graduate" of several Agile methodology courses, I found myself ill-equipped to answer these questions. There isn't much information in the trainings or on the web about the role of QA in the Scrum team. What little I have found is primarily anecdotal, rarely specific, and never comprehensive. While Product Owner and Scrum Master roles are very well defined, QA Engineers fall under the generic label, "Team Member". We were on our own to define the role of QA in Scrum.

What we quickly learned delighted us: Agile methodology, with its fundamental shift away from waterfall software development, gives QA an opportunity for broader and deeper involvement in the overall software development lifecycle. This enables us better to ensure quality -- not by finding and reporting defects -- but by preventing the introduction of defects in the first place.

This paper provides a deep dive into the specific, day to day role of QA in Scrum, as our team developed it through the course of our yearlong transition to Agile. You will learn specific responsibilities and activities you can bring to your Scrum teams to transcend the traditional test-and-report, after-the-fact approach to quality assurance. And you will learn to leverage the unique components of Agile to significantly expand your scope of influence, and measurably improve the quality of your software. If you're not currently working in Agile, you will find these concepts relevant and applicable to other methodologies as well.

Biography

Karen Ascheim Wysopal is a Section Manager in Software Quality at Hewlett-Packard. She currently leads the Quality Assurance program for HPConnected.com, ePrintcenter.com, and related HP web-connected printer technologies. With over 20 years in the software QA industry, Karen's professional *passions are building high functioning innovative QA teams from the ground up, defining new processes, and expanding QA focus on the user experience.* She can't seem to stop breaking software *.*

1. Introduction

Traditional thinking about the role of QA has been *defect-oriented*: QA Engineers test, find, and report defects.

Agile methodology gives QA engineers an opportunity for broader and deeper involvement in the overall software development lifecycle. This enables us better to ensure quality -- not by finding and reporting defects -- but by *preventing the introduction of defects into the code in the first place*.

What is it about Agile that makes this possible? How does QA's role in the Agile Scrum Team enable us to release higher quality software?

2. Terminology

While this is paper does not set out to define Agile methodology, practitioners use its terms differently, so it's important to establish an understanding of the terminology used here.

Scrum Team refers to a group of people working together, with specific roles as defined by Scrum methodology. Scrum Team Roles:

- Product Owner (PO)
- Scrum Master (SM)
- Engineers (QA and Development)

Sprint refers to the time interval which the team has to complete the agreed upon – *committed* – set of work, after which the completed work is deployed to production. Generally sprints run at 2-4 week intervals. This discussion is based on a 2-week sprint cadence.

User Story is the Agile term for Requirements. And the *Product Backlog* is the set of User Stories yet to be implemented.

3. QA Responsibilities during the Sprint

The first days of the sprint emphasize user story analysis, sprint planning, and a lot of coding. New implementations may not be test-ready for several days. What does QA do during the first part of the sprint, when there's nothing yet to test? The answer is, a lot. Let's review the numerous responsibilities QA Engineers may have as part of overall quality assurance (regardless of methodology.)

Here are the responsibilities that our QA engineers are involved in throughout our sprints, starting from Day 1:

Daily Standup

Sprint Planning

Backlog Grooming

Test Plan Prep

Submit Infrastructure Requests

Cross-Team Meetings

Defect Triage

Backlog Grooming Prep

Test Plan Create/Execute

Exploratory Testing

Inform SM of Blockers

Story Acceptance Testing

Defect Disposition Exercise

Retrospective Prep

Backlog Grooming

Review Branch Merges

Sprint Demo

Sprint Retrospective

Some activities, such as Defect Triage, are tasks which must be attended to on a daily basis to keep the project on track. Others are one-time events, or occur periodically at key intervals along the sprint timeline, for example Backlog Grooming. In our team's 2-week sprint timeline, our QA tasks are scheduled something like this (each team makes adjustments to best suit their needs):

Week 1				
Day 1	Day 2	Day 3	Day 4	Day 5
Daily Standup ***Sprint Planning*** ***Test Plan Prep*** ***Submit Infra Requests*** ***Set Cross-Team Meetings*** ***Defect Triage***	*Daily Standup* *Cross Team Meetings* *Test Plan Prep/Create* *Defect Triage*	*Daily Standup* *Inform SM of Test Blockers* *Test Plan Prep/Create/Execute* *Backlog Grooming Prep* *Defect Triage*	*Daily Standup* *Backlog Grooming* *Test Plan Create/Execute* *Inform SM of Acceptance Blockers* *Defect Triage*	*Daily Standup* *Test Plan Create/Execute + Exploratory Testing* *Inform SM of Test Blockers* *Cross-Team Check-ins* *Defect Triage*
Week 2				
Day 6	Day 7	Day 8	Day9	Day 10
Daily Standup *Test Plan Create/Execute + Exploratory Testing* ***Inform SM of Blockers*** ***Defect Triage*** ***Defect Disposition Exercise***	*Daily Standup* *Test Plan Execute* *Cross Team Meetings* *Inform SM of Acceptance Blockers* *Defect Triage*	*Daily Standup* *Test Plan Execute* *Final Defect Disposition* *Defect Triage*	*Daily Standup* *Retrospective Prep* *Backlog Grooming* *Review Branch Merges* *Defect Triage*	*Daily Standup* *Defect Disposition Cleanup* *Sprint Demo* *Sprint Retrospective* *Defect Triage*

Figure 1: QA responsibilities across a 2-week sprint timeline

Figure 1 illustrates QA responsibilities each day throughout the sprint. It's important for the entire team to be familiar with these tasks, because many of them involve other team members engaging with QA.

Many of these tasks and activities are familiar to QA engineers coming from traditional methodologies, such as reviewing requirements, writing test plans, testing and logging defects. Others may be less familiar. Earlier we asked the question, “How does QA’s role in the Agile Scrum Team enable us to release higher quality software?” We’re going to explore that in detail. But before we do, let’s take a moment to review some well-known QA activities not unique to any methodology. I refer to these as “the usual stuff”.

3.1 The Usual Stuff

- Test plan creation
- Test execution
- Logging Defects

These are familiar to QA engineers within any methodology. And of course they’re cornerstones of the work in Scrum as well. They absolutely have their place throughout the sprint schedule.

But this is stuff we know well. So let’s look at what’s unique to the QA Role in Scrum that takes us beyond “the usual stuff.”

3.2 Backlog Grooming: Quality Starts with the User Story

The process of reviewing and refining user stories to prepare them for commitment in a sprint is called Backlog Grooming. Two to four backlog grooming sessions should be planned for during the course of the sprint, so that stories have been fully defined (“groomed”) and are ready to be committed in the next sprint.

Scrum training discusses story writing and backlog grooming in depth, defining this as a Product Owner task, perhaps with some input from developers. So what is QA’s role in backlog grooming? What we discovered is that QA’s input is essential.

Backlog grooming is our first opportunity to prevent defects from being introduced into the code in the first place. In other methodologies, QA rarely has input into requirements definition. And we often have limited interaction with developers during implementation. The approach of, “throwing code over the fence to QA” makes it impossible for us to prevent the introduction of defects. We can only find and report on them after the fact.

In Agile, we have the opportunity to ask key questions during requirements definition. And we can ask technical questions of developers before they start implementing. This results in more thoroughly defined user stories. QA’s questions often prompt developers to consider issues they hadn’t previously thought about. They can resolve these with the first check-in rather than waiting for the question to be asked in the form of a defect report. This is prevention vs. detection in action.

3.2.1 The Importance of Acceptance Criteria

Acceptance Criteria are used to determine if the completed user story meets the team’s “Definition of Done”. A story isn’t complete without them. While this isn’t a presentation on how to write good user stories, it’s important to discuss the significance of well-defined acceptance criteria.

It’s not enough to have criteria like, “Works as designed”, or “Works like it did before the change”. Acceptance criteria should be specific to what’s being implemented, and speak to the customer experience and value proposition. They should include system integration expectations where applicable, and negative use case handling. Well defined acceptance criteria ensure quality both by driving good testing, and by enabling accurate story sizing.

3.3 Sprint Planning for Success

3.3.1 Don't Forget the Defects

During sprint planning, the scrum team reviews the product backlog and decides which work can be committed to the sprint. But there's more work to plan for than just implementing user stories. We must also plan time for defect fixes. Our defect database contains defects found (but not fixed) during previous sprints. Like the product backlog, the defect database is another "backlog" which must be reviewed during sprint planning. The team must decide which known defects it can commit to fixing during the sprint.

3.3.2 Planning for the Unknown

In addition to known defects, we must also allocate time for fixing defects which are *not yet known* -- that is, the defects which will be found during the upcoming sprint. These defects may be related to the stories under development in the current sprint. Or they may be found during system integration or other testing which could only take place after an earlier sprint had been completed. In any case, each new defect must be evaluated during our daily triage. Some will likely need to be fixed as part of the current sprint. Neglecting to allocate time to fix unknown defects makes the sprint plan inaccurate.

3.3.3 Sizing: Story Points = Dev Time + Test Time

"Story Points" are units of measure used in Agile to reflect the amount of work needed to complete a user story. Agile refers to the estimation exercise as "story sizing". When teams do story sizing, they often forget to include *test time* in their overall estimation. (This is not unique to Agile.) More often overlooked is the *test-fix-retest cycle* that's necessary when a critical defect is found that prevents the story from being accepted. For our sprints to be successful, it's critical to include these in our sizing. Here's a guiding principle to ensure the best accuracy in story sizing:

Story Points = Dev Time + Test Time

Some engineering teams resist including test time in their story sizing. In our teams, once we added test time to our story sizing we saw a dramatic increase in the percentage of stories completed in a sprint.

3.3.4 Timing

When planning a sprint, avoid sprint-end bottleneck! Remember that the last step in accepting a finished story or defect fix is QA verification. If stories aren't complete until the end of the sprint, QA won't have time to do all the verification testing. When committing stories to the sprint, be sure to select stories which will be ready for QA verification throughout the sprint timeline, rather than all at the end.

3.3.5 All User Stories are Not Created Equal

Part of story sizing is recognizing that all stories are not created equal. When considering the lifecycle of a User Story, we tend to think Develop->Test->Accept -- a simplistic model that looks something like this:

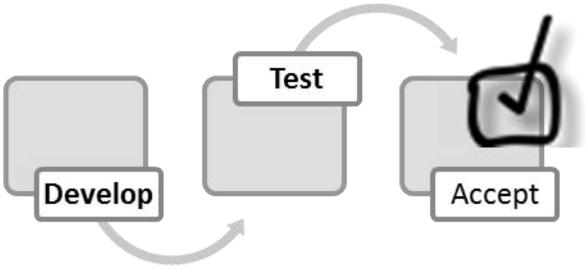


Figure 2: Simplistic model of user story lifecycle

How does this simplistic vision of story lifecycle impact our ability to plan our sprint realistically? Applying this model to multiple user stories in the context of the Sprint Timeline, our sprint plan might look something like this:

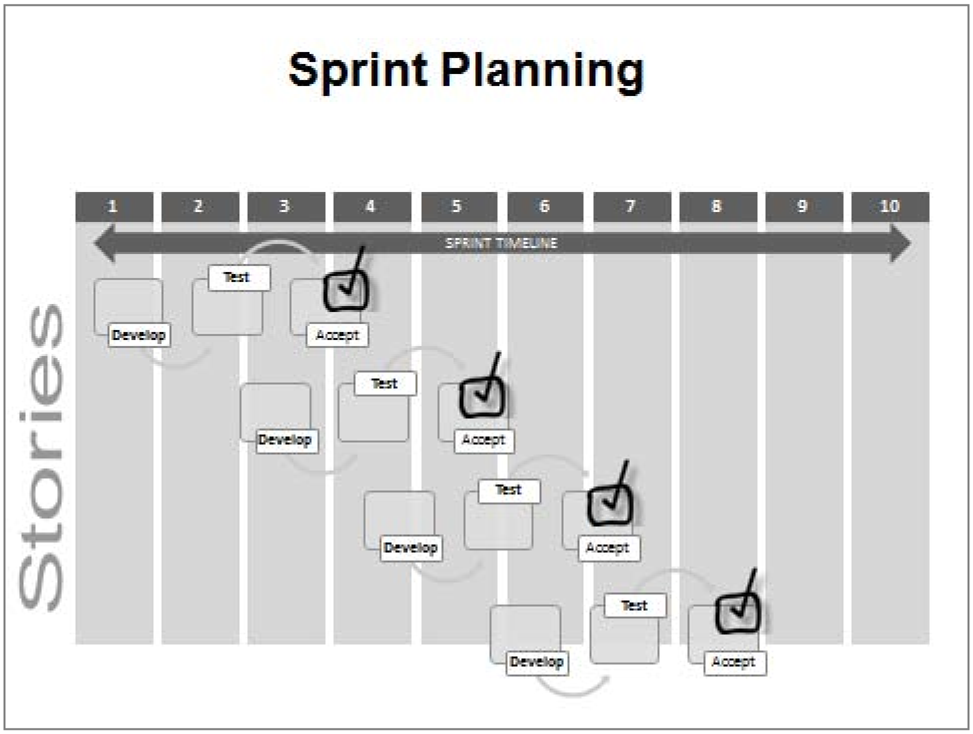


Figure 3: Simplistic model of user story lifecycle on a 2-week sprint timeline

Unfortunately, this straightforward model isn't a realistic representation of a sprint plan. What are some problems with this sprint plan?

- Dev and test times are represented as equal and predictable
- One task doesn't start until the next one is finished
- Possible test failure is not accounted for – no time is allocated for the unexpected.

Expecting our units of work to fall into this simplistic model isn't realistic, and sets us up for failure.

Instead, let's consider some more realistic examples of user story lifecycles:

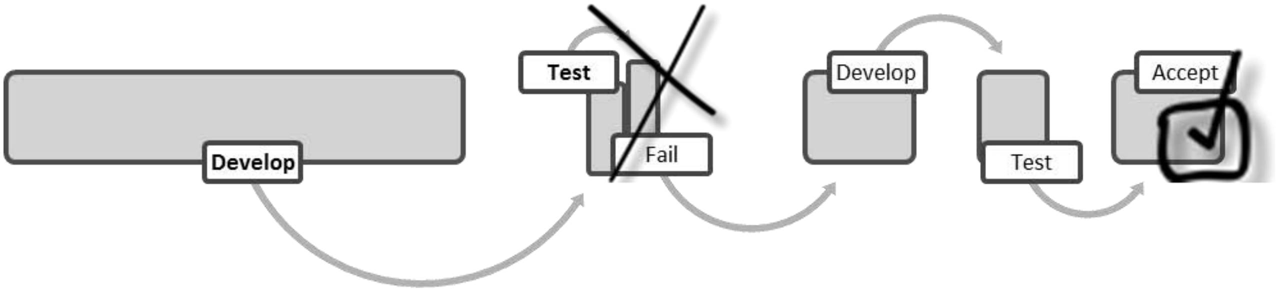


Figure 4: Example story lifecycle: long development time, short test time.

In this example, we have a story in which development times are relatively long, and test times are relatively short. This might be the case when a large refactoring is called for, which takes significant development time. Test time would be relatively short, however, because no additional work is required on the existing test automation needed to test it.

Notice also in this example, we've allowed for the possibility of acceptance test failure, and the need to rework and retest the code.

Another example illustrates the opposite condition: a story in which development times are relatively short, but test times are relatively long.

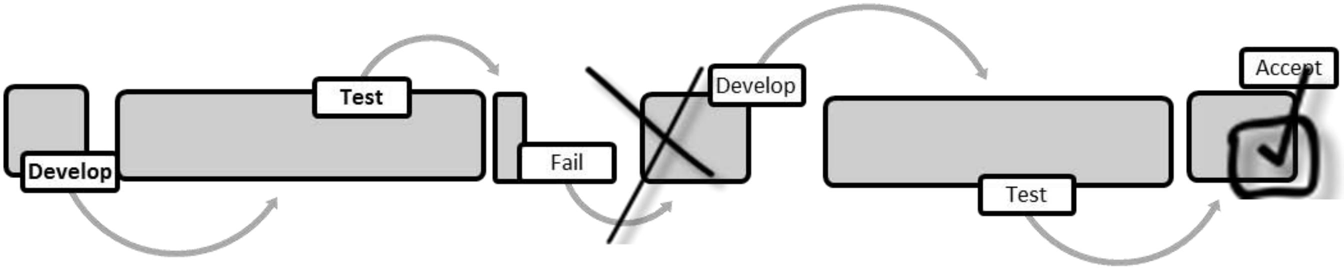


Figure 5: Example story lifecycle: short development time, long test time.

3.3.6 Putting it all Together

Now that we understand that Story Points = Dev Time + Test Time, and that not all stories are created equal, putting all the units of work (user stories and defect fixes) together into the sprint plan, the reality looks more like this:

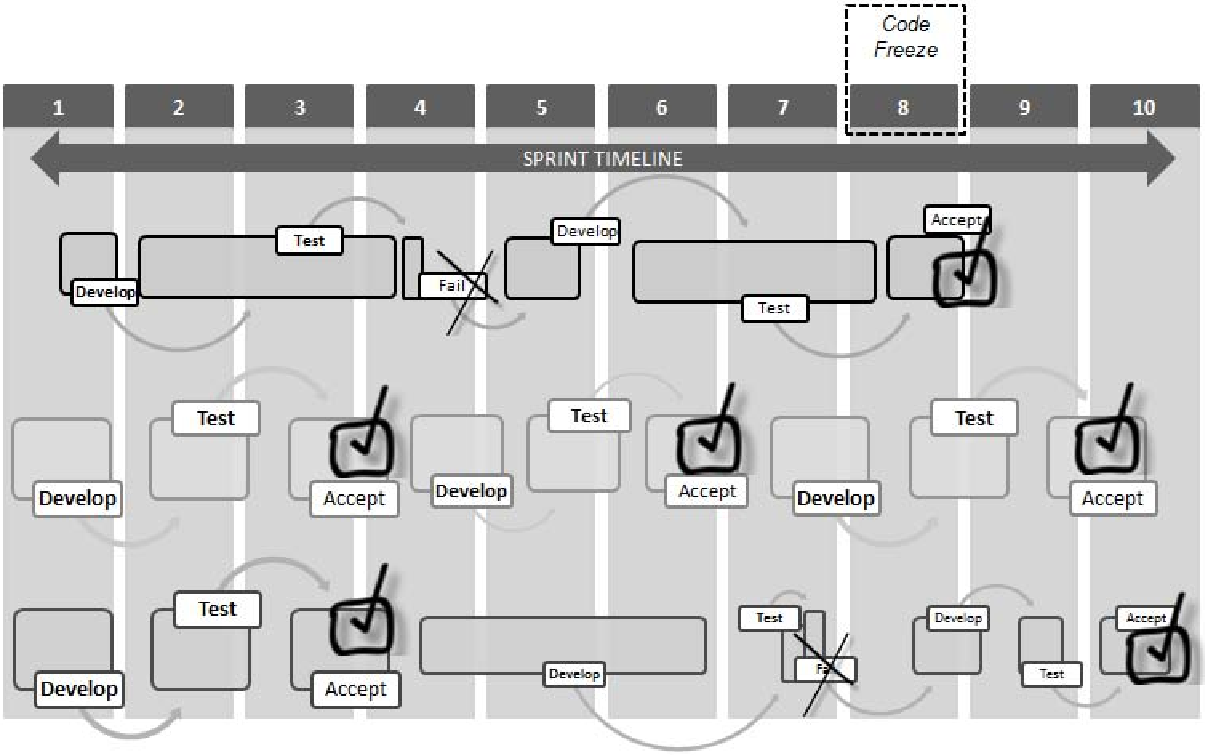


Figure 6: Planning stories in the sprint

In this example we see some effective planning, and some unknowns accounted for:

- Stories are represented as having varying lifecycles, some demanding more or less dev and test times than others
- Time is allocated for potential test failure, requiring more dev and test time

But what are some of the planning considerations we've discussed which are not accounted for in this sprint plan?

- Expectation for first testable code comes too early in the sprint
- Too much code delivery at the end of the sprint – creates a bottleneck in QA
- No time allocated for necessary sprint wrap-up activities including code freeze, release branch check-in, release notes creation, final defect disposition, sprint retrospective and demo.

How do we fit everything in? A sprint plan which is set up for success will look more like this:

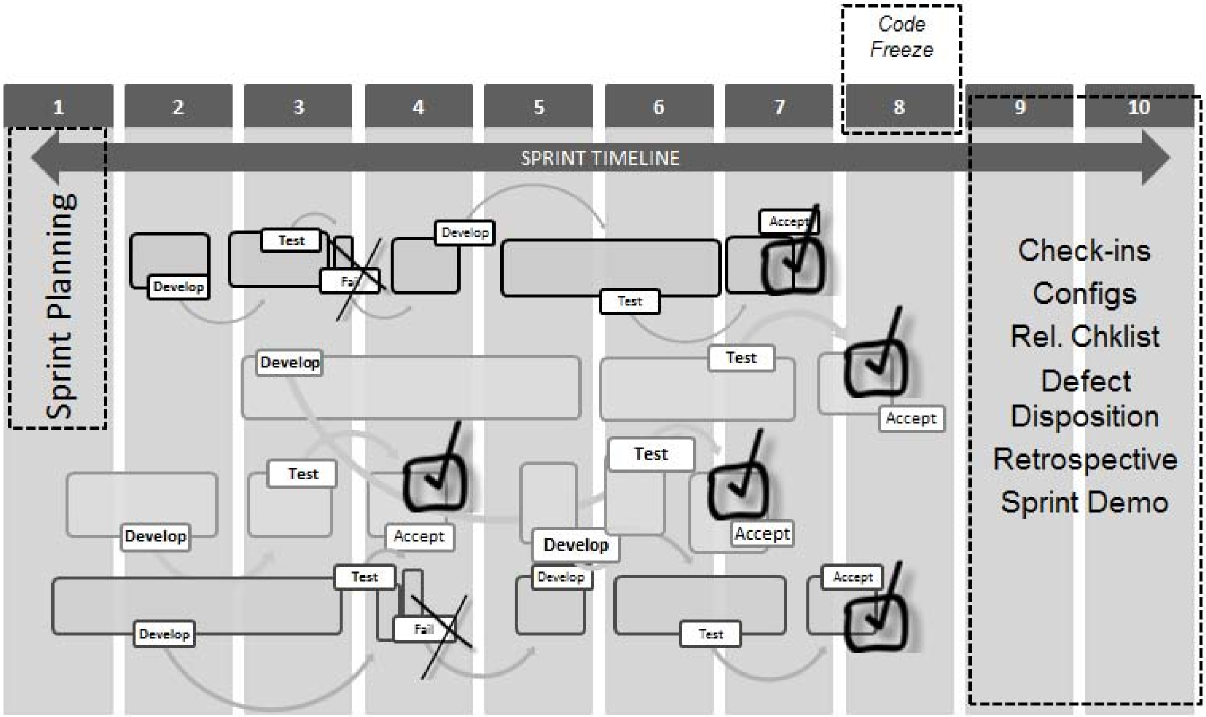


Figure 7: Sprint planning for success

This example allocates time on Day 1 for Sprint Planning. It's key that the entire team take time together to create the sprint plan. This plan looks complicated, but it works, because it accounts for all work needed to be successful, not just development and test.

This plan has one less story than the last. But it's well positioned to deliver 100% of the work committed. The previous example was not. The more granular your stories are, generally the more you can commit to in a sprint. Here we're less concerned with quantity; the emphasis is delivering quality, and completing 100% of the work committed. Here's why this plan is positioned for success:

- Time is allocated on Day 1 for sprint planning
- Test-ready code is not expected until Day 3, some later
- Story sizing is carefully planned, and includes both dev and test time
- Time is allocated for acceptance testing before Code Freeze
- Time is allocated for sprint-wrap up work

3.4 What about System Integration Testing?

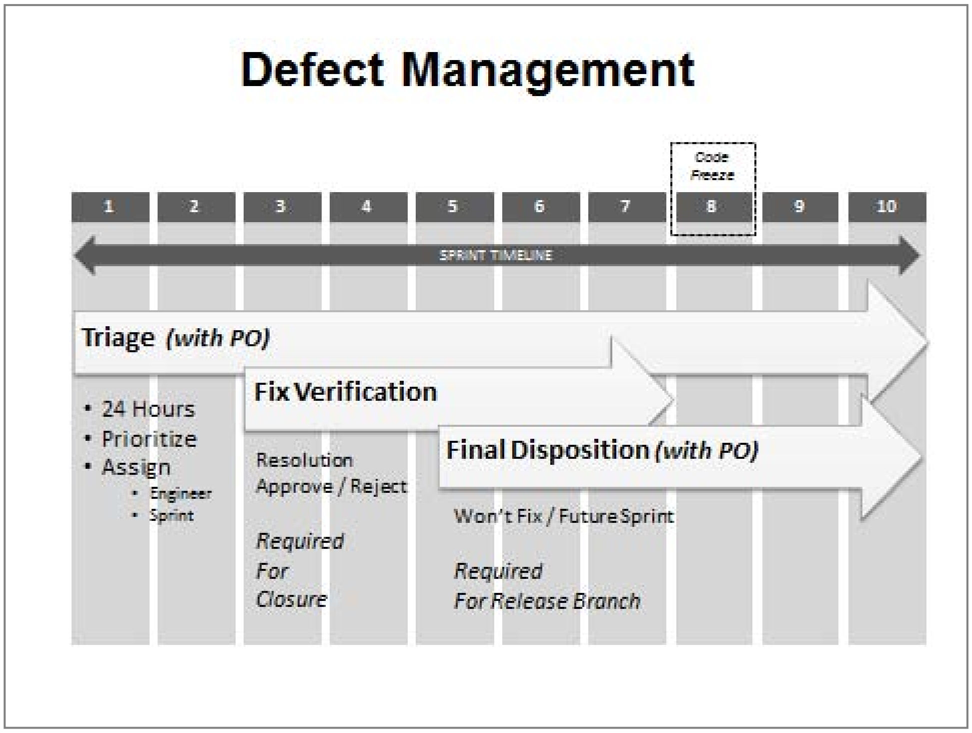
Thus far, we've been talking about sprint planning in the context of only one Scrum Team. But what about scaled Agile, where multiple Scrum Teams deliver code into the same code base? Many of our user stories have dependencies and integration points with stories being developed by another team.

Within the scrum we focus our testing on the specific stories and defect fixes being worked on by our team. This work has a tendency to become “silo’d”: our intense focus on the work of our own team results in loss of visibility to what other teams are working on. How do we prevent testing gaps across the full system integration? We’ve addressed this by setting aside time for **Cross-Team Test Planning**.

To close integration testing gaps, QA leads from different scrum teams need regular communication. They need to understand stories being worked on in other teams, so that together they can plan testing which ensures coverage of all integration points. This also ensures dependencies are accounted for in acceptance criteria and test plans. These meetings should occur weekly. Tuesday of each week works well for our team. By the first Tuesday of the sprint, sprint planning will have been completed. Testers will have reviewed all the stories and are prepared to share integration and dependency information with other teams. By the second Tuesday, having completed some work, testers now should have increased insights to share with each other.

3.5 Defect Management

There are three critical steps in defect lifecycle which need to be managed throughout the sprint: Triage, Fix Verification, and Final Disposition.



Defect Management

Figure 8: Defect management tasks along the sprint timeline

3.5.1 Triage

Defect triage involves reviewing newly submitting defects, verifying the defect is reproducible, determining priority, target sprint, and development ownership. In our team, Product Owners are responsible for prioritizing units of work. This includes prioritization of user stories as well as defect fixes. Product Owners and QA perform triage together. In a two-week sprint cycle, defect triage must happen daily. Many teams I’ve worked with balk at the idea of a daily triage. They feel it’s too time consuming. But consider the alternative. If at sprint’s end we find that there’s an outstanding defect that must not go to production, we have to make a choice: either disrupt the release cycle by fixing urgently, rebuilding, and redeploying the release branch, or don't merge this code into the release branch at all. We hope never to be faced with this choice at the end of the sprint, but it does happen. The way to minimize this is by holding daily defect triage sessions so that everyone is keenly aware of all the existing defects throughout the sprint cycle.

3.5.2 Fix Verification

As soon as a defect is resolved by a developer, QA is responsible for verifying the resolution, whether it's a fix, or another resolution such as "can't reproduce", "not a defect", "will never fix", etc. As with story acceptance testing, it's critical that QA provide quick turnaround on defect fix verification. Developers need to learn as soon as possible that a defect fix didn't really work, or that the issue they thought was invalid really is a defect. Receiving the information when the issue is still fresh in the developer's mind ensures a faster turnaround in the event that more work is needed. QA's approval of the final resolution is required before the defect can be closed.

3.5.3 Final Disposition

In our organization, the term "disposition" is a verb. To "disposition" a defect is to decide when it will be fixed, or that it will not be fixed at all. One of our criteria for code merging into the release branch is that there are no outstanding defects assigned to the current sprint. Earlier we talked about the two kinds of defects to consider during sprint planning: known, and unknown. As the sprint comes to a close, there will be a number of defects falling into that second category: they were unknown – didn't exist – at the start of the sprint. These may have been reported near the end of the sprint, in which case they are likely still open and assigned to the current sprint. However, there's no time to fix them in the current sprint. Final disposition involves reviewing all defects which are still open at the end of the sprint, and making sure each is a) acceptable to be released, and b) targeted to be fixed in a future release. Without final disposition, open defects remain assigned to the completed sprint, and get overlooked as we move to the next sprint.

3.6 Story Acceptance

Story Acceptance is the process of determining whether a user story meets the Definition of Done. Our team's Definition of Done is this:

1. Meets Acceptance Criteria
2. Passes Acceptance Tests
    - This could be viewed as part of #1. We found the need to explicitly call it out so that QA would be an active participant in the story sign off process.
3. No outstanding Priority 1 or Priority 2 defects against the story.
    - Why allow for *any* outstanding defects against the story? Our testers find all kinds of defects, many of which, while relevant, don't warrant rejecting the story. Examples include minor cosmetic defects unique to non-priority browser versions, or minor issues with code which won't be visible to the user yet but is needed by a dependent team for further work in the next sprint.

QA should be prepared to perform story acceptance testing as soon as a story is complete. Just as with defect resolution verification, developers need immediate feedback as to whether or not their story is accepted. This enables them to either address any outstanding issues immediately, or turn their full attention to the next backlog item.

3.7 Sprint Wrap-Up

We talked earlier about the need for QA to have a strong voice in “non-traditional” aspects of their role, such as backlog grooming. Our voice is also needed in sprint wrap-up activities.

3.7.1 Sprint Retrospective

The sprint retrospective is an opportunity for each team member to voice ideas for improvement. QA should prepare feedback for the team about what went well, what didn’t go well, and what can be done better in the next sprint to further the goal of preventing, rather than detecting defects.

3.7.2 Sprint Demo

The sprint demo is the time for the team to show off the great work they’ve accomplished during the sprint. Generally this takes the form of a software demo, or maybe even a brief code review highlighting a sophisticated solution to a complex implementation. Development accomplishments are easy to show because they’re ready for production release. If QA has done its job well, which we assume it has, the software performs brilliantly and the demo goes off without a hitch. QA tends to remain the quiet observer. While the audience is wowed by the software being shown, they don’t think to say, “Hey, no bugs! Great work, QA!” Rather, they just expect it to work. So what does QA have to offer in a sprint demo? We think it’s important to use the sprint demo to showcase QA’s contribution. We do this by allotting time for QA to either demonstrate or talk through a testing accomplishment. Some team members have demonstrated test automation in action. Others have presented a brief automation code review, demonstrating innovation which makes the automation faster, or more extensible. Still others have talked through a strategy for test coverage of a critical new feature. This raises everyone’s awareness of the important role QA plays in the scrum team. And it increases confidence in the quality of what’s about to be released.

4. This Really Works!

By implementing the core concepts and processes described here, our team has delivered measurable quality improvements consistently over multiple releases. Having implemented just some these processes, in less than six months we successfully transitioned from quarterly to 2-week release cycles. We reduced overall defect count and deploy time, and achieved a 95% reduction in production hotfixes. Everyone finally stopped working weekends!

5. Conclusion

As the transition to Agile continues its rapid growth across the globe, it’s essential to fully define and understand the role of QA in Scrum. Software organizations depend on us for success.

I hope that this paper has provided you with insights, information, and ideas that you can take back to your own scrum teams to add to your continuous improvement process. Share with your teammates the important voice QA must have in sizing units of work during sprint planning, defining acceptance criteria, managing the defect lifecycle, communicating dependencies across teams, and actively contributing to all scrum ceremonies.

Although Agile has been in full swing for over a decade, there still remains an exciting opportunity for us to further the success or our organizations by bringing clarity to the role of QA within the Scrum Team.

Colocation: A Case Study in the Rewards and Perils

Mary Panza

Parametric Portfolio Associates

mpanza@paraport.com

Abstract

Some Agile software teams struggle to colocate their members so the software developers and testers can be near each other at work. The Parametric development team members had the luxury of sitting together in the IT department from the beginning, but sought even greater flexibility and efficiencies. We took the bold step of colocating some development teams with the end-users of the software they write. We had lofty goals for the experiment and some specific expectations, but ended up with some unexpected results, as well.

This paper serves as a case study and a retrospective on our effort. It discusses our reasons for moving the teams, as well as the lessons we learned and the changes made to our process. The colocation experiment caused our development organization re-examine its best practices and processes. We would like to share our findings with other teams and organizations.

Biography

***Mary Panza** is the scrum master at Parametric Portfolio Associates in Seattle, Washington. She has supported, tested and managed software for the past twenty years for various companies around the Puget Sound area. Mary's passion has always been for problem-solving and process improvement, as well as seeing the human side of software development.*

The views and opinions expressed in this article are those of the author and do not necessarily reflect the policies or positions of Parametric Portfolio Associates LLC.

1. Introduction

Parametric Portfolio Associates LLC (“Parametric”) is a registered investment adviser focused on the delivery of engineered portfolio solutions. Technology is leveraged to deliver a scientific and pragmatic approach to a rules-based, but dynamic investment approach. Parametric maintains a small development team of approximately twenty people that write, test and support custom software for in-house business users. This environment affords flexibility not commonly available to development teams in most standard software companies. The development team’s mission is to build proprietary software to carry out Parametric’s business strategy with the most efficiency and least risk possible. How the team meets these goals is unrestricted by company management. Several years ago, we, the development team, realized the method we used to develop software – in small increments with early user input and immediate feedback – already had an industry name: Agile. Rather than attempt to “reinvent the wheel”, we set out to learn more about Agile Software Development and began to apply Agile tenets to our own process within the team.

The Agile Manifesto speaks about “individuals and interactions”, and the principals behind it weigh open and easy communication heavily as a factor for success (Cunningham, 2001). Collaboration through face-to-face conversation is an ideal goal for sharing information among members of the software development team (the “delivery team”). It is also just as important for interactions between the delivery team and the actual end users of the software. Often the end users are represented by a single individual seen as the key stakeholder on a project (the “product owner”) to simplify the communication routes, but the goal remains the same: the people creating the software and the people using the software should talk openly and freely about the software. Colocation is seen by many Agile practitioners as the best way to achieve this collaboration. At a basic level, the members of the delivery team should sit close together to reduce the physical impediments of sharing information among the members and to enhance the willingness to share by building the team’s sense of community. (Cockburn, 2001) Having the product owner and delivery team physically located together, or at least in close proximity, increases the collaboration further. (Krumins-Beens, 2012)

At Parametric, the product owners are members of functional business units and have regular responsibilities to perform for their departments; they act as product owners in addition to their standard duties. This arrangement inhibited moving the product owners away from their business team to sit with the delivery team assigned to their business unit. Instead, we decided to move some delivery teams closer to their respective product owners. This paper serves as a retrospective on the effort to colocate some of the delivery teams with their product owners and business users. We had a few specific goals and expectations going into the change, and we received a few surprises along the way. It is our hope that others can benefit from our experience.

1.1. General description and overview of our original process

The overall development team is comprised of smaller teams assigned to work with specific business units; for simplicity’s sake, these delivery teams can be referred to as Team A, Team B, Team C and Team D. Originally, the entire development team sat together in the same physical space in an open building design. Application Support, DevOps, and Development Management were also located in this large group. Each delivery team had its own pod of desks to facilitate an easy flow of communication all day long. The desks and computer set up allowed for two or three members to work together easily at the same time, an Agile practice called “pairing”. White board space was abundant, and each team had a physical task board and calendar that were visible to all. All the teams were using some form of Scrum or Kanban to manage their work and had a quick daily meeting that included the product owner, whose desk was located elsewhere in the building with their functional business unit. The individual teams met regularly with their product owners to review completed work and plan upcoming projects. The frequency and formality of such meetings was determined by working agreements between the delivery team and the product owner

1.2. Initial Perceptions: Strengths and Weaknesses

The working proximity of the delivery team members enabled them to eavesdrop on each other. If a software developer needed a code review, it was easy to find someone. If a tester couldn't figure out why tests were failing or the build was breaking, help was close by. Questions about why a feature was implemented in a particular manner could simply be asked out loud to the general group; chances were someone could explain in detail. When a question came up about a business area, it was easy to find someone who could answer the query or suggest an appropriate resource. The open atmosphere created a collective consciousness around the internally produced software, which benefitted not only the delivery teams, but the support teams, as well. Together, the overall development team had a shared understanding of the domain knowledge and technical skills needed to complete projects.

This environment also fostered strong team-building opportunities. A delivery team could easily solicit and receive a design review from another member of the development department who had directly related experience; for example a developer who had written code for that application in the past or a support technician familiar with a targeted bug. Information and experience flowed between delivery teams seamlessly. The team members shared ideas through informal conversations, collaborated over coffee and worked out design issues during lunch.

The biggest perceived weakness in the original environment was the physical separation of delivery teams from the product owners and business users. In order to write software that serves specialized business needs, the delivery team must understand how the end users accomplish their tasks and what problems they face. In the initial arrangement, the delivery teams had limited daily interactions with the product owner via the daily meeting, as well as time-boxed interactions regularly through project planning. Additionally, some team members would infrequently shadow a business user to watch them work, but this did not provide enough information. This left the delivery teams with a potentially limited understanding of how their users really completed their work. When writing highly customized software, this arrangement proved to be problematic. In the year prior to the colocation exercise, 60% of hotfix releases could be tied to a lack of understanding of the business process. A misunderstood requirement or an incomplete test case resulted in the incorrect implementation or a bug. Almost one third of all feature release required a follow up release. And these numbers do not show the cost of time spent in extended rounds of user-acceptance testing where these problems were caught prior to release.

2. Colocation changes

Since the Product Owners have positions in their business units as functional team members, i.e., their "regular job", they didn't have abundant time to train, demonstrate or explain the daily processes and procedures. Realistically, the delivery teams required this critical business information in order to produce quality software. The developers and testers needed a way to gain this knowledge and spend less effort doing so. We hypothesized that since information and experience flowed easily between development team members simply because they sat in close proximity to each other, perhaps the same thing would happen if we located a delivery team to sit with their product owner and business users. The arrangement we hoped to create, where information could be exchanged without much interruption to either the delivery team or the business users, was termed "Osmotic Communication" by Alistair Cockburn (Cockburn A. , 2004). He described an environment where the cost of communication was low, yet provided team members with a rich information source. If a delivery team could glean information about business processes and problems simply by hearing about it in the background, then moving them to sit near their respective business unit could bridge the knowledge gap.

Over the past year, we moved three different delivery teams to sit with their respective product owner and business users. The following section covers our decision to move the teams and what we expected would happen.

2.1. Why – The Goals

Our goals were straightforward:

1. Deepen each delivery team's understanding of the business processes and problems typically found in the area for which they were writing software.
2. Improve the relationship between each delivery team and their respective product owner and business users.
3. Leverage that relationship to inspire new creative solutions to the current and future business needs.

We theorized the delivery teams would be more efficient in generating quality technology solutions if they better understood what the end-users needed to accomplish in their daily tasks. We wanted to improve our own productivity and reduce our own risk as we built software that would help our business to do the same.

2.2. What we wanted to happen

When we decided to move the first team – Team B, we thought they would gain enough knowledge of the daily tasks and processes to perform those same tasks later in their own development and test environments. Thanks to the cooperative benefit we expected, developers would ask questions, watch how the software was used, and understand how the business users really work. In turn, the product owner and the rest of the business team members would get a better idea how their software was developed. Ideally, together they could bounce ideas off each other to improve the quality of the product, determine the best approach to a given challenge and quickly obtain feedback on new feature development all without scheduling extra meetings or formalized training. At that point in the transformation, they would cease being "the Delivery Team" and "the Business Unit" and become "unified group who make portfolio management work".

One concern we did have was that the relationship between the development team and the business users would become so integrated that the normal controls in place for product support issues would be bypassed informally. If that happened, it would cause the delivery team to be unduly distracted with requests from the business users. In anticipation of that event, we reminded the delivery teams and their product owners of the accepted procedures for routing production support. The business users and application support team all agreed to follow the current protocols for production issue resolutions. Additionally, for the first few months, the application support manager planned to monitor the situation closely and intervene if necessary.

And finally, we thought the collective consciousness around development knowledge, practices and procedures would endure. In short we expected to improve standards; the move was supposed to increase overall quality. After an initial period with only Team B colocated with their product owner and business users, Teams A and D were moved to sit with their respective business teams, as well. Team C stayed in the main development location.

3. The Outcome

After watching the colocated teams work for a few months, we were surprised to see that the expected changes weren’t happening as planned. The goals were not being met and more concerning, two of the delivery teams started to drift from our standard development practices. Initially we addressed the individual issues as they came up, thinking the issues were small adjustments needed along the way. Eventually, we realized a bit of a retrospective was needed to determine the root cause of the difficulties experienced. The development managers stepped back to sort out the pros and cons of colocating to determine how and when it would work in this environment. This section covers the observed results of our experiment and some analysis of our experience.

3.1. Overall results

Happily, none of the teams were barraged with direct requests to handle production support issues. All of the business units respectfully followed the directive to continue using the standard procedures of opening a support ticket first. The Application Support team would handle the request and loop in the delivery team on escalations as necessary.

From a technical aspect, spontaneous and relatively instant design reviews were common in the old scenario, but no longer available for the colocated teams. The business users are not software developers; they can't give input on the delivery team's design. The other development team members, who were located in teams elsewhere, were simply not available to critique implementation plans as the plans were being discussed. There were no formal design reviews, and the colocated delivery teams did not seek out additional points of view. Their pool of problem-solving resources and idea-generators shrunk to only their team. This impacted all of the delivery teams, even those not colocated with their business users. The colocation experience highlighted a need for a formal process, which will be discussed in the next section.

All those good habits we thought our teams had? They were not habits after all. There was no alignment with accepted practices - just peer pressure to follow procedures. Without the peer group and the constant reminders about code reviews, robust integration tests and sound deployment plans, the teams started to abandon protocols.

3.2. Team A results

Team A was well-established with a defined process that worked for them. They already had a very good relationship with their product owner and business unit and had a good grasp of the business domain. While colocating was useful, it was not responsible for a major leap forward in efficiency for the delivery team. During the initial period of being embedded with the business users, the team’s productivity and quality of work stayed the same, and then eventually improved. The team’s release cycle time went from almost 3 weeks per release down to one and half weeks per release. Their hotfix rate remained less than 20%. The only re-work required of the team after colocating with their business users involved a project that touched other business groups. The team did initially decide to discontinue some of the routines included in scrum, specifically daily meetings and release retrospectives.

Overall the product owner for Team A reports increased satisfaction with the colocation and improved efficiency for himself by having the delivery team literally within arm’s reach. Team A’s members have gained a better understanding of the ways their end users interact with other business units in the company and have used this information to influence their design and user-acceptance testing practices. When asked about the change, this is what he had to say, “Closer collaboration has made both halves of our collective team better. The business owners cannot always “speak technology”; having the developers learn to “speak business” has proved successful in reducing implementation struggles arising from communication gaps. I have been pleasantly surprised at the rate of feature delivery. There have been numerous instances where the developers built with the future in mind; they were able to implement incremental features far quicker and easier than in the past.”

3.3. Team B results

Team B consisted of development team members who had experience in the general development department at Parametric, but the individuals had not spent much time as a small team when they were moved to sit with the product owner and business users. After colocating with the business users, the communication did improve between Team B and their product owner and business users. However the knowledge transfer regarding the business processes was not as fruitful as expected. During the initial period, the team discontinued most of project processes, including daily meetings, defining discrete tasks during iteration planning and retrospectives. Their aim was to follow a very lean implementation of Kanban and felt it was feasible given the close proximity of the team members. Surprisingly, these changes resulted in longer development cycles and an increase in post-release production issues, including several that required a rollback to the previous version or immediate hotfixes. When the team first moved to sit with the business users, their release cycle time was roughly two weeks per release. By the end of the year, it crept up to more than 5 weeks per release. Over the year, one third of all their releases required a hotfix or unplanned rework of a feature.

While the relationship between the team B and their product owner and business users did improve in some areas, it was in a more casual sense. The groups don't share a common language to exchange "technical" information. The individual delivery team members possess a varying degree of understanding of the business domain and jargon. The greater the understanding, the more information the team member picked up during the immersion. For the team members with less knowledge, the business conversations going on around them were a source of noise and chaos rather than enlightenment - a distraction from their work. When that occurred, headphones went on and it didn't matter what was discussed around them; that information would not be assimilated. We misjudged the base level of business knowledge needed and the motivation required to gain that knowledge independently. In turn, this stifled the desired collaboration.

In retrospect, the very lean approach to development meant the team could not rely on specifications or detailed task-tracking to ensure all requirements were met before finishing a feature. Often acceptance-testing turned up missing criteria at the end of the cycle and required the team to rework the feature and delay the release. Sometimes a design flaw was not caught until after release resulting in the need to back out the new feature or create a critical patch. The lack of business knowledge hampered the team’s ability to be forward-thinking with regards to making design decisions that would also support future development.

3.4. Team C results

Team C remained in place, not moving to sit near their business users. Their process and performance remained steady during this time.

3.5. Team D results

Team D did not experience any benefit by being closer to their product owner and business users. The team was a newly formed team, with a majority of the members new to the company. When located with the business users, the only space available was near staff members who were also new to the company. This arrangement did not provide Team D with a useful source of expert information in the business domain. After two months of inconsistent efforts to perform maintenance updates to an existing product base, it became obvious the members were struggling to work together as a team and keep up with the accepted practices and processes while located away from the main development department. During that time, they did not meet the required standards for release quality. Their work re-introduced some bugs they had fixed in prior releases and resulted in the need to roll back the released versions. The decision was made to move Team D back to the general development location to give them support from another existing development team and the department management while they established themselves as a team.

4. Process changes and results so far

In response to these findings, we have made some significant process changes and continue to adjust as we go forward. The goals for these changes are aimed at improving the communication among the teams as well as giving management a better overview of all the projects currently in process and upcoming. By exposing the high level details of each team’s projects, we are able to take advantage once again of the knowledge contained in other teams. We have also gained more control on planning for upcoming projects. These changes applied to all teams, whether they were located with their business units or in the development area.

We also looked at the processes used by the individual delivery teams. After separating from the main development team, two delivery teams (Teams B and D) drifted into inconsistent practices; if they thought no one would notice or be impacted. (And they deemed this decision to be valid under the guise of self-organization.) The declining condition of their output - inconsistent quality and sometimes chaotic delivery schedules resulted from that decision. To that end, the management team and scrum master began to work with each team to establish processes that meet the company and development department goals, while still allowing the individual teams some ownership over their work.

4.1. Overall development team goals

The first change was a rework of our daily "Big Scrum"; the meeting consists of a representative from each delivery team, plus a representative from DevOps. It is still a time-boxed, ten minute meeting, but rather than giving a generic status for each group, high level details are presented. Reported items include general areas of the code base that are affected by current work, changes in expected release dates and the progress of the feature. This raises the level of notification among the teams and has cut down on implementation conflicts in co-owned code and dependent services. Deployment scheduling has improved, as well. When a potential conflict arises or a blocking issue is raised, the attendees can figure out the best way to address the issue or seek out additional resources to assist the resolution.

In an effort to correct the deviation from standards we observed and prevent it from occurring further, we formalized our architecture and design review processes. Rather than relying on it happening organically, we added official "steps" to our overall project plan to review architecture, test planning, infrastructure and documentation needs. The extra process may not sound particularly "Agile" to some, but the small increase in planning is showing results. We have seen a decrease in post-release production problems and repeated refactoring. Additionally, a small group of developers embarked on an exemplar project to illustrate our accepted patterns and practices. The overall team has this project as a reference to guide their design decisions and verify they are meeting the expected standards. After applying these changes, Team B's release cycle time dropped to slightly less than two weeks per release, and their hotfix rate dropped to less than 20 percent. Team D had several successful releases.

Our functional teams now meet regularly to determine and discuss best practices, explore new directions and technologies, and work out problems facing individuals in their roles on the delivery teams. For example, the Quality Assurance team gets together weekly to review their latest test efforts and generate ideas for approaching upcoming projects on their assigned team. One tester reported she "felt disconnected from the rest of the testers after moving to the business area, but the weekly meeting helps to know what everyone else is doing. What they learn and what they struggle with." Another example is an architecture team that meets regularly to discuss the big picture for our projects and make platform level decisions. If one team wants to employ a new technology, it is reviewed by the architecture group to determine whether it's a good fit overall for our project portfolio. This alleviated the need for each team to make its own third-party choices for implementing grid-handling, data-mapping and test tools.

We have also made an active effort to generate more team-building opportunities while the development team members are located in disparate physical areas. We hope to strengthen the personal relationships and contribute to the pool of knowledge by building connections and developing common interests. Some examples of this include a weekly lunch outing that any member of the development department can attend and a subscription to a technology webinar series attended by a cross-functional group interested in exploring new options for software development. After all, it is easier to seek out advice or a second point of view from a coworker with whom you feel comfortable.

4.2. Individual delivery team goals

Each delivery team met to define their own working agreements to guide projects and daily work. This activity enabled them to take ownership and be self-organizing. Their only requirement was to have a defined process and follow it. For Team A, it was a matter of capturing the process they already followed and getting formal agreement from all members of the team. For Team B, it took several iterations to come to an agreement to which everyone could commit and follow consistently. Team D continued to refine their processes even after moving back to the main development area. This effort is still a work in progress, but is starting to pay dividends.

Teams A and B had drifted away from daily standup meetings. Their rationalization was that they all sat together in close proximity to each other and the product owner, so everyone should know what the other members of the team were doing and when they needed help. By not having a daily meeting, it became apparent that the quick meeting served a bigger purpose than to simply inform. It served to reinforce the agreement between team members to finish the tasks to which they committed by the next meeting. We reinstituted the daily meeting as "little scrum" as a requirement for each team. The product owners had a clearer picture of work progress and could make better decisions to adjust the scope and target release date for the current project. The team members checked each other on the progress of individual tasks. For example, if a team member said he would complete a particular task on Tuesday, yet he still reported it as "in process" at Wednesday's stand up, the rest of the team offered to help or at least would want an explanation why it wasn't done yet. This motivated a struggling team member to ask for help early, rather than hiding his problem. The daily meetings also had the benefit of quickly exposing any ambiguities in the acceptance criteria or implementation design. More than once a simple status update spawned an hour-long discussion that clarified a misunderstanding, saving the team complicated rework later.

Teams B and D suffer from their lack of business knowledge. Rather than relying solely on their product owners and business users to fill in the gaps, the more knowledgeable members of the development team stepped in. They organized a series of presentations to give the newer members of the department a baseline understanding of the business we do, the jargon and where to find more information. As the team members better grasp the industry specifics, they ask more intuitive questions of their product owner and business users.

And finally, we revamped our tooling to make each team's work visible to the team and the product owner. We switched our electronic tracking tool to one that would let each team design their own task board and backlog, rather than making them all fit into the same template. This change tied in with the working agreements and gave the teams the freedom to be truly self-organizing in their process and to make that individuality visible. We found the delivery teams are more likely to keep their task boards up to date in both content and structure if they have ownership of that board and can tailor it to their needs.

5. Conclusions and recommendations

In conclusion, we found that well-established delivery teams who have engaged product owners make good candidates for colocating with their business users. Teams that are in the "Norming" (Tuckman, 1977) stage of development may find that colocating with their product owner launches them into the "Performing" stage. They have standard practices and working agreements set; they are "in the groove". Further immersion in the business domain elevates them to the next level of productivity. Colocating with the business users is not a reason to abandon the accepted practices and processes that make a team cohesive and effective. If anything, it is more important to hold to those best practices and routines that bring consistent quality to the software development effort.

On the other hand, new teams or teams with existing problems will likely not benefit from the move. They need support of their peers to establish themselves as a team first. It may help to have the product owner to move closer to the development team, if possible, and be a part of the team formation. They need to get comfortable as a team before embedding with the business users. Likewise, the team should work out any team issues, i.e., get out of the "Storming" stage, before attempting to be present with the business users. These teams have all the instability they can handle; moving them to a new environment would incur too much change. At the very least, a move could stall their effort to become a cohesive team; at worst, a move could decrease their productivity or damage their credibility with the business users. These teams can be assisted by coaching to define – and refine - their working agreements and processes to improve the team first. After the team stabilizes, consider colocation with the business users.

At Parametric, we plan to continue offering the option to colocate a delivery team with the product owner and business users, but will make the decision on a team by team basis. When we have a delivery team willing to take the next step in their performance progression and an engaged product owner able to incorporate the delivery team into the normal business flow, we will make the move. As always in an Agile environment we will continue to inspect the outcome and adapt as necessary.

References

Cockburn, A. &. (2001, November). Agile Software Development: The People Factor. *Computer, 34*(11), pp. 131-134.

Cockburn, A. (2004). *Crystal Clear: A Human-Powered Methodology for Small Teams: A Human-Powered Methodology for Small Teams.* Addison-Wesley Professional.

Cunningham, W. e. (2001). *Manifesto for Agile Software Development*. Retrieved June 23, 2013, from Agile Manifesto: http://agilemanifesto.org/

Eccles, M. e. (2010). "The impact of collocation on the effectiveness of agile is development teams.". *Communications of the IBIMA.*

Krumins-Beens, I. (2012, February 14). *Sitting the Teams Together: Value of Colocation*. Retrieved from Ilio on Agile: http://www.iliokb.com/2012/02/sitting-teams-together-value-of-co.html

Tuckman, B. W. (1977). "Stages of small-group development revisited.". *Group & Organization Management, 2*(4), pp. 419-427.

Waters, K. (2008, May 20). *Agile Colocation*. Retrieved May 2013, from All About Agile: http://www.allaboutagile.com/agile-colocation/

Scrum in a Land Far, Far Away…

(or Using Scrum Across Dispersed Sites)

Rick D. Anderson
rick.david.anderson@gmail.com

Abstract

The agile project management methodology Scrum was designed for teams that are co-located. Unfortunately that is not a common reality in the high technology world we live in. Indeed, companies continue to expand geographically and overseas, more employees are working from home and many large corporations now hire the best talent regardless of project location.

When co-location is not possible, Scrum can still be used with some tailoring. This presentation will talk about Dispersed Scrum – how to use Scrum in with remote workers. Several successful techniques will be shared on how to use Scrum when the project team is not co-located including dealing with team communication, local site leadership, handling Daily Scrum Meetings and ScrumMaster and Product Owner challenges. Most of the techniques presented will be valuable to anyone who works with employees at remote sites regardless of methodology.

The paper assumes a basic understanding of the Scrum methodology. Primers on Scrum can easily be found on the web (see reference [8]) or in book form [9].

Biography

Rick Anderson has been with Wind River for 4 years and is the Engineering Director of the Android Solutions Technology Center, a world-wide group of experienced Android engineers who bring forth unique features into the Android ecosystem. In his over 25 years of embedded, real-time experience, Rick has worked at Tektronix, Interactive Systems and Lucent Technologies in a variety of engineering and management roles.

As a recognized expert in Software Engineering processes, Rick has presented numerous papers on improving software quality and productivity. He is currently serving on the program review committee for the Pacific Northwest Software Quality Conference and he chairs the Wind River Solutions & Services Software Technology Leadership Council. Rick is a certified Agile Scrum Master, an Agile coach and is a member of the IEEE and the American Society of Quality.

Rick graduated summa cum laude with a B.S. in Management and a B.S. in Computer Science from Oregon State University. In this spare time, Rick coaches competitive youth sports and leads Boy Scout high adventure activities.

1. Introduction

In a perfect world, product development teams are all located in the same city, building and even floor. They sit side-by-side and through the incredible power of co-location, they collaborate to dream-up incredible innovations, as well as design and implement killer products. Unfortunately for most of us, this perfect world does not exist. More often than not, development teams are spread across multiple and geographically separated sites. The expectations of these so-called “dispersed teams” are no different than their co-located counterparts – they must develop high quality products on time and with robust features sets.

While the number of dispersed teams continue to grow, so too does the move towards agile software development. Scrum, which combines software development and project management methodologies, seems to be the method that is making the most traction in the industry today. If one goes back to the original Scrum methodology as outlined by Hirotaka Takeuchi and Ikujiro Nonaka in 1986 [11], or even the modern versions captured by Dr. Jeff Sutherland in 1993 and Ken Schwaber in 1995 [9,12], one sees that Scrum is designed for co-located teams.

There are many challenges when using Scrum with dispersed teams. These include communication issues such as how does the ScrumMaster run the Daily Stand-Up Meeting, physical location issues such as how to show a demo to a Product Owner who isn’t in the room and a variety of team organization challenges around training, accountability and schedule tracking.

So how does one then do Scrum with a dispersed team? While many experts have said this is not possible, with tailoring of process and setting clear expectations with your team, Scrum can successfully be used with dispersed teams.

This paper will further outline the challenges of using Scrum with a dispersed team and then provide specific techniques to overcome these. By following this advice, there is no reason a dispersed team cannot be as successful as a co-located one and remain agile at the same time. While most of the advice contained herein is targeted at those who have teams in multiple geographies and time zones, much of the practice outlined here applies also to Scrum teams who are potentially regionally co-located, but still cannot work together face-to-face on a daily basis. In addition, regardless of whether Scrum is used or not, those working with dispersed sites on a regular basis will probably find good advice in this paper as well.

2. Why Dispersed Teams?

In this paper, we will use the term “dispersed teams” to denote any product development team that is not co-located within face-to-face meeting distance of each other. This “dispersed team” could be at home workers, workers who are remote in the same region or country, or workers who are spread across multiple countries and time zones. For the most part, we will assume this dispersed team is spread across multiple geographies and time zones.

Before we dive in, one might ask “What is driving the increase in dispersed teams?” There are a few trends that seem to be causing more dispersed teams. First, outsourcing of work to low-cost geographies continues to occur as businesses focus on driving down expenses. Second, corporations are increasingly willing to hire the best candidate regardless of location, as long as they can report to one of the company’s local offices. Third, companies continue to purchase other companies and often keep their local sites open, thereby creating a more dispersed company. Finally, more companies are allowing employees to work from home to avoid long commutes and help reduce pollution. While this does not contribute to geographically dispersed teams, it does create teams that are not always co-located.

3. Typical Solutions for Dispersed Teams

There are three primary methods of performing Scrum with a dispersed team. These are:

Integrated Scrum – This is your typical Scrum process. It consists of a single Scrum for the team, where dispersed team members participate just like they are co-located. Allowances must be made for the Daily Stand-Up, planning, retrospective and demo meetings. The integrated Scrum approach can work with a local or regionally dispersed team, but is rarely practiced with a globally dispersed one that is spread between multiple time zones (especially more than eight time zones away).

Isolated Scrums – In this model, each major local site runs their own Scrum process. These are not coordinated (hence the name, isolated). This works well when sites don't depend on each other. Local ScrumMasters and Product Owners drive the work. The isolated Scrum approach is rarely chosen because work often cannot be clearly separated by site with no dependencies between sites.

Scrum of Scrums – Each dispersed team runs its own Scrum process, but the Scrums are synced (start and end on time) and the outputs roll-up into a master Scrum process. Local ScrumMasters (and if possible, Product Owners) drive the work, along with an overall ScrumMaster who is driving the roll-up of the individual Scrums. This process can be complicated and dependencies between groups/Scrums must still be dealt with. This can be one of the best methods for using Scrum with large teams.

This paper will propose a fourth method which is a hybrid of the Scrum of Scrum and Integrated Scrum methods. This method has worked well for the author over the past six years. Let's call it a Dispersed Scrum.

Dispersed Scrum – A single Scrum across all sites (like an Integrated Scrum), but run locally by each major local site (like a Scrum of Scrums). In this model, you need one ScrumMaster per site. There is a single Sprint Backlog used by all teams, and during their local Daily Stand-Up Meeting, the local ScrumMaster updates the Sprint Backlog for all local employees. For example, in North America we have an 11am PDT Stand-Up run by ScrumMaster A who is located in North America, and in China we have a 2am PDT Stand-Up (but represents a local work time for them) run by ScrumMaster B who is located in China.

The Dispersed Scrum has the advantage of keeping a single Scrum which is managed locally and integrated across all sites. The following table shows the methods and trade-offs:

Method	# of Scrums	ScrumMaster	Coordinated
Integrated Scrum	1 ☺	At main site only	Yes ☺
Isolated Scrum	2 or more	At local site ☺	No
Scrum of Scrums	2 or more	At local site + main site SM ☺	Yes ☺
Dispersed Scrum	1 ☺	At local site ☺	Yes ☺

A Dispersed Scrum does not solve all problems, however. The planning, demo and retrospective meetings are still a challenge if individuals are located in different time zones. The majority of the planning step generally takes place over a 24 hour period where each site contributes. Specific user stories are selected from the product backlog prior to planning and loosely assigned to each individual and they are responsible for coming up with the task list and estimates for each task. This is supplemented by peer review instead of good practices like Planning Poker. These are added by the local ScrumMaster into the Sprint Backlog. Work generally starts immediately on the next sprint (it doesn't wait for the 24 hour period to end). Given planning is iterative across multiple time zones, it often takes 48 or even 72 hours to get a four week sprint completely planned out by the head ScrumMaster.

The Daily Stand-Up meeting updates a single Sprint Backlog, so it's easy to see what work was accomplished by co-workers at other sites the day before. Roadblocks are added to a common roadblock sheet and worked by local teams where possible, or saved for another team if appropriate. For demo and retrospective meetings, we put these back to back and attempt to get together as many people as possible even if it's an early or late meeting (once a month is generally deemed acceptable for a non-working hours meeting).

For the remainder of the paper, we will cover some of the key areas for success in using Scrum across dispersed sites. For each area, we'll share one or more challenges and then some techniques for success.

4. Crisp Team Communication

Challenges:

#1: Scrum and agility require a faster pace, which can stress a dispersed team. Said another way, agility often exposes communication weakness in a group.

#2: Scrum requires daily interaction, yet time zones and distance get in the way.

#3: Scrum means everyone must be involved. This can be more difficult for the introverted engineer. [Note that this is not technically a challenge that is shared only with dispersed teams.]

Techniques for Success:

#1: All email which requires an answer must be acknowledged within your working day. Where possible, email is answered during your work day. But if the answer is not available, you at least communicate receipt of the email and a target date when you can respond. This "rule" helps sets clear expectations to all team members around the speed at which communication must occur in the group.

Beyond this base rule, asking team members to check email early in the morning (7am when they first get up) and late at night (7pm or 9pm before they go to bed) greatly increases email turnover and the speed at which the team can get things done. This is generally a quick check to answer easy questions, not a lengthy email session. In some cases, we only ask for this level of commitment during critical times during the project or we rotate the responsibility among team members if it can be rotated. Another technique is to work split shifts (6am-10am and 2pm-6pm) which some actually find refreshing. The highest performing teams have incorporated some form of this secondary rule.

#2: Over communicate. In all communications, reiterate the question, provide an executive summary of the answer, then respond with or point to details and quantitative data to back up the answer. Document everything using common tools and common workspaces. It is important to put team information into a well-organized tool so the question and answer can be found by team members on their own. We've used wiki's and Jive (http://www.jivesoftware.com) successfully within our team. The team should work towards perfection in their communication, as every miscommunication costs the team 24 hours.

Perfection, in this case, means the responder does not have to ask follow-up questions to get the information they need (the communication cycles is limited to one question and one answer exchange). We use the phrase: “Perfect communication every time.”

#3: Standardize on the communication tools used and the formats expected. For example, where are status reports kept, when are they due, what format should they be in? Beyond the typical Scrum questions “work done yesterday, work expected today and roadblocks”, what do you need the team to report on and track to be successful. Be very clear about what is required and help the team by creating templates. This is especially important when more information is required than what is typically found in a Sprint Backlog. Again, tools like Jive and wikis can be helpful, although the information does have to be well organized and managed by the team to be of use. Recognize that because people are dispersed, having tools in the cloud can often be useful.

#4: Make it clear to the team that there is no hierarchy in the team and that everyone must speak up. In some organizations, a junior engineer would report to a senior engineer and very little of the junior engineers’ communications would be passed along unfiltered. In order to be agile, this cannot happen. Every team member needs to speak-up quickly and internal team responses should be unfiltered. This idea might take some encouragement, mentoring and socializing within the team, but it is required if you want to truly be agile.

#5: Cultural and language training for all Scrum team members can go a long way towards helping the team work together more closely. Such training improves communication and helps the team members understand why their co-workers might be behaving in a way they are not normally use too.

5. Accountability with Bottom-Up Estimation

Challenges:

#1: Scrum exposes your best and your worse estimators. While this is not necessarily a unique issue for dispersed teams, off-shore resources often do not have the same training and experience doing estimation.

#2: Related, Scrum also exposes your highest and lowest productivity engineers. It is more difficult to “hide” your performance. While this is also not necessarily a unique issue for dispersed teams, it can bring into question the value of dispersed resources.

#3: The Product Owner is watching more closely when using Scrum. This can be intimidating to those not use to working with a real customer or a strong marketing person. This is exacerbated when dispersed team members do not speak the same native language as the Product Owner.

Techniques for Success:

#1: When using Scrum, it is very important to discuss the meaning of accountability and schedule ownership. Everyone needs to understand they own a piece of the schedule and if they do not deliver on schedule, it sets the whole team back. Scrum team members must understand the importance of doing solid estimation up front and providing feedback on others estimates if they feel they are incorrect. In my experience, many team members are afraid to add new tasks in the middle of the sprint, or to raise roadblocks early. The ScrumMaster needs to encourage this behavior. Additional training might be required here. The ScrumMaster must be willing to work with and mentor those who are struggling in any of these areas.

#2: Emphasize the schedule every day in your Daily Stand-Up meeting. Show the burn down and/or burn up chart. Highlight progress. Let the team know if they are ahead or behind. Engineers like to stay ahead of schedule. And of course, they don’t like falling behind. Open and visual communication helps ensure positive behavior.

#3: Don't plan 100% of your available hours during your planning meeting. If you do, any unestimated tasks (the #1 source of missed schedules) will surely put you behind because you cannot recover. This also gives you a little flexibility to take on some stretch goals or to deal with emergencies that occur during your sprint.

#4: Use six hour work days. On my teams, tasks are estimated assuming you are doing nothing but working on the task (so called "heads down estimation"). But the reality is that the typical engineer has email to respond too, meetings and other interruptions during a typical work day. By planning for a six hour work day, you don't have to estimate with "overhead" built into your estimates. I have generally found this to be a more accurate way to perform estimation.

#5: Product Owners should recognize the enormous influence they might have over Software Engineers who are not use to working with a customer or a customer representative. Providing lots of positive feedback and give-and-take dialog will help establish a partnership of trust. The ScrumMaster should watch these relationships carefully and help both sides understand roles, project goals and convey ways to work together more efficiently.

6. Daily Stand-Up Meeting

Challenge:

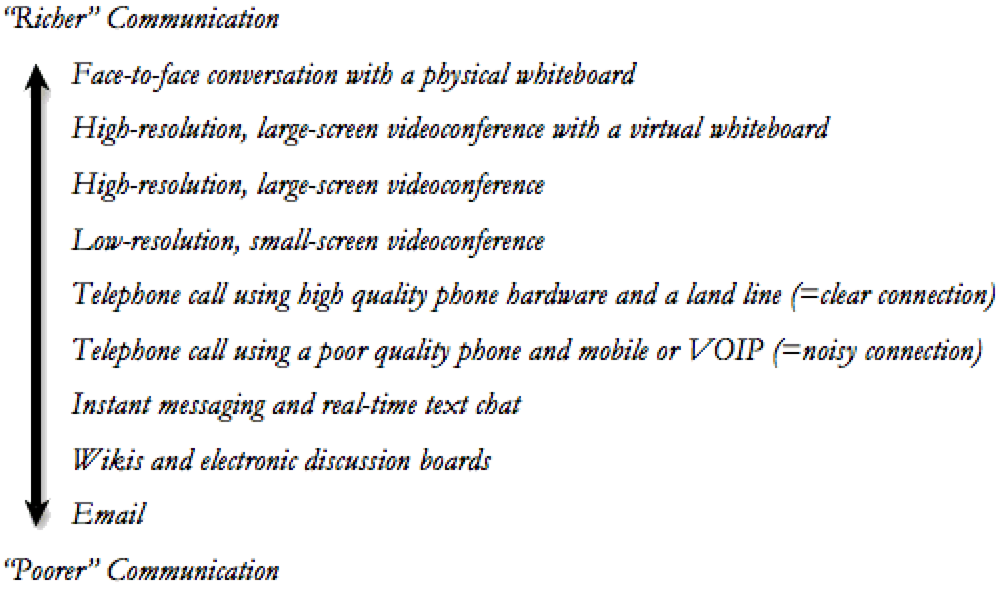
#1: A co-located Daily Stand-Up is impossible with a dispersed team.

Techniques for Success:

#1: Have local Daily Stand-Up Meetings. Using the Dispersed Scrum model described above, each major local site has a Daily Stand-Up meeting run by a local ScrumMaster. In essence, you have multiple Daily Stand-Ups. The Dispersed Scrum method uses a single sprint backlog, burn down chart, roadblocks list, etc. so all team members see everything else the team is doing. These process assets are stored in a commonly accessible place (the cloud is good given its accessible regardless of location). It is important to make sure they are under configuration management control and backed-up.

#2: Use the most appropriate tools designed for distributed work. While email is often an engineers' first choice for communication, it is often the worst choice if you wish to have a robust conversation.

Peter Deemer in his document on Distributed Scrum [6] talks about a "richness" scale for communication which can be seen here:


"Richer" Communication
Face-to-face conversation with a physical whiteboard
High-resolution, large-screen videoconference with a virtual whiteboard
High-resolution, large-screen videoconference
Low-resolution, small-screen videoconference
Telephone call using high quality phone hardware and a land line (=clear connection)
Telephone call using a poor quality phone and mobile or VOIP (=noisy connection)
Instant messaging and real-time text chat
Wikis and electronic discussion boards
Email
"Poorer" Communication

Source:http://www.goodagile.com/distributedscrumprimer/commsdiag.gif.

The basic idea is that visual communication (face-to-face, video conference) is better than verbal communication (telephone) is better than interactive written (chat rooms, text messaging) is better than non-interactive written (email).

When dispersed, video conferencing is great for Daily Stand-Up Meetings if you have the equipment and the bandwidth because you can see the other team members. Skype, WebEx, Google Hangouts and other tools and services exist if you don't have dedicated video equipment. When video conferencing doesn't work, text messaging using IRCs, IM or Chatrooms can work almost as well. When combined with searchable logs and the ability to go back into history, these even become more valuable.

Having status typed-up ahead of the meeting start, using an agreed upon format (for example, task/hours worked/hours remaining), starting on time and proceeding in a set order helps things move along. This method can also overcome some language barrier issues that might exist with video and spoken communication. I've also seen teams record a Daily Stand-Up meeting and post this video for dispersed teams. Whatever method you use, really try to maximize team communication while keeping the meeting short and to the point.

#3: Continue to have weekly group meetings. While this is often the responsibility of the Engineering Manager or Technical Lead/Architect instead of the ScrumMaster, having this additional interaction when you have a dispersed team remains critical. Having short training sessions, design reviews and other sharing sessions despite having a dispersed team helps keep the team together and productive.

7. Strong Local Site Management

Challenge:

#1: It can be hard for leaders (ScrumMasters, Product Owners or Engineering Managers) to understand exactly what is going on at dispersed sites. They have no eyes and ears to see the folks and hear the hallway conversations. There is often no one person you can go to for issues that fall between typical job duties. In an international setting, understanding local laws, customs and culture might be difficult.

Technique for Success:

#1: Having a strong designated leader at your remote site helps greatly. This can be any combination of a local ScrumMaster, Product Owner, Engineering Manager or Technical Leader. These individuals can act as the local translator if language issues are involved, they can be your eyes and ears, and they can help explain local laws, customs and culture. Finding the right leader is critical and once you find them, keep them! They will become very critical to your success. Have weekly 1:1's with them to really understand the heartbeat of the dispersed site.

8. Self-Organizing Teams

Challenge:

#1: Problem solving must be pushed down to the individual engineer with a dispersed team. Take an extreme example, where every team member is located in a different location. In this scenario, there is often no one to ask for help, and certainly no one sitting next to you that can help you. Co-located teams have a much larger "pool of knowledge" sitting next to them. Further, the number of common work hours might be limited due to geographically different time zones for project team members.

Techniques for Success:

#1: Build empowered and self-organizing teams. To build a really strong team, you must let the team figure out things for themselves. One of the great investments you can make here is to teach the team problem solving techniques. These add tools to their toolbox. Examples of some great problem solving techniques to be taught include Fishbone Diagrams and 5 Whys. Having weekly technical sharing meetings also helps and is a good forum for quick-n-dirty tips around tools and other areas.

#2: Leverage your retrospectives by listening to your team and driving problems to root cause. Sharing of lessons learned is so obvious, but rarely done well. See previous PNSQC paper by this author and others for some great suggestions on holding good post-project reviews [1], [2]. Also see Martin Fowlers article on Lessons Learned when using Agile with Offshore Development teams [10].

#3: While a strong ScrumMaster or Engineering Manager could solve most problems themselves, this doesn't help make the team stronger. Instead, identify problems, schedule a meeting and act as a moderator while letting the team figure the problem. This guided problem solving approach takes a bit longer, but makes the team stronger.

#4: The Four Hour Rule. This rule says that if you have spent more than four hours trying to solve a problem and you still haven't done so, you should ask for help. This means raise it as a roadblock and alert the team you are stuck. It's amazing how well this simple rule works at removing roadblocks by using the collective knowledge of the team and not letting a problem churn in an unproductive manner.

#5: Adopt a Defect-Free Mentality. This concept is one where you push quality upstream and the Software Engineers focus on producing work products free of defects. In essence, they try to put the SQE team out of business since there are no defects to find. It's a mental mindset that is lacking in most organizations. See this PNSQC paper for more on this [3].

9. ScrumMaster Issues

Challenge:

#1: It's difficult for a ScrumMaster to manage a dispersed team. They might have to attend daily stands-up in weird time zones, it's hard to get a real feel for the status of the project when you can't touch the product, and you don't hear the hallway conversations or see the panic on their faces.

Techniques for Success:

#1: Follow the Dispersed Scrum model. This says you should have one ScrumMaster at each local site. A local ScrumMaster can overcome many of the challenges a typical project faces. Certainly all ScrumMasters on the project must work closely together as well.

#2: Rotate the ScrumMaster role between those that have the right traits (see below). This strengthens the team. And there is no reason why the ScrumMaster can't be outside of Software Engineering, like Software Quality Engineering or Technical Publications.

A good ScrumMaster must be:

- Organized and prompt
- Helpful
- Patient and flexible
- Accountable
- Be detail oriented
- Have good communication skills

#3: Use the right communication medium for the job. As previously discussed, engineers have a tendency to use email for their default communication device because it's fast and easy. However, often times it is not the best method for achieving timely resolution. The preference for agility is interaction, so my encouragement to teams is to use the phone or video conference first, then some interactive text messaging method next and email last.

#4: Learn the local language, customs and culture. Certainly this is easier said than done, but there are books, websites and courses on all of these. The more a ScrumMaster understands and can communicate with the team, the more effective they will be.

#5: Visit your remote teams at least once per year and work in their time zone every once in a while to increase overlapping communication time. Sometimes a simple shift in work hours goes a long ways towards making a good ScrumMaster a great one.

10. Product Owner Issues

Challenges:

#1: A remote Product Owner has the same challenges as a remote ScrumMaster. This includes time zone and communication issues. Product Owners who cannot see and touch their product as it is being developed is certainly a challenge that must be overcome.

#2: Not having an active Product Owner is equally challenging. While this is not specific to dispersed teams, the challenge becomes great when you are dispersed.

Techniques for Success:

#1: See all the ScrumMaster Techniques for Success above. The Distributed Scrum Primer has a whole section on trust and the benefits of the Product Owner travelling to dispersed sites [6, and specifically page 2].

#2: Make sure your Product Owner understand their role. Level with them on the time commitment, participation expectations, communication within the group, etc. Be sure there is agreement up front on how planning and product demonstrations will be taken care of.

#3: Make adjustment to the product demo meeting so it's still relevant for all. Which method is best depends on your product and team location. There are a few different ways to do this:

- Assuming your product can be controlled via remote means, perform the demo live over the Internet.
- When the product can't be controlled remotely, using a live video feed via the Internet is next best. The Product Owner can ask questions in real time and the engineers can adjust the demo on the fly.
- Have the same physical hands-on demo at all dispersed sites, but at the same time. With this approach, the "demo master" guides everyone step-by-step on the execution of the demo.
- Video the demo ahead of the meeting and share the video via WebEx or other sharing mechanism during the demo meeting.

Related to this, be sure to still celebrate project success. While it might not be possible to have ice cream or a team lunch with a dispersed team, there are other mechanisms that can be utilized to share incremental milestones on your projects.

#4: Supplement the primary Product Owner with a local Product Owner. This might be an Engineering Manager, a Technical Leader/Architect, a Product Owner from another team, or anyone else who can represent the owner. Certainly the supplemental Product Owner must work closely with the Product Owner on all decisions.

11. Separate Software Quality Engineering Teams

Challenges:

#1: On many project teams, the Software Engineering (SWE) team is co-located and the Software Quality Engineering (SQE) team is also co-located, but not necessarily together. Specifically in many organizations, the SQE function is off-shore.

#2: Is the SQE team part of the Scrum or not? While the answer to this seems obvious and it is not specific to dispersed teams, it seems this question comes up a few times a year.

Techniques for Success:

#1: Have a separate SQE function. It remains a best practice to have a team responsible for product quality, writing automated test cases and watching over the Software Engineering teams work as a voice of the customer.

#2: Have the SQE team 100% integrated in the Scrum. So while they can be a separate role or group, they are absolutely part of the Scrum team. SQE retains their typical duties of test planning, test case creation and test case execution. A focus on daily builds and test automation is recommended. Part of the SQE group's roles should be to push quality upstream (software process improvement). There are many articles being written about how SQE processes can be modified to work with Scrum.

#3: As much as possible, treat the SWE and SQE teams as the same. They should be equals as far as job status, pay, responsibility, etc. If you have Software Engineering leaders, have Software Quality Engineering leaders. Their value is the same.

#4: Challenge the SQE team to do more than just manual testing. They are SWEs, but their role is to write test code instead of product code. Also encourage them to push quality upstream (software process improvement). These items give the SQE role more variety and make it more exciting.

12. Track Plan versus Actual

Challenge:

#1: Scrum normally tracks the original estimate and hours remaining (what did you do yesterday, what will you do today, how many hours are remaining). This makes it hard to identify slipping tasks and those who have estimation problems. While this challenge is not specific to dispersed team, it can be a bit more difficult to identify those who have estimation issues if you aren't in a daily stand-up with all team members every day.

Technique for Success:

#1: Track actual work hours. With just a minor change, your Daily Stand-Up Meeting and Sprint Backlog sheet can be modified to keep plan versus actual hours. I found this to be a huge win in tracking the success of my projects and helping those who needed it with estimation and problem solving.

Below is an example of a spreadsheet-based Sprint Backlog. You will see that it shows the Initial Estimated Hours during the planning meeting (Initial Est'd Hrs column), but also the running total of the number of hours worked on the task (Hrs Worked column). Each day during the Daily Stand-Up Meeting, individuals report the task they worked on, the # of hours they worked on it and the number of hours remaining. The # of hours worked is added to the # of hours in the Hrs Worked column. The # of hours remaining get recorded in the appropriate day column (for example, 14-Sep). Looking at this table during your retrospective, it would be easily to see that the "Add alpha blending feature" was under-estimated.

Backlog Item	Task	Resource	Task Status	Initial Est'd Hrs	Hrs Worked	10-Sep	11-Sep	12-Sep	13-Sep	14-Sep
Jelly Bean Security	Test and debug	Rick	Not Started	9	0	9	9	9	9	9
	Add alpha blending feature	Rick	Completed	12	18	12	12	6	6	0
	Gingerbreak security exploit	Rick	In Progress	6	0	6	6	6	6	4
	Combine existing code & open source	Rick	Completed	6	6	6	0	0	0	0
	Video	Rick	Not Started	12	0	12	12	12	12	12
	Demo image creation	Rick	Defer	12	0	12	12	12	12	0

13. Other Tips for Success

There are several other tips for working with a dispersed Scrum team that won't be covered here, but include:

- Use of appropriate tools that can be used across geographies (cloud-based tools come to mind)
- Daily builds used by all sites
- Good configuration management techniques (CM becomes very important to project success; consider having a dedicated person responsible for this, potentially at each dispersed site).
- Visit your team on a regular basis. Getting the team together even when dispersed helps with team morale and productivity.

14. Conclusion

This paper has presented a number of challenges that a dispersed Scrum team faces as compared to a co-located Scrum team. For each of these challenges, one or more Techniques for Success have also been given. Hopefully you will find these helpful in your efforts to become more agile. I'd like to leave you with the following final suggestions:

- Building a high performance dispersed Scrum team takes lots of patience and work, recognition that it will not happen overnight (nor in your first sprint) and two-way trust (empower the team, hold them accountable).
- Create an environment where "we are all in this together".
- Use retrospectives as constructive events.
- Celebrate success

Good luck!

15. References

[1]. Anderson, Rick D. and Gillmore, Bill, 1997. “Maximizing Lessons Learned: A Complete Post-Project Review Process”. 15th Annual Pacific Northwest Software Quality Conference Proceedings, page 129. http://www.uploads.pnsqc.org/proceedings/pnsqc1997.pdf

[2]. Lavell, Debra, 2007. “Facilitating Effective Retrospectives”, 25th Annual Pacific Northwest Software Quality Conference Proceedings, page 329. http://www.uploads.pnsqc.org/proceedings/pnsqc2007.pdf

[3]. Anderson, Rick D., 2007. “Flexing Your Mental Muscle to Obtain Defect-Free Products”. 25th Annual Pacific Northwest Software Quality Conference Proceedings, page 49. http://www.uploads.pnsqc.org/proceedings/pnsqc2007.pdf

[4]. Buecker, Inge, 2008. “Architecture tip: Adapting the Scrum project management method for geographically separated teams”. http://www.ibm.com/developerworks/library/ar-scrumremote/

[5]. Rothman, Johanna, 2012. “Agile Lifecycles for Geographically Distributed Teams, Part 1”. http://www.jrothman.com/blog/mpd/2012/01/agile-lifecycles-for-geographically-distributed-teams-part-1.html

[6]. Deemer, Pete. “The Distributed Scrum Primer”, http://www.goodagile.com/distributedscrumprimer/index.html

[7]. Puopolo, John, 2007. “Be There or Be Square – A Case for Collocated Teams”, http://www.scrumalliance.org/community/articles/2007/august/be-there-or-be-square .

[8]. Deemer, Pete and Benefield, Gabrielle, 2007. “The Scrum Primer – An Introduction to Agile Project Management with Scrum”, http://www.scrumprimer.com/

[9]. Schwaber, Ken and Beedle, Mike, 2002. “Agile Software Development with Scrum”.

[10]. Fowler, Martin, 2006. “Using an Agile Software Process with Offshore Development”. A list of several lessons learned from ThoughtWorks experience.

[11]. "New New Product Development Game". Harvard Business Review 86116:137–146, 1986. January 1, 1986. Retrieved March 12, 2013.

[12]. Sutherland, Jeffrey Victor; Schwaber, Ken (1995). *Business object design and implementation: OOPSLA '95 workshop proceedings*. The University of Michigan. p. 118. ISBN 3-540-76096-2.

Others:

http://msdn.microsoft.com/en-us/library/jj620910.aspx

http://www.nicta.com.au/pub?doc=4637

http://www.solutionsiq.com/Portals/93486/docs/distributed-scrum-why-some-teams-make-it-work.pdf

Achieving Right Automation Balance in Agile Projects

Vijayagopal Narayanan

Vijayagopal.n@cognizant.com

Abstract

When is testing complete and How much testing is sufficient is a fundamental questions that has been asked for a long time. A convincing answer to this question will help in optimizing the effort spent by Development and QA teams in testing the systems that they build.

Multiple approaches have been proposed as a silver bullet to achieve high quality, from TDD, BDD, ATDD, Service layer testing etc. To traditional requirements based testing. One view is to breakdown the system into its architectural layers and use tools to verify what is being build in each layers. In such a scenario some of key questions that arise are, how do I know testing at these layers brings value? How can I be sure that there are no redundancies? And where should I invest more? An attempt has been made in this paper to come up with a framework which provides a unified platform on which testing across these layers can be integrated. In addition, a set of metrics are defined to measure the effectiveness of testing in various layers and their contribution to overall quality.

Biography

Vijay Narayanan is a Senior Manager at Cognizant, currently working at Chennai, India. He has carried out multiple consulting and delivery engagements in Testing and Test Automation for customers in different industries. The engagements covered various dimensions like Automation Strategy, Testing Tool Strategy, Test Process Assessment, and Setting up of Testing Center of Excellence. He has strategized and deployed Testing Transformation Program for Large Testing engagements.

He also led development and deployment of solution accelerators like Automation frameworks, and QC integration tools. He led the Automation and SOA COEs and was responsible for new service offering, partnership management and business development in this area.

He has 13+ years of experience in software services industry, out of which 10+ years in Testing and Test Automation.

Vijay has an M.E in Computer Integrated Manufacturing and B.E Mechanical Engineering from Bharathiar University.

1. Introduction

Different viewpoints have evolved to the question of "How much testing is sufficient?". The answers are provided from one of the two perspectives, one is to verify the smaller parts which make up the whole (i.e. unit testing, code coverage etc.), and the other is to look at the behavior of the emergent system as a whole (i.e. a black-box point of view). The development in test automation has also followed on the similar lines with different tools and frameworks addressing each of these.

In the bottom-up view of building-in quality the system is broken-down into its architectural layers and testing tools are used to verify what is being built in each layer. In this scenario one of the key questions is, how do I know testing at these layers brings value? In addition a top-down method is also adopted typically during later stages of testing. In which case, How can I be sure that there are no redundancies? And where should I invest more – at each layer or at system level? In this paper an attempt has been made to come up with a framework which provides a unified platform on which testing across these layers can be integrated. In addition, a set of metrics are defined to measure the effectiveness of testing in various layers and their contribution to overall quality by leveraging and extending static code analytics tools, and analytics. It helps in painting an integrated view of various testing and how effective they are in exercising the system.

2. Testing Multitier Architecture

Most of the systems built in today enterprises follow a multi-layer (N-Tiered) architecture. The tiers typically have a GUI layer to render the user interface, a business layer which encapsulates business rules and orchestration, a services layer which implements business rules\functions and a data layer for persistence. Even with logical separation of responsibilities, the complexity involved in testing such a system can be very high.

Organizations often employ a pure black-box testing strategy to validate such systems. A requirement based testing approach is used and traditional GUI test automation tools are employed to bring efficiency to test execution. In more matured organizations a grey-box test strategy is adopted using a combination of GUI and Service testing tools and frameworks, thereby providing feedback at two of the layers (GUI & Business Process) involved.

In cases of Agile projects teams may employ techniques like unit testing, GUI automation for behavioral and acceptance testing and API\Service layer testing. Though a combination of these efforts may improve the quality of the system being developed, it is very difficult to balance these test approaches and decide on how much testing is required at different levels.

3. Limitations with Existing Solutions

In this context it is essential to put in place a feedback mechanism so that the test efforts are optimized. Some of the mechanisms that are already available are,

A) Static Testing and Code coverage tools: Static Testing tools can automate the process of code review for standards and best practices. It reveals sections of code which can cause errors and supports implementation of white-box testing. Code coverage tools provide information on the parts of code executed as part of black-box testing. When a project team makes a refactor or implements a new feature code coverage tools can be used to understand whether the refactored or added code has been covered by testing or not.

B) Behavior Driven Development (BDD): Behavior driven development is an Agile development methodology that focus on user needs captured as behaviors in ubiquitous language and form an executable acceptance criteria. Business, development and quality teams collaborate to capture behaviors that are to be met by the system under development. These are then tied together in a validation script and run using BDD frameworks like JBehave. When these scripts are run feedback on the system is provided at behavior level.

What we need here in addition to the above methods is a mechanism which can verify the effectiveness of coverage at different architectural layers. This mechanism needs to be light weight and also should be able to automatically collect data and analyze the feedback that is received from across the layers. It must also provide a view to integrate the business domain models with system model, which consists of implementation units like Classes and Web services.

4. Integrated Test Effectiveness View

4.1. Why we need an Integrated Test Effectiveness View

If we look at the feedback mechanisms available from the perspective of the multitier architecture, we see that code coverage and behavior coverage address the opposite ends of the spectrum. The code coverage tools provide very granular information and helps in understanding the inner working of the system, while the BDD helps in focusing on the bigger picture, as shown in the Figure 1 : Integrated Test Effectiveness given below. Coverage effectiveness measures for other layers are either not available or not in common use. It will be interesting to have feed-back mechanism for other layers. A framework that can integrate feed-back data to provide a seamless in an out-side-in or inside-out fashion will be valuable from the quality assurance stand-point. It can also help in identification of gaps in testing and to highlight the layers\areas to be focused based on the change is being implemented.

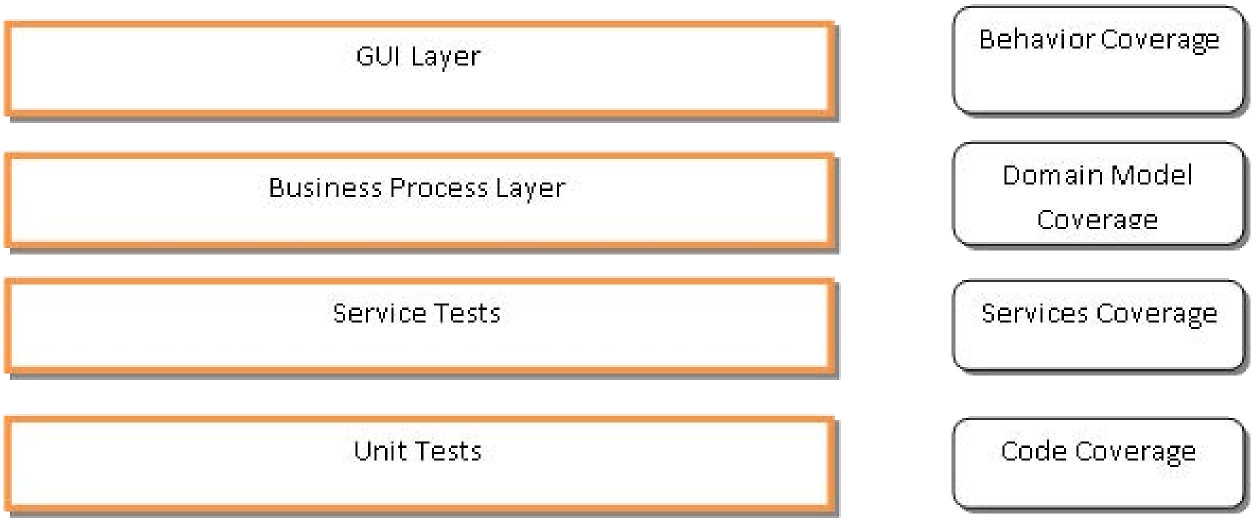


Figure 1 : Integrated Test Effectiveness

We are at an interesting point of time where the developments in testing concepts and technology field provides us various tools which can be used to arrive at a better feedback mechanism.

4.1.1 BDD approach

BDD frameworks allow testers to hook the test scripts at deferent levels of the architectural layers, while at the same time keeping focus on behavior of the system. At the GUI layer the step definitions can be implemented using one of the GUI\Browser drivers like Selenium, WatiN etc. Again at the services layer SOAP UI or other such tools can be used to invoke the services and check for the system behavior. Similarly for the API\Class\Method layer.

4.1.2 Model driven engineering

Model driven engineering approach focuses on domain models and deriving systems designs\patterns from there. This provides tools for testers to understand the underlying systems and compare coverage between the actual and modeled system behavior. It even allows automation of the test design processes using techniques like state-machines to derive scenarios for the system.

4.1.3 Analytics

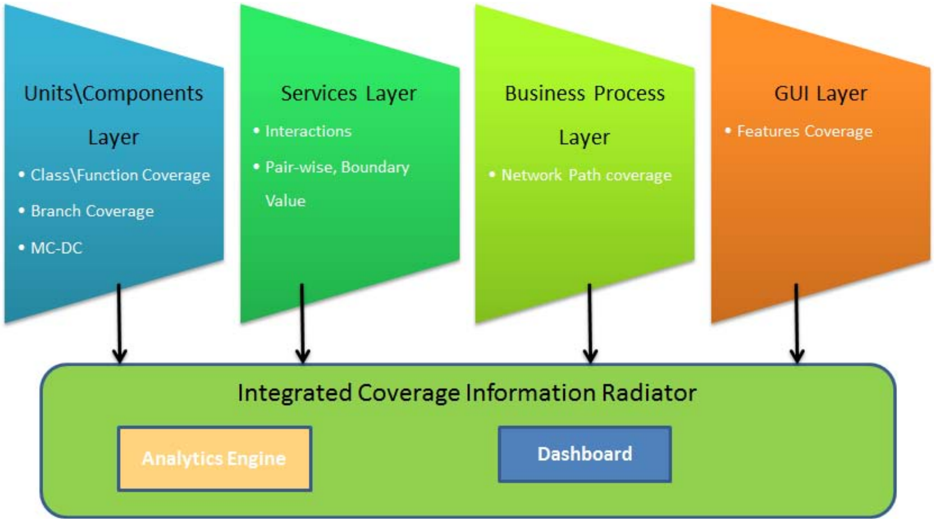
This is a technology which has not been leveraged optimally to understand the testing coverage, progress and other interesting attributes. As systems become increasingly complex, it will require more of analytics to bridge the gap between the domain-model and application model. There by, helping us to understand emergent behavior is produced through interactions between the system components. e.g.: How do hundreds of unit test scripts map to a service and how do they in-turn map to a behavior of my domain model.

4.2. Conceptual Model of Proposed Integrated Test Effectiveness View

By leveraging the above mentioned advances an approach is proposed to provide the testing team a valuable Information Radiator to address the key shortcomings in the current approaches. This can be used in real-time to understand the quantum of testing carried out by various team and vector the test efforts on a real-time basis.

It is assumed that the project team has implemented static testing and code coverage tools, service and business process layer testing as well as traditional functional testing at GUI layer.

The conceptual components of these solutions are explained in Figure 2: Conceptual Model below.



4.2.1 Unit and Components Layer

This component extracts the code coverage and static code quality data for the various packages that are part of the system being tested.

4.2.2 Business Process Layer

A component will analyze the business processes that are covered by testing. One of the approaches that can be used is to, leverage the middleware\BPM component. Typically the business process orchestration happens through a middleware component or Business process engine. A BPEL model is prepared and used to define the orchestration mechanism. This information can be matched with actual business process orchestration data generated during the testing.

The other approach is to leverage service virtualization tools to stub all systems other the System under test (SUT). By analyzing the invocation records from the virtualization tool, information could be gathered to trace the systems that are called as part of a business process.

Both the approaches can provide high-level as well as granular data on the coverage achieved through testing at the level of business process layer.

4.2.3 Services Layer

Services level coverage data can be generated by extending the capabilities of service testing tools and gather information on the atomic and composite services that are tested at the courser level. Detailed coverage at data element level like boundary value\pair-wise coverage can also be achieved.

4.2.4 Functional\Story Coverage

The next element is the coverage at system behavior or feature level. Projects adopting any one of the BDD frameworks can gather this information. This will show at a business level which features are being tested.

4.2.5 Analytics engine

This is the key component that will run algorithms to integrate and piece together a system view from the coverage data that is made available from all the layers. All the information that is gathered at each of the layers, as mentioned above, will be integrated and a complete picture of the solution will be constructed in the Analytics engine. This will include performing analysis, such as A) which class and services render a particular behavior, B) Which services are orchestrated during a business process etc. We envisage integration with Business modeling and BPM tools to compose this integrated picture of the system.

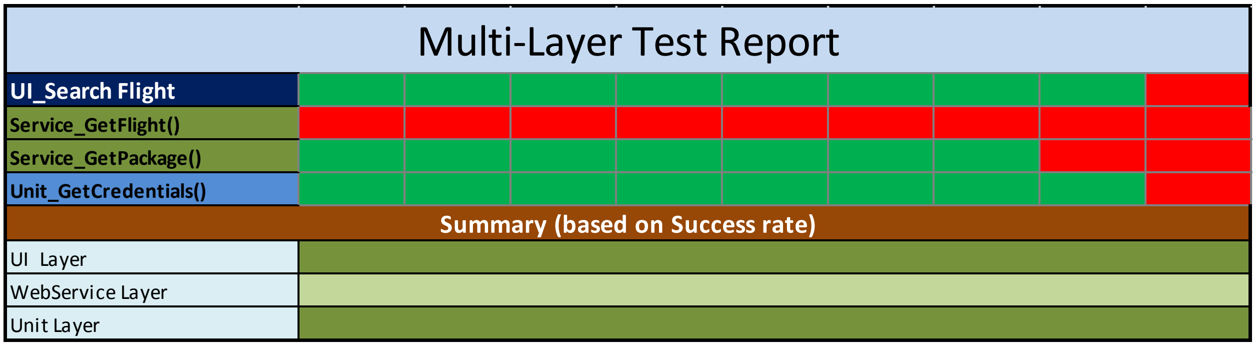
4.2.6 Dashboard

The insights produced by the analytics engine are provisioned through the dashboard which will serve as an information radiator for the teams. For this purpose BI tools like Informatica or light weight implementations likes Google Analytics APIs can also be used.

5. Implementation Approach

The above conceptual model can be realized by employing tools\frameworks like SONAR, SOAP UI, Selenium and JUnit for multilayer testing. Reports from these tools like test coverage, test failure rate etc. will be fed into the Integrated Test Effectiveness Synthesizer, which can be a Java based component. This component will have necessary logic to integrate the feedback from each layers and the resultant dataset will be stored in a database like MySQL. A dashboard generator component will query and format the data in JSON to feed the Dashboard. Dashboard rendering engine can be based on Google Analytics API along with an agile project management tool.

A template reporting format for integrated test effectiveness dashboard is shown in Figure 4: Template Report given below. In this view the failure of test scripts at GUI, Service and Unit level are shown in red. They can be aggregated at the level of features, web service methods and classes respectively for Acceptance, Service and Unit test levels.



Multi-Layer Test Report									
UI_Search Flight	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥
Service_GetFlight()	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Service_GetPackage()	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
Unit_GetCredentials()	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟥
Summary (based on Success rate)									
UI Layer	🟩								
WebService Layer	🟩								
Unit Layer	🟩								

Figure 4: Template Report

6. Conclusion

The challenge of doing the right amount of testing at right time increases due to system complexity and multiplicity of test approaches that are adopted; there is a necessity for a "Crystal Ball" of sorts to view the big picture. In the current approach different testing techniques are used validate various layers of the system. Still there is no mechanism available to verify the cumulative effectiveness of these testing efforts. This paper proposes an Integrated Test Coverage Radiator that can integrate with testing tools used for multi-layer testing and provide a synthesized view into system behavior at these layers. This approach can be adopted for situations where the applications being built are complex, or connect to multiple dependent systems to keep the testing on track and thereby to achieve the right balance in automation.

References

Larsen, Michael.2012. "Get the Balance Right: Acceptance Test Driven Development, GUI Automation and Exploratory Testing", *Excerpt from PNSQC 2012 Proceedings.*

Bach, James.1998. "A Framework for Good Enough Testing." *Computer, IEEE Computer Society (Oct):124-24.*

From Dinosaur to Cutting Edge: 5 Practical Keys to Avoiding Extinction in An Agile QA World

Robert Gormley

hotspur99@hotmail.com

Abstract

The advent of the mobile world has placed product quality and customer satisfaction squarely on the shoulders of the company's IT department because the speed and transparency of customer feedback in the world of social media means poorly designed or low quality software goes "viral" for all the wrong reasons. The response of the IT community to this challenge has been to rid itself of long-winded iterative software development methodologies like the Rational Unified Process (RUP) or Waterfall. In their place, dynamic, ever-evolving methodologies like Agile and Scrum have begun to dominate the landscape like a ferocious T-Rex because they offer more flexibility in the development process and expedite the time to market. While the adoption of a new development methodology can be difficult on all departments, the transition of the QA Department to Agile is laced with the harsh realities of having to overcome a change to their philosophic core and meeting the challenge of aggressive skill adaptions required for Agile.

Based on my experiences (from skeptical beginner to implementation leader) and supported by research, this paper points out 5 practical keys for making Agile QA more successful for those QA departments already engaged in Agile. These keys may not provide a Rosetta stone for perfect Agile QA but the unique perspectives derived from them may help other practitioners of Agile QA in their march to avoid extinction.

Biography

Robert Gormley is currently a Principal Consultant at SWAT Solutions where he focuses on executive level TQM strategy and developing bleeding edge test automation frameworks. He has worked for 12 years in Quality Assurance and has worked his way up through the ranks in industries like finance, healthcare, retail, security software, and device QA. From his early days to present day, Robert's focus has always been on creating lean, flexible processes that allow organizations to produce quality software that meets a consumer's needs. Leveraging his Master's in Education, Robert has passionately trained and mentored all types of resources (regardless of title or function) in the ways of Total Quality.

Robert holds a Master's in Education from Loyola College in Baltimore as well as BAs in English and History from the University of Maryland at College Park.

1. Introduction

About 6 years ago, I was asked by my wife (who also happens to be in software QA) what I thought about Agile. Her company had just made the announcement that they were going to transition to Agile. Based on some quick research and a lot of skepticism, I answered her in the following way:

> Well, you better start looking for a new job. Agile isn't a methodology, it's an excuse for people not to write down requirements and a way for business to take control of the development process. People like your product manager who knows nothing about IT are going to be running your life. Good luck!

As I review my response today, I note the general sense of fear I had about Agile and what it would mean for QA. One of the most difficult things to imagine for me at the time was how anyone in QA would be able to test without a comprehensive requirements document. The principles of RUP and Waterfall were so ingrained in my way of doing QA that the idea of changing my thought process was both scary and anger inducing. I'm not alone either as a study in 2012 shows:

Out of over 200 participants**,** 64 percent said that switching to Agile Development was harder than it initially seemed. Forty percent of respondents did not identify an obvious benefit to the practice. Out of those who did, 14 percent thought it resulted in faster releases, and 13 percent – that it created more feedback. Seven percent of participants noted that Agile developers were happier due to reduced future planning and documentation.[1]

The study's conclusions show how much psychology plays into the successful transition to Agile. At the time, I wouldn't have been part of a successful team because I was defiant on what Agile was and what the benefits were.

I now realize how naïve I was and how much of what I knew about Agile was based on fear of the unknown and how immensely dependent I was on highly inflexible processes. I had spent 6 years of my professional career dedicated to learning and mastering RUP and Waterfall then suddenly the world I knew was evolving into something I didn't recognize. The other fear I had was based on my lack of technical ability. I wasn't an automation guy; I was a thinker, a planner, a strategist. The skill set for an Agile world seemed to favor someone who could do development work along with QA. How on earth was I going to fit in if I wasn't willing to adapt my skills to the new world order?

Fast forward to today and the experiences I've had with Agile for the past 6 years have made me a better QA professional. But for every 1 person like me, I have encountered what seems like 5 others who have had a terrible experience with Agile and who continue to struggle with the transition. Being in leadership positions, I've had the unique privilege of being able to help implement Agile and have been able to watch with both joy and terror as companies I've worked for have tried to transition to Agile development. To this point, I've been able to identify 5 practical keys to being successful in the Agile transition when it comes to QA teams.

2. Collective Commiseration Can Be Positive

Nothing seems to bring people together more than being able to commiserate about a shared experience. What I've learned in my Agile transitions is that collective sympathy can be a positive thing for your team and it can contribute to providing motivation[2]. It's hard to change the way someone approaches their profession, especially if they have spent years becoming an expert in it and they feel like they haven't had much say in the decision to change the way things are done. Let people "vent" while you empathize or you risk being one of the organizations that fails because Agile is "forced" on a team that is accustomed to other methods.[3] What I've found in my teams is 3 types of commiseration bubbles to the top: warm fuzzies, cautious optimism, and destructive blowhards. Managing and encouraging the first two kinds of commiseration while dealing swiftly with the third can make or break an Agile QA transition.

2.1 The Warm Fuzzy

Everyone hopes that the team will feel warm fuzzies when they transition to a new methodology which may be unfamiliar. I've always been surprised that the most unlikely of people will experience the warm fuzzy the quickest. These feelings have often been expressed by people saying things like "wow, I really feel more involved in this process" or "I never knew how our business worked but now I do having sat with the business owners". Another key indicator of warm fuzzies is the body language. Instead of shying away from having people share personal space, invitations for people to come over and talk increase. Instead of standing up with your arms folded, people relax and smile. If you see a warm fuzzy, encourage it. Also, don't be afraid to add a warm fuzzy by praising someone (especially someone who may have been skeptical to begin with) who has really embraced the Agile transition.

2.2 Cautiously Optimistic

Let's face it; there are 2 types of people in this world: people who see the glass half full and people who see the glass as half empty. Don't let the half empty people get you down, see it as an opportunity to step in and work to alleviate feelings of confusion, fear, or frustration. It is natural for people who are transitioning into something new to have these feelings; success in the transition can be determined by how you manage these feelings. Empathize with everyone and give them a forum to voice their opinions so they feel like they have a stake in the game. Work with your team to make sure they feel like they own the process, even if the process isn't fully developed. I recall a quote from Scott Barber that is apropos here: The reality is that Agile is far more a mindset and a culture than it is a collection of practices, processes and tools.[4] Focusing on making people feel comfortable is just as important as outlining new processes or pushing people to learn new skills.

2.3 Destructive

Regardless of how much you try, you'll always have one or two people who just won't gracefully accept change. They become the individuals that seek to make destructive comments to try and undermine both the process and the people. These individuals say things like "I hate this, it's never going to work" or "I'm not going to do it that way and no one can make me." It is best to identify people who may fall into this category as quickly as possible and move them. Find them a new role in the organization or move them to a role where they are more comfortable. Nothing destroys team chemistry and momentum more quickly than a squeaky wheel that you can't fix. There's a pretty good chance that they won't hold up their end of getting work done too. Don't let one or two people ruin the experience for the team.

3. Training and Communication Are Key

When faced with the task of transiting into a new development methodology, most IT resources are faced with some skills learning curve in order to become truly effective team members. For Agile QA teams, the skills adaptation process is aggressive as the time line for adoption is relatively short and the skills required fairly complex. From relearning to communicate to learning new tools, QA teams are asked to reshape their knowledge domain quickly. In order to help facilitate a speedy and successful transition, training is a must.

3.1 Training

One of the most obvious answers to the challenges associated with an Agile transition is training and the most common solution is to hire an Agile coach or a scrum master to the project team. While there may be great Agile coaches and scrum masters out there that can and do understand QA, I have found that few have the experience or the chops in QA to be able to help QA Teams with their transition. If an Agile coach or scrum master is not the answer, then where should a QA Team turn for help and training? In my experience, the best resources have been other QA managers who have already gone through the Agile transition, Agile QA specific forums/blogs, and QA conferences/classes. QA managers who have already gone through an Agile transition, whether successfully or not, can arm you with 2 powerful pieces of information:

- What worked well – tools, tips, processes that helped their team succeed in their transition
- What didn't work well – retrospective advice on what they wish they would have done differently so that their transition would have been smoother

One of the great things about talking to a QA manager who has gone through the Agile transition is that the information you receive from them will directly relate to what you ultimately have to accomplish. Find a resource or two you can tap into and do not worry if the resource you find is not involved in the same business field.

The cost of training QA resources can be a deterrent for many companies not only in dollars spent but also in productivity lost. Some of the best free resources available for QA teams in transition are the Agile QA specific forums and blogs. You'll find a lot of useful perspectives on the Agile QA transition from a large number of resources in all different business spheres and from people in different places in their own transition. It's easy to see the positives from what you read but being focused on the common mistakes can help you avoid the pitfalls others have already experienced. Encourage everyone on your team to research on their own and discuss findings as appropriate in meetings.

When cost is not an issue and you are able to attend or send resources to training, consider QA specific conferences/classes. At most QA conferences, you will find good Agile QA sessions which will give you valuable insight into your transition. Be mindful not to spend your entire training budget on sending everyone to the same conference or class. Instead, make the money go further and get more out of it by designating one resource that will go and report back to the team. If you use this approach, you will be able to have training all throughout your transition and will have an opportunity to address the issues you may be experiencing at the time of the conference/class.

3.2 Communication

Most organizations are either matrixed or siloed and rarely cut across functions. A byproduct of differentiated organizations is that resources within them rarely have the opportunity to communicate cross-functionally. Unless you happen to be in a leadership role, you may never actually have the opportunity to communicate outside your team. Being placed in a cross-functional Agile team where precise communicate is important places most resources at a disadvantage. It's even more difficult when you have multiple Agile teams on a large scale project. To help your QA team "re-learn" to communicate in an Agile team, hold a couple QA only meetings that run like scrum meetings so your QA team has practice at communicating. Here are some meetings you can hold that will help your team communicate more effectively:

- Automation design meeting - allow your technical team members to geek out or use the meeting as an opportunity to address a particularly vexing technical problem.
- Team meeting - all QA team members should meet to discuss what's going on in their projects and how their particular tasks are going for the Sprint.

During these meetings, encourage team members to ask questions. Keep the meetings brief and to the point but share experiences (good and bad) especially solutions. By sharing both good and bad experiences while focusing on solutions, you will avoid the meeting becoming a “gripe” session. Don't be afraid to rotate the job of leadership in your QA Team meetings. Make sure everyone has a chance to lead a meeting so they gain the valuable experience of understanding what types of questions to ask, what information is needed, and how to help solve problems. These skills can and should be practically applied during everyday tasks and honing them is an important part of the Agile skills adaptation.

4. Paired Testing

As difficult as learning communication skills can be, the challenge of adopting technical skills required in Agile QA can be daunting for the majority of QA Teams. For most QA organizations, the team is comprised of two groups: the automation test team and the manual test team. Implicit in this team dynamic is the idea that automation resources are more technical then manual test resources because of their understanding and experience in 3 areas: development language(s), problem solving in technically complex automation frameworks, and integrating and building of automation components. What’s more, automation resources tend to be more comfortable with Agile test tools which lean toward the development side of IT. Based on their understanding and experiences, automation test resources seem naturally fit for an Agile QA transition where whitebox testing, design and code reviews, and paired programming become central tasks for QA resources.

If a good percentage of your typical QA Team was comprised of automation test resources, the Agile QA transition would be fairly smooth. The reality that faces most QA Teams is that manual testers far out-number the automation resources giving rise to the challenge of how best to transition a mostly manual test team to a more technically demanding Agile QA.

One of the great successes I’ve had in the Agile space is the application of the paired-programming technique to the QA team. In paired testing, a manual QA analyst works closes with an automation QA analyst. Here’s why I’ve had such great success using the paired-programming technique within QA: In addition to having a business and operational focus, manual testers can audit the automation to make sure the right things are being tested. Manual testers can also execute one-off, specialized tests much more easily and can provide the automation analyst with precise directions on how to reproduce the steps programmatically. Having concrete steps to automate the most difficult one-off tests can mean the difference between being “done done” at the end of the Sprint or carrying-over to the next Sprint.

If you use paired testing, make sure you rotate pairs regularly and make sure they work together often. Your pairs should work together at least an hour a day and should focus on doing peer reviews, technical reviews, or learning about each other’s projects. Rotating pairs helps avoid the pesticide paradox[6] of testing. Adding a fresh set of eyes and perspective can help avoid missing bugs because testers can begin to have tunnel vision when executing the same tests over and over again. Having to explain the test to someone can also jog important discussions which can lead to evaluating whether or not your testing is as effective as it could be.

One of the most important advantages of paired testing is the collective spread of knowledge amongst your QA team. With the help of the test automation analysts, your manual testers will be able to gain valuable technical expertise which will help them in the technical Agile world. What’s more, your automation analysts will get to hone their qualitative skills by participating in tasks like reviews, audits, and traceability analysis. Honing these skills will allow them to develop more complete technical solutions as they will be more aware of all cases and decisions in their automation.

5. “Done Done” Means Test Automation is Complete

Test Automation is a must and “done done” should include having automated, integrated tests for each work task included in the Sprint. I’ve often observed in other organizations that people say they are Agile even though the test automation is 1 or 2 Sprints behind. I say they aren’t truly Agile and I often debate with my QA brethren this concept of “done done” being defined by having high quality automated QA tests which are part of the Continuous Integration (CI) process at the end of each Sprint. My logic is this: how are you Agile if you are behind? My defense of this idea always brings to bear the following questions.

5.1 How did I come to such a conclusion?

I arrived at this conclusion by thinking through the following:

- Exhaustive Testing is not possible - CTFL 7 QA principles[5] - Anyone who has ever been on a large or complex project knows you can’t manually exhaustively test. What’s more, you can’t expect to leave everything to the end and hope to catch up because you are then slowing down the development process which is contrary to the idea of Agile.

- Immediate feedback is imperative – If your automation is 1 or 2 sprints behind and your manual testing team can’t exhaustively test everything, then it stands to reason that there is a gap in the feedback loop for the areas that manual testing can’t get to in each sprint. Your team may recover the gap in the feedback loop in a later sprint but delaying feedback to your developers works contrary to the idea of Agile where an immediate feedback loop helps develop quality software faster.

- No business wants to spend days, weeks, or months doing Regression Testing at the end - if you have a certain number of requirements, the number of tests grow exponentially with each Sprint. Building a large automation “backlog” doesn’t work in favor of the QA team as they hold up the release and reverse all the delivery gains of the Agile team.

5.2 How do you get the time to build test automation infrastructure?

You can always make time for infrastructure by planning ahead. You should be applying the Agile development principles to the test automation framework development so change is inevitable. Don’t try and craft a solution from the get-go, try and slowly develop it over time. Here are some hints:

- Make resource (tools and personnel) decisions before Sprint 0.

- Make good use of Sprint 0: infrastructure, design, and architecture is important.

- Aggressively hire for automation but avoid the “developers can do QA Automation” last resort.

- Have coding standards and have a tool or a peer review process to make sure standards are being followed. Having standards will allow additional automation resources to slide in and help if needed in a “packed” sprint.

- Collaborate on enterprise framework design and architecture but have an ultimate authority so you don’t get bogged down by unnecessary “perfectionist” tendencies of group think.

- If your budget allows, have an extra automation resource that can float. You will always come across a situation where a sprint will be too heavy for the assigned QA resources so having a spare resource that is plugged into the environment and can lend a hand in a pinch could save the day.

Test automation is too important to leave to chance. Work diligently to plan for the long-term solution by building the foundational pieces one-by-one.

If you fail to automate all of the testing in a sprint, your team must fail the sprint. It sounds harsh to fail a sprint because the test automation isn't complete but holding your team accountable is important. If the developers on your team are required to deliver good, unit-tested code for all the stories included in a sprint, then the QA team should be held to the same standard for its test automation.

6. Customer-Driven

Regardless of which test methodology you use, being customer-driven can lead you to be more successful in identifying whether or not your software is fit for use. Borland recently conducted a webinar called Is Traditional QA Dead? In the webinar, they make the following point: QA is much more than just validating functionality – it's about customer satisfaction.[7] Borland adds to this point by stating that the raise in social media has created greater "transparency" and speed to customer feedback. These trends are putting tremendous pressure on QA teams to be able to predict how and what customers will do with their systems. If your system meets requirements but isn't fit for use, it won't take months, weeks, or even days for feedback to hit the airwaves; it will take a matter of seconds.

So, here are two things to consider as it relates to customer-driven Agile QA:

6.1 Fit for Use vs. Meets Requirements

At a conference for Healthcare software early on in my career, I heard a vendor say the following: if doctors don't use it, it's the same as being broken. I hadn't thought about software in that way before but I realized that designing software for customers who don't use it is just as bad as having buggy software. What I learned is that thinking like the customer drives out more bugs. In Agile, the beauty of acceptance criteria is being able to clearly define what a system should do; the short-coming is in defining what a system should NOT do. Thinking like a costumer often forces you to consider complex decisions and shake out all possible outcomes not just the happy path. A complex decision requires difficult code to be written which means more bugs will cluster in those areas. Also, when you focus on being customer driven, you help the software become more usable and improve the user experience.

6.2 Include Operations in the development process

In my experience, one of the major short-comings of Agile is in the negative impact it has on most organization's Operations teams. Volk's study on comparing Agile at over 200 companies states, "The Agile movement shifts the broad, inter-departmental process of software engineering to one that is focused on software development to the exclusion of QA and operations".[8]

When things go right for a company engaging in Agile, they put software into production quickly and continue to iterate against the product in production. This quick development cycle puts enormous strain on Operations teams from hardware/infrastructure teams, call center/customer support teams, deployment teams, and even supply chain teams. Think of the Operations teams as being consumers of the software as they are directly influenced by the software in one way or another. Engage them early and often in the process so you know how your software decisions are impacting their needs as well as the needs of the user. What's more, Operations teams like the customer support team may have valuable insights into how consumers are using your products. That feedback may help you avoid developing needless features and functionality allowing you to deliver faster.

7. Conclusion

Six years ago my understanding of Agile was based on fear, skepticism, and naiveté and I incorrectly believed it was the end of QA as I knew it. At the time, I believed that nothing anyone in the QA team did would help with an Agile transition. What I've learned in six year of Agile implementations since then is that there are practical keys that can guide QA teams in their journey to evolve into dynamic Agile QA teams. This paper examined some key pivots within an Agile QA transition and offered up advice on how to better manage the process. The following pivots were examined:

- How the psychology of a team's shared experience could help draw the team closer together. By pulling together, a team could face their fears, skepticism, and confusion and turn them into a positive, galvanizing force.

- How training and communication are vital for QA teams in transition. Learning to be an effective member of a cross-functional Agile team requires that QA teams get training and practice what they learn as often as possible.

- How applying the principal of paired-programming to the QA teams can not only help manual tester resources get technically up to speed faster but can also lead to more fully developed, complete automation tests.

- How "done done" in test automation means having fully integrated test automation running within each sprint and not delayed.

- How focusing on being customer-driven will allow QA teams to focus on what matters most: delivering high quality software that's fit for use. You don't want your product to go "viral" for all the wrong reasons so focusing on the user experience and integrating Operations into the testing process is important to quality product delivery.

By exploring the practical keys and understanding how they may be applied, QA teams in transition should be able to focus in on areas of weakness and take necessary steps to improve them.

References

1. Smolaks, Max. "Agile Development Comes With Hidden Costs Warns Report". *The fashionable development practice may be bending IT out of shape.* TechWeek Europe, July 16, 2012. Web http://www.techweekeurope.co.uk/news/agile-development-report-86052

2. Rooney, Dave. "Agile Transitions and Human Nature". *Practical Agility,* January 25, 2012. Blog http://practicalagility.blogspot.com/2012/01/agile-transitions-and-human-nature.html

3. Smolaks, Max. "Agile Development Comes With Hidden Costs Warns Report". *The fashionable development practice may be bending IT out of shape.* TechWeek Europe, July 16, 2012. Web http://www.techweekeurope.co.uk/news/agile-development-report-86052

4. Johnston, Matt. "Testing the Limits with Scott Barber Part I". *Testing the Limits.* UTest, April 26, 2010. Web http://blog.utest.com/testing-the-limits-with-scott-barber-part-i/2010/04/

5. International Software Testing Qualifications Board, 2011. ISTQB_CTFL_Syll_2011. September 2011

6. International Software Testing Qualifications Board, 2011. ISTQB_CTFL_Syll_2011. September 2011

7. Roboostoff, A & Rechberger, G. (2013, July 10). Is Traditional QA Dead? [Webinar]. Web Seminars, A TechWell Event.

8. Smolaks, Max. "Agile Development Comes With Hidden Costs Warns Report". *The fashionable development practice may be bending IT out of shape.* TechWeek Europe, July 16, 2012. Web http://www.techweekeurope.co.uk/news/agile-development-report-86052

Bridging the Gap Between Acceptance Criteria and Definition of Done

Sowmya Purushotham, Amith Pulla

sowmya.sudha@gmail.com, amith.pulla@intel.com

Abstract

With the onset of Scrum and as many organizations are trying to adopt Agile test processes and tools, the test resources on the Agile teams need to help the product owner define Acceptance Criteria during backlog grooming or the Iteration planning process. Acceptance Criteria may include a collection of Story tests that need to be passed for Story acceptance. The Acceptance Criteria and Story tests can be confined to the Story and may not consider all the aspects of the application or product.

The Definition of Done (DoD) is a wider and more comprehensive criterion that needs to be defined and met at a Story, Sprint and Release level in addition to Story acceptance criteria for the new set of features or capabilities to be released to production or users. Story acceptance criteria can include a series of functional tests to ensure the functionality meets the business or user needs, but does not necessarily define all conditions to say the Story is done and no more work needs to be performed by the team. For instance, the acceptance criteria can have list of Story tests or test cases that cover functionality, but a Story DoD can cover non-functional and technical requirements of a User Story like performance testing, browser compatibility, stakeholder acceptance, unit testing, regression testing and automation scripting.

The Scrum team(s) needs to collaboratively define DoDs at Story, Sprint and Release levels. The Story, Sprint and Release has to meet all the conditions defined in the DoDs for the Scrum team(s) to say the Story, Sprint and Release is Done and no further activity is needed.

This paper discusses the need for a clear Definition of Done at Story, Sprint and Release levels and sheds some light on few examples that are common in software applications and products, with a goal to help Agile and Scrum teams to better define their DoDs as a team and improve the overall quality of the application and releases.

Biography

Sowmya Purushotham is a software quality engineer with Clinicient Inc. based in Portland, Oregon. Sowmya has an extensive background in Agile testing practices and tools. As a software quality engineer, Sowmya works with a large global application development team in an Agile environment.

Amith Pulla is a QA Manager at Intel Corp, currently working at the Intel site in Hillsboro, Oregon. Over the last decade, Amith has been involved in developing software testing strategies and processes for applications mainly focused on sales and marketing. Amith also worked on developing test methodologies and techniques to meet the business's needs and timelines. As part of his Agile QA Manager role, Amith focused on improving and refining QA processes and standards for efficient software testing. Amith also worked as ScrumMaster and Project Manager on short assignments.

Amith has a M.S. in CIS from the New Jersey Institute of Technology; Amith got his CSTE and PMP certifications in 2006. Amith got his CSM certification in 2012.

1. Introduction

Acceptance Criteria which are defined by the product owner and then elaborated and further broken down into Story tests by the Scrum development team can be confined to a specific Story. In Scrum, the development team will have developers, testers and others working together to design, build, test, and deliver a product or application. The Acceptance Criteria defined for a Story does not take into account all the other stories, both under development and in production, of the application or product. For instance, if a particular Story under development may impact the other Stories, features or capabilities in the application, certain regression testing and other activities are necessary to make sure this new Story has no negative impact to the product, but works as intended. The acceptance criteria defined for a Story may not cover the regression testing aspects necessary to successfully integrate the new Story into the application, but the Story Definition of Done needs to cover the regression testing aspects.

A Definition of Done (DoD) is typically defined at Story, Iteration or Sprint and Release levels by the entire team, and DoDs are applicable to all Stories, Sprints and Releases. In Agile, “Done” means all the activities necessary to deliver a specific Story are completed. The Story Definition of Done captures all the conditions that needs to be met, once DoD is met it means no further tasks or activities are needed for that Story. A DoD looks at all aspects of the application or product and defines specific conditions that need to be met for the team to say the Story is Done. This may include regression and performance testing of the entire application, cross browser testing, test automation scripting, user acceptance, end user training or horizontal capabilities like BI.

In this paper, we’ll discuss Acceptance Criteria and Definition of Done with a few examples, illustrating the importance of defining, documenting and agreeing on DoD. We’ll offer some guidance on what to look for when defining Definition of Done for a Story, Iteration and Release.

2. Acceptance Criteria

2.1 Definition

Acceptance Criteria are defined as conditions that a User Story must satisfy to be accepted by a user, customer or other stakeholder.

In an Agile process, product owners write User Stories and discuss them with development team. The User Story is presented and a review process is started. The team collectively decides on the Acceptance Criteria and further elaborates into specific Story tests like functional test cases with Pass/Fail conditions. Acceptance Criteria can be a set of statements or conditions describing both functional and non-functional aspects of the product or application. Typically, there is no partial acceptance of a specific condition; it is either satisfied or not satisfied.

Depending on the product and application under development, Acceptance Criteria can also define boundaries and constraints of the User Story that determine whether the Story is working as expected and ready to be accepted. Acceptance Criteria needs to be defined before the Iteration and development on the Story starts, but can be refined during the Sprint based on the product owner’s feedback. A User Story in itself usually doesn’t provide enough details on the functionality and feature to the development team to start designing and coding; developers use the User Story and Acceptance Criteria to start any development work.

Generally in Scrum, Acceptance Criteria are discussed and defined as part of the product backlog grooming sessions where the whole team meets including the product owner and ScrumMaster. Sometimes backlog grooming sessions are also called Story Time sessions. A product backlog grooming session is not a formal Scrum ceremony like Sprint Planning or Daily Stand Up (DSU). Ken Schwaber, one of the founders of Scrum, recommends that teams dedicate five percent of every Sprint to this process (Scrum Backlog Grooming, CollabNet). There are other activities that happen during the backlog grooming process like ordering (prioritization) and estimation, but defining Acceptance Criteria is a key activity.

Acceptance Criteria needs to be clearly written in a language that customers, product owners, and the development team can easily understand. In Scrum, development team is the cross-functional team that has developers, testers, QA, UI designers, test automation experts and other roles. When defining Acceptance Criteria, there should not be any ambiguity on the expected outcome and expected behaviors. Well written Acceptance Criteria should be easily translated into one or more manual or automated Story tests.

2.2 An Example

Here is an example of a User Story:

As a User (Account Manager), using the letters feature, the User should be able to fax letters to a contact directly from the system instead of printing them out and manually send fax to the contact.

Questions and discussion for the product owner by the development team may include:

> Will the User be able to fax multiple letters from the application to same contact?
> Where does the User choose content to fax?
> Can the User edit a letter for faxing?
> Does the User need to manually enter fax number?

The issues and ideas raised in this session are captured in the Story's Acceptance Criteria.

The Acceptance Criteria could include:

> User is able to see a 'Fax' button under Actions > Create a Letter menu
> User should be able to choose a Client, Account and Contact
> User can select a letter template and edit as needed.
> User needs to click 'Fax' button to fax a letter.
> User should see the fax chart notes pop up on clicking 'fax' button to choose a Fax cover page.

2.3 Story Acceptance

In Scrum, when the development team has completed the development and testing of the Story, the product features and functionality that are part of the User Story are demonstrated to the product owner showing how each item in the Acceptance Criteria has been satisfied. The product owner accepts the Story by verbal approval or by updating the project management, Agile Lifecycle Management (ALM) or QA tool such as Rally, Quality Center, RTC or VersionOne.

Typically, Acceptance Criteria is written as part of the Story in ALM tools; ALM tools have indicators that will show if a Story is accepted or not and who accepted it.

3. Definition Of Done

The "Definition of Done" (DoD) is the project or team's agreement stating that all the tasks or activities are completed for a Story, Sprint or a Release. It is a set of common conditions across all Stories, Sprints and Releases that states that no more work is left to be done for a Story, Sprint or Release. The primary purpose of a good DoD is for the Stakeholders and teams to better understand when a product increment can be called "Done." The DoD makes sure that the team adheres to a shared understanding of completion. Unlike Acceptance Criteria, Definition of Done is not specific to a Story or a feature, but defined such a way that it's applicable to all Stories. It also defined at a Sprint and Release level and is applicable to all Sprints and Releases that the team works on.

The DoD can also serve as a contract between a development team and its stakeholders. It can be seen as the team's standard of excellence or quality. The Definition of Done is defined and agreed upon as a collaborative effort between the ScrumMaster, development team, product owner and other stakeholders. The Scrum Guide (Ken Schwaber, Jeff Sutherland 2011) describes the Definition of Done (DoD) "as a tool for bringing transparency to the work a Scrum Team is performing." The Definition of Done is more about the quality of an overall product and user experience and less about the functionality and features. Typically, the Definition of Done will be written and reviewed before the first Iteration or Sprint starts, but it is regularly revisited, reviewed and updated in Retrospectives to meet customer, product owner and quality expectations.

Like the Acceptance Criteria, the Definition of Done needs to be clearly written in simple language that the development team, product owner and stakeholders can clearly understand. The Definition of Done should be written as a set of conditions like a checklist that the team can check off using the known information in the project management or tracking tools. The Definition of Done is usually posted in a prominent place in the online collabration and project management tools so that everyone on the team sees it on a regular basis and fully understands all the conditions. Having a clear DoD can foster team collaboration, common and consistent development practices, as well as better visibility for customers which leads to better overall product quality.

The book "Agile Testing: A Practical Guide for Testers and Agile Teams" talks about the importance of technology-facing tests in addition to business-facing tests. It says "Technology-facing and business-facing tests that drive development are central to agile development." (Lisa Crispin, Janet Gregory, 2009). When defining good Definition of Done statements at a Story level, the team needs to consider both technology-facing and business-facing tests that are not already defined in the Story Acceptance Criteria and add them to Story DoD as needed.

Some suggestions on writing good DoDs: (Rally Publications, 2013)

- Use conditions like, "all code checked in" or "unit test coverage > 80%".
- Use "Code review completed" instead of "code review".

3.1 Technical Debt and Quality Risk Considerations

A good Definition of Done is important for the velocity of the agile team and the quality of the delivered product. A poorly defined or incomplete Definition of Done can lead to gaps and defects in potentially-shippable software as development practices vary from Story to Story. The team needs to decide whether an activity or condition belongs in the DoD for a Story, a Sprint or a Release. As we move the conditions from a Story to Release level, the team is temporarily creating technical debt and adding to the risk. The team should try to keep as many conditions or activities as possible at the Story level and move them up to Sprint or Release level only if it's inefficient to do it at Story level. There are many factors that will influence these decisions. The team should ask themselves if a task can be done at the Story level, and if not, move it to the Sprint DoD. If it cannot be done at the Sprint level, it must be done at the Release level and needs to be added it to Release DoD.

For instance, if the teams want test automation scripting as part of the Iteration or Sprint Definition of Done, the team should consider whether test automation scripting can be added to Story DoD and completed with the Story. This means test automation scripting is removed from Sprint DoD and added to Story DoD, so that team completes test automation scripting before they call each Story done.

Performance or load testing can be a good example too. Performance or load testing can be added to Story DoD, but if it's inefficient or expensive for the team to run performance or load testing for each Story individually, the performance or load testing can be moved from Story DoD to Sprint DoD. This allows the team to run a single performance or load testing cycle towards the end of each Sprint on multiple Stories developed in that Sprint.

As part of defining the Definition of Done, the team should consider and discuss all the impediments that could prevent them from meeting the DoD. Based on the impediments, conditions can be moved from Story DoD to Sprint or Release DoD.

3.2 Examples of Definition of Done

3.2.1. For a Story:

- Code Completed and Reviewed
    - Code is refactored (to support new functionality)
- Code Checked-In and Built without Error
- Unit Tests Written and Passing
- Release Configuration Documentation Completed (if Applicable)
- Acceptance Tests written and Passing
- Pass all Non-Functional Requirements if Applicable (Cross browser compatibility tier 1, 2)
- Product Owner Sign Off /Acceptance
- User Acceptance
- Manual regression scripts updated
- Test Automation Scripts Created and integrated
- Localization (truncation, wrapping, line height issues, string array issues, etc.)
- Analytics (Non-Functional Requirements) integrated and tested
- Story level device support (big browser, tablet, mobile device) tested

3.2.2. For Iteration:

- Unit Test Code Coverage >80%
- Passed Regression Testing
- Passed Performance Tests (Where Applicable)
- End user training team hand-off
- UAT (User Acceptance Testing)
- Production Support Knowledge Transfer done

3.2.3. For a Release:

- Regression tests completed in an integrated environment
- Performance or Load Tests completed
- Open defects reviewed by stakeholders and production support team
- Workarounds documented
- UAT and end user training completed

3.3 Definition of Done for Enterprise-Class Agile Development

Many Agile teams today in large enterprise companies work in Scrum of Scrums setup or adopt an Enterprise-class Agile development method like SAFe (Scaled Agile Framework) or DAD (Disciplined Agile Delivery). In the Scrum of Scrums setup, usually there are two or more Scrum teams working on a common platform or different aspects of the same product. The teams need to adhere to common enterprise architecture, release cadence, UX design and platform constraints.

When multiple Scrum teams are working together, it’s important that the Definition of Done is same for each of the Scrum teams involved to avoid inconsistencies in the development and testing practices and quality of the application or product. Generally in Scrum of Scrums set up, multiple Scrum teams work in the environment as separate Scrum teams but release the code on a same platform and in same cadence; this means the Release DoD must be shared by all the Scrum teams in the Scrum of Scrums setup.

3.4 SAFe Definition of Done Example
(Scaled Agile Framework, Dean Leffingwell, 2013)

This is an example of Definition of Done from Scaled Agile Framework (SAFe), one of the most popular Enterprise-Class Agile Development methods adopted in the software development industry today. In this example, the Story DoD is defined at Story, Feature and Releasable Feature Set.

All items defined for Story DoD needs to be met before the Story is declared Done and then it will be considered ready to be added to a Feature. In the same way, all items defined for Feature DoD needs to be met before the Feature is declared Done and only then it will be added to Releasable Feature Set.

Story	Feature	Releasable Feature Set
Acceptance criteria met	A stories for the feature done	All features for the releasable set are done
Story acceptance tests written and passed (automated where practical)	Code deployed to QA and integration tested	End-to-end Integration and system testing done
Nonfunctional requirements met	Functional regression testing complete	Full regression testing done
Unit tests coded, passed and included in the Build Verification Tests (BVT)	No must-fix or Showstopper defects	Exploratory testing done
Cumulative unit tests passed	Nonfunctional requirements met	No must-fix defects
Code checked in and merged into mainline	Feature included in build definition and deployment process	End-to-end system, performance and load testing done
All integration conflicts resolved and BVT passed	Feature documentation complete	User, release, installation, and other documentation complete
Coding standards followed	Feature accepted by Product Owner or Product Manager	Localization and/or internationalization updated
Code peer reviewed		Feature set accepted by Product Management
No Showstopper or must-fix defects open		
Story accepted by the Product Owner		

3.5 Meeting the Definition of Done

Stories are marked or stated as Done by the team anytime during the Sprint or Iteration when all the items in the Story DoD are satisfied.

For Sprint DoD, at the end of each Sprint the development team conducts a Sprint review attended by the whole team. During the Sprint review the team reviews the Sprint DoD and checks off the Sprint DoD list.

The Release is stated as Done once all the activities to deploy the code to production are completed and new features are ready for the Users. The Story, Sprint or Release is not done until all activities and conditions are met and satisfied.

4. Bridging the Gap

The QA Leads and Test Engineers in Agile teams play an important role in bridging the gap between the Acceptance Criteria and Definition of Done. QA resources, with their extensive background in product quality and testing, can help their teams define the optimal Definition of Done that will help the team build higher quality software.

Below are the items to consider for the teams when discussing the Definition of Done.

4.1 Horizontal Capabilities

Every product or application has some horizontal capabilities. This is especially true for enterprise applications. Below we'll discuss few examples.

4.1.1. BI (Business Intelligence) and Analytics

In today's applications or products, BI and Analytics are an important aspect. For an application, the analytics may be tracking site usage, page views, user behaviors or transactional data. For a web application, each page can have some analytics code that sends data back to a central repository to be used for BI. In a client or mobile native app, generally there are several built in reports and transactional data tracking tools that are used for system analytics.

4.1.2 UI (User Interface) Design

If the team is building a large product or application that has multiple components or sub-sites, it's important to have a consistent UI design and UX (User Experience) across the site. Creating a common Definition of Done at a Story or feature level for all stories within a product can ensure that all teams use the same UI design patterns and guidelines.

4.1.3 Training and Help Files

In enterprise applications, the end user training material and documentation is usually created by a separate team within the IT organization. The training and help documentation must be updated for every production Release. Capturing this activity in a Sprint or Release DoD ensures that users have the most up to date documentation and training available after each Release.

4.2 Agile Engineering Practices

For Agile teams, it's important to adopt Agile engineering or development practices that will enable the team to have stable velocity while meeting the quality standards. Engineering practices need to use common tools and frameworks for greater efficiency and tracking. Typically, the engineering practices are captured in DoDs.

4.2.1. Unit Testing and Code Reviews

A unit test is a piece of code written and maintained by the developers that exercises other areas of the code and checks the behavior. The result of a unit test is primarily binary, either pass or fail. Developers write a large number of unit tests based on the functions and methods in the code. Unit tests are usually automated and can be run frequently as the code changes. Unit testing frameworks are used to write, maintain and execute the unit tests in an application. Unit tests are written as part of development or coding and this activity is typically part of Story DoD. NUnit and JUnit are two popular tools for unit testing. Some Agile teams refer to unit testing as developer testing.

Code reviews and pair-programming are popular in Agile development teams; many Agile experts believe that effective use of code reviews and pair-programming can improve the overall quality of the application by identifying and eliminating defects during the coding or development phase itself. The book "How Google Tests Software" emphasizes the importance of code reviews in the development process. The book says "Google centers its development process on code reviews. There is far more fanfare around reviewing code than there is about writing it" (James A. Whittaker, Jason Arbon, Jeff Carollo, 2012).

4.2.2 Test Automation

Building a robust test automation framework is critical for Agile teams to deal with regression issues. Test automation tools are used to automate functional, integration and system tests. Automated regression tests can find issues faster and help the team move faster by saving time spent in manual testing. Automated tests can be created as soon as the Story is ready for testing. The team can keep this activity as part of the Story DoD.

4.2.3 Continuous Integration (CI)

Test Automation in and of itself doesn't offer much value to Agile teams if the tests are not run frequently to find defects. A Continuous Integration model allows teams to check in code and relevant automated tests frequently, sometimes multiple times a day. Once the new code is checked in and a working build is created, the automated regression tests can run on the build and find the defects faster. As soon as the Story and a relevant test automation scripts are developed, they can be checked into the CI model. Again, this activity can be tracked in the Story DoD. The Story is not done until the automated tests are created and added to the CI and regression tests are run without any failures.

4.2.4 Test-Driven Development (TDD)

Test-driven development (TDD) is a software development process in which the developer writes an automated test case that states the desired behavior; the test should be initially failing. Then the developer writes the minimum amount of code to pass that test. The new code should be refactored to acceptable coding standards. TDD, which is intended to build the code right, as an engineering practice is encouraged in some Agile circles for better design and improved quality.

4.2.5 Acceptance Test Driven Development (ATDD)

ATDD (build the right code) is a complete paradigm shift from other Agile software development practices. In ATDD, developers build the application code based on User Stories and acceptance tests and automated tests are run on them to capture feedback from Users and product owners as the development is still in process. The automated tests are defined by product owners and Users using a WiKi mechanism, and then an ATDD framework like FitNesse or Cucumber is used to integrate the tests to working code using fixture code. ATDD integrates developers, testers and product owners and Users into the development effort in a kind of forced collabration. ATDD encourages team's acceptance of quality as everyone's responsibility.

4.2.6 Automated Deployments

Automated deployment or infrastructure automation is an emerging engineering practice in Agile teams and is considered a key aspect of DevOps. DevOps is a set of practices to improve collaboration and alignment between development and operations. Automated deployments allow teams to push working software or shippable increments to production in a matter of minutes. Automated deployments use frameworks and tools that can deploy and test code on servers in a fraction of time compared to traditional deployment methods; sometime these tools can deploy code to multiple severs simultaneously. These tools integrate with version control systems and the team needs to change the configuration and setup to work with a planned production release. The setup and configuration activity can be tied to the Release DoD.

4.3 Integration Testing

In large scale enterprise applications, there are multiple platforms and applications in the ecosystem that need to communicate with each other to bring the right data and experience to the Users. The application may need to communicate with several common enterprise User authentication, Business Intelligence (BI) and security systems. Integration testing ensures that all the integration points and data flows are happening as expected. Integration testing needs to be done for each release. This activity can be tracked as part of the Release DoD.

4.4 Performance and Load Testing

Performance and load testing are important aspects of testing that ensure the application meets the performance expectations of the Users under typical production load. It's not always efficient to run performance tests for every Story, but they should be run for key features to ensure that response times are below the expected thresholds as defined in the Acceptance Criteria. Performance tests can be run at the end of each Sprint when a set of stories are completed by the development team. Performance testing activity can be part of Sprint Definition of Done.

4.5 Mobile Device Compatibility (if applicable)

In today's mobile era, the application functionality often needs to be delivered not only for traditional form factors (desktops and laptops) but also for mobile form factors (tablets and phones). The Story DoD can capture activities that can enable a product's features for mobile devices.

Any mobile development and testing activities that are common for all Stories and features for a product can be added to Story or feature DoDs.

5. Conclusion

Acceptance Criteria and Definition of Done are two important aspects of Agile development that will help teams deliver quality to the users or customers. Teams should invest time and collaborate to define and document these and make them accessible and visible to the overall team including stakeholders, development team members and management.

The QA team members or Testers on Agile teams, with their knowledge of product quality and experience in developing test strategies, can help the team define and implement good Definition of Done conditions at Story, Sprint and Release levels, bridging the gap between Acceptance criteria and Definition of Done.

References

Lisa Crispin and Janet Gregory. 2009. Agile Testing: A Practical Guide for Testers and Agile Teams

James A. Whittaker, Jason Arbon and Jeff Carollo. 2012. How Google Tests Software

Rally Publications. 2013. Agile Definition of Done Guide
https://www.rallydev.com/sites/default/files/defining_done_guide.pdf

Dean Leffingwell. 2013. Scaled Agile Framework
http://scaledagileframework.com/

Ken Schwaber and Jeff Sutherland. October 2011. The Scrum Guide

CollabNet. Scrum Backlog Grooming.
http://scrummethodology.com/scrum-backlog-grooming/

Turning a Marathon Runner into a Sprinter:

Adopting Agile Testing Strategies and Practices at Microsoft

Jean Hartmann

Test Architect
Office Client Services
Application and Services Group
Microsoft Corp.
Redmond, WA 98052
Tel: 425 705 9062
Email: jeanhar@microsoft.com

Abstract

For many years now, Office has successfully released its Productivity Suite including Word, Excel, PowerPoint and Outlook with a regular cadence of about three years. These major releases have relied on a substantial amount of testing to ensure high-quality and compliant products are shipped worldwide. However, with future releases, the client division is moving to a more agile model for software development, which has major ramifications for software quality and testing. The move is compounded by the need to also deliver these products on a wider variety of devices and platforms. This paper will highlight improvements such as greater emphasis on developer unit testing for early bug detection, smarter regression test selection to enable faster, higher quality deployments and streamlining of mandatory ship criteria and improved product telemetry for evaluating and assessing test automation quality.

Biography

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

1. Introduction

A waterfall-based, endurance-like approach to software development has worked well over the years for teams developing Office desktop products like Word, Excel, PowerPoint and Outlook. Going forward, however, teams did not want their customers to have to wait for several years before receiving the latest set of features and enhancements. Instead, our teams wanted to deliver a steady stream of new features at shorter time intervals or sprints as well as across a broad range of platforms and devices. It was time for these marathon runners to transform themselves into sprinters!

To achieve this goal and successfully make the transition to an agile world, teams realized that they needed to strike a careful balance [1]. They needed to embrace the best aspects of agile (and cross-platform development) and then blend those with best practices that already existed within Office.

Planning teams were assembled and tasked to explore key themes. These would need to be promoted, developed and supported by the organization as it geared up for a fresh round of development. In some cases, the topics and investments were obvious with ROI (Return On Investment) data available from within and outside the company to demonstrate their benefits. In other cases, the decisions to invest in a theme were based more on engineering intuition and experience, especially when empirical data was not readily available. A good example of the latter was the broad adoption of unit testing practices. Much anecdotal evidence existed within and outside the company regarding unit testing, but little ROI data was available regarding its cost vs. benefit.

Key test- or quality-related themes included:

- *Moving quality upstream.* We anticipated that by supporting a better developer (unit) testing experience during refactoring and componentizing of existing products, bugs would be caught earlier, reducing defect leakage and maintaining quality. Such work would also be a cornerstone of efforts to improve code velocity and deployment in the context of continuous integration.

- *Better leveraging product telemetry.* Collecting and analyzing product telemetry data for key attributes, such as bugs, performance, reliability, security and network latencies, etc. represents a powerful mechanism to continuously improve on product quality and can also be leveraged to better assess, guide and influence process improvements, such as the quality and scope of teams' test automation.

- *Adapting cross-platform testing for agile development.* With Office products sharing a number of common components across different platforms and devices, it was important to improve agility in the context of cross-platform development. Rationalizing resources around validating those components via a single platform-agnostic test suite per product rather than multiple, platform-specific test suite became important.

- *Streamlining mandatory shipping criteria - Quality Essentials (QE) - including security, accessibility and localization testing in the face of faster release cadences*. In previous ship cycles, such non-functional testing required substantial resources and effort to satisfy. Through clear definitions of deployment quality gates (so-called “rings of validation”) and better integration of associated work items into the agile process, the additional effort incurred was intended to be kept to a minimum.

- *Ensuring rock-solid tools support and integration*. It was clear that shifts in product development philosophy and cadence could not be accomplished without good tools support and proper integration into the new agile development process. As a result, the central engineering and tools team needed to quickly evaluate their existing tools and delivery mechanisms, to gather requirements and close outstanding gaps.

The remainder of this paper summarizes our ongoing efforts with respect to each of these themes.

2. Emphasizing Developer Testing

Many years of development effort have resulted in large, monolithic code bases for classic Office products. So the need to refactor and componentize these products was steadily growing. The broad adoption of developer (unit or component) testing was seen as one way of ensuring product quality was maintained during refactoring and componentization. It also provided the benefit of early bug detection, reducing defect leakage. Developers were able to implement and validate their classes or components quickly and frequently and as early as possible.

Apart from driving quality improvements upstream, developers needed confidence with integrating and deploying their code into production faster. Continuous integration became a key issue. This required more extensive testing, against a wider set of integration and system criteria than just unit or component tests. At the same time, check-in times needed to stay reasonable. Smarter regression test selection criteria were developed to address this issue and are currently being evaluated.

2.1 Introducing Unit/Component Testing

While most Office developers have not (yet) embraced true, agile testing practices such as TDD [2] – they still write the code first, then create the tests to validate it – they are making strides in developing automated unit test suites. Developers have rapidly accumulated hundreds of thousands of unit tests to validate core functionality and are continuously improving their levels of coverage.

Rather than rigidly adhering to the classic definition of unit testing that prescribes validating every class and its methods, developers have the flexibility to validate larger chunks of code, namely components or sub-systems. The decision is largely based on the developers’ assessment of the code, its complexity and dependencies.

When writing unit/component tests, developers typically produced C++ code, ensuring that it was portable across multiple platforms (compilable and executable using platform-specific compilers). Unlike system or scenario-based tests, these test suites were executed as part of the same execution process as the product (or “in-proc”), making their execution very fast and efficient; and more importantly, enabling quick and easy debugging of the product.

In some teams, developers and testers shared the development of component tests. The anticipated benefits for testers included getting a deeper (white-box) understanding of the product code, its algorithms, data structures and event models. It also gave testers the opportunity to improve product testability by adding appropriate tests hook that made it easier to verify the products. Apart from developers' workloads being reduced, this collaboration also represented a source of independent verification and validation for their designs and implementations. From an agility point of view, having testers more informed about product code intent and purpose resulted in deeper, more meaningful testing.

Our initial belief that unit testing would be costly in terms of mocking and stubbing due to the large number of potentially complex dependencies of legacy code was groundless, although in many cases this may have been due to developers sidestepping the issue by validating at the component level instead.

As this unit testing initiative is still in progress, its benefits and ROI (additional bugs found vs. effort invested) are still being evaluated. It is too early to draw any definitive conclusions about the broader, long-termer benefits. Having said that, more immediate benefits, such as leveraging unit tests to help developers correctly resolve merge conflicts, have been reported. Also, the stability of some products *appears* to be improving with system-level regression test runs passing at higher rates than in the past. However, at the time, we have not been able to correlate unit testing efforts with the latter observation.

2.2 Smarter Regression Test Selection

Smarter regression test selection techniques and criteria enable developers to validate code changes quicker and with more confidence, enabling faster deployment to production. The aim is to give them ways to more effectively and efficiently validate code by selecting an appropriate subset of tests for rerun, instead of having to execute all tests [3]. The techniques enable developers to perform *pre-checkin validation* themselves rather than relying upon *post-checkin validation* efforts conducted by testers. In the former case, regression testing acts as a powerful gatekeeper mechanism to raise developers' confidence in selecting and running the most appropriate test automation from amongst a team's unit, component and system tests as well as informing them about code changes that coincide with sections of code, not previously tested. Selecting from a range of different types of test cases ensures that the code is being exercised in the broadest possible context. Continuous integration and deployment now becomes a more reliable, predictable and measurable process.

At this time, smarter regression test selection for developer check-ins has just been introduced. Its benefits and ROI (additional bugs found vs. tests selected for rerun) are still being evaluated. Based on results achieved by these techniques during post-checkin validation, they will provide significant, additional value.

2.3 Improving Product Testability

As part of the drive towards refactoring and componentization of the various products, developers and testers also had an opportunity to collaborate to improve the testability of their product code. Better testability focuses on improving controllability and observability of a product and is typically achieved by exposing more of the internal behavior and properties of the product via a set of application interfaces. As a result, developers and testers can, for example, place their product into specific states or add a specific data value to a document property (set). Conversely, they can easily observe state or document property changes (get).

Improving product testability helped developers, making their unit and component testing efficient and effective. In addition, testers benefitted as they could now significantly reduce their reliance on user interface (UI) testing and instead, access the equivalent application logic[1]. The resulting component and system tests executed faster, proved to be more reliable and exposed deeper, more fundamental flaws in the application logic. After seeing the early benefits of testability, developers have now become accustomed to exposing internal product details.

3. Adapting Cross-Platform Testing

In a previous PNSQC paper, the topic of cross-platform testing at Microsoft was described in detail [4]. The use of this approach was thought to be particularly compelling, when a significant portion of core product code could be shared across multiple platforms as in the case of many Office applications. It was also anticipated that it could contribute to our agile development practices as a single platform-agnostic test suite would replace multiple, platform-specific ones. A good example were the cross-platform File I/O or file fuzzing activities where Office applications need to be validated against multiple platforms and devices using a common, shared scenario test. Previously, multiple tests were written; now the same goal can be achieved by one, shared test!

Several Office teams recognized the potential for leveraging such cross-platform test suites and are starting to adapt the approach described in the paper. They are currently authoring new shared, desktop-based C# tests, which in turn validate their applications on different platforms and devices. They are, in effect, future-proofing their test investments and improving their agility in terms of being able to quickly validate shared, common code on new platforms and devices, as these arrive in the marketplace.

Having a single test entry point (PC desktop) also enables teams to leverage existing, mature and very useful tools for the same application on different platforms, making the ROI even more compelling. A good example of this are in-house fuzzing tools, used on the PC for years, but not available natively for other platforms. Now, the generated tests can also drive the same application on those non-PC devices, improving agility and quality!

4. Leveraging Telemetry

Product telemetry enables developers and testers to add additional code to their products in order to track various quality attributes and ensure minimal deployment downtime for our applications. Telemetry typically focuses on collecting data to ensure a better customer experience. In the past, there was little telemetry

[1] The limitations of UI testing are well-documented, including lack of test stability and performance due to frequent changes in UI layout during development and potential masking of bugs by the UI, which were generated within the application logic, but were never ‘visible’ in the UI.

gathered to give better visibility into customer issues. It also took a considerable amount of time for the organization to react, namely provide bug fixes, in response to issues. One of the few forms of product telemetry available to Office was delivered courtesy of the Windows operating system. The Windows Error Reporting (or Dr. Watson) service reported various operating system and application errors and crashes [5].

With the advent of online services such as Exchange, SharePoint and now Office365, this has changed. By extensively instrumenting these services with telemetry points and monitoring their functional and non-functional behavior, Office online services teams are able to resolve problems and deploy fixes much faster than their Office counterparts on client platforms and devices. This is, however, changing as those client teams are also intending to leverage telemetry as part of their improvement efforts. They are essentially using the same backend infrastructure to collect telemetry data as their services counterparts, but developing a unified logging mechanism that can be used across various platforms and devices, browser-based or native applications. The result is the development of a unified telemetry infrastructure where products spanning client platforms and devices as well as servers can be analyzed.

Product telemetry is an important consideration when moving to an agile development philosophy as products and services are being continuously integrated, built and deployed. While developers and testers do a good job validating main product features and workflows in any given sprint, they also need more visibility into the product to minimize downtime when something does go wrong. Beyond debugging, the question arises – can this product telemetry data also be used to assess, guide and influence the need for test automation?

5. Integrating Release Criteria

Ship or release criteria represent an extensive list of non-functional quality criteria that need to be satisfied by every team within Microsoft before deploying a product or service to external customers. Examples of such criterion include accessibility, security, privacy, world-readiness and localization [6].

In the past, teams spent considerable development and testing resources on satisfying these criteria. They would often spend dedicated time after completing their functional features to validate against these non-functional criteria, reworking designs to satisfy them. In each product release, a significant portion of the total number of work items and bugs would be attributed to such work.

With the shift to agile development methodologies, test teams no longer have the luxury of validating and satisfying these criteria after completing their features. The challenge here is to spread the required testing effort across the new agile development cycle without lengthening it. The hope is that better techniques, tooling and procedures can help. For example, localization requirements can be pushed upstream, so that developers consider them as part of their early design discussions and planning. Tools could be deployed to perform certain checks at code check-in time rather than running them later on entire products. In the past, testers often verified these criteria during the later phases of development, for example, by performing accessibility testing. The intent of this initiative is to revisit the existing set of criteria and more effectively integrate specific aspects into the agile development workflow. The benefit is that teams satisfy the criteria as an integral part of their daily workload rather than as a distinct and separate workload after the product functionality is already completed and is ready to be deployed.

6. Streamlining of Engineering Tools and Services

An important and yet often overlooked consideration when moving to an agile development methodology is how the engineering tools and services can support this transition. Key technological improvements, such as continuous build, virtualized test and debugging improvements, need to be adopted and integrated into the new agile development processes. Improvements to test automation were particularly important to developers and testers who needed to frequently run their respective automated test suites and debug products. They needed fast and reliable tests and test results that detected product bugs rather than highlighting test failures. In the past, they were often hampered by slow test execution and unreliable test results. In order to remedy this situation, the engineering team initiated efforts to better measure and improve the stability of the test collateral to achieve higher pass rates. To complement that, test execution leveraged virtual rather than physical machines, which significantly reduced set up times and helped both debugging and test throughput. Whenever tests failed and product needed to be analyzed, the virtual machines (VMs) containing the buggy product could be saved elsewhere and debugged at leisure with the hardware being reclaimed for further test runs. Average test job turnaround times have been reduced by up to 50%. As part of these improvements, the engineering team also promised more visibility into their services including timeliness and reliability by developing better “dashboards”.

7. Conclusion

In this paper, I have highlighted key areas of improvement to our development and testing processes as Office teams move away from a waterfall-based, endurance-like model to embrace a more agile, sprint-like model. While I discuss important aspects of each area, work is ongoing and the verdict is still not out as to their ROI. Having said that, teams are starting to reap some of the early benefits.

8. Acknowledgments

I would like to thank Marty Riley, Principal Test Manager, Curtis Anderson, Director of Test and Tara Roth, Corporate VP of Office Client Services for their ongoing support. I also want to express my gratitude to all Office team members for their contributions, support and discussions concerning this work. Thanks also go to my reviewers whose invaluable feedback and comments are greatly appreciated.

9. References

1. K. Sureshchandra, J. Shrinivasavadhani, "*Moving from Waterfall to Agile*", IEEE Agile '08 Conference, pp. 97-101, Aug. 2008.
2. K. Beck, "Test Driven Development: By Example", Addison-Wesley Longman, 2002.
3. J. Hartmann, "Thirty Years of Regression Testing", PNSQC 2012.
4. J. Hartmann, "Exploring Cross-Platform Strategies at Microsoft", PNSQC 2011.
5. K. Foster, "Windows Error Reporting: Elementary, My Dear Watson", www.tabpi.org/2006/rch3.pdf
6. Introduction to the Microsoft Security Development Lifecycle (SDLC),link

Agile Test Automation: Transition Challenges and Ways to Overcome Them

Venkat Moncompu

venkataraman.moncompu@cognizant.com

Abstract

Agile development is now becoming mainstream, recent Gartner research highlighted last year 21% of development projects were based on agile methodologies. Though the adoption of agile is burgeoning, the fundamental challenges in adopting agile testing methodology remain. Nuances in Agile transition at an organization and process level are constraining the desired business outcomes.

A survey conducted by a leading tools provider mentions the two major resistances to agile adoption are lack of management support (27%) and company culture (26%). One of the prime reasons of failure at a project level is the inability to integrate test automation in an agile lifecycle. The traditional practice of automation towards the end needs a paradigm shift to an integrated and incremental approach across the lifecycle.

In this session, I will share my experience with teams making the agile transition and talk about technical and non-technical challenges at different organization levels. Discuss the Agile based rethink approach for time-to-market, cost reduction and increasing ROI. I will also share techniques around transparency and project visibility, waste elimination, technical debt management, continuous feedback, just-in-time decision-making for test automation, and what worked well and some that didn't.

I conclude with how these were successfully applied to transforming the test automation practice in some organizations of different sizes, levels of maturity and cultures.

Author: Venkat is an agile testing evangelist and enthusiast with 16 years of industry experience. Over the past few years, he has worked with teams that have mastered the agile software delivery as well as teams that have struggled through the painful process of adoption until abandoning agile development methodology altogether.

Currently, Venkat is a director in the automated testing practice within Cognizant Technology Solutions' test Center-of-excellence. He has a master's degree in engineering from Arizona State University and has participated in technology and quality conferences as a presenter and speaker - Software Quality Days 2013, PNSQC 2012, Rational VoiCE Conference 2011. He has also published in the Software Test & Performance magazine and Stickyminds in 2008.

1. Introduction

Organizational changes are some of the hardest to come by. Deep rooted culture, habituated practices and a deep sense of resistance to changes are some of the biggest obstacles to overcome. In the context of agile software development, these are amplified as the people part of the equation of people, process and technology simply lack proper guidance in their adoption to begin with and the perseverance to make the transition.

Given the significant advances in technology to enable agile development practices, their contexts of use are sometimes lost in the ways teams have adopted them in their projects. Prior to the advent and popularity of agile practices, engineering heavy practices such as architectural milestones were emphasized, accentuating the waterfall type of staged development with rich ceremonies of review and base lining. The processes were also supportive of the command and control leadership and encouraged a rich set of metrics that were rarely predictive of team dynamics and goals of the project.

In his book the agile survival guide, Michael Sahota talks about the ills that are plaguing the agile software development movement – change agents talk about adopting agile rather than transforming the culture of the organization to an agile mindset (Sahota, 2012). Adoption might be easier in large organizations, but transforming cultures to scale is a significant challenge altogether.

In the following sections, I will share some of my experiences as a tester and quality professional. I have faced challenges as a member making the transition to agile methodology and in trying to be an agent of change in larger organizations. I will also narrate some of the cultural challenges to self-organizing teams that span cultural boundaries as off-shore teams attempted these transformations.

2. Challenges of Engaging the Business

One of the benefits of agile principles is better alignment with business and engaging business throughout the development period. The adoption of continuous integration practices using unit tests provide guidance to the development and automated acceptance tests give an indication of 'Doneness' for the stories under development. The difficulty of customers and business to engage with development is that customers find it difficult to find time in joint application development (JAD) to actively engage with the teams that helps development teams to identify data or control flow through the application.

In some of the agile projects I have been involved with, we addressed this challenge by describing the specification as behavior itself. To the extent we could use predicates to describe outcomes and behavior; we succeeded in deriving acceptance criteria that we could validate with the customer. However, program-defined behavior specifications are particularly ill-suited for use in designing software systems (Wileden, et al., 1983). The problem with behavior is that describing them at the structural layer differs greatly with the functional layer. Practice of acceptance test driven development (ATDD) is a good way to design these tests and engage the customers. Studies have shown that there can still be significant gaps between the test coverage and code coverage for such automated tests. Robert Glass writes in facts and fallacies of software engineering that even when developers think they have "fully" tested their code, only 55-60% of the logic has been tested (Glass, 2002). We have also experienced defect leakage past these tests attributable to incomplete data flow coverage. We have tried to address this challenge by adopting boundary value decomposition and path analysis techniques (McCabe, 1982). We have been able to qualitatively show that with such techniques, we can increase confidence in early automated testing and the value therefrom. We captured benchmarks and baseline metrics of complexity with coverage using some of the commonly used metrics. By correlating the number of tests identified against complexity measures, we were able to show a corresponding increase in coverage through the increasing number of tests with higher complexity. When we tied the higher coverage with the number of defects, we could not necessarily get a quantifiable measure of effectiveness but could demonstrate subjectively the *defect prevention* had improved. This has addressed the concerns regarding return-on-investment (ROI) which we are often faced with to demonstrate quantitatively on the benefits of these behavior-acceptance tests to the customer.

3. Challenges Faced By Traditional Test Automation

When teams transition to agile testing, the traditional testers who are used to the late testing mind-set have to think very differently about the value provided of early testing on partially complete code. The automated testers have to develop tests behind the GUI which they are not used to. It calls for programming skills, tools knowledge and understanding of the solution architecture – at least to the extent that they can relate objects and behavior. This transition is very profound and the learning curve fairly steep but not insurmountable.

Building the competency, skillsets and enabling the tester is one aspect, the bigger challenge is the shift in mindset of the tester to think about continuous value realization during the sprints of an agile cycle. Moving the teams from functioning in siloes to highly collaborative development involves overcoming a high level of distrust and antipathy toward the QA teams. Additionally, in some of the organizations I was involved in, we resorted to coaching, facilitation of team games to build collaboration and cross-functional training to build self-organizing teams.

The notion that automated unit tests and acceptance tests can serve as a safety net for the team to maintain the cadence is significant in that testers are able to directly relate to the value they could provide. In my view, this form of empowerment of the testers and the value that developers realized through continuous integration was instrumental in bringing the teams together. Complemented with exploratory manual tests, the coverage increased and the automated tests could be adapted throughout the lifecycle.

We adopted a script-less approach for the GUI-based regression tests. Our approach was tool-agnostic and relied on object oriented techniques such as reflection and dependency injection. Object oriented languages allow instances to dynamically inspect code in the same system referred to as reflection. Using this technique, users can extend the framework by defining custom functions that can be invoked for handling those object interactions that are not supported natively by the web driver. And with dependency injection, objects can be injected to a class to instantiate it rather than relying on the class to instantiate it. The approach addresses issues of *technical debt* of automated scripts that accrues with tighter timelines and quicker turn-around as tests scale and story points grow, reduces waste by self-documenting tests instead of *apriori* documented scripts by leveraging the action-keyword implementation of test scenarios and just-in-time automated test execution through late-binding. This approach also fostered collaboration between the user experience (UX) and testing teams as this approach relied on communication of the UX design through wireframes and user navigation to develop automated test scenarios early in the development sprint.

The script-less approach also enabled teams to realize benefits such as accepting late changes, lower maintenance costs, concurrent automated test design with code development and quicker turnaround. This also ensures that test automation effort progresses in-line with iterations.

The other challenge GUI test automation teams face with automated testing tools is that of object recognition and the extent of identification support by different accessors. By engaging early, the automation teams can work to resolve them while the code is being developed and get the help of developers to include 'testability' into their design and implementation, which addresses these problems. The benefits observed as shown in figure 1 is based on an audit application project that had over 6500 regression test cases (or steps) automated by a team of 7 spanning 3 months over 1 major release and 2 minor releases, the duration of the project release being 1 year and 3 months. With over 80 business user stories that were automated, the maintenance effort of the test assets was less than two percent (2%) of the investment (fixed cost + variable cost) to keep up with the product features. The highly readable automation test scripts also serve to communicate with cross-functional teams very effectively.

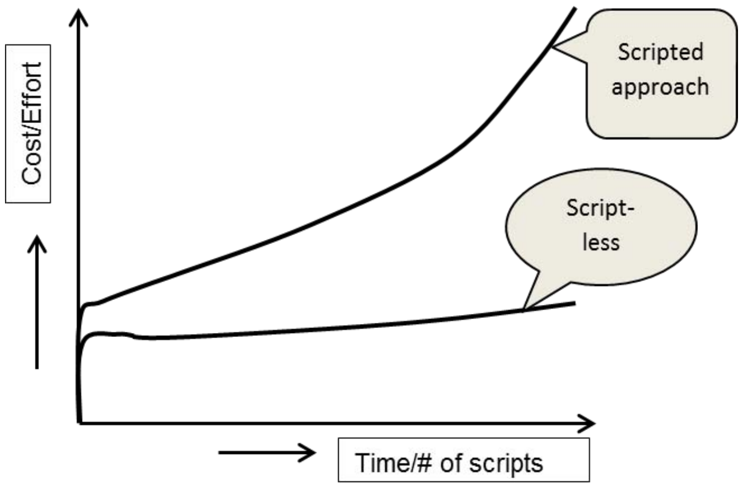


Figure 1: Cost savings from script-less compared to scripted test automation

Agile testing looks to test automation to providing insights and feedback on application quality as the iterations progress. The classifications of tests where automation can provide this feedback to the different stakeholders are best captured in the agile testing quadrants described by Brian Marick (Crispin & Gregory, 2009). These are reproduced here for reference in the figure 2 below. These test approaches need some discussion here: BDD or behavior driven development tests are those that describe the behavior of systems and is used to guide design and development. This behavior of system can be specified in a domain specific language called Gherkin that lets you describe software's behavior without detailing how the behavior is implemented. A Gherkin DSL example follows in the subsequent section; however the same can be used to describe acceptance criteria for product backlog item or user story. When this is done to verify *doneness*, it is called Acceptance Test Driven Development (ATDD). Both BDD and ATDD are primarily business facing tests and typically automated. While these tests leverage a specification that is at a higher-level of system abstraction, the unit tests written prior to the writing of the code facilitates test driven development (TDD). At the end of the day, all of these tests provide the safety-net for developers to focus on managing technical debt through the red-green-refactor cycle.

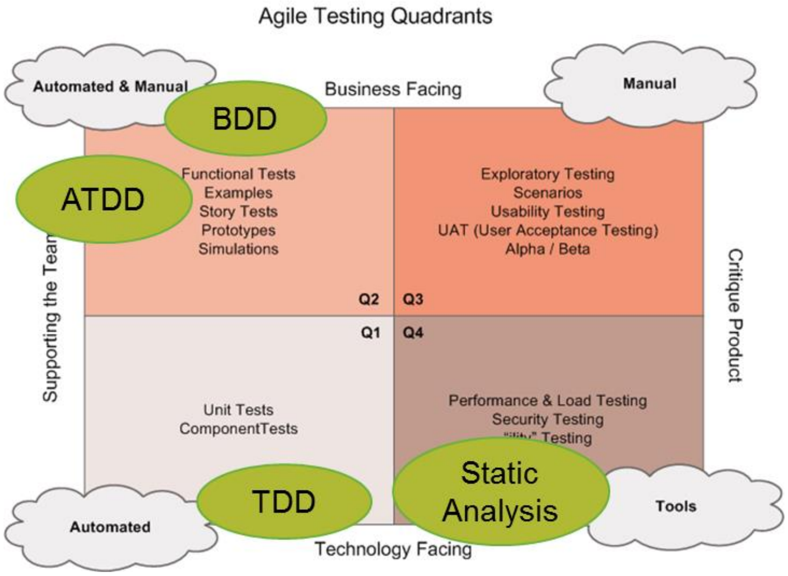


Figure 2: Brian Marick's Agile Testing Quadrants (Reproduced with permission, Courtesy: Crispin & Gregory, 2009)

The quadrants can be used to articulate the value proposition from automated testing in agile projects. The significant difference between the linear development cycle and iterative cyclical agile testing is that the testers have the opportunity to continuously adapt tests in line with the principle of welcoming changes late in the development process. Customers are often heard retorting, “that’s what I asked for, but that’s not what I need.’ Without continuously validating the builds, a lot of waste (rework) ensues. Teams have to look beyond GUI testing to build-in quality and get early validation toward a converging solution. I often use the quadrants overlaid with the techniques available to the testers to highlight the insights these provide and the respective groups that benefit therefrom. In order to ensure continuous test adaptation, I encourage teams to combine different approaches to review and balance the automated tests with their manual counterparts. This helps in capturing the metrics such as reduction in defect leakage, defect detection by sprint normalized to the story points realized. Through high visible information radiators, teams have achieved seen greater participation by the stakeholders. These metrics in addition to the commonly used ones such as burn-down charts, burn-up rates and velocities is very helpful. The steps to create these highly visible information dashboards are straight-forward – determine the defect detected by user story, then using the story estimation point as weights to compute the weighted average. This ensures that the normalized defect detection is also relative to the team just as estimates and velocities of different teams also vary for the same set of stories and seen in the context of the team. A couple of the metrics shown in the information dashboard are shown in figure 3 below.

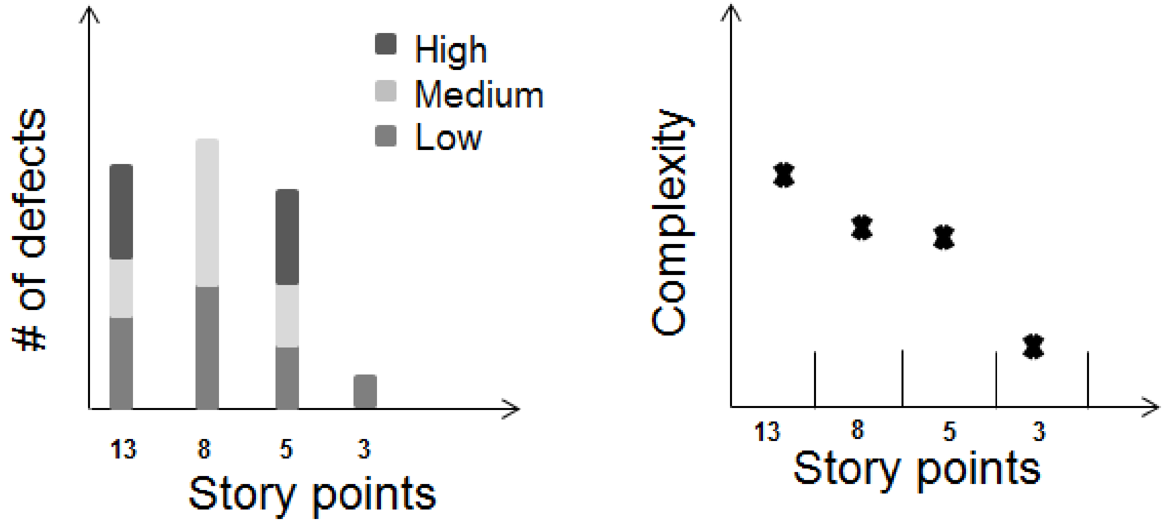


Figure 3: Some Measures Tracked & Presented in the Sprint Dashboard

Note that the complexity referenced here is one of the commonly used measures one of which is the cyclomatic complexity but for the purposes of this discussion this could well be replaced by another measure without losing the point.

4. Technological Advances and Innovative Practices

Technological advances in the recent decade in the areas of continuous integration and delivery have encouraged teams to realize the value from completed iterations. This fundamental change in approach also increased collaboration among the teams as the automated tests quickened the feedback cycles and increased the opportunities to *build-in* quality.

In and of-itself, testing teams face challenges of re-tooling and building skills of software development for designing *adaptive* automated tests – reusable (Hunt, 1999), extensible, maintainable and receptive to late changes (Beck, 2000). With script-less GUI-automation, most of these challenges are addressed implicitly as the development of action keyword processing engine is a one-time activity and skills of the development team can be leveraged to design, develop and test the script-less test execution engine. The action keyword processor is the part of the engine that processes the interaction and navigation flows of the application whereas the script-less test execution engine provides other capabilities as scheduler service, threaded parallel execution capability and also self-executable test script that can be reused by cross-functional team. The low-level action keyword sequence would look something like the following as shown in table 1 – typically they follow a page-object model. A higher level sequence of these low-level tests can be grouped by logical business function and managed by the script-less execution engine as shown in table 2. This also gives the flexibility of creating self-documenting tests and increases re-usability. The low-level action keyword processor is an open-closed solution in that it is open to extension but closed to modification (Martin, 1996).

Object	Action	Parameter	Execute
Browser	Launch	IE	Y
URL	Load	https://abc.company.com/login.asp	Y
Login_Field	isVisible		Y
Login_Field	SetText	“user_1”	Y
Password_Field	SetEncryptedText	“Ah#i@0JWb&”	Y
Submit_Button	isEnabled		Y
Submit_Button	PerformClick		Y

Table 1: Low-level action keyword sequence for a login function

Script	Execute	Argument
Login	Y	<test data filename>
NavigateToAccountsSummaryPage	Y	<test data filename>
PerformBillPay	Y	<test data filename>
Logout	Y	

Table 2: High level execution sequence

With the development team involved in the design and development of the keyword-driven script-less framework, the development team begins to ‘own’ and support the product. The developers could also run these regression scripts as some of the agile projects have reported doing it (Ambler, 2010). Lately, tools support Domain Specific Language (e.g., Gherkin) for test definition breaking down the barriers of niche-skills and alien programming languages that traditional commercial tools brought with them (Pettichord, 2000). With a language like Gherkin, you can specify the behavior in unambiguous terms and develop code that can verify in the language of implementation. For example, with Cucumber which is in plain text, the test harness can be generated to verify the feature and steps within the test as shown in table 3.

Feature: Overdraft amount withdrawal exception management
If a user attempts to withdraw funds that exceeds account balance to throw an error that indicates insufficient balance unless the user has an over-draft account

Scenario: Insufficient account balance exception handling during withdrawal attempt
Given the user is a regular account holder is logged in
And has selected the withdraw amount option
When the user requests an amount exceeding his current balance
Then the system should inform the user that the current balance is insufficient to honor the request

Scenario: Over-draft account withdrawal subject to an upper limit
Given the user has logged into his margin based account
When the user has selected to withdraw an amount exceeding his current account balance
Then the user should be allowed a one-time over-draft withdrawal subject to an upper limit

Table 3: Example of a behavior description using Gherkin DSL

The behavior definition as described in table 3 translates to a test runner in the specific implementation language.

5. Cultural Challenges and Agile Adoption

Agile estimation is very inclusive and iterative as opposed to the traditional top-down command-and-control style that traditional development lifecycle entails. This is another challenge faced by teams making the transition to agile. From the cultural perspective, outsourced quality assurance engagements involving external teams find it difficult to break-free from the 'being-told-what-to-do' frame-of-mind to a highly engaging, self-organizing and collaborative participation in the development process. Some of these traditional off-shore teams find moving from the top-down estimation which they are habituated to give absolute numbers for a set of requirements thrown 'over the wall' to an iterative and relative sizing that agile practice recommends difficult. And the myth that agile estimates are absolute has to be broken. That different teams can weigh same stories differently is perfectly acceptable in agile estimation (Kelly, 2013). The concept of continuous learning and retrospectives is also an alien practice to traditional off-shore teams who rarely get a place in the table to provide inputs that materially affect the practice of application development. Teams that are beginning the agile transition journey may err on the side of caution by estimating more than the actual efforts needed. This is ok and over time as the team gets into a cadence; the team is able to learn from the sprint velocities, getting better at estimations.

In some of the cultures, it is believed that a good process can make a great product as long as you can demonstrate rigid adherence to the process, the outcome will turn out to be good. Unfortunately, with software development this is far from the truth (Hartman, 2010). Agile methodology describes *principles* when followed through *practices* that are contextual result in addressing most of the issues that stem from other software development methods. It is to be realized that Agile is a culture rather than a software development process. This acknowledgement for some of the traditional teams is hard to come-by. In highly metrics-driven leadership cultures, this can be a chicken-and-egg problem. Without adoption, you cannot show the statistics and without statistics, you can't get the traction and support for the agile transition. In organizations that are run by metrics and measures, the teams inherently imbibe the culture of process efficiency at the expense of the product.

I have resorted to building the mind-share within the organization, in such situations, by constantly deliberating and giving my points-of-view backed by industry reports, thought leadership articles and references to business values and outcomes for agile adoption and transition.

6. Team Velocity and Cadence

A common challenge that teams face in the early formative stages of agile adoption is in finding the right cadence. This is driven by the urge to start working on newer tasks before completing the ones committed to for the current sprint. Teams often lack this discipline of starting many tasks and activities without committing to complete them. While inability to complete the ones started is not unusual for even the more mature agile teams, the mature teams realize the importance of completing them before starting to work on newer ones. The Kanban approach to limiting the work-in-progress and driving to completion the stories that are started is the most effective way to manage the velocity of teams. *Kanban* (Japanese) literally means a signboard that was first used in the Toyota Production System by Taiichi Ohno to control production between processes and just-in-time manufacturing. They were effectively used to minimize work-in-progress and reduce cost related with holding inventory (Gross & McInnis, 2003). Even today Toyota continues to use the system not only to manage cost and flow, but also to identify impediments to flow and opportunities for continuous improvement

7. Conclusion

In this paper, I have tried to articulate some of the major challenges to agile transition with a propensity to agile test automation. This challenge for test automation teams are to think of the value automated testing effort brings to business and development teams continuously look to optimize the test coverage by vectoring the automated tests and balancing the manual exploratory and automated tests. I have also touched upon some of the key challenges such as shifting from a command-and-control team structure to a self-organizing style of functioning, coming to terms with relative sizing and estimation, limiting work-in-progress to focus on completion of user stories, value realization through working software etc. that I have faced in working with teams that have struggled with the transition.

References

1. Ambler, Scott, 2010. *How agile are you? Survey,* www.ambysoft.com/surveys.
2. Beck, Kent, 2000. *Extreme Programming Explained: Embrace Change*, Addison-Wesley.
3. Crispin, Lisa & Gregory, Janet, 2009, *Agile Testing: A Practical Guide For Testers and Agile Teams*, Addison-Wesley.
4. Glass, Robert, 2002. *Facts & Fallacies of software engineering*, Addison-Wesley.
5. Gross, John M. & McInnis, Kenneth R. 2003, *Kanban Made Simple*, AMACOM books.
6. Hartman, Bob, 2010, *Doing Agile Isn't the Same as Being Agile*, PHXSUG presentation.
7. Hunt, Andrew, 1999. The Pragmatic Programmer, Addison-Wesley.
8. Kelly, Allan, 2013. *Top Twelve Myths of Agile Development*, Agile Connection – online community, www.agileconnection.com/print/article/top-twelve-myths-agile-development.
9. Kniberg, Henrik, 2010, *Kanban vs. Scrum – making the most of both*, lulu.com.
10. Martin, Robert J., 1996, http://www.objectmentor.com/resources/articles/ocp.pdf
11. McCabe, Thomas J., 1982. *Structured Testing: A Software Testing Methodology Using the Cyclomatic complexity Metric*, NBS Special Publication, National Bureau of Standards.
12. Pettichord, Brett, Jan/Feb 2000. *Testers and Developers Think Differently*, STQE Magazine.
13. Sahota, Michael, 2012. *An Agile Adoption and Transformation Survival Guide: Working with Organizational Culture*, lulu.com.
14. Wileden, Jack C, et al., 1983. *Behavior specification in a software design system*, The Journal of Systems and software, 3, 123 – 135, Elsevier Science Publishing Co., Inc.

Software Analytics by Storage (SAS)

Sundaresan Nagarajan – McAfee/Intel India

Sundaresan_Nagarajan@mcafee.com

Abstract

There are a number of challenges in software, especially in its effective testing and the software maintenance problems. Can there be something done to improve the testing efficiency and help deal with the software maintenance problems? This paper presents a case that software analytics based on a storage-based approach can help deal with these issues. This database storage-based system along with simple graph-based queries, applies to different types of software, platforms, domains and also to complex multi-tier software.

This system works by making use of a lower level view of the software and mapping it to high level user-domain features and tasks with the help of a database. For example, when a test case is executed, the test case gets tagged to the path traversed by it in the structural graph of the software which is stored in a database. The objective is to exploit the inherent knowledge present in the software to improve its quality. There are two parts in this solution: a minimal database-based storage layer that acts as a base for overlay applications which solve a specific problem. There are two such overlay applications discussed here: parallelized testing and automated root-cause analysis.

The minimal storage layer consists of a relational database which is populated with both static and runtime behavior of the software. This storage layer is queried by overlay applications working on graph based queries to solve a specific problem like: the right set of test cases to execute for a bug/feature, parallel testing (concurrent execution of test cases), root-cause analysis for a bug or test case design

Some of the benefits of this system are that it helps right from software development testing to software maintenance post release. The relational database which is part of the storage layer matures with time based on testing done and issues found in the field deployment which is looped back into the system. As we close the loop with every issue found in the field and testing of the software, the system gets better as new software features and fixes are introduced. Some of the other benefits of this system are identifying parallel test cases so that better understanding and more efficient testing is possible, and also efficient and accurate root cause analysis based on depth querying and graph traversal algorithms. It is a step towards automated root cause analysis.

On the whole, the storage approach captures the dynamic relational knowledge of software and its execution. The silos of information, that is code logic at different layers of the software, are brought together in a relational database to make a combined sense. This offers greater insight into the software, thus increasing its efficacy.

Biography

Sundaresan Nagarajan is a Senior Software Engineer in Test working at McAfee/Intel India. Prior to this, he has worked in the storage domain at EMC Corporation. He has around 6 years of experience working in the software industry in both the fields of development and software testing. He has resolved over a 100 software customer issues reported from the field and also worked on software development projects initiated from scratch. He is a graduate in Computer Science and Engineering from Anna University, Chennai, India.

1. Introduction

Is there something that can be done to improve the software testing process and the numerous software maintenance problems? Are developing code, software testing and software maintenance related? Why don't we avert software maintenance problems and increase software maintenance efficiency with time?

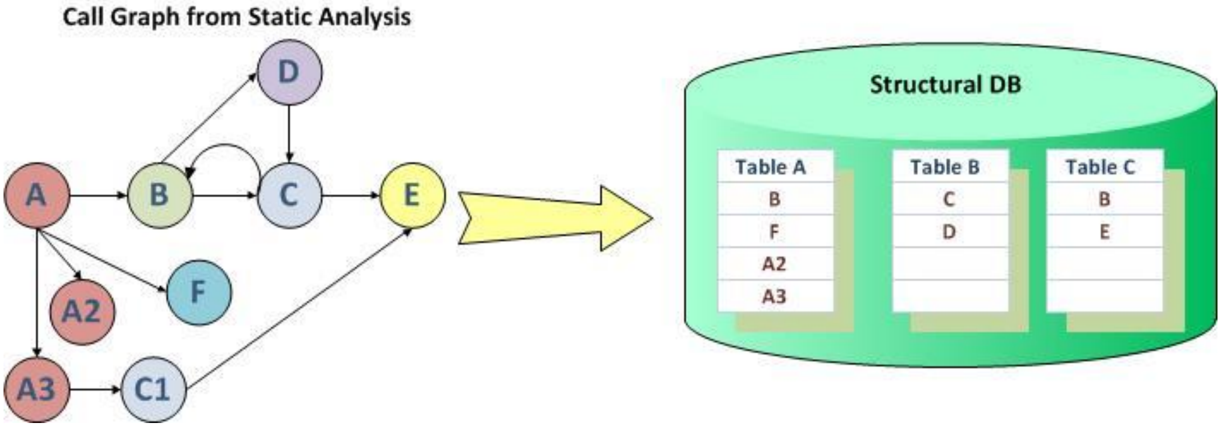
These are the some of the questions, which led to the Software Analytics by Storage (SAS) system. The objective of this system is to exploit the inherent knowledge of the software to improve its quality by improving the testing process and help in solving the numerous software maintenance problems.

This paper proposes a storage and database based solution to improve the software quality. The database system proposed here matures over time. With issues root-caused, testing already done need not be redundant and issues are averted.

From the user perspective, the SAS system provides a simple user query system. Complex applications like graph search can be built on top of it.

2. Database and Storage based approach

The approach to solving the problems mentioned here is take the output data from static analysis software (like Doxygen) and its call graph (Ryder 1979, Grove 1997, Callahan 1990) details and store them in a database. Also the computations performed on the stored static data are captured in a database which enables subsequent user querying. This forms the central theme of the storage-based approach. This approach is illustrated by the diagram below.



The relational database identifies the relationships between the data records of a set of data, commonly referred to as tables. The referential integrity property of databases ensures that the correct relationships are formed as the data is fed into the database.

The other major advantage of a database approach is its ability to mature with time, as relationships that have been captured gets augmented as new features and fixes are added, and new issues and testing related static data are fed into the system. This approach is capable of helping in applications like root cause analysis. Even during the pre-release phase of the software, the maturing aspect of the storage-based approach ensures that more effective identification of complex issues can be done while testing. This will ensure that the customer gets a high quality product at first usage.

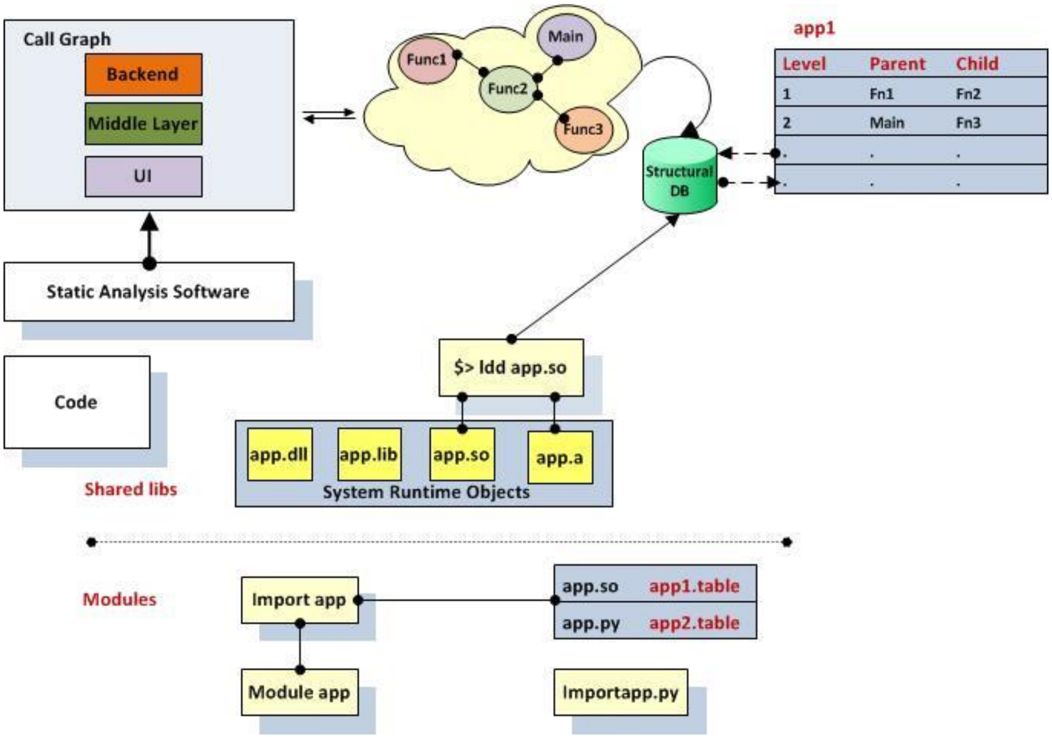
Now that we have an overview of the database and storage based approach to software testing and maintenance, in the next section we can look at how a typical storage base structural database is modeled.

3. Concept of Structural Database

The structural database captures the structural details of the software being analyzed. Subsequently the run time data of the software (like debugger recordings/stack trace analysis (Gavrilov 2000)) of the software features are linked with the structural database to form the relational mapper database. This association between the structural database and the runtime captured information takes place by common key and record attributes.

The figure below illustrates this concept. The software static analysis generates the methods and the call sequences. The structural database links the method calls associated with a library. The information in the structural DB contains not only the call graph information but also libraries and system runtime objects (.dll, .lib, .so, .a) across platforms and across the languages (like .cpp and .py) which are the components of a multi-tiered software system.

This is useful for the system as we go through an end-to-end scenario of testing, runtime analysis, etc where the structural DB contents such as the call graph of code, libs, and modules could be part of a single test-case path.



Level	Parent	Child
1	Fn1	Fn2
2	Main	Fn3
.	.	.
.	.	.

app.so	app1.table
app.py	app2.table

4. Change set and Sequence of Trees

In this section we introduce the concept of a change set, which forms the core of the SAS system. The change set is a part of structural DB tagged and extracted by feature. All the feature tags corresponding to a feature are tagged under a sequence. The change set is a hierarchical grouping of tags related to a feature. The change set can be formed from the output of different types of testing (white box/black box testing).

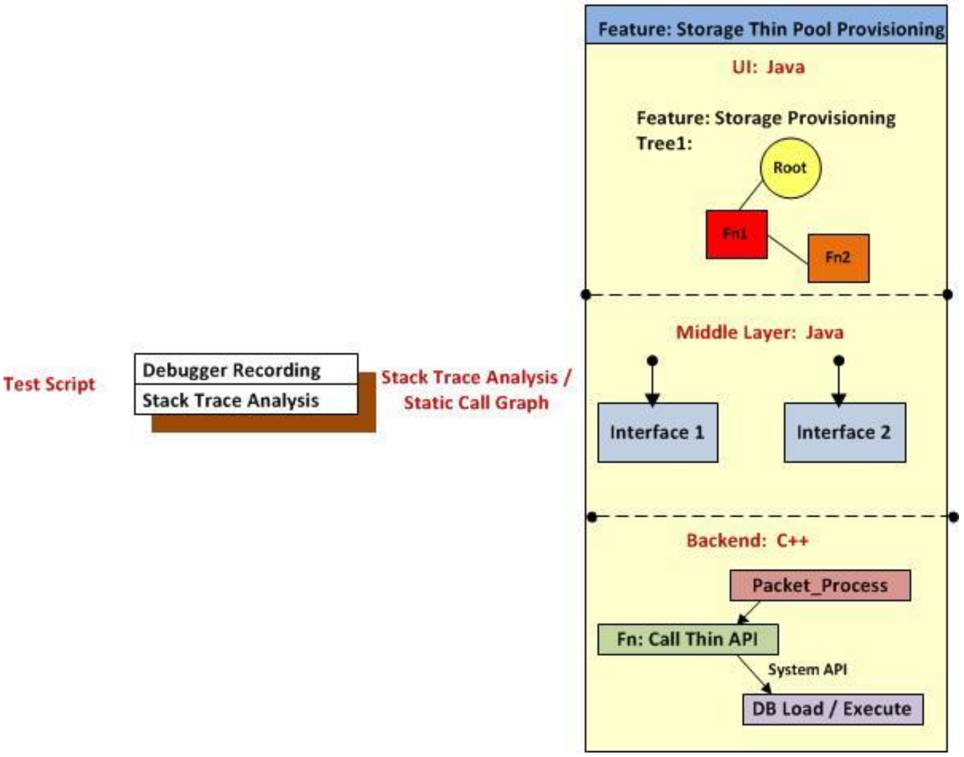
The change set refers to the code and the relationships between the code of different components, applicable to a particular feature. Normally the change set information is obtained from a configuration manage system only while a feature is developed. In this paper we are able to derive the change set for a feature from the structural database as well as runtime recordings.

The change set internally can be referred to as a hierarchical group of relational trees derived from the structural database. The change set can contain sub-change sets corresponding to sub feature test cases. The sub-change sets are merged and augmented incrementally to form the feature change set which is a group of trees at each layer of the multi-tier software (as illustrated in the later diagram).

The rationale behind the concept of a change set is to provide a semantic meaning to the paths of the structural DB. When a test case is executed, the test case is tagged to the path traversed in the structural graph (which is stored in a structural DB). This forms the central idea of this paper. Once we have these tagged paths, we can process them, extract information from them, and build overlay applications from them (such as parallelized testing and automated root-cause analysis as discussed later). The next section and accompanying diagram show how a change set is generated from the structural database and the test case execution.

We have considered a multi-tiered application and a use case feature that is being tested. This usecase encompasses logic from the different tiers of the system like UI, Middleware and Back-end layer. As we can see in this example such a feature can be implemented using different programming systems but the heterogeneous data can be stored in a database.

Change Set & Relational Group of Trees



5. Relational Extractor DB

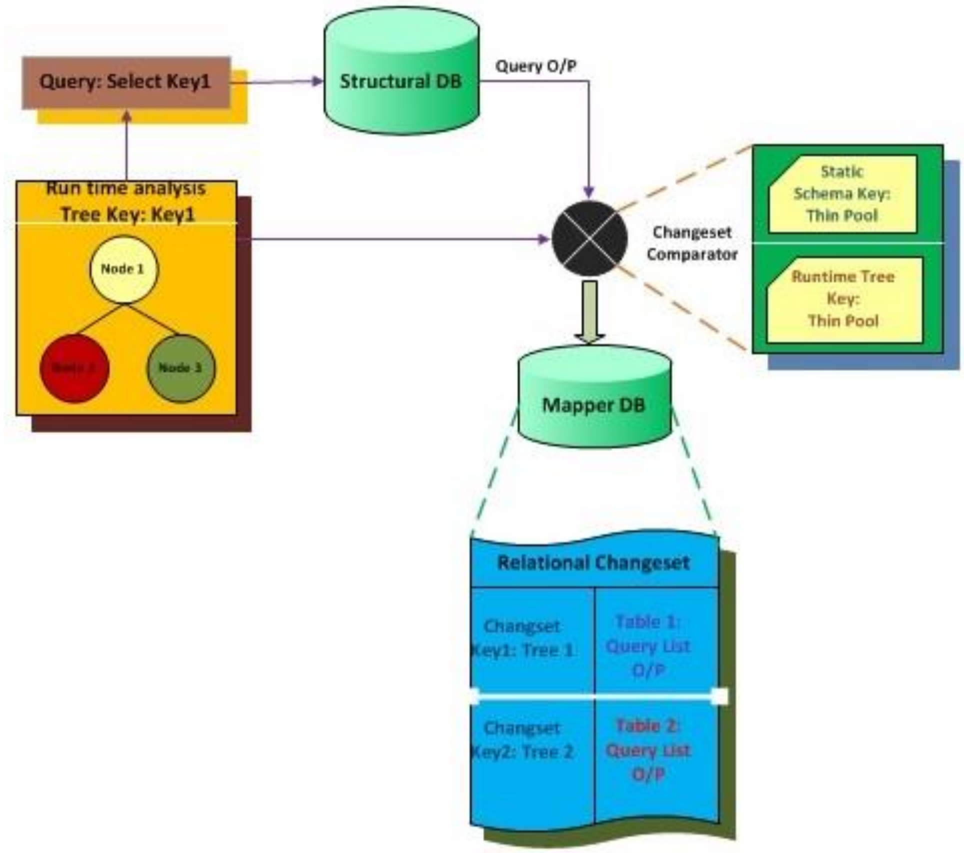
The purpose of the Relational Extractor DB is to merge the structural database content and the runtime feature based content. This feature-based content comes from the testing and the corresponding runtime content recordings.

The merge operation of the Mapper and Relational Extractor works by means by identifying paths obtained from dynamic analysis and, matching them with paths of the static analysis output i.e. the structural call graph DB. The matched path of the structural DB across all layers is extracted from the DB, decorated with a tag name (feature name) and added to the change set which is present in the Relational extractor DB. The output of the merge operation is a change set for a feature. This change set operation is ‘augment change set’ as the change set for a feature gets augmented as more and more paths are added to it, as more and more testing of the software takes place.

The querying capabilities of the system are dependent on this merge operation. As is discussed in the next section, the overlay applications are based on the Mapper DB created after the merge operation.

The figure below shows how the merge operation is performed from the operands of structural database and feature recordings. The merge operation can also be called as tagging, as it essentially assigned the feature tags to the corresponding paths of the structural database. The output of the operation is a relational extractor DB which stores the change set which has been defined in the previous section.

The test case maps to paths in the call graph database, known as the structural DB.



Relational Changeset	
Changeset Key1: Tree 1	Table 1: Query List O/P
Changeset Key2: Tree 2	Table 2: Query List O/P

An aspect of the maturing DB is the change set expansion as the feature is covered more and more in testing.

Tagging of Structural DB (also known as the Merge Operation):

The change set is a part of structural DB tagged and extracted by feature. All the feature tags corresponding to a feature are tagged under one sequence. The change set is a hierarchical grouping of tags related to a feature. The change set can be formed from the output of different types of testing (white box/black box testing)

The aggregation of paths with tagged data but with gaps is a candidate for end to end path to be tested. This leads to a new test case not covered before in the different forms of testing in the system.

The concept of tagged paths is how the relational change set is created from the static and runtime analysis output.

The flow of the system is as follows:

Tagged Paths -> Categorization -> comparison based on testing tag.

The pseudo-code for the implementation of the merger operation is as follows:

```
Merge<dynamic_analysis_graph_1, call_graph_2>
 {
    Loop {
        Search for Keys<Path1 of graph_1, call-graph>
        Path-compare and extract compared path;
        Add to Changeset_1
    }
    Returns Changeset_1
}
```

The numbering scheme for the tagged paths is similar to that of below:

Testing_type[i].Majorno[i].Minorno[i]

This kind of numbering scheme enables user querying of tagged path. The Relational Extractor DB stores the test case table with the tag numbering scheme associated with the test case. The numbering scheme is derived from the type of testing, test case id and the feature and sub-feature wise classification of test cases. The test case table is as follows:

Test case Table:

Test Case Table: **Testing Type**	**Major number**	**Minor Number**
Unit		
White-box		
Black-box		

The aggregation of paths into a change set forest is depicted in the following table of change sets:

Feature ID	**Change set ID**	**Path IDs**	**Forest ID**
		List	

The Change Set implementation:

The change set for a feature is an expanding tree which gets augmented with more and more testing done and more and more tagged paths identified.

So the change set implementation should support the augment operations on the change set already created.

The pseudo-code for change set implementation is as follows:

```
Class Changeset

{

    List <Tree*> lt;

    Add()

    Modify()

    Merge(this, changeset1);

};
```

The high level pseudo code for merging a change set with its potential child change set is as follows:

```
Changeset::merge(this, Changeset1)

{

    For(;;) {

        Parent = this;

        If(this->tag.feature_name == changeset1->tag.feature_name) {

            Parent->addChild(changeset1);

            Break;

        }

        Else {

            This=this->next;

        }

        }

}//End of Changeset::merge() operation

Class Changeset_Track : public Changeset
```

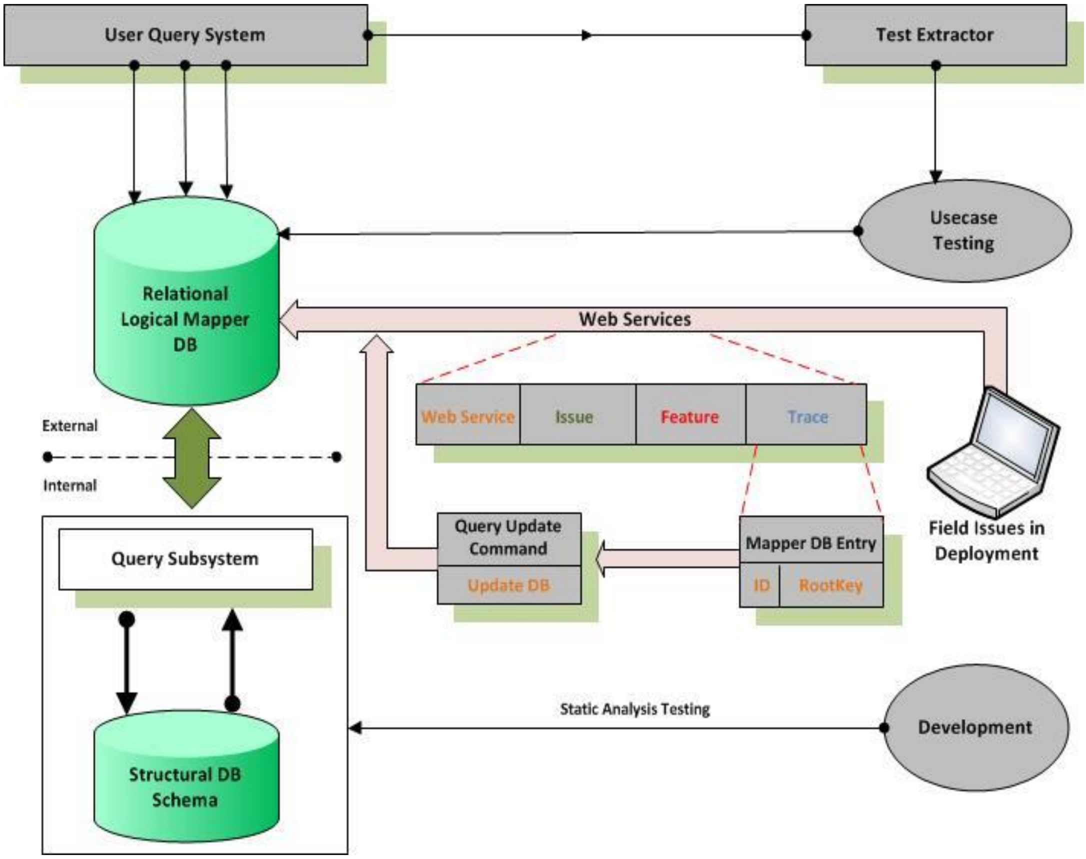
To summarize, the change set in the Relational Extractor DB maps the higher level test cases to the lower level paths. In other words the change set DB captures the path traversed in the structural DB for a feature test case or a group of test case for a feature. The change set of the Relational Extractor DB is extracted from structural DB for the paths matching the dynamic analysis, tagged as feature paths. There is a version match between the change set DB and the structural DB. The change set DB gets augmented in a loop-based fashion as more and more of a feature is tested. The change set DB stores the change sets which are hierarchical in nature, because they correspond to the higher level test cases.

6. Overall System

The overall system closes the loop between testing and feature development/fixes for issues. The database-based system keeps maturing as the data is fed into it and gets stored in a relational way. This enhances the capabilities of the user query system.

The interface between the field deployment of the software and the mapper DB is in terms of web services APIs. The issue (bug identified in the field) is mapped to the corresponding feature of the software and its identification in the mapper DB. This enables the updating of the DB using the corresponding query update command.

The overall system is described in the diagram below. The fixes made to the software are reflected in the overall system as the updating to the structural DB and the corresponding updating of the relational mapper DB. This ensures that the overall system works in a cycle leading to maturing of the database over time- which helps both in the software development / testing cycle and in post-release maintenance.



To summarize, the central idea of the SAS system is to deconstruct the structural DB by means of test cases (and the corresponding change sets) and assign meaning by assigning user defined, test case defined tags to paths in the individual paths of the call graph. In other words, the paths of the call graph, structural DB is extracted by its correspondence with the test cases and decorated with the test case keys namely tags. Then it can be constructed back by merging the change sets incrementally so that our goal of maximum coverage of the original call graph is achieved.

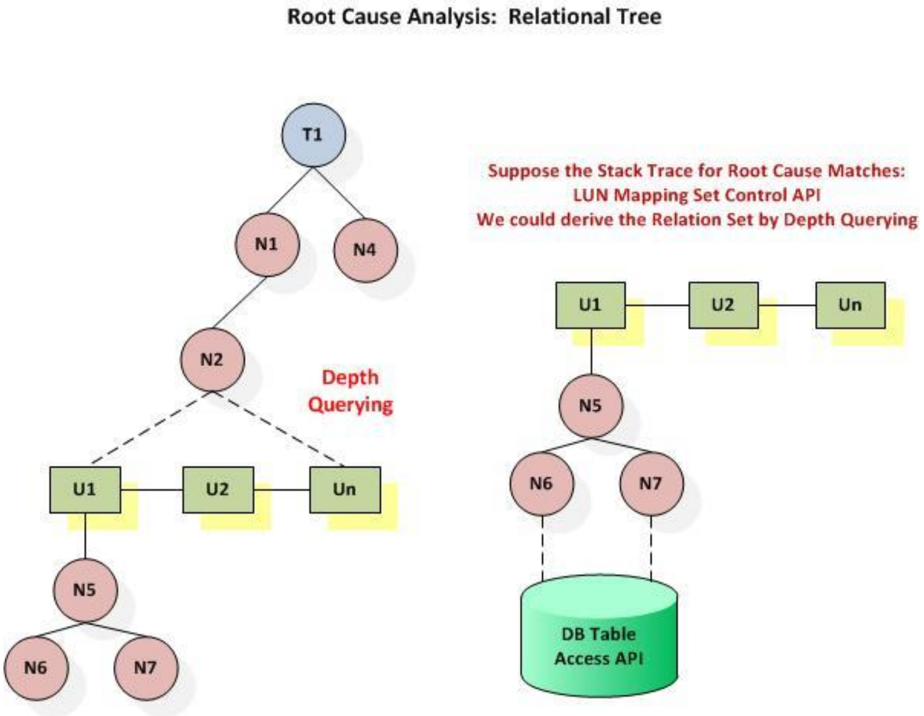
7. Case Study

The SAS is designed as a minimal system, which provides the user querying capabilities over the vast depth of software knowledge. This system enables developing simple queries, which apply this knowledge to the problems of software testing and maintenance. This section provides two such applications which helps in root cause analysis and parallel testing

7.1. Overlay application Relational Tree Visualization by Depth Querying for Root Cause Analysis

The advantages of the SAS system relational tree visualization over the available visualization of existing static analysis software (like (Doxygen)) is that it provides a heterogeneous change set which can aid root cause analysis over a multi-tier software application.

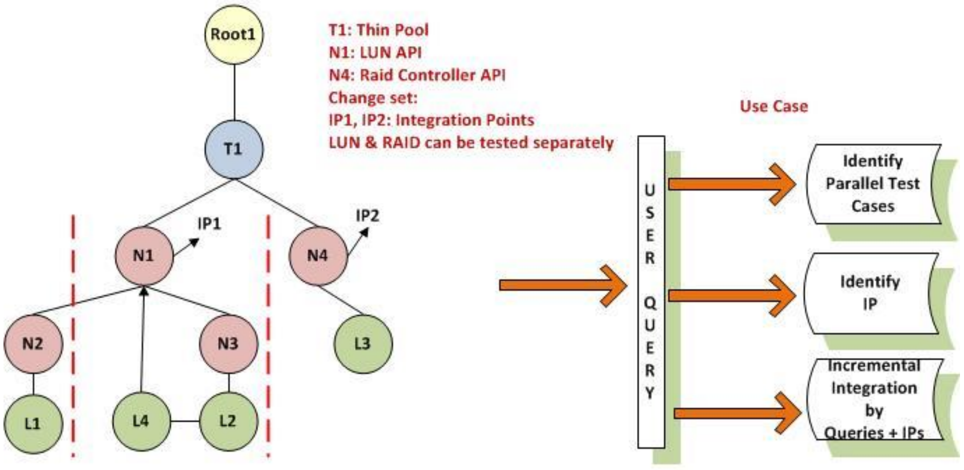
In the figure below from the stack traces of a bug, we could identify the corresponding test case affected by the bug. And by querying the SAS database we could identify the visualization of paths affected by it using Depth Querying across the layers of a multi-tier application. In the figure below, U1, U2 and U3 refers to the starting points of interfaces in a different layer for an end-to-end path scenario.


Root Cause Analysis: Relational Tree
T1
N1
N4
N2
Depth Querying
U1
U2
Un
N5
N6
N7
Suppose the Stack Trace for Root Cause Matches:
LUN Mapping Set Control API
We could derive the Relation Set by Depth Querying
U1
U2
Un
N5
N6
N7
DB Table Access API

7.2. Parallel Graph Search in relational tree for Parallel Testing

One of the frequent problems of software projects is that feature completion of development is a dependency for software testing. By parallel testing, the aim is to perform parallel unit testing when the software is being developed and incremental integration testing.

Test Extractor and Parallel Testing


Root1
T1: Thin Pool
N1: LUN API
N4: Raid Controller API
Change set:
IP1, IP2: Integration Points
LUN & RAID can be tested separately
T1
IP1
IP2
N1
N4
N2
N3
L3
L1
L4
L2
Use Case
USER QUERY
Identify Parallel Test Cases
Identify IP
Incremental Integration by Queries + IPs

Identify Partitions in Graphs with Integration Points

In the figure we introduce the concept of integration points. An integration point is the point in the call graph where an independent logic branches out. Integration Points help in identifying parallel test cases.

The integration point can be thought of as the point up to which duplication/common paths are allowed in two potential test cases of the feature. This is to avoid false negatives in identifying parallel test cases caused by various factors such as common driver code, or code common to all paths such as common database code, or common networking library code.

The application queries the SAS system and identifies the partitions in graphs with integration points. The procedure to identify the test cases is to identify maximal length paths in the relational graph with no duplicates.

The following is an overview of how Parallel Testing test cases can be derived from the change set functionality of the Relational Extractor DB. The API accessed is the Identify_Parallel() API for the parallel test cases.

The pseudo-code for the parallel test case functionality is as follows:

```
Identify_Parallel(FeatureID [IN], ChangesetID [IN], PathsList(paths) [OUT])

{

    //Iterator over the PathsList

    If(PathsList != NULL)

    While( i <= PathsList.length())

    {

        addTestCase[PathsList[i]];

    }

}
```

The API for Test Executer is: TestExecute<TestCaseID, <ChangesetID,FeatureID combo>>

The logic for Test Tracker is to query the DB and the above said process of tagged path creation, updating of the change set for the feature happens again for this cycle.

New Test cases (Missing Gaps of Testing):

A query of the uncovered path (that is paths that are not tagged by any previous testing) in the call graph will yield data for the new test cases to be designed.

Examples

1. SAS - Parallel Regression Automation (Feature development)

This example scenario is to manage a storage software system which is made of a RAID group of disks and has advanced storage provisioning features like that of the storage pool. Here is some information on the software being developed in an agile scrum model.

- RAID group Feature
- Virtual Provisioning Feature
- Storage pool feature

Each feature is built in incremental iterations of the product development process using the agile scrum model.

The key feature of this model is that there is a demonstrable, testable deliverable of software development at the end of each iteration of the project.

RAID Group feature can be classified into two categories:

1. Discovery of the RAID Group component and its sub- components.

2. Active management of the RAID Group feature which performs an action on the existing RAID Group in the system, thereby changing its configuration. After the active management operation a discovery of the changed RAID Group component is needed to reflect the changed component in the User Interface.

Now consider the virtual capacity calculation problem: It is a function of the RAID Group capacity calculation. There is a dependency here.

Now the inclusion of the additional features means the following changes to the implementation (code portion) of the software:

a) New members being added to the discovery object, updating of existing members of the discovery object.

b) New members being added to the database tables, new addition to the record set retrieval functionality and the corresponding changes in the User Interface.

Now consider the changes to the capacity calculation logic:

In which way new functionality affects the previous iteration existing code.

Previous:

Total Capacity = ∑ (RAID Group capacities of all the RAID Groups in the system)

Now:

Total Capacity = ∑ (RAID Capacity) + ∑ (Storage Pool Capacity) + ∑(Meta Capacity)

The representation in the Relational Extractor DB for the Storage Pool scenario:

- End to end paths tag, testing type: end-to-end
- Unit testing path tag, testing type: unit-test

Change set testing path1: StoragePool

1. End to end path tag – storage pool discovery,
2. End to end path tag- storage pool active management,
3. End to end path tag within active management :
    a. Create storage pool,
    b. Associate storage pool to RAID group,
    c. Storage pool expand.

All the three above active management paths (3.a-c) belong to the active management change set. The active management change set along with storage pool discovery path forms part of the storage pool change set.

The changes sets are hierarchical in nature, representing the structure of test cases in the testing domain.

The example of tags for the above three examples (1-3) are as follows. In the example E1 ..E7 are function calls which are part of the change set. For instance E1-E2-E3 is one such path of function calls traversed by the end_to_end.storagepool.discovery change set.

1. End_to_end.storagepool.discovery: E1 - E2 - E3
2. End_to_end.storagepool.active.create_storagepool : E1 - E4 - E5
3. End_to_end.storagepool.active.associate_storagepool_to_RAIDgroup: E1 - E6 - E7

In the above change sets (1) and (2) have E1 in common so it’s a common interface method traversed by both the test cases i.e. storage pool discovery and storage pool active management operation: create storagepool.

The representation of the above change sets in the database table is as below:

Testing Type	Change Set ID	Feature Level1	Feature Level 2	Feature Level 3	Call graph function calls in path for this change set	Child Change set IDs
End to end	CS1	StoragePool				CS2, CS3
End to end	CS2	StoragePool	Discovery		E1→E2→E3	
End to end	CS3	StoragePool	Active			CS4, CS5
End to end	CS4	StoragePool	Active	Create StoragePool	E1→E4→E5	
End to end	CS5	StoragePool	Active	Associate StoragePool to Raidgroup	E1→E6→E7	

As can be seen from the database table, change sets CS1, CS2 and CS3 are related by the hierarchical relationship CS1→CS2, CS1→CS3. And CS3 is the parent change set for the child change sets CS4 and CS5 represented by the relationships, CS3→CS4, CS3→CS5. The above change sets are an example of hierarchical change sets.

Storage Pool Feature Testing:

Find parallel test cases in storage pool change set:

Query for paths in the changeset.storagepool.

Identify Integration Points: E1

Independent paths which can be run in parallel: E2-E3, E4-E5, E6-E7.

These independent paths that can be run in parallel are identified by the procedure provided in Case Study of Section 7.2. Basically, it is achieved by taking the integration point (E1 in this case) and identifying parallel paths in the graph of related change sets. From the graph we can deduce that beyond the Integration Point- E1, the paths in question E2-E3, E4-E5, E6-E7 are all independent paths with no common function calls. Therefore they can be run in parallel.

The corresponding test cases for the above paths:

- storagepool.discovery,
- storagepool.active.create_storagepool,
- storagepool.active.associate_storagepool_to_RAIDgroup

These change sets and their subsets can be tested in parallel.

New Test cases(Missed Gaps):

Query for Paths in Structural DB with unit testing paths in each layers but no continuous paths in end-to-end testing paths:

The query result will have such paths that have not been tested in the end-to-end scenario.

1. So proceed to test one of these end-to-end untested paths,
2. Rerun the merge operation of relational extractor db, this tested path becomes a tagged path.
3. The tagged path based on the tag detail gets aggregated to the corresponding change set.
4. This tagged path shows up in the query for parallel test cases. And if it is an independent path, the test case can be regressed as a parallel test case.

The high-level use cases are:

1. Identify module-by-module testing dependency.
2. Carry over testing from iteration to next. In integration testing, lower level layer functionality is tested with a stub. If the stub is replaced by actual higher layer, with SAS we are able to identify the testing dependencies. This is possible due to the longest subsequence match of the change set which has been affected from one iteration to next. The regression test can be calculated in the current iteration and testing is performed efficiently based on it.

2. Bug Scenario

The logic for bug is similar to parallel regression for feature.

Example Defect:- Capacity column of UI – Erroneous value due to missing virtual capacity.

Change set contents:

Back end:

Before Defect fix: the path flow in the change set:

StorageArray.Insert(Array_Group.capacity[IN], float x [OUT]) →

AClass.populate(StorageArray) →

StorageArray.setCapacity(float xval) →

Normalized_collected_capacity(RAIDGroupCapacity, xval) →

CollectCapacity(RAIDGroupCapacity) →

Parse_CommandExecuted(output,command) →

ExecuteCLICommand(“getrg –capacity”) → (1)

After Defect Fix (Changed portions/ new portions in Bold):

StorageArray.Insert(Array_Group.capacity[IN], float x [OUT]) →

AClass.populate(StorageArray) →

StorageArray.setCapacity(float xval) →

Normalized_collected_capacity(RAIDGroupCapacity + VirtualPoolCapacity, xval) →

CollectCapacity(RAIDGroupCapacity) → **CollectCapacity(VirtualPoolCapacity) →**

Parse_CommandExecuted(output,command) →

ExecuteCLICommand(“getrg –capacity”) → **ExecuteCLICommand(“getvpool –capacity”)** → (2)

The contents above in path (2) in bold show the additional changed paths in the call graph and in the change set after the defect fix.

Now the contents of the change set<Storage Array> before and after the defect fix are :

Before:

Tag<GetCapacity> ➔ (1)

Tag<Show Connectivity & Capacity> ➔ (1) → Tag<list of RAIDGroups>

Change set after the defect fix:

Tag<GetCapacity> ➔ (2)

Tag<Show Connectivity> ➔ (2) → Tag<list of RAIDGroups> ->Tag<list of VPools>

The above shows that apart from GetCapacity test cases, the test cases that need to be regressed for the specific bug fix are

“Show Connectivity” which in turn contains the test case for:

a) List of RAID groups

b) List of Virtual Pools

This is deduced from the path graph queried from the change set for the feature, Storage Array.

The concept of tagged path and change set also narrows down the combinatorial explosion of paths returned from the query of the structural DB.

Since the black box test case is tracked with the actual path traversed in the code, the accuracy of the method in regression identification is possible.

In the above example, Tag<Show Connectivity> change set and Tag<GetCapacity> change sets are the sub-change sets of the Tag<StorageArray> change set.

Like this there could be <StorageTapes>, <DiskLibrary> and <Router> change sets modeled after the feature test cases in the product.

Defect Regression:

> Related (connected) paths changed paths in the change set + paths not covered in the changed portion of the latest call graph (code). This is an example of new test cases identified by the SAS system.

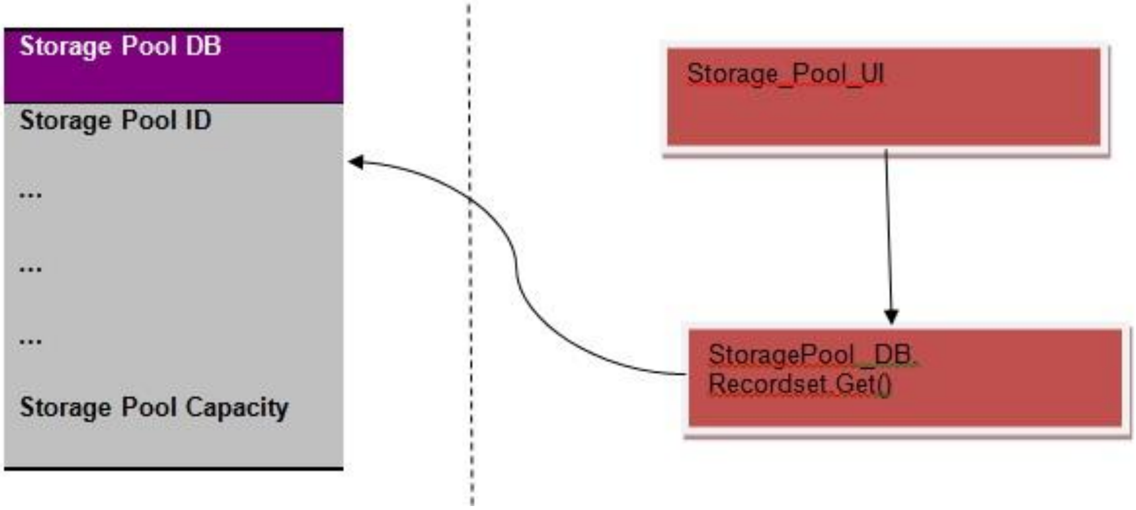
Consider a case where there are N number of defect fixes given in a patch. Then the group of defects can be regressed as an incrementally developed change set. This is the advantage of the hierarchically organized change set in the SAS system.

3. Another Bug Trace through the SAS tool

Bug 2: Negative value in capacity field of storage pool attribute

Query the Extractor DB for Storage Pool Feature.

- Identify the Change Set – for UI, DB and Back-end layer for the storage pool feature. The output will be something like this:



Changeset ID: Array.StoragePool

PopulateDB(StoragePool, StoragePoolID, float Capacity)

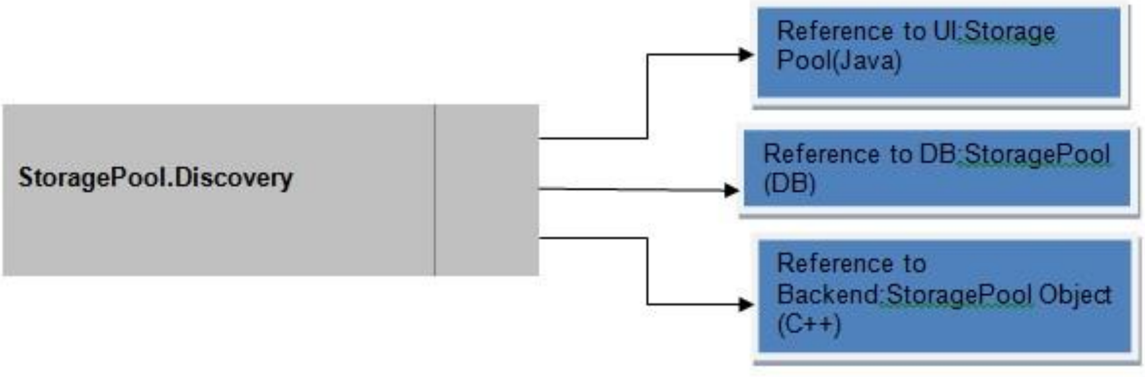
Change Set is a list of references to the call graph (represented in DB) queried at each layer of the software for each feature.

Query:

Get the information about “StoragePool.Capacity” in StoragePool.Discovery change set

By discovery we mean how the Storage Pool data found its way in to the UI field.

ChangeSet --------- List


StoragePool.Discovery
Reference to UI:Storage Pool(Java)
Reference to DB:StoragePool (DB)
Reference to Backend:StoragePool Object (C++)

Hierarchical queries will fetch the following information:

UI:

StoragePool_UI.field

StoragePool_DB.recordset.Get(Capacity)

Query:

DB.StoragePool(StoragePool_DB, capacity)

The query output of the above query will lead to the following functionality in the Back End (C++ layer):

StoragePool.insert(StoragePool.AClass.capacity, float x);

[AClass is a structure in which data is sent from the C++ Backend layer]

SAS Query: Get the info of StoragePool.AClass(capacity) =>

Aclass.Populate(StoragePool) =>

Aclass.StoragePool.SetCapacity(float xVal) =>

SAS Query for xVal in Backend Layer:

Yields normalize([in]arrayCapacity, [out] xVal) => This is the function where the bug lies.

Regression:

Execute Test => Stack Recordings

Compare Path => Verify coverage takes place (minimal set regression)

This enables rigorous testing during each iteration of development before the product gets shipped.

Here the advantage of SAS concept of change set is that because it tracks the changes across the layers we know the changes corresponding to a feature/bug in totality.

A group of change sets for the corresponding features can be provided to testers to test based on coverage. Based on the change set coverage in the overall call graph structural DB, we know the paths covered, yet to be covered, identify parallel automatable test cases and uncover new defects.

How SAS tool helps in testing during product development:

SAS-tool based actual dependencies of test cases on the basis of real code of the software are more accurate than perceived dependencies of the test cases.

Bug: Now the fix for the bug is related to the field size of the data member leading to overflow error. This fix needs to be made at all the layers of the software (UI, middle DB, Backend C++). The utility of SAS is that the call-graph changes for this fix is stored in a change set (remember the fix is at all layers, so the change set is a list of changes at each layer). This storage of change set can be retrieved quickly for further queries thereby leading to a system which keeps improving.

Regression Testing:

Once the bug fix is made,

Execute Test => Stack Recordings

We can query the DB for paths relative to this path, if it is an independent path or else it's a dependent path. All the dependent paths are extracted from the tags we can identify the corresponding test cases and they need to be rerun for minimal set regression.

The use of SAS here is not just for regression of a single bug but dependencies between bug fixes (testing of incremental patches that have been provided). This helps the tester tremendously.

8. Applications of SAS

The applications of SAS can be summarized as follows:

1. Identify Parallel test cases in test case regression of a feature/group of related features.

   The interesting aspect of this kind of parallel testing is it easily applies to the Agile model of software development, which calls for a demonstrable partial product of the overall product in each iteration cycle (sometimes called a sprint) of the software development process.

   The following questions can be answered by the SAS tool:-

   - How could testing take place?
   - What are the regression implications as more and more features are added to the product?
   - How to we identify opportunities for parallelism?

   Another use of the SAS tool is the test case categorization in terms of identifying the embedded test cases and those which are not.

2. Bugs and Root Cause Analysis.

An example for this application of SAS tool is given in the case study 1.

3. Bugs and Regression

   A bug fix is made with the help of SAS RCA analysis. The structural DB is updated after the bug fix. Changes set for bug is calculated – Minimal set regression of the bug across a multi-tier application is possible in the SAS tool.

4. Test Execution Tracking (based on comparator and Structural DB)

   The SAS tool helps in automatically identifying test cases that were not covered in one level of testing- say earlier iteration or unit testing. This helps in identifying missing pieces in development testing.

Logical to Physical Mapping of test cases helps in:

1. Minimization of regressions (feature).
2. Dependency – Visual dependency of test cases - an application of which is parallel test cases. Visualization of test cases -
   a) An organization of test cases that is from ad-hoc organization to actual dependency based organization of test cases. It helps tremendously in testing optimization and automation.
3. As we model every static and run time object, it helps in root-cause analysis as well.
4. Minimization of bug regression.
5. Complex end-to-end test cases can be easily designed and validated. New test cases can be validated before execution itself by querying the SAS database.

9. Effectiveness of this approach:

1. Visualized Testing – it merges the benefits of white box/black box testing

2. Agile Process – Avoid delayed testing - SAS allows early and effective testing, SAS helps in integration testing and end-to-end testing.

3. More confidence in the quality of the product – visualized tracking of testing effectiveness. Metrics can be derived about the testing coverage as a proportion of the call graph DB.

   An important metric for testing using SAS is: Overall Project Testing effectiveness in terms of coverage. This can measured in SAS by means of cumulative change sets for a feature. This cumulative merged change set should be as close as possible to the Structural DB. The percentage of the match between the two gives the percentage of the overall project testing effectiveness

The effectiveness of this approach is that Queries of the SAS system can be cached and result set retrieval optimization increases the efficiency of the system.

The other aspect of its effectiveness in the maturing database is the fact that there could be many instances of the same bug found by many customers. All of them have similar RCA and bug regression, such instances can easily be tracked and handled by the SAS system.

The new metrics of testing possible due to SAS is the analysis of nature of bugs, inter-dependency of paths in the call graph structural DB leading to identification of inter-dependence of modules from the testing perspective. There has been work related to dynamic analysis for identifying the code impact (Law 2003) and also identify test cases (Frechette 2013), but they do not cover a complex multi-tier scenario and also the host of advantages of the SAS tool listed above. The storage based approach helps in internal vulnerability assessment leading to new test cases and possible new types of bugs in the multi-tier software system.

10. Conclusion

The objective of this paper is to provide a glimpse of what can be done with software analytics based on structural analysis and storage. For this purpose the SAS tool has been designed to work on call-graphs of the software at different architecture levels, by storing it in a database and processing it. In this we have specifically shown how the logical, user domain organization of test cases is mapped to physical path organization of the test cases based on the call graph and storage based approach. This enables a host of applications from developers to testers from Root-Cause Analysis to test case design, execution, tracking and measuring overall effectiveness of the testing.

References

Callahan, D.; Carle, A.; Hall, M.W.; Kennedy, K., "Constructing the procedure call multigraph," Software Engineering, IEEE Transactions on, vol.16, no.4pp.483-487, Apr 1990.

Frechette N, Badri, L, Badri, M, Regression Test Reduction for Object-Oriented Software: A Control Call Graph Based Technique and Associated Tool. In the proceedings of ISRN Software Engineering, March 2013.

Gerter O. 2004. A Database File System. An Alternative to Hierarchy Based File Systems. University of Twente.

Graham, Dorothy. The Foundations of Software Testing. Wiley Publications.

Grove, D., DeFouw, G., Dean, J., and Chambers, C. 1997. Call graph construction in object-oriented languages. SIGPLAN Not. 32, 10 (Oct. 1997), 108-124.

Law, J, Rothermel,G. “Whole program path-based dynamic impact analysis,” in *Proceedings of the 25th International Conference on Software Engineering*, pp. 308–318, May 2003.

Oxygen Static Analysis Tool for C, C++, Java, Python, etc.(http://www.stack.nl/~dimitri/doxygen/)

Patent: Method of implementing an acyclic directed graph structure using a relational data-base http://www.google.com/patents/US6633886

Ryder, B.G., "Constructing the Call Graph of a Program," Software Engineering, IEEE Transactions on, vol. SE-5, no.3pp. 216- 226, May 1979.

Stack Trace Analysis. https://profes2013.cs.ucy.ac.cy/files/GavrilovStackTraceAnalysis.pdf

Challenges with Test Automation for Virtualization

Sharookh Daruwalla

sharookh.p.daruwalla@intel.com

Abstract

The scope of software validation is always expanding with the ever-increasing test matrix making automation a necessary tool to alleviate the unmanageable workload on validation teams. However, extending automation for Virtualization OSes (e.g. VMWare ESXi) come with unique challenges not seen with traditional OSes (e.g. Windows, Linux). Most test execution engines support only one specific traditional OS running on a single System under Test (SUT). Further, they are also designed for static resource management and usually support only the handful of scripting languages and communication protocols preferred by the supported OS. This paradigm does not work for virtualization, where a SUT is only a host or manager of virtual machines (VM) running different Guest OSes. In addition, virtualization test times are also longer due to the additional steps required to setup VMs. This paper presents a detailed look into the challenges faced when automating test validation for Virtualization OSes and the design choices used to overcome them.

The proposed ITE-Berta test automation framework implements dynamic resource management within its libraries to support VMs created within tests, adds communication protocol support for multiple OSes, and uses cross-script execution to extend support for languages not natively supported in the test execution engine. To reduce virtualization test times, we implement a Test Continuation Mechanism that maintains setups and performs selective clean-up between similar tests. Finally, we analyze our proposed solution and how it may be applicable to different Virtualization OSes and test frameworks.

Biography

Sharookh Daruwalla is a network software engineer at Intel Corporation, currently working at the Jones Farms Campus in Hillsboro, Oregon. He has been with Intel for 3 years and, for the past 2 years, been working on automation architectures and extending their test automation framework for virtualization. Sharookh has a wide range of experience from driver development and network storage at Intel to research in hardware-software simulation methodologies at Portland State University. Sharookh has two M.S. degrees, in Computer Science and Electrical & Computer Engineering, both from Portland State University.

1. Introduction

Software validation is the process of confirming that a software product meets its design specifications and that it performs as intended. Software is normally validated against its test matrix, which can be defined as all hardware, operating systems (OS) and configurations as dictated by its use-case. As more hardware, OSes and configurations gets added, the test matrix increases and validation workloads can easily get out of control. Intel, for example, has more than a dozen Converged Network Adapters (CNA) that are validated on various OS distributions. With more than 800 test-cases and over a dozen supported configurations, the validation teams are tasked with assuring software quality across hundreds of thousands of unique setups. Further complicating matters are the relatively lengthy setup and breakdown times associated with storage network technologies. Future products will only increase the test matrix and the associated validation workload. In this scenario, test automation becomes a necessary tool to alleviate the unmanageable workload on validation teams.

1.1 Traditional versus Virtualization OS

While test automation is essential, its design and implementation need to be very different across different OSes. Today, operating systems can be categorized into two types - traditional and virtualization. The main difference between a traditional OS (e.g. Windows, Linux) and a virtualization OS (VMWare ESXi) is that traditional OSes [Figure 1 (Left)] have direct and dedicated control of their system's hardware resources, while virtualization OS [Figure 1 (Right)] share their hardware resources between multiple Virtual Machines (VM), where a VM acts as an independent system running its own traditional OS. The biggest advantage of the virtualization setup is that it enables maximum utilization of hardware resources. Understandably, this also means virtualization should support most, if not all, traditional OS distributions. This requirement makes test automation for virtualization more complicated than for traditional OS.

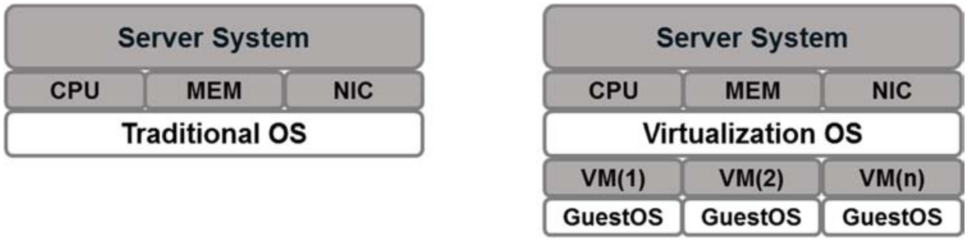


Figure 1: Traditional OS vs. Virtualization OS

1.2 Test Automation Frameworks

A typical test automation design involves three major components – a test scheduler, a test execution engine and a System under Test (SUT), as shown in Figure 2. The test scheduler's main responsibility is triggering tests on the SUT via the test execution engine. It also acts as a database of test automation resources including the various SUTs available, operating systems, configurations, test-cases, etc. The test execution engine holds all the test scripts and is responsible for executing them on the SUT.

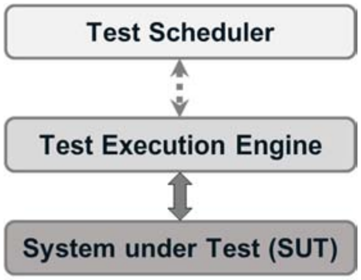


Figure 2: Typical Test Automation Design

When moving test automation from traditional OS to virtualization OS, one of the biggest paradigm shifts is the expected ratio of SUT to OS. Traditional OS have a 1:1 ratio i.e. one OS running on one physical system. In contrast, Virtualization OSes create multiple VMs (N) that share the SUT's physical resources but run their own traditional OS. Therefore, Virtualization OSes have a 1:N SUT to OS ratio. This new paradigm is unknown to typical test execution engines. They handle system resources statically i.e. the details of a SUT and its OS are known before a test is executed. But this does not work for virtualization because the VMs are SUTs that are not created till after the test execution begins. Therefore, for virtualization, resource management needs to be handled dynamically.

Other than the need for dynamic resource management, virtualization automation efforts have to deal with other complexities such as supporting all traditional OS including various communication protocols, programming languages, etc., setup and teardown overheads to create-remove VMs per test, lack of support for ghosting various OS flavors and lack of support for advanced virtualization techniques (e.g. clustering, migration, etc.). Without these features, test automation framework for virtualization would be incomplete and have limited use. In this paper, we present a working solution that overcomes most of the challenges faced with virtualization automation. The solution has been successfully implemented to extend our Windows test automation framework for VMware ESXi. Further, the solution can also be used to extend automation for other Virtualization OS such as Hyper-V and KVM.

1.3 FCoE Storage Networking

Storage Area Networks (SAN) are dedicated networks built to provide fast and reliable access to consolidated data storage systems. The main purpose of SANs are to provide a centralized data storage location easily accessible to servers such that the data appears as if it were locally present on the server. In SANs, servers initiating the data requests are known as *initiators* while data storage devices that service these data requests are known as *targets*. Each target maintains a large number of storage hard drives that are identified by their Logical Unit Number (LUN). The initiator can access data from targets using a number of protocols including Fiber Channel (FC), SCSI over TCP/IP (iSCSI), and Fiber Channel over Ethernet (FCoE). For this paper, we take a look at a test automation framework used to validate the FCoE protocol.

Datacenters typically use Ethernet for LAN networks and Fiber Channel (FC) for SAN networks. Integrating storage-popular FC networking with the more widely adopted Ethernet medium is very desirable since it allows FC networks to take advantage of faster 10 gigabit or 40 gigabit Ethernet networks. Fiber Channel over Ethernet (FCoE) is the networking technology used to enable FC network over Ethernet by encapsulating Fiber Channel (FC) frames within Ethernet packets. Figure 3 shows the typical FCoE setup from an initiator to a storage target. The initiator is typically connected to one or more Fiber Channel Forwarder (FCF) switches that are responsible for transporting FCoE frames over the Ethernet network. Further, FC frame are extracted from the FCoE frames by a Fiber Channel switch (FSW) to access a target within the SAN network.

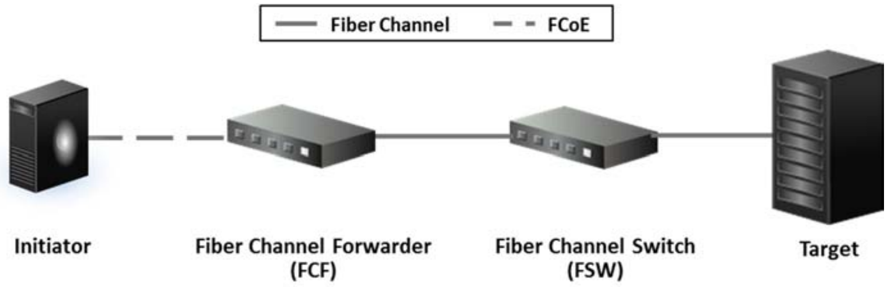


Figure 3: Fiber Channel over Ethernet (FCoE) Setup

The remainder of the paper is divided into following sections: Section 2 overviews the ITE-Berta test automation framework. Section 3 takes a detailed look at the challenges with automating virtualization while Section 4 presents our solution in ITE-Berta. Section 5 present and analyze our results, and we conclude in Section 6.

2. ITE-Berta Test Automation Framework

The ITE-Berta test automation framework was originally developed with Windows storage validation in mind. It was extended to support virtualization storage validation at a later stage. The main goals of the ITE-Berta automation efforts were:

1. **Extensible Design with Shared Libraries** – Libraries should be OS-agnostic and easy to extend by other teams. The central repository for these shared libraries will be maintained by one owner, while each team will also have an owner to maintain the team repositories with the central repository.
2. **Maximize Resources** – Maximize software re-use by leveraging existing solutions wherever possible. Also, share the same infrastructure with manual testing and use virtualization to maximize available resources.
3. **No-Touch Automation** – Should be capable of performing complete *no-touch* testing, on new component and project builds, from Basic Acceptance Tests (BAT) to Product Tests.
4. **Extend Validation** – Should enhance overall software quality, not replace manual validation. This is achieved by categorizing test-cases as either automatable or not-automatable. Not-automatable indicates either unavoidable manual intervention or that the effort required to automate is too large.
5. **Adding Value** – Not only use the software tools available but also contribute in their development.
6. **Consistency** – To maintain consistency within testing, automated test will be mirror-images of their manual counterparts.

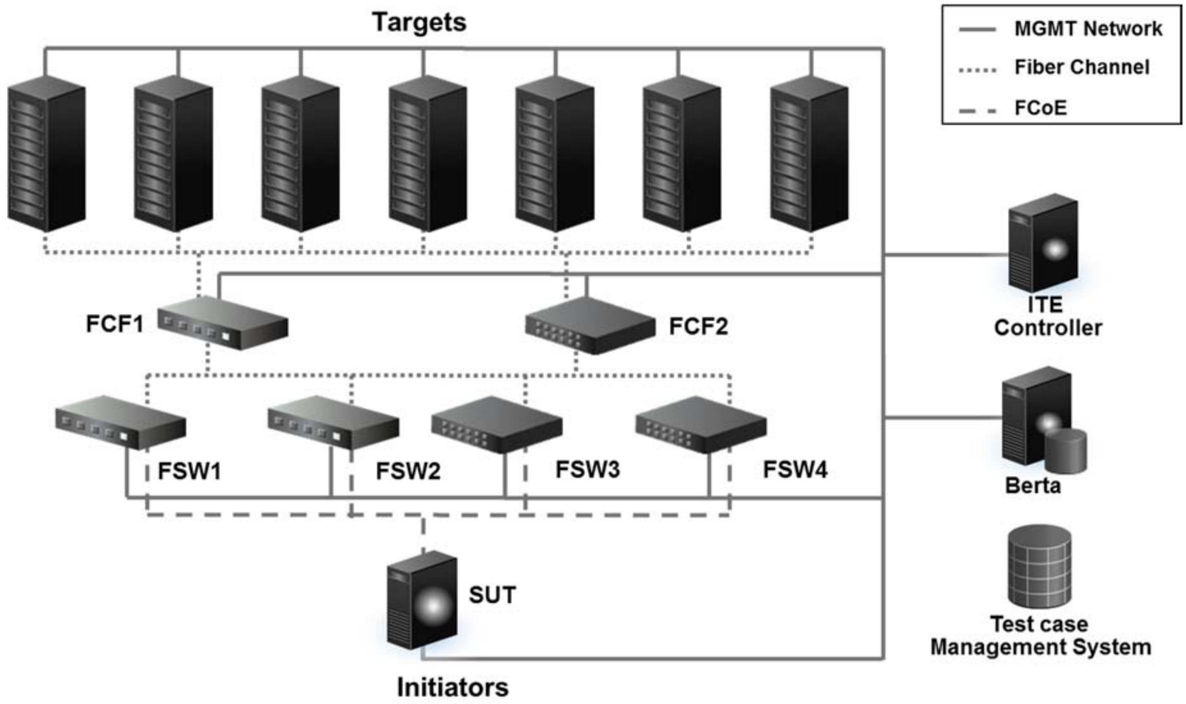


Figure 4: Test Automation Setup for FCoE Validation

The ITE-Berta test automation framework consists of two in-house software tools – the Integrated Test Environment (ITE) and Berta Test Scheduler. ITE is the Test Execution Engine responsible for maintaining and executing all test scripts with their associated libraries. ITE was originally developed for Windows and has native Windows support. It supports VB Script and Java Script and has a built-in communication protocol, within the ITE Agent, to communicate with the Windows test OS. Berta is a test scheduling solution that is used to automatically trigger tests. Berta also maintains a database to store various SUTs, adapters, OS flavors, and other test configurations. It uses a web-based interface to schedule tests and is not only capable of managing multiple ITE test controllers but can also intelligently select SUTs for testing. It is capable of providing test metrics, archive results and send email notifications at the completion of a test. Using Berta in conjunction with ITE allows us to build a complete *no-touch* test automation solution. Figure 4 shows the storage team's test automation setup including different FCoE Targets and switches. In order to best utilize our resources, we went through our test-cases and identified the test-cases to automate. Based on the *non-automatable* criteria described above, we identified that approximately 70% of the storage test-cases were *automatable*.

3. Virtualization Automation Challenges

Virtualization is based on the idea that a system's hardware resources can be shared across multiple virtual machines (VM) where each virtual machine runs its own traditional OS. Virtualization OSes run directly on the hardware and are commonly known as either a *host* or *virtual machine manager*. The host's main responsibility is managing the VMs as well as acting as a bridge between them and external environment. Therefore, when adding support for Virtualization OSes within a test automation framework, we need to add support for not just the host but also VMs. The major challenges with virtualization automating include static resource management and single OS support within the test execution engines. Other challenges include long setup and teardown times associated with setting up virtual machines, lack of support to test advanced features like migration and clustering.

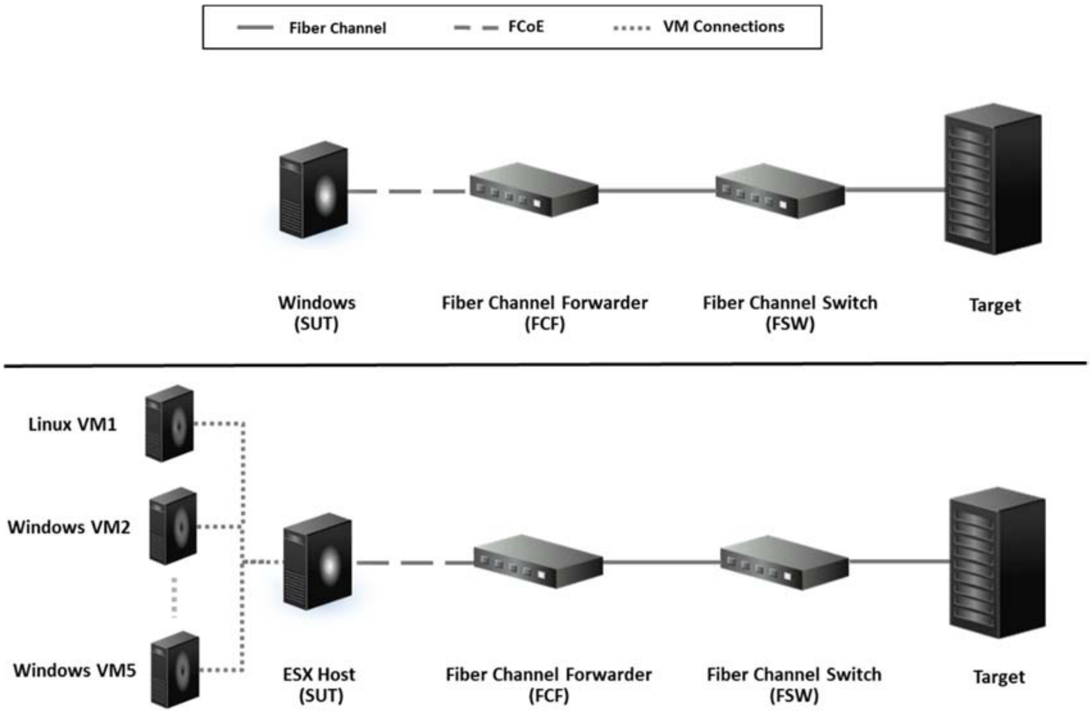

Fiber Channel
FCoE
VM Connections
Windows (SUT)
Fiber Channel Forwarder (FCF)
Fiber Channel Switch (FSW)
Target
Linux VM1
Windows VM2
Windows VM5
ESX Host (SUT)
Fiber Channel Forwarder (FCF)
Fiber Channel Switch (FSW)
Target

Figure 5: Typical FCoE Setup for Windows versus ESXi

To better explain the complexities brought in by virtualization, let's take a look at a trivial storage FCoE test comparing Windows versus VMWare ESXi. Table 1 compares the various steps required for Windows versus ESXi while Figure 5 shows the final setup for both OSes. As shown in the figure, the Windows OS will run directly on the SUT (top), while ESXi will host the VMs (bottom). We can also see that the SUTs in virtualization are multiple OSes, unlike that for Windows. Initially, the test OS will be the Host. However, once the VMs are setup, the OS running on the VMs become the test OS. Further, since there are multiple

VMs and each is to be considered as an independent machine, we need to dynamically connect to each VM to execute its required operations. Effectively, for virtualization, the test validates one physical system running the ESXi Host along with 5 virtual machines, each with its own OS.

	Traditional OS (Windows)	**Virtualization OS (ESXi)**
1	Establish connection from SUT to a Target (via FCF - FSW)	
2	Add 1 LUN of required size	Add 5 LUNs of required size
3	*Not applicable*	Create a DataStore on the LUNs, if required
4	*Not applicable*	Create 5 VMs with Windows Guest OS
5	*Not applicable*	Attach 1 LUN to each of the VMs as VM disk
6	Run FCoE traffic between SUT and Target	Run FCoE traffic between VMs and Target

Table 1: Comparison of Test Procedures for Traditional vs. Virtualization OS

3.1 Dynamic Resource Management

ITE manages its hardware resources such as SUTs, Peers statically. This means that all SUTs and Peers for a test need to be defined, configured, and selected before the test run. While this model works well for traditional OS validation, it is not practical for Virtualization OSes. In the example discussed in Figure 5, the Windows OS is running on the SUT and all required information (e.g. MAC Address, IP Address, Hard drive, etc.) is static and known from the very beginning. In comparison, for ESXi, we have the required host information but not the VMs that are yet to be created in Step 4 (Table 1). Each of the five VMs, when created, will be given MAC addresses, static/dynamic IP addresses, hard disk names and numbers, etc. This information is not likely to be known prior to running the test because the VMs do not exist then since creating VMs is part of the test itself. Also, once the VMs have been setup, we need to control them dynamically as well. Because of these complications, we need to find a way to dynamically manage and control resources while running a test so that we can create new systems as needed as well as select which system from all the systems is the *current* system under test.

3.2 Multiple OS Support

Most test execution engines are developed with a particular OS in mind. Since Windows and Linux differ substantially, from their choice of communication protocols to preferred development languages, focusing on one OS makes sense too. ITE was originally developed for Windows automation and all its features are Windows focused. It, therefore, supports only VB and Java scripting languages and offers built- in support for .NET libraries as DLLs. It also has its own proprietary agent to communicate with the Windows test OS. However, there is not sufficient support for virtualization where both Windows and Linux need to be tested. If ITE only supports Windows, then we are really only testing the Virtualization OS for half its capability.

Extending ITE support for Linux while absolutely necessary comes with its own complexities. First, since Python is the preferred scripting language in Linux, either Python needs to be supported within ITE or Linux libraries using VB or Java scripts need to be developed. The former approach requires adding a Python compiler within ITE but comes with the advantage of being able to leverage existing python scripts. The latter approach is faster and easier but requires building everything from scratch. This gets further complicated when we try to extend ITE to support ESXi which is PowerCLI based, where PowerCLI is a Windows PowerShell module. Here, too, we face a similar problem – either support PowerCLI and use ITE or discard ITE and lose all the libraries that can have been built for Windows. Since VMWare does not support any other language, using PowerCLI within ITE would mean invoking PowerCLI scripts from VB Script or C#. This workaround would not only make script execution slower but has the potential to make debugging PowerCLI within VB Scripts tedious and difficult.

3.3 Complex Setup, Teardown

Virtualization OSes requires a more extensive setup than traditional OSes since they need to setup multiple VMs, with their own OS, along with all associated configurations. The total time it takes to complete its setup is dependent on the number of VMs. The percentage of the total test time consumed by virtualization setup in an automated process is quite high. To add to it, we must also allocate time to teardown the setup at the end of a test so that the SUT is ready for its next test. This teardown time is usually the reverse of the setup adding to the total test time for virtualization.

Table 2 compares step-by-step the average time required for both traditional and virtualization OS to complete the test-case in Table 1. Notice that not only do Virtualization OS setups have more steps to complete but also that these steps need to be multiplied with the number of VMs we want to setup. In the reverse, teardown too is dependent on the number of VMs to remove. The actual test time is the only step that both traditional and virtualization take similar time for. As expected, Virtualization OSes take close to 2x test time when compared to traditional OS. Further, this difference will increase with the number of VMs. Therefore, there is an urgent need to find a way to reduce these test times else much more hardware will be needed to run similar number of tests as traditional OS validation.

		Traditional OS (minutes)	**Virtualization OS (minutes)**
1	Establish connection between SUT-Target (via FCF-FSW)	25	25
2	Add required LUNs from Target	4	(2mins/LUN) 10
3	Setup 5 VMs (DataStores, VMs and OS, LUNs)	-	(10mins/VM) 50
	Total Setup Time:	**29**	**85**
4	Run SAN IO (SUT/VM – Target) for various packet sizes	30	30
	Total Test Time:	**30**	**30**
5	Remove VM setup (delete DataStores, VMs, etc.)	-	(4mins/VM) 20
6	Release Target LUNs, SUT to Target connection	24	(4mins/LUN) 20
	Total Teardown Time:	**24**	**40**
	Total Test Time	**83**	**155**

Table 2: Comparison of required testing time for Traditional vs. Virtualization OS

In order to successfully implement virtualization automation, these three challenges need to be addressed and resolved. Without addressing them effectively, virtualization automation will be stuck in a semi-automated state requiring constant manual supervision and intervention. In the next section, we present a working solution for Virtualization OSes.

4. Test Automation with Virtualization

When implementing Test Automation for Virtualization OSes, we evaluated three feasible design options:

1. **PowerCLI Only:**
   Create a new framework with scripts written in the preferred language for each OSes, instead of the existing ITE-Berta framework. This approach gives the freedom to develop a framework from scratch with preferred languages per OS i.e. PowerShell for Windows and ESXi, and Python for Linux. This

would not only allow the use of existing scripts but also simplify scaling across other teams. However, its biggest disadvantage is having to re-write all the ITE-Berta libraries (VB Script, C#).

2. **C# Wrapper:**
   This approach would encapsulate all relevant ESXi and Linux scripts and operations into the C# libraries. It would allow us to use the existing ITE-Berta framework and its libraries. However, it also means double-wrapping ESXi and Linux scripts inside C# libraries. This could potentially increase execution and debugging time, create various library maintenance problems, leading to lower framework adoption across teams.

3. **Hybrid:**
   The third option is a hybrid solution. In this approach PowerCLI libraries would be used for ESXi but would be executed within ITE via VB Script. Also, Linux OS support would be extended within the VB Script, C# libraries, thereby maintaining consistent libraries for both traditional OSes. The biggest disadvantages are maintaining Linux in VB and invoking PowerCLI from within VB scripts but has the advantage of using PowerCLI for ESXi within the stable ITE-Berta environment.

Ultimately, we decided to go Hybrid because it gave us the best trade-off between either creating a new framework to implementing complicated double-wrapped libraries. Figure 6 below shows the solution used to extend the ITE-Berta framework for Virtualization OSes. Once the ITE-Berta framework was decided to be extended for Virtualization OSes, we addressed the three major challenges discussed i.e. dynamic resource allocation, multiple OS support, and reducing virtualization test times.

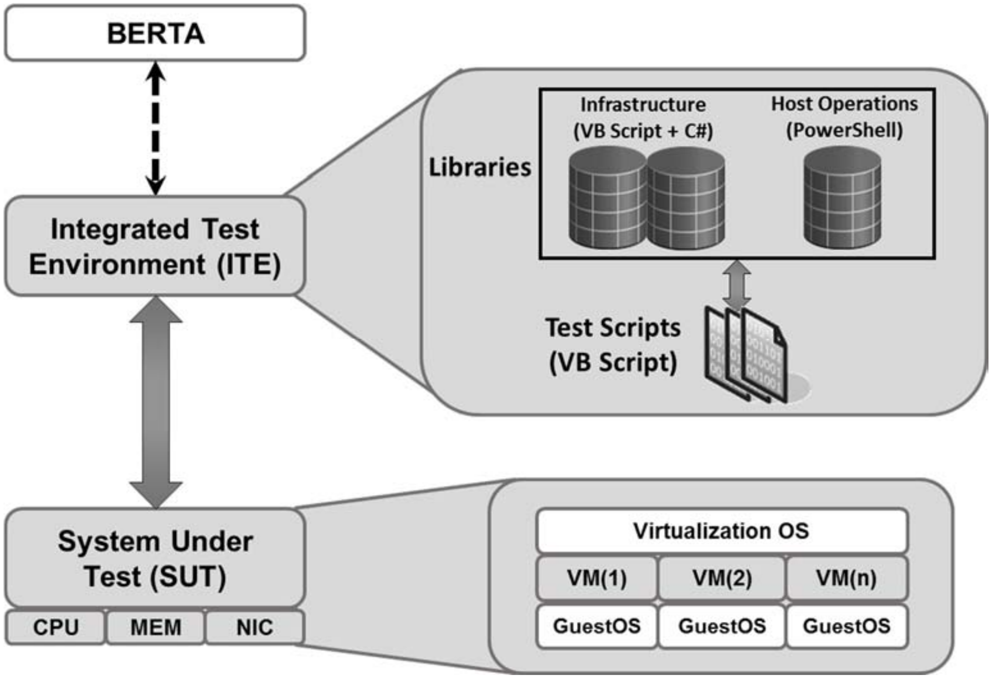
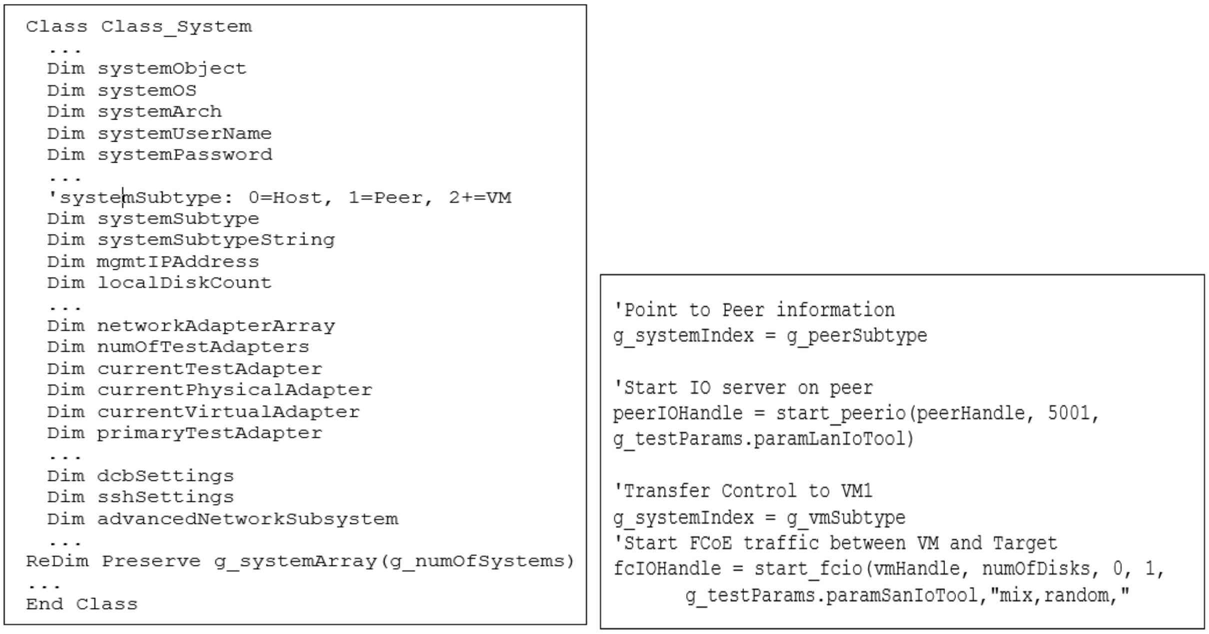


Figure 6: Virtualization Test Automation using the Hybrid Solution

4.1 Dynamic Resource Management

Since ITE only supports static management of test resources, dynamic resource management was implemented within the libraries itself. Integrating resource management within the libraries required major refactoring of all libraries and was implemented in two parts – 1. Record the new resource and 2. Assign control to correct resource. To do this, the libraries were modified the *system* class to support 3 types of test systems - SUT, Peer, and VM. For virtualization, the initial SUT would be the host followed by the VMs. In networking, *peers* are traffic partners added to the testing environment. Each of these 3 types have data structures that hold all relevant information based on the type of system. Figure 7 (Left) shows the `Class_System` defined in the libraries and the relevant OS and network information it holds.

Each time a new dynamic system is created, a corresponding new entry is added into the systems array. Once all systems (physical and virtual) have been setup and recorded, a global pointer (`g_systemIndex`) is used to toggle control between the various systems. Figure 7 (Right) shows sample code used to first assign control to a peer in order to start LAN traffic between it and the SUT after which control is handed to VM1 to start SAN traffic between it and target. Therefore, use of the global pointer `g_systemIndex` allows a form of dynamic resource management to be implemented on top of ITE's static resource management structure.



```
Class Class_System
  ...
  Dim systemObject
  Dim systemOS
  Dim systemArch
  Dim systemUserName
  Dim systemPassword
  ...
  'systemSubtype: 0=Host, 1=Peer, 2+=VM
  Dim systemSubtype
  Dim systemSubtypeString
  Dim mgmtIPAddress
  Dim localDiskCount
  ...
  Dim networkAdapterArray
  Dim numOfTestAdapters
  Dim currentTestAdapter
  Dim currentPhysicalAdapter
  Dim currentVirtualAdapter
  Dim primaryTestAdapter
  ...
  Dim dcbSettings
  Dim sshSettings
  Dim advancedNetworkSubsystem
  ...
ReDim Preserve g_systemArray(g_numOfSystems)
...
End Class
```

```
'Point to Peer information
g_systemIndex = g_peerSubtype

'Start IO server on peer
peerIOHandle = start_peerio(peerHandle, 5001,
g_testParams.paramLanIoTool)

'Transfer Control to VM1
g_systemIndex = g_vmSubtype
'Start FCoE traffic between VM and Target
fcIOHandle = start_fcio(vmHandle, numOfDisks, 0, 1,
      g_testParams.paramSanIoTool,"mix,random,"
```

Figure 7: Dynamic Resource Management

4.2 Multiple OS support

The ITE libraries were originally designed for Windows automation but could also be extended for other traditional OS. However, for virtualization we had to categorize the libraries into two types: common and virtualization. The common libraries were extended for features common to all OSes including virtualization while the virtualization libraries held only virtualization specific features. Further, the virtualization libraries were designed to extend to other virtualization solutions such Hyper-V, KVM in mind.

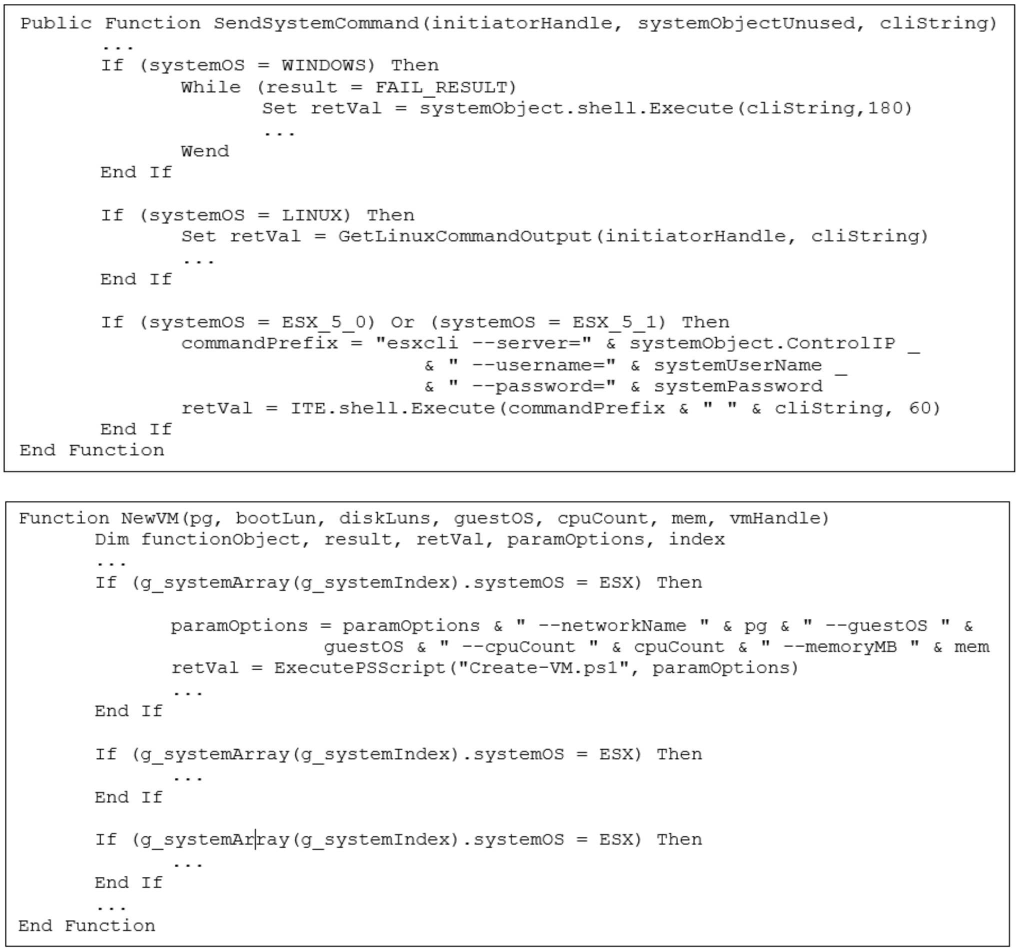
Figure 8 shows sample code for both types of libraries. The `SendSystemCommand` (top) encapsulates communication methods for all OSes. Windows is directly supports by ITE while Linux and ESXi are use the Secure Shell (SSH) protocol. The major difference between Windows and Linux is that output or error parsing is handled by ITE for Windows while the libraries handle them for Linux and ESXi. The `NewVM` (lower) is a virtualization specific function responsible for setting up new virtual machines on a virtualization host. As can be seen, the ESXi calls have a mix of both VB Script and PowerShell. PowerShell scripts are executed wherever ESXi host operations need to be performed while all other operations are worked out in the VB libraries.

4.3 Test Continuation Mechanism

As discussed in the section 3.3, total test times for virtualization OSes can be 2x or longer than those for traditional OS and are usually directly dependent on the number of VMs required by the test. Further, we also demonstrated in Table 2 that while the actual test for ESXi only required 30 minutes, its setup time needed 85 minutes (3x) and its corresponding teardown time took 40 minutes (2x). Since creating VMs is part of the testing itself, it's unlikely that we would be able to reduce the setup-teardown times for the individual test. However, there is a potential to reduce them when placed in groups.

Validation teams rarely completely teardown setups per test. They only clean-up enough of the setup to be able to run the next test without any bias from a previous test setup. For example, there is little need to remove VMs after Test 1 if Test 2 is going to require similar number of VMs. The quickest clean-up would ensure the VMs are cleared off of any setup related to Test 1. This type of selective clean-up can save valuable time and trouble associated with VM setup. We have, therefore, implemented a similar mechanism called Test Continuation Mechanism to use a similar procedure with similar-setup test groups.

Berta allows us to create test scenarios, essentially a group of tests that are run sequentially in a given order. We performed a detailed test-case classification of all available virtualization test-cases and found that there were four major differentiation points within them: number of VMs, number of LUNs, SAN/LAN IO packet sizes, and IO run times. We decided to group together test-cases that used very similar setups into the same test scenario so that they would be executed in single test run. Since these tests would be run together and required the same setups, we then added logic within the framework to only run the entire setup for the first test, while performing minor required clean-up in between the rest of the tests. Finally, the setup would be torn down at the end of the final test. Figure 9 shows the flowchart for the Test Continuation Mechanism.



```
Public Function SendSystemCommand(initiatorHandle, systemObjectUnused, cliString)
        ...
        If (systemOS = WINDOWS) Then
                While (result = FAIL_RESULT)
                        Set retVal = systemObject.shell.Execute(cliString,180)
                        ...
                Wend
        End If

        If (systemOS = LINUX) Then
                Set retVal = GetLinuxCommandOutput(initiatorHandle, cliString)
                ...
        End If

        If (systemOS = ESX_5_0) Or (systemOS = ESX_5_1) Then
                commandPrefix = "esxcli --server=" & systemObject.ControlIP _
                                        & " --username=" & systemUserName _
                                        & " --password=" & systemPassword
                retVal = ITE.shell.Execute(commandPrefix & " " & cliString, 60)
        End If
End Function
```

```
Function NewVM(pg, bootLun, diskLuns, guestOS, cpuCount, mem, vmHandle)
        Dim functionObject, result, retVal, paramOptions, index
        ...
        If (g_systemArray(g_systemIndex).systemOS = ESX) Then

                paramOptions = paramOptions & " --networkName " & pg & " --guestOS " &
                                guestOS & " --cpuCount " & cpuCount & " --memoryMB " & mem
                retVal = ExecutePSScript("Create-VM.ps1", paramOptions)
                ...
        End If

        If (g_systemArray(g_systemIndex).systemOS = ESX) Then
                ...
        End If

        If (g_systemArray(g_systemIndex).systemOS = ESX) Then
                ...
        End If
        ...
End Function
```

Figure 8: Multiple OS Support

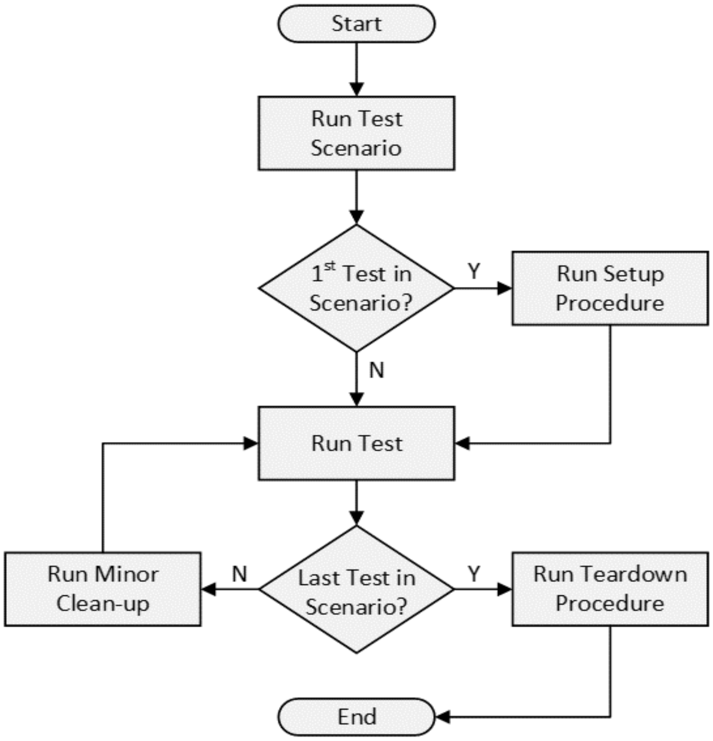


Figure 9: Test Continuation Mechanism

5. Results & Analysis

We have completed implementing dynamic resource management within the ITE-Berta libraries, as well as adding support for Linux and VMWare into the libraries. While the library implementation works well, there are some compatibility issues for Windows Hyper-V. We are, therefore, moving dynamic resource management into ITE's source code itself. We have also completed our test-case classification and have begun implementing the Test Continuation Mechanism into the test scripts. Currently, we have automated more than 50% of the ESXi test-cases. We have also setup the framework for no-touch automation for both product as well as component testing and plan to add developer-level smoke testing. Our next step is to start extending our framework for Hyper-V and KVM support.

Extending automation for virtualization has had its advantages and disadvantages. One of the biggest advantage has been the reuse of the existing ITE-Berta framework itself. It also creates a feasible common automation framework that can be used for both traditional and Virtualization OSes. Further, the PowerCLI scripts have had a dual purpose – they are used within the automation framework as well as to ease manual testing by automating parts of the test. In contrast, a big disadvantage is related to PowerCLI scripts execution from within VB. While the execution has not seen a high performance drop, debugging PowerCLI scripts in this scenario has been particularly painful. Also, the current arrangement makes scaling more tedious for other teams as they need to be familiar with the VB based ITE libraries.

6. Conclusion

In this paper, we took a detailed look into the challenges with test automation for virtualization and design choices that can be used to overcome them. We showed that with dynamic resource allocation, multiple OS support and test continuation mechanism, it is possible to extend current traditional OS test automation frameworks for virtualization testing. Finally, we discussed results from our ITE-Berta framework implementation, and analyzed our solution's strengths and pain-points.

How Can I Automate Stuff?

A guide for the beginner on making their testing life easier

Alan Ark
Software Development Engineering in Test - Senior
Eid Passport
aark@eidpassport.com
http://www.linkedin.com/in/arkie

Abstract

Automated testing is a hot topic right now in the QA space. But how do you get started in it? The answer is one step at a time. This talk is aimed at the beginner and hopes to provide some background in how to start to learn how to automate stuff. Stuff that can make your job easier.

Even if you **think** you don't have any automation experience, you probably actually have something that you can build on. This paper gives you one path as a guide, but ultimately it will be up to you to complete the journey.

Biography

__Alan Ark__ is a Software Development Engineering in Test (SDET) - Senior at Eid Passport, in Hillsboro, Oregon. Alan has gained tremendous experience working for Viewpoint Construction Software, Compli, Unircu, Switchboard.com, and Thomson Financial – First Call. Mr. Ark has previously presented 'Euro: An Automated Solution to Currency Conversion' at Quality Week '99, 'Collaborative Quality: One Company's Recipe for Software Success' at PNSQC 2008 and 'YES! You CAN Bring Test Automation to Your Company!' at PNSQC 2011. His LinkedIn profile can be viewed at http://www.linkedin.com/in/arkie

1. Introduction

Looking at the open jobs listings, it is pretty easy to find companies asking for test automation experience. Software Development Engineer in Test (SDET), test frameworks, Selenium, Java, .Net. These are just some of the skills listed on job descriptions. When the phrase “test automation” is used, many people think of large scale test frameworks that exercise the system under test from end-to-end. That can be hard work and it is not recommended that a novice automator think about solving that problem just yet. The question is - how do you get that experience with automation if you do not know where to start? One place to start is with the definition of what automated testing can mean.

The promise of test automation is a very broad one that is fraught with danger. Many times, managers think that automation is the silver bullet that can solve all manual testing ills that live at their companies. Others have the opposite thought and believe that all automation projects will end up as shelfware and represent a large cost with little return on investment. This paper will help those on both ends of the spectrum find the right balance to learn automation and set up the effort for success.

The idea of test automation can be expanded to include ideas that are not test cases in themselves, but things that need to be in place to support the running of tests - automated or even manual ones. Some examples include setup of test environments, clearing out log files, and creation/deletion of data. If a Windows application exists for a piece of Microsoft functionality, there usually is a way that the same functionality can be accomplished from the command line. Put a few of these commands together in a batch (Microsoft TechNet website) or powershell script (Script Center website) and you have created automation!

Anything that you do manually over and over again might be a candidate for automation. If you can automate that process, it will free you up to finish other tasks. And this is where our journey begins. This paper is aimed squarely at the beginner QA engineer, who does not have a programming background, but is looking to get more involved with automation. One of the goals of this paper is to give you some ideas on how to learn the skills needed for automation (not just test automation!). Another goal is to give you some leads on further learning on your own.

The section titles come from song titles by the Beatles. Just as this talk is aimed at the beginner, many musical artists took their cue from the Beatles at one point.

2. "Don’t Let Me Down"

How some managers view automation efforts can be summed up in one word – disappointment. Some automation projects have failed to the point that all automation efforts are viewed in a negative light. To prevent this from happening, we need to show that failure is not always the case. They need to know that not all automation efforts are bad. With the right scope and design, small efforts can show great gains in productivity – the ability to work smarter, not harder. The creation of automation can be a difficult thing to achieve when you have no experience in automation to begin with. The idea is to pick the right project to start off with, to have it (and you) succeed, and show that you will not let your manager down.

Learning automation is a skill that does not come overnight. As with all learning activities, you get out only what you put in. You might have to do some research outside of office hours. You might have to read books, blogs, and white papers. You will have to talk to other people for ideas and reviews. You might have to work outside of your comfort zone. You may think that learning how to automate is hard. You may think that you cannot do it. You may think that no one will help you. Don’t let yourself down by dwelling on these thoughts. Learning how to automate is a challenge, but like other challenges, you can rise to it and learn from it. While it is true that automation is not for everyone, most people can show some level of success using the ideas that they learn from the research that they have done.

While the goal of this paper is to inspire questions and thoughts about how to learn automation skills, there are a few items that are beyond the scope of this paper. These are placed here as to not let you down.

- Build a test automation framework
- Learn how to program
- Teach you how to use Selenium. (SeleniumHQ webpage)

3. "Come Together"

What is the plan? People like to hear what has worked for others. This provides validation that someone else has achieved some level of success with their approach. The real question that needs to be answered is “What are MY pain points?” Only by learning what YOUR pain points are, can a solution that works for your situation be crafted. Only then will the plan come together and the solution start to sing.

This is the beginning of a software development project. Don’t focus on what tool or technology you think you need first. This is custom automation outside of any framework whose sole job is to get some “stuff” done. You should not feel that the solution needs to be wedded to a particular tool at this point. Let the exploration of the problem lead to the tool or tools to be used.

First, think about the problem space. What are some activities that are done over and over again? Which of those activities take little time to do, but are error prone? Those are your first candidates for automation! The smaller the perceived scope of the task, the higher the likelihood of success for the automation project. Examples of small automation tasks like this would be

- Copy files from one location to another
- Backing up log files and clearing out the old ones.
- Modifying a configuration file
- Stopping and restarting services
- Modifying registry settings.

While these activities in themselves are not a test automation project per se, they indeed are steps that are needed to get a testing environment set-up. Since they need to be done anyways, they make GREAT candidates for automation, and will help you learn how to automate in the long term.

Once you identify an activity for automation, you need to define a clear goal. Defining a goal is a very important step to any software development project. This is the purpose of choosing a small activity to start off with. The activity should be of such a small scope that by identifying what the task is, the goal is perfectly described by the definition of the task. The examples listed above should give you an idea of the size of the problem to begin automation with. Once the goal has been defined, keep sight of this goal. Don’t be distracted by shiny objects that might take you off course. By focusing at the task at hand, you will have a better chance of completing what you had originally started off doing.

Now you need to work on the design. Every software project should have some design to it. Your projects will as well. The purpose of choosing an activity that you have done over and over again is that you will know exactly the steps that are needed. If you know exactly which steps are needed to finish the task, it will immediately help with the design of the solution. If nothing else, the design of the automation could be to implement each step that you would have to do manually. Believe it or not, that is an acceptable design for automation projects! Simpler really is better when you are first starting out. Don’t make things more complicated than they need to be.

When you are successful with very atomic tasks, the next step would be to string a few of the tasks together to form something larger. A great example of this is pushing out a build to a web server. A batch script can be created to do the following (familiar) things:

- Stop Internet Information Services
- Clean out directories
- Push out the new files to the proper location
- Reregister the libraries used by your product
- Copy the correct configuration files to the correct directories
- Start Internet Information Service (IIS) on your servers.

The net effect of putting these tasks together in the right order is that you have now automated a one-step deploy of code to a web server under test. This is automation at work!

4. "Get Back"

With larger automation projects, simply identifying the steps that are taken will not translate directly into automation design. In these cases, a little more thought will be needed. One common method to coming up with a workable design is to breakdown the problem into a set of smaller problems that are a bit easier to solve - the divide and conquer approach. If it turns out that you are having trouble using this approach to tackle the candidate for automation, then it’s a red flag that maybe you need to start with a different project first. If you find these red flags popping up, then maybe you need to take a step back and get back on track.

Red flags raised during the goal definition and design process are valuable pieces of information. They might identify that the project you have chosen is not quite the right one to try to implement at this moment in time. Knowing when you might be over your head is a valuable skill to have. It will save both you and your manger time and money. Remember when some managers thought that all automation projects were doomed to failure? Not knowing when the staff was in over their heads could have been a reason for that failure. This is something to avoid. A good technique to address this issue is the time box. The time box should force you to think creatively and enlist the use of your peers to come up with a solution and also prevent you from spending days or weeks on any one particular issue.

Another thing to avoid, are automation activities that are just hard to automate. An example of something that is difficult to automate would be an operating system install on a bare metal machine. Another example would be an activity that is done infrequently enough that running the steps by hand would take much less time than creating automation for it. Some things are not meant to be automated. The threshold for this discussion would really depend on the situation it comes up in. It is a formula that takes into account the people involved, the business value, and the perceived risk among many other facets. There is no *magic* formula that works in all situations, but there is certainly guidelines to be followed by your company.

What do **YOU** want to try?

5. "Help!"

Everyone gets stuck somewhere along the way when creating software. The best thing that you can do is get unstuck quickly. How do you do this? Ask for help from people you know. Asking for help is not a bad thing. Believe it or not, your peers are more than willing to help you learn if you ask for their help.

Don’t forget to let Google be your friend. Search on keywords to the problem that you are stumped on. It should lead you to message boards and blog posts that open up more doors to exploration. Maybe it will lead to alternative solutions that you did not think about. Also, don’t expect that all of things that you try will work. As with most software activities, there is usually more than one solution that could be used. Don’t be afraid to try something new. Sometimes, investigating a different solution will be the beginning of the solution that you implement.

As mentioned in a previous section, don't try to do anything fancy. Simpler is better in most cases. Also, don't try to solve all the world's problems in one swoop. Divide and conquer! Even if the solution cannot be totally automated, maybe you can get most of it automated and have some manual steps to finish the process off. That could be thought of as a form of success. At least *some* of the work is now automated. Now you'll gain more time to work on other tasks. Build on these small victories and celebrate them! By building on these victories, you will soon have a portfolio of success that you can build on.

You will start to recognize patterns, and reuse some of the ideas from your past in new projects in the future. Leverage those past successes into future solutions. You should find yourself using some code snippets over and over again. This is a good thing! Recognize this pattern early so that you can be secure that the solutions to some tasks already live in your toolbox, ready to be deployed at will! Also by showing success in your endeavors, you gain more trust from your peers and management.

Learning is a process where you see what works and what doesn't. Each idea that you try is a part of this process. Remember both the things that went well and didn't go so well. Creative brainstorming, problem-solving skills, and experimentation can all help achieve the wanted outcome. Sometimes reframing the problem statement can help you get to the final destination.

Let's revisit the difficult automation problem of installing an operating system onto a bare metal machine. What if we reframed the question? One could use virtual machines as a part of the solution. You still need to manually setup a machine once per OS/software install combination to create the base virtual image, but now you could clone them to make an army of machines! This leads to saved effort down the road.

6. "Old Brown Shoe"

Until now, this paper has shied away from touting any language over another for the novice automator. There are lots of code examples out on the Internet that people can peruse to examine how to connect to SQL servers, drive a browser, or interact with a CSV file. Feel free to use any language or tools that will help you move closer to your end game. This section will showcase how to do these things with Ruby (Ruby – A Programmer's Best Friend website). Ruby is a comfortable old brown shoe that can be used when one needs to get things done. There are other options such as Python (Python Programming Language website) and powershell that could be used as well.

In the opinion of this author, one of the strengths of using Ruby is in that it is platform agnostic. It is available for free on just about any platform. No license needed! It is an interpreted language which usually means its fast to edit and run programs. You can use the Interactive Ruby Shell (irb) and start investigating the language and its libraries (called gems in Ruby-ese). It is also open source with a strong community of contributors.

Another great "feature" with Ruby is that most of the gems will come with its set of unittests that are run to ensure that the libraries are working properly. In other words, the gems come with a set of code examples that you can run! You can use those tests to see the code in action and try to use it to suit your own needs. Be sure not to blindly cut and paste code from these tests (or any other examples) without understanding what that code is doing. If for some reason, there is a bug in the code that you pasted, you better know how to fix the problem!

The following Ruby examples can be cut and pasted into irb and used as the basis of your experimentation. You will need to install the following gems

- Watir-Webdriver (WW) (Watir-Webdriver website)
- Database Independent Interface (dbi) (Tutorialspoint – simple easy learning website)
- Database Driver – Open Database Connectivity (dbd-odbc) (ODBC bindings for Ruby website)

Pointers to more information for the above gems are included in the references. There is also a link to a WW tutorial (Ruby – A Programmer’s Best Friend website) contained within the references. The WW example page referenced in the below code is a good place to start to familiarize yourself with some of the things that WW can do, and because WW is built upon Ruby, you have the entire Ruby interface available to use. The possibilities are endless. Each possibility should share something in common – judicious use of comments. It is the comments in the code that will make it easier for someone (perhaps you!) maintain the code in the future. Without comments, your code will be much more difficult to manipulate in the future.

To use the SQL example, you will need to use a custom connection string to match your database and login credentials. You may want to modify the actual SQL query to something that would return results from your database as well.

To use the CSV example, you should have it point to a CSV file that contains a header row with at least two columns – ‘TestName’ and ‘URL’. Or feel free to modify the code to use the headers in your own CSV file.

You may wish to enter commands into irb one line at a time. This way you can experiment with syntax and see the results of your handiwork right after you hit the <ENTER> key!

```
# The hash mark at the beginning of a line marks the line as a comment.
# This script uses the following libraries - called gems in Ruby-ese

# For the web example, keep these prerequisites in mind
# Be sure to have Firefox installed.
# For chrome support install chromedriver as well.
# For IE support install IEDriver as well.
require 'Watir-Webdriver'

# For the database examples, keep up to date with the Ruby ODBC
# documentation for any prerequisites that are needed.
require 'dbi'
require ‘odbc'

#  The CSV gem is part of the core package.  There is no need to install this
as a separate gem.
require 'CSV'

#---- Use WW to fill in a web form
# Using a browser developer tool to investigate your web control is very
useful here
# The web page pointed to here is actually part of the Watir tutorial.
#   It is the watir example page to experiment with different input controls.
# Go find a text field by its name, and set it’s value
browser = Watir::Browser.new
browser.goto("http://bit.ly/watir-example")
browser.text_field(:name => "entry.0.single").set "Watir"

#---- Use Ruby to interact with SQL Databases
# Create the connection string
# Connect to the database and execute the SQL statement
# Iterate over each result to print out the first column - which should be
the company name in this case
# Close out the SQL connection
# NOTE:  The actual SQL to be used is dependent on your schema.
```

```
sConnect = 'DBI:ODBC:Driver={SQL
Server};Server=MyServer;Database=MyDB;Trusted_Connection=yes;'
hDBI = DBI.connect(sConnect)
sSQL=hDBI.prepare("select companyname from company")
sSQL.execute()
sSQL.fetch do |row|
      # Change this statement based on your schema.
      puts 'CompanyName  --  '+ row[0]
end
sSQL.finish()

#---- Use CSV to interact with CSV files
# Use a CSV that has a header row
aList=CSV.read('myFile.csv',:headers=>true)

# Iterate thru each line in the file and print out the test case name and
target URL
# Assign each instance to a temporary variable
# For each instance, print out two columns - the name of the test and the URL
that it should run against.
aList.each {|aPage|
      # NOTE: The following statement may change based on your schema.
      sName=aPage['TestName']
      sURL=aPage['URL']
      puts sName + " - " + sURL
}
```

Pointers to the documentation for each of these libraries are contained in the References section. Other great references are the unittests that come bundled with each gem. The directory where your gem is installed should contain a directory named unittests, spec, or test. With a typical install, one may find the Watir-Webdriver unittests at C:\Ruby\lib\ruby\gems\[*ruby version*]\gems\watir-webdriver-[*gem version*]\spec, where *ruby version* and *gem version* are the respective build versions of Ruby itself, and the version of the Watir-Webdriver gem being used by the current install. The unittests are all written in Ruby. Coupling the unittests with irb should give one better insight into how to better use the gems that you are interested in.

7. “Long and Winding Road”

This is not a talk about how to program, but recognize that to gain the most from automation, you really will need to understand basic programming concepts. This is beyond the scope of this talk, but here are some concepts that you should become more familiar with when you start to look at different programing languages. Remember that learning how to program is a long and winding road. Topics for further research include:

- Data Structures
- Flow Control
- Looping constructs
- Function/Method calls
- Handing of Input, Output, and Errors.

A good resource is a cookbook. One of the best was the ‘Perl Cookbook’ (Perl Cookbook Wikipedia entry). Since its original publication, cookbooks for other languages have been published using the same recipes. Cookbooks for a multitude of languages such as ruby, Java, and C# can be easily found. They provide recipes that you can easily incorporate into your project. But remember you should understand what the code is doing before using it.

There are also more rigorous free online courses available thru the OpenCourseWare group (OpenCourseWare Consortium website). These are free classes offered by some of the top schools around the world. They cover a range of topics including non-technical categories such as history, writing, and many other subjects. These courses appear to be a promising source for additional learning.

8. Conclusion

This paper has presented some ideas to keep in mind while you are exploring and learning more about how automation can help you.

Remember that automation is more than just test automation. It is anything that can be created to make your job easier. Tasks that are done repeatedly are great candidates for being automated. The likelihood of a successful automation project is increased by recognizing tasks that are done often and are error-prone when done manually.

Have a clearly defined goal for each of your automation projects. Keen focus on the project goal will help keep the project on track. Being able to complete automation projects within a time budget will increase the likelihood that you will have the opportunity to work on other automation projects.

Break the project into bite size pieces if needed. These smaller tasks should be well bounded and have a better chance of success. Build on these successes as you build your skills. Use these opportunities for success to show that automation can work and work well for you and your company.

Use all of the on-line and off-line resources that are available to you. Using a search engine should be the first resource that is consulted, but it should not be the only one that you rely on. Your peers are a great point of contact that should be leveraged as well. They can lead to other paths of research for you to examine. They can also provide a different perspective on the problems that you are trying to solve.

Congratulations on taking the first step to learning about automation - researching it! Take the opportunity to examine a few different languages to see what works for you. Feel free to experiment with the different libraries that are available for you. Never walk away from an opportunity to learn and you will do well with automation. Hopefully these ideas will serve you well in your quest to learn how to automate stuff.

References

Places to start

Ruby community. "Ruby Programming Language." Ruby – A Programmer's Best Friend, http://www.ruby-lang.org/en/ (accessed August 5, 2013).

Python Software Foundation. "Python Programming Language." Python Programming Language, http://www.python.org/ (accessed August 5, 2013).

Ruby community. "Ruby in Twenty Minutes." Ruby – A Programmer's Best Friend, http://www.ruby-lang.org/en/documentation/quickstart/ (accessed August 5, 2013).

Wikipedia community. "Perl Cookbook." Wikipedia, http://en.wikipedia.org/wiki/Perl_Cookbook (accessed August 5, 2013).

Microsoft. “Scripting with Windows PowerShell.” Script Center, http://technet.microsoft.com/en-us/scriptcenter/dd742419 (accessed August 5, 2013).

Microsoft. “Command-line reference A-Z.” Microsoft TechNet, http://technet.microsoft.com/en-us/library/bb490890.aspx (accessed August 5, 2013).

OpenCourseWare Consortium. “The OpenCourseWare Consortium.” OpenCourseWare Consortium, http://www.ocwconsortium.org/ (accessed August 5, 2013).

tutorialspoint. “Ruby/DBI tutorial.” Tutorialspoint – simple easy learning, http://www.tutorialspoint.com/ruby/ruby_database_access.htm (accessed August 5, 2013).

Libraries referenced

Selenium community. “Selenium – Web Browser Automation.” SeleniumHQ, http://seleniumhq.org/ (accessed August 5, 2013).

Scott, Alister. “Watir WebDriver | the most elegant way to use webdriver with ruby.“ Watir-Webdriver, http://watirwebdriver.com/ (accessed August 5, 2013).

Werner, Christian. “ODBC bindings for Ruby.” ODBC bindings for Ruby, http://www.ch-werner.de/rubyodbc/ (accessed August 5, 2013).

Considerations Before Starting Test Automation

Steven Vore

Steven.Vore@Telerik.com

Abstract

Software is complex, and complex software doubly so. Our testing teams, comprised of fallible humans, are challenged with finding the tiniest of errors in a seemingly infinite domain. Once we're satisfied with a version we start all over again. It's no wonder that teams are looking to automation in hopes of easing their workload.

Company after company, testing team after testing team is asking, "How do we get started with test automation?" The wrong answer for our testing teams is to buy a product or download a framework and start coding against it. This paper and presentation focus on key questions to ask and discussions to have before pulling out the checkbook, keyboard, or mouse.

We will begin with assessing team skills; determining the level of technical skills and interest amongst the testers and developers. I will discuss the need for finding allies and for getting management buy-in before starting – or forgiveness after getting underway. You'll learn methods for identifying the areas of your project that would be good starting points for automation, and the importance of determining a course of action.

Wise testing team managers should remember to "look before you leap" into automation. Just as a development team would be mistaken in choosing a programming language or database tool before understanding a new project, we should be asking the important up-front questions before diving in.

Biography

Steven Vore is a Test Studio Evangelist at Telerik, currently working from his home office in Atlanta, Georgia. He has worked in software support and testing for the better part of two decades, and enjoys exploring ways to make software easier to use. Steven has worked on projects in defense contracting, networking software, telephony and medical information.

Steven has a degree in Information Systems Management from the University of Maryland and has earned several technical and project management certifications.

1. Introduction

The world of software development is a complex one, often encompassing multiple frameworks and technologies, using various patterns employed by architects and programmers of varying skill. Our testing teams, also comprised of humans with multiple backgrounds and skills, are challenged with finding the tiniest of errors, and once found and fixed we're expected to ensure that errors don't creep back into the product. It's no wonder that testing teams are looking to automation in hopes of easing their workload.

At conferences, in online forums, and over coffee tables, managers around the world are asking themselves and each other "how do we get started with test automation?" As with any question in a technical realm, it is easily answered with a pointer to a blog post to read, product to purchase or framework to download – but that is the wrong answer. Wrong answers are easy to get when the wrong question is being asked.

Tools and frameworks do not automate testing, people do. Before jumping to a technical solution, testing managers need to evaluate their organization and team members, asking themselves instead "are we ready to automate?" Some important questions to consider, the answers to which will help make this determination include:

- What are the norms and expectations in our organization?
- What level of skill and interest do the testers have
- What sort of acceptance and assistance can we expect from other teams?
- What level of support will our management provide?

Only after these have been discussed can a testing team realistically begin to identify the areas of their project that would be good starting points for automation, and to choose the tools to do so.

2. Determining your goals

There are many reasons for test automation; the most often-stated business goals include reducing errors and saving time. Traditionally testing has been a manual and often scripted, repetitive process, and humans are easily bored and make mistakes. Manual testing is also slow, and the time required grows as an application is built. Subsequent modifications and versions require re-testing, which increases both the time required and opportunities for errors overlooked due to tedium.

An underlying goal for the testers themselves, which should be made more explicit, is that of having an opportunity to do their best work. Recent studies have shown that there are two internal desires that drive knowledge workers – challenge & mastery and making a contribution. Testers, developers, and others are motivated and do their best work when they are given the freedom to learn and master new skills and then use those skills in a meaningful way. (Pink 2011) Automating the repetitive and boring tasks can result in organizations that are more effective and, in the long run, more profitable.

It is important that not only the QA manager or test team lead make sure that all members of the team are in agreement on their goals, and that the rest of the organization – "upper management", developers, and others – understand and are on board as well.

3. Assessing the organization

3.1. Organizational Support

Software testers do not function in a vacuum. We interpret requirements from business analysts and customers. We seek information from and provide feedback to developers. We report our findings to product owners. Each of these interactions comes with some level of expectation and support. It is important to understand how others in the organization view the testing team, and how changes to our testing methods will be supported (or not).

Automation, while streamlining our tasks and saving time in the long run, will require some amount of additional time in its startup phase. If the testing team is valued as part of the development process, this time will be seen as an investment.

For many organizations, however, the "QA" or testing team is considered to be a necessary evil and an expense. This can be seen by a lack of budget for salaries, tools, and training. Other, more subtle, indicators come in the form of attitude: development teams who do not consider testability to be a desirable feature of their code, managers who allow scope creep to squeeze testing into a last-minute rush of late nights and weekends. "When an organization lacks an overall quality philosophy and pressures teams to get the product out without regard to quality, testers feel the pinch." (Crispin and Gregory 2009)

In this sort of environment, it is difficult to justify the investment required to properly automate testing. Test managers should begin by first increasing awareness of the value testing brings to the organization. Defect counts and time required for bug fixes before product ship are perhaps the easiest metrics to gather but are easily gamed and are not seen as terribly useful. Overall product quality, though much more difficult to measure, should be the goal. Product quality – or the lack thereof – can be seen by product owners, customer support and client service groups, and by looking at the number and severity of customer-reported issues. These people and metrics can have considerable influence in raising the testing team's perceived value.

Managers of testing teams should look to insert themselves and their team members into the product planning, design, and architecture process as early as possible. Working directly with product owners will afford an opportunity to point out adjustments to features which would make them more testable and often more helpful by end users as well. The testing team will also become more familiar with features' intended purpose, which helps design tests that do more than verify that the product matches the specifications – they can aid in insuring that the specifications are describing a product that will meet the users' needs.

3.2. Own the data

In order to automate any testing – UI or not – the tests must be repeatable without human intervention. This means that the results need to be predictable, and for this the underlying data must be in a known state. You may or may not think of your system as having data, and it may or may not use some sort of database back-end; for all but the simplest applications there is some sort of data on which you need to be able to rely. Your system likely also has user accounts (usernames, passwords, permissions and other settings). Think of any element of information, which, if changed, would cause a tester to diverge from a specific character-by-character script. For automation to be viable, the testing team must be able to control all such data in the testing system.

Does the development team use the testing system's database for experimentation and exploration? Does the product owner or sales team use the test system for demonstrations and training? Does the production support or operations team use the test system for their own "playground," verifying data-manipulation processes prior to implementing them on a production system? If any of these are normal in your organization – especially without the testing team's knowledge or authorization – the data may not remain in a stable.

Evidence of this is easy to find: simply examine the language used when describing testing tasks. If testers are required to look up values or independently calculate results to verify tests' success or failure upon each execution of the test, it is unlikely that the data is stable enough for the tests to be automated. The lookups or calculations themselves could be codified into the automation, but this would then become code which itself would be subject to inspection and testing.

In an optimal world, the testing team would have its own environment. The team would control software builds, configuration, and data upon which tests would be performed. There are many reasons why this is often not possible; the most-often cited are financial and manpower. Organizations are unable to purchase or lease additional servers and licenses, and developers, testers, and system administrators are not given sufficient time to setup and manage another set of systems.

Fortunately, most applications can be sufficiently managed so that this is not an insurmountable problem. As is often the case, the solution comes down to conversation, agreements and cooperation rather than technology. Each team can, for example, agree to use a their own set of user accounts. As long as changes in the data used by the various accounts does not impact other user accounts' data, each team can function independently. These agreements can be – in fact need to be – seen as beneficial to all. Not only will a stable data set give the testing team an opportunity to streamline, and perhaps automate, their testing; training and demonstrations can become more stable.

Other teams may be reticent to agree to such a split; often this is simply due to the time required to setup new data or a lack of knowledge of how to do so. In cases such as this, the testing team can offer assistance with creating data setup and cleanup scripts. This can ease the agreement, making this an even more palatable to other teams. It can also give you time and opportunity to document the data model and setup process, if that has not yet been done.

As part of any discussion, all parties will need to come to an understanding and agreement on management of application-wide configuration as well as scheduling of deployments.

4. Assessing team skills

4.1. Technical skills amongst the testers and developers

Do any of the testers have a background or training in programming, scripting or automation? Are they comfortable writing scripts for your system (Unix/Linux/OS X shell scripts, Windows PowerShell scripts, SQL query scripts), or will they need to be trained from scratch?

Let your testers' skill set determine what portions of the application's testing to automate first. For example, testers with strong SQL skills may start with writing queries, scripts and stored procedures to create or setup data, and then to test backend processes and application-level stored procedures. Testers with knowledge of HTML could begin with a record-and-playback tool, preferably one that will allow them grow into custom-coded steps as they progress. Team members who have knowledge of programming could work with the developers to enhance unit tests, or explore one of the frameworks available for User Interface testing.

Start with some wins, providing positive feedback to the team and showing positive movement for managers. This will ease the transition into training, learning, and experimenting with less-familiar territory.

4.2. Interest amongst the testers and developers

Not all testers are interested in making the move to automation. As with any profession, unless your hiring and training practices specifically target individuals with scripting and automation skills you will have a range of interest and motivation among your team members.

Managers should being by asking themselves “do I have any team members who have already shown an interest in automation?” Look for testers who have spent some of their spare time looking into what can be done to streamline their tasks. When putting data together, for example, you may have testers who have created spreadsheets and batch files rather than typing fresh data or copying from documents. These individuals can also often be found talking with developers, looking to understand how the product is being developed. Given encouragement and a little freedom, testers such as this can grow into the stars they want to be.

Developers: how testable have they made the application thus far; are they using Test Driven Development methodology or unit tests at all? Testers will often need support when coding automated tests. If testing, in your organization, is considered to be a menial task to be done by “the QA team,” what amount of support or assistance will the developers be willing or able to provide? Many of us have worked with developers who eschew any form of testing, choosing to “throw code over the wall” to testers without any earned confidence in its quality.

A former co-worker of mine often claimed that his “unit tests passed” – though he had coded no tests at all; his claim was based entirely on the fact that the application opened in a browser. Accompanying this was a refusal to add IDs to HTML elements. His attitude toward testing made his relationship with testers uncomfortable and their attempts at automation extremely difficult if not downright impossible.

5. Identifying allies

5.1. Why

Although most tool vendors wouldn’t consider it a popular statement, automated testing will always require some level of coding for several reasons.

Code-generation and Record & Playback tools have improved over the years. However, so have the applications they’re being used to test. Applications grow not only in technical complexity but also in user interface creativity. The likelihood of tool creators to be able to foresee every desired set of actions/verifications is dubious at best.

Most testing scenarios include a need for setup before executing or cleanup afterwards. For example, a set of tests may be required to ensure that new user accounts are created with the proper set of attributes based on different authority or roles of the creating user. As discussed under ‘own the data,’ these scenarios are only repeatable if the underlying data is in a known state. A well-written test scenario – for humans or automation – should include steps to ensure that the test does not fail merely due to duplicate data, and that data created during the test is removed in a proper fashion afterwards. These are not “test steps” in the classic sense, they are not steps upon which failure or success should be reported, but they are required nonetheless. Such setup and cleanup steps typically involve more direct access to the system than the test steps; they may call a stored procedure created expressly for, if not by, the testing department. Setup steps should be automated wherever possible; not only as part of test automation but also to assist manual testing.

Testers often produce code for tasks which simply cannot be done by Record & Playback tools. Samples of this include data creation, manipulation, and cleanup; defect tracking and consolidation; task and product documentation and reporting. This type of work supports the testing effort, and is “code that helps the product get out the door sooner.” (Page 2013)

The idea that testers “might code in different ways, for different reasons, but they can be coders too” (Christie 2010) is not new. Some test team members are comfortable writing their own code. Their coding skills are often not formally recognized, and in many organizations training in programming techniques and practices is not provided. The result is that testers rely heavily on their programming-centric peers for assistance. In some cases assistance is sufficient, lending suggestions and support as a tester writes their own code; in other cases there is a complete dependence, with little or no programming actually done by the testing department.

Perhaps of most importance is automating tasks that directly support not only testers but developers as well. This list includes automated build and continuous integration systems as well as unit tests. These are typically built and maintained by development groups, but if they are not in place the testing team may need to take the lead in making that happen.

5.2. where to find them

Given that testers need assistance with coding and related tasks, from where does this support come? The obvious answer is “the developers,” the organization’s formally-hired and -trained coders. Testers should be cultivating good professional ties with the developers even if this were not the case, and this symbiotic relationship benefits the organization as a whole as it fosters quicker, higher-quality product delivery. Product development and testing should be seen as a team effort, with developers and testers both bringing their skills to bear.

There are likely others in the company who have development expertise, and to whom testers should extend a hand. There is often quite a bit of coding done in the production support or “DevOps” team, for example, by the individuals who are tasked with keeping systems running day-in and day-out. These teams quickly learn, usually through hard-earned experience, the internal workings of the systems. Database and scripting tasks are their specialty, as are the connections required to deploy and manage the application. This is invaluable information for a tester.

Additional support – technical and morale – can be found outside of the company as well. In many cities, user groups can be found using online tools (meetup.com, lanyard.com, etc). These groups are valuable resources for more than QA/testing topics; groups exist for discussing and learning processes and methodologies, such as Scrum and Acceptance-Test Driven Development, as well as technology-specific topics including programming languages and frameworks. Organizers and attendees welcome – and rely on – guests and new members who are eager to learn. In-person provide an opportunity not only for learning, but also for general discussions during which ideas can be vetted and to see how our practices compare with those of our peers. Managers of test organizations should not only attend external meetings but also encourage their team members to do so as well.

6. Discussions with Management

Often, when the topic of test automation is discussed with IT Directors or other “upper management,” the initial thoughts are of UI Test Automation and questions about cost. Testing team managers should be prepared to address these topics. Approaching “upper management” unprepared can lead to unanswerable questions and projects being shot down before they even get off the ground.

As discussed above, automation within the testing realm consists of a variety of coding activities, of which UI Test Automation is a visible part. I suggest that the team discuss their end-to-end environment with this in mind, determining a list of all the possible points of automation, which have the potential for best long-term time savings, and suggestions for methods of attack. Each area of potential automation should be examined with regard to tools available; in-house, open-source, and commercial.

Where interest is shown, and small slices of time can be made available, small “skunkworks” projects can be valuable. These have a short timeframe and specific goal in mind – to determine the viability of a particular tool or technology for use in automating a specific slice of the testers’ duties. The idea is to give the team room to grow and prove or reject a concept without interrupting the existing workload. After one or more of these have been completed, a testing team can make an informed presentation.

Test managers should take care with regard to presentation of in-house or open-source tools and frameworks as “free.” There is a tendency to see cost of software separately from the cost of the time required to use it properly and effectively. Discussions should always include this investment in time as well as any monetary requirements, and a reminder that results may not be seen immediately.

7. Where to begin

Testers, being technical people, have a tendency to pick a tool or technology that appeals or is popular and dive right in. It is important for the team to step back and ask some questions first – “where will automation be the most beneficial,” and “what’s the most painful or boring part of our jobs today?” Taking the answers these questions in conjunction with tools such as the Test Automation Pyramid (Cohn 2009) will lead to good starting points. To many, “automated testing” means using tools or code to drive an application via its user interface, but this should constitute the smallest effort as it provides the least ROI. These often test then entire application rather than individual components, and are therefore relatively slow to execute. Tests at lower levels of the pyramid are more less fragile, longer-lasting, and more effective.

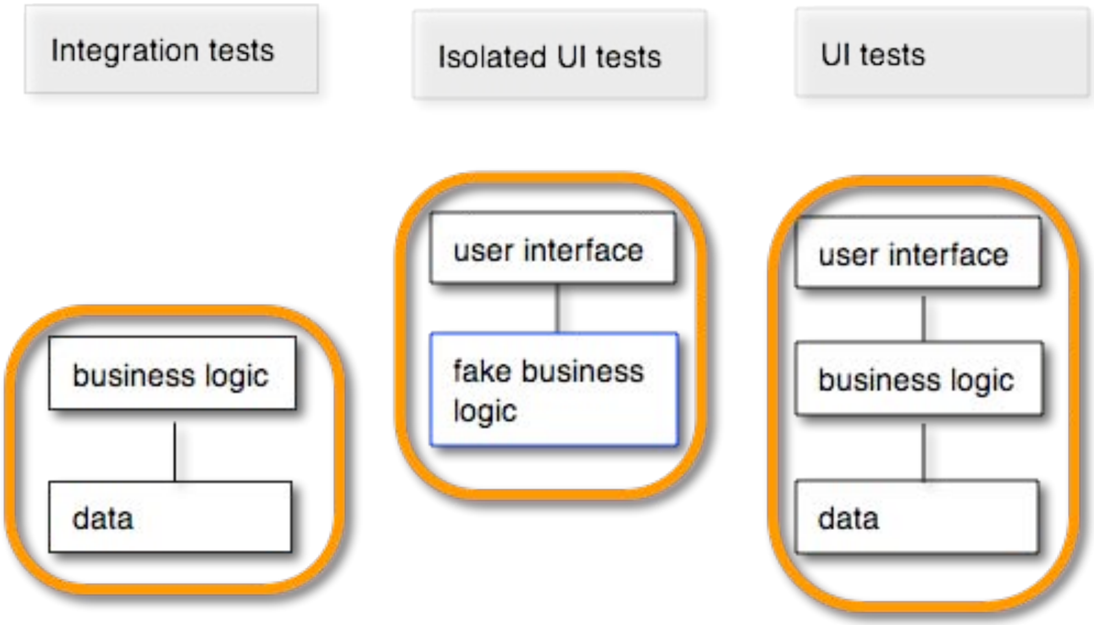
The lowest level possible is the base on which the pyramid sits, the foundation of good current development practices. Without a stable, automated and quick build process, both developers and testers suffer; this should be the first thing on any team’s automation list. This will need to be a joint effort between developers and testers, perhaps with assistance from system integration or production support teams in the case of new environments.

Unit tests, in theory at least, would be the province of the development team rather than the testing team. Unit Testing is sufficiently important, and testers should be aware of the developers’ use of unit tests, encouraging and assisting wherever possible.

The bulk of testers’ time, with regard to automated testing, should be spent working – alone or with developers’ assistance – in integration or “under the UI” testing. Integration tests exercize business logic, often combined with database or other back-end access.

User Interface tests where possible, should use an interface which calls “fake” logic. This will allow the UI iteself to be tested quickly and independently. This is becoming increasingly important with client-side validation and actions become popular.

Finally, a minimal number of traditional end-to-end UI tests will be created.



8. Summary

As we have seen, Test Automation can cover a wide variety of activities, from unit tests and continuous integration to driving the actual user interface. Testing and development managers and team members have several topics to discuss while considering automation; as with any activity, preparation is key. Wise testing team managers will remember to "look before you leap" into automation, leading their team through the important – and often non-technical – discussions that can lead to success.

9. References

Christie, James. *Testers and coders are both developers.* October 12, 2010. http://clarotesting.wordpress.com/2010/10/12/testers-and-coders-are-both-developers/ (accessed June 2013).

Cohn, Mike. *Succeeding with Agile: Software Development Using Scrum.* Boston, MA: Addison-Wesley Professional, 2009.

Crispin, Lisa, and Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams.* Boston, MA: Addison-Wesley, 2009.

Page, Alan. *Coding, Testing, and the “A” Word.* June 5, 2013. http://angryweasel.com/blog/?p=649 (accessed June 2012).

Pink, Daniel. *Drive: The Surprising Truth About What Motivates Us.* New York, NY: Riverhead Books, 2011.

Agile Testing at Scale

Mark Fink

Author contact: mark@mark-fink.de

Abstract

"Agile Testing at Scale" is an important topic in Agile software development of enterprise business applications. More and more companies adopt Agile development methodologies which introduce much shorter release cycles. Frequent releases push requirements regarding continuous integration and test automation.

This talk introduces methods on how large automated test suites are structured and maintained efficiently.

At scale, test automation will eat up a significant part of your development teams project budget and if it is not running effectively you will not even gain the desired benefits. Consequently you need to set goals and metrics so you can easily keep your quality initiative on track and / or being able to make adjustments to the life cycle of your project at any point.

For software development projects which have not yet introduced it the continuous integration tool chain really opens up the next level of software quality. Continuous integration ensures that the application can be build and integrated at any given point of time. Feedback to developers via email regarding integration problems will be given immediately after a code change is applied. So by introducing continuous integration big massive integration problems are reduced to a few tiny ones.

Successful test automation does not only depend on processes, tools, and infrastructure. It is also necessary to enable the person who writes the testcases in the first place. I will introduce the ACC methodology on how to approach writing tests. I will also discuss a styleguide for writing new tests.

I will round up the paper with internet resources that you can access to download a continuous integration server and configuration that demonstrates all relevant test automation aspects on a working Javascript application.

Biography

Mark Fink runs an independent consultancy providing software testing services. In 2013 Mark released his book "The Hitchhiker's Guide to Test Automation". The book summarizes his practical experience with software testing methodologies and tools in the context of agile web application development. His combination of academic qualifications with 20 years' practical experience have made him a popular speaker at international conferences on software test automation and performance testing.

1 Introduction

Many aspects need to be considered for an agile software development project to be successful in test automation. Continuous integration is a precondition and needs much consideration in addition to installing another tool. Another example is the integration of the test-related tasks into the agile software development process. Of course all these important topics can not be covered within a single conference paper. They could fill books. In fact I wrote one myself: “The Hitchhiker’s Guide to Test Automation”. So in this paper I will focus on the aspects that I think need the most attention when it comes to “Agile Testing at Scale”.

With scale I mean:

- the scope of the project is in the area of 100 man-years or more
- the team consists of 5 developers or more
- high velocity - new features and changes are added to the code base at high pace (the above mentioned developers are frequently adding changes to the code base)

In a situation like this it is crucial for your test initiative to have methods in place and tool support for writing and maintaining tests. This topic is covered in section 2 “Writing Tests”. Section 3 “Evolving Automated Testing” covers an approach on how you structure your tests in a way so that they stay efficient and effective. I give examples where ever possible. Section 4 “Installing the Continuous Integration Server and Demos” gets you started with the continuous integration server and a sample application.

2 Writing Tests

This section is aimed at writing automated tests, tooling and styleguides.

Successful test automation not only depends on processes, tools, and infrastructure. It is also necessary to enable the PEOPLE who write test cases in the first place. Consequently this chapter introduces the activity-testing pattern for approaching test-writing and elaborates on the environments necessary for writing. We also discuss a styleguide for writing and / or maintaining tests.

2.1 Activity-Testing Pattern

When implementing GUI regression tests, it is a good idea to start implementing them in a top-down fashion all the way from the intention of the test to the technical activities [ADZ2010].

The activity-testing pattern recommends describing the test and the automation at the three levels depicted in Figure 1 (an example is provided for each level).

Table 1 provides definitions and examples for each of the three abstraction levels of GUI test automation.

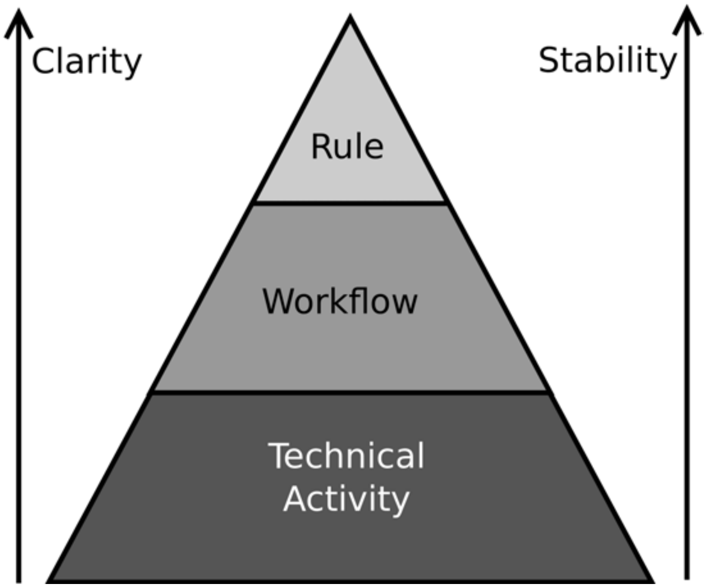


Figure 1: Three levels of abstraction in GUI test automation.

Table 1: GUI test-automation abstraction levels.

Level	Description	Example
Business Rule / Functionality	What is the intention of the test?; what is it demonstrating or exercising? A good start when implementing tests is to use examples gathered during specification of the story regarding the feature in question	Free delivery is offered to customers who order two or more books
GUI Workflow	What does a user have to do to exercise the functionality through the GUI on a higher activity level?	Put two books in a shopping cart, enter address details, verify that delivery options include free delivery
Technical Activity	What are the technical steps required to exercise the functionality? On this level the API of the automation tool (in this case Selenium) is used. There is no need to wrap the features of the automation tool!	Open the shop homepage, log in with 'testuser' and 'password', go to the '/book' page, click on the first image with the 'book' ID, wait for page to load, click on the 'Buy-now' link and so on

FitNesse's main purpose is to provide a means to build the abstraction levels described above. Robert C. Martin has compiled an excellent series of video tutorials on how to achieve this using FitNesse [MAR2008]. The video *Writing Maintainable Automated Acceptance Tests* is particularly relevant in this context.

2.2 Page Objects

Page objects are a layer that guards you against changes in application frontend technology. This can be very useful if you need it but it comes at a cost, too. Using page objects it is also possible to fence yourself against the technology used for browser automation. But the same mechanics apply here. Unless you need the flexibility to exchange the automation technology, I advise against taking on the cost. I observe a trend toward maximising the use of pattern and abstraction layers. These bring additional complexity and costs and you should only consider using them if you foresee significant benefits in the near future.

The page-objects pattern has the disadvantage of structuring the writing of test cases in a bottom-up way. This leads to an overly technical focus instead of a focus on the test intention and workflow steps relevant to a user story.

I strongly recommend using the activity-testing pattern. This is a way superior concept, structuring testing activities in a top-down fashion. The activity testing pattern works well with the S/M/L scopes (see *Evolving Automated Testing*) that provide a model for structuring your test automation. If you have doubts, I recommend reading Eric Evans' book [EVA2004].

2.3 Standardised Working Environment

To lower the entry barrier on test automation for your colleagues it is very important that you provide a standardised environment for running tests, analysing test results, improving tests or writing new test cases. Your environment might consist of test tools, IDEs, source code and test code repositories and the like. You have to make sure that installing and using this environment is as easy as possible.

I found it good practice to hold everything related to the standardised working environment in a repository. Everybody in the team (including analysts) knows how to work with a repository, so upgrading the test environment is only a few clicks away. In one project, the team developed multiple applications in parallel. I have split the working environment from the test suites, so one can work with multiple test suites within the same working environment.

Of course some people have different requirements for their working environment, just as developers have different IDEs or different clients for the source repository. But they benefit from the low entry barrier provided by a standardised working environment. This way they have a smooth start and can integrate the test tools into their customised workflow later once they have seen how everything goes together. They just pick what they need from the standardised working environment.

By maintaining a single working environment, you can reduce need for maintenance and make sure it works smoothly. Broken test tools are not exactly what you need when building a reputation with your team.

2.4 Staging Environment

Test the test! Your automated test suite, wall display, and working environment become essential components to support your development process after a little while. Now if you change anything, such as maintaining or adding a test or refactoring some test function, you might break that development process. If the test automation reports a failure, it must be in the application or infrastructure. If the test automation reports false positives, this becomes an issue. Once you have to deal with lots of false positives, developers will start to ignore the results.

This means you have to strictly separate the test-automation runtime that supports the development process and test executions that you need to test the test. In other words, you need a dedicated environment to test the test.

Newbie Test Suite

> All new and changed tests get tagged as "Newbie". They do not run with the regular test suite. So if they fail, a tester needs to look into them. Their failure does not show on the wall display. The Newbie Suite runs multiple times a day and can be triggered manually. So within two days we have more than ten executions of a new test and we get a first indication of whether there are timing issues we must fix immediately. If after 10 or more executions a new test case turns out to be stable, we promote it to the test suite.

Staging Environment

> Once you have a test suite of significant size, you cannot just run the whole test suite on your tester's machine. Neither can you check in the refactored test code. If you have changed anything in the test infrastructure, such as upgrading to a

new Selenium version, you need to make sure everything works well before you promote the changes to your development process. This is why you should plan for a staging environment from the beginning. Without it, you cannot improve your test infrastructure without compromising the acceptance of the test automation.

When sizing the hardware for your test infrastructure, you have to plan for newbie and staging environments as well. If you share the environments with the regular test execution, you run the risk of slowing down test execution considerably, which in turn could lead to acceptance problems. Section *hardware* goes into calculating the sizing for test infrastructure.

2.5 Styleguide for Writing Tests

Styleguides are very specific to your company, team and application, so I can only provide you with a common styleguide as a starting point. Update your styleguide regularly to address issues you face in your day-to-day work.

Naming Convention

The naming convention is an important aspect of your test-automation initiative. It heavily influences the maintainability of your test suite. If you need to find test cases for regression testing or for adding verification steps for new features, it is important that you find them quickly.

I consider the following aspects to naming tests:

- Business rule
- Workflow
- Sub-Application
- Feature
- Dialog / Panel
- When naming the test case think “How would I ‘google’ for this?”
- If you have similar tests, highlight the differences
- Test intention
- A naming example for a test for the Supercars application would be “SupercarEdit”
- Do not use project or change request numbers as part of the name

If I create new test cases I usually complete the documentation section first before I define the name for a test case.

Documentation

I start each test case with a documentation section. It is difficult at first to find the right amount of documentation (the right abstraction level) necessary. Reviews by colleagues can help here.

Aspects relevant for test-case documentation are:

- Document the intention (the business rule) of the test case with a few sentences
- Do not repeat the test-case name
- Document which features or functionality you intend to test
- Document the expected results (validations)
- Do not repeat the complete set of test steps

Journal of Changes

Document recent changes to the test case. There's no need to journal that you fixed a typo, but significant changes should be recorded.

Test-Case Blocks

Often it makes sense to structure tests into multiple blocks (sections):

- If the test consists of multiple sections, then add some documentation / headlines on each section
- Make explicit which sections are preparation, which are closing / cleanup, and which is the relevant test block

Timeout

Timeouts are important to manage the execution time of your test suite. You do not want to wait forever for something that is missing.

When setting timeouts in tests, consider the following aspects:

- Use reasonable general timeouts (60 seconds)
- Use specific timeouts for steps that are known to take longer. Set back to the general timeout at the end of the step

Abort Conditions

If you are testing on test environments with real backends and databases such as (integration - or system - test environments), you will experience unexpected downtimes and failures in these systems. It is essential that you make yourself immune to these issues. If you do not prepare for these issues, your suites will run forever and you will have difficulty finding out why. Abort conditions are the right tool to manage execution time for larger test suites.

Following is some guidance on using abort conditions:

- Abort when condition is not met
- Abort if some of your test infrastructure is missing (e.g. Selenium server)
- Abort for situations such as "500 system is not available", or user login failed
- Abort when not on the right page
- Abort as soon as possible to save valuable testing time

Locating HTML Elements

There are different ways to locate HTML elements. If you do HTML element location wrong, your tests will eat up lots of resources. Use element IDs or CSS selectors whenever possible. Try to avoid using XPATH altogether.

You should agree with your developers that HTML elements need useful IDs or name attributes assigned. In my projects this is an acceptance criterion that falls into the category of testable application.

Waiting for Events

Always wait for specific events. For example "wait for element visible" is much more reliable than "wait 500".

Tagging of Test Cases

Tags are applied to individual test cases. The tags are used orthogonally to the test suite structure (see *structure-test-suite*). You can use the tree structure and tags as additional filter criteria to include or exclude specific tests.

Some applications for tagging test cases are:

- Tagging a test for specific environment (DEV, INT, SYS)

- Tagging new tests for newbie environment (NEWBIE, ...)
- Tagging for specific purposes (POWERON)

If you think this styleguide lacks something important, please let me know so I can add the missing information.

3 Evolving Automated Testing

In an Agile development process, automated testing is an ongoing activity that is interwoven with the development process.

In this section I introduce the concept of small / medium / large scopes that as a mental model will allow you to keep your test suites effective and maintainable while your application evolves over time.

3.1 Complexity

Complexity is an important topic in software development and consequently, in software testing. Scientists from various disciplines such as Mathematics, Biology and even Engineering have made huge advances in the last few years in understanding complexity, but we have not yet come to a solid understanding of how complexity influences engineering and software development processes in particular. Nevertheless, we understand that complexity drives efforts in software development and also creates the risk of failure. For this reason, in test automation, we set ourselves the goal of reducing complexity by replacing large test cases by multiple test cases of medium or small scope that are easier to maintain and faster to execute.

Google proposed classifying test automation into three scopes of large, medium and small tests. The former Google tech-leader Prof. James Whittaker explains this approach as determining testing scope by "emphasising scope over form" [WHI2012].

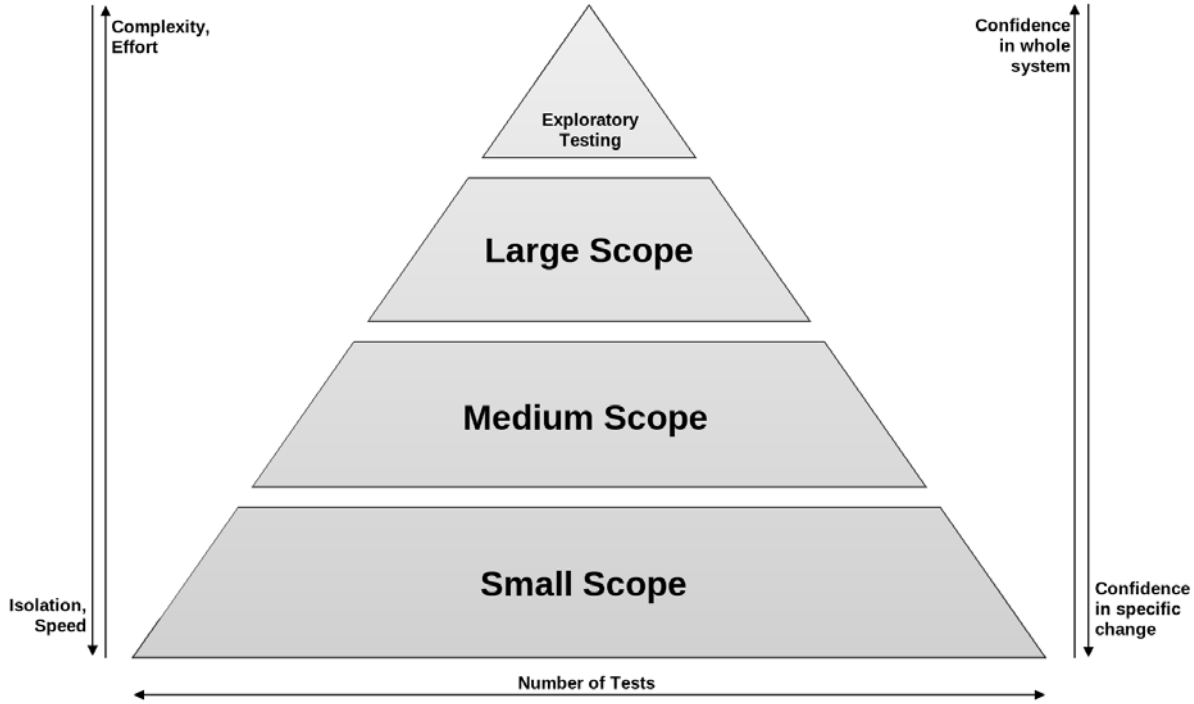


Figure 2: The testing pyramid consists of exploratory testing and large, medium, and small test-automation scopes.

Figure 2 shows how the different scopes in test automation relate to each other in the testing pyramid. The figure also contains a scope for exploratory testing, a subject covered in Part IV of the book.

3.2 Large Scope / Acceptance Tests

Large Scope tests exercise the entire application in an end-to-end fashion, typically through the GUI. This scope should be limited to checking for catastrophic errors. The question large scope tests attempt to answer is, "Does the application operate the way the user would expect?" Execution time for large tests is on the order of minutes or hours.

Table 2: Classification of Large Scope.

Benefit	**Weakness**
Test the most important aspects of the application	Nondeterministic (flaky) results because of dependency on external systems and test data
Test external subsystems	Broad test scope makes result analysis difficult and drives analysis efforts
	Time-consuming test data setup
	A high level of operation often makes it impractical to exercise specific corner cases. That's what small tests are for
	Long running
	Resource-intensive

Use large-scope tests sparingly! This is important for keeping your test-automation effort on track and ensuring the test suite stays maintainable. A word of caution: if your testers think it is best to test everything in the large layer because it is the "best" way to identify defects, you are soon destined to suffer test-automation congestion. Your test suite will become ineffective and inefficient. This means it simply takes too long to run and due to its flaky nature, it requires a lot of analysis to decide whether results point to a problem.

3.3 Medium Scope Tests

Medium Scope tests exercise interaction between components. They may have system dependencies or run multiple processes. The question medium-sized tests attempt to answer is, "Do near neighbouring functions interoperate the way they are supposed to?" They run on the order of seconds or minutes.

Table 3: Classification of Medium Scope.

Benefit	**Weakness**
Loosened mocking requirements and runtime limitations provide developers with a stepping stone that can be used to move from large tests to small tests	They can be non-deterministic because of dependencies on external systems
Tests run in a standard developer environment, so developers can run them easily	They are not as fast as small tests
They run relatively fast, so developers can run them frequently	
Tests account for interfacing with external subsystems	
Testing of configuration variants. Because the tests run fast, you are able to run thousands or even hundreds of thousands of variants, something that is prohibitive in the large scope	

Mocking technologies are often used to make components testable by taking out technical infrastructure such as message queues and email-based workflows.

3.4 Small Scope / Developer Tests

Small tests are very finely grained and exercise code within a single function or class. No external resources are involved (maybe some data files). Implementation of small tests requires mocks and fake databases. Developers often use small tests when diagnosing a particular failure. The question a small test attempts to answer is, "Does this code do what it is supposed to do?" Small tests execute on the order of milliseconds or seconds.

Table 4: Classification of Small Scope.

Benefit	**Weakness**
Lead to cleaner code because to be testable, methods must be relatively small and tightly focused; mocking requirements lead to well-defined interfaces between subsystems	Do not exercise integration between modules. That is what the other test categories do
Because small tests run quickly, they can catch bugs early and provide immediate feedback after code changes have been made	Mocking subsystems can sometimes be challenging
They run reliably in test environments	Mock or fake environments can get out of sync with reality
They have a tighter scope, which allows for easier testing of edge cases and error conditions, such as null pointers	
They have focused scope, which makes analysis of errors very easy	

Generally speaking, there is no need for exact classification criteria with the intent to achieve a sharp distinction between the three test scopes. On the contrary, we do not want to have a sharp separation, since we want to use this as a mental model to establish a consistent migration tendency for test automation from large to medium and small scopes.

3.5 How to Use the Model of S/M/L Scopes

If you are going to add new tests to your test suites, add them as low in the testing pyramid as possible.

Keep an eye on the number of large scope tests you use. If you count too many, identify areas where large scope tests can be replaced with one or more medium / small scope tests. In this way, you keep your automated tests in shape while your application evolves over time.

Within Google, project teams strive towards well-balanced test suites with an approximate goal of about 70% small, 20% medium and 10% large tests.

4 Installing the Continuous Integration Server and Demos

For "The Hitchhiker's Guide to Test Automation" book I created tools and demos with installation scripts and all necessary configuration. The Supercars application and tests are intended for to demonstrate all the different test automation aspects for a web application. The whole installation procedure should take no more than half an hour, depending on your Internet bandwidth and the machine you are working with.

Overview of the book's Github Repositories

The tutorial files and demo software can all be found at *http://github.com/markfink*. The files are contained in git repositories.

Relevant Github repositories are:

- supercars - the Supercars application

- fitnesse_jukebox - tutorial on using FitNesse
- SelRunner - using FitNesse for browser automation
- tutorial_ci - Jenkins plus test automation runtime
- tutorial_jasmine - tutorial on using Jasmine for testing the jukebox sample

Further installation details and installation procedures you can find at https://github.com/markfink/tutorial_ci.

5 Conclusion

The paper provides a brief overview on the topic of "Agile Testing at Scale" and reference to more detailed information. The paper does not provide an exhaustive documentation on the topic or technologies used for the implementation of test automation. Its intention is to reach people interested in software testing, in order to enter into fruitful discussions on software testing and test automation methodology. The acompanied installation scripts and demo applications aim to foster adaption into a variety of project environments.

References

[ADZ2010] Gojko Adzic, "How to implement UI testing without shooting yourself in the foot", http://bit.ly/9OoUf6

[MAR2008] Robert C. Martin, "FitNesse video tutorials", http://bit.ly/tBfUQ

[WHI2012] James Whittaker, 2012, *How Google Tests Software*, Addison Wesley

Championing Test Automation At A New Team: The Challenges and Benefits

Alan Leung

AlanLeung@sierrasystems.com

Abstract

Testing System-to-System (S2S) messaging requires painstaking attention to detail and a level of technical skill beyond verifying most GUI interfaces. Regression testing requires the same disciplined approach despite being a tedious and time-consuming task. S2S messaging is an obvious candidate for test automation but automating any process takes another level of understanding, expertise, and effort.

The mature project that I joined was responsible for several SOA (Service-Oriented Architecture) interfaces involving SOAP-based and REST-based web services, XSL transforms, and an EMPI data source. With limited staff and time to test these interfaces, I wanted to promote automated testing as much as possible in order to ensure software quality with constrained resources.

While the test team was aware of the potential benefits of automation, they needed some initial assistance to begin their efforts. Over the course of approximately a year, my assistance to the test team decreased from frequent interactions until the team was relatively self-sufficient and taking the testing framework in ways I had never explored.

The result of the test team's work is that five of seven interfaces have fully automated testing and the number of test scenarios has increased by 305%. Considering one interface, testing that was not as complete and that took approximately 7 days to complete now takes 7 minutes to execute. Because of the result of the team's work, an automated regression test suite (rather than one-time manual efforts), their product has also helped other teams downstream of the system testing phase. The test team has also gained valuable skills that can be deployed on other projects and other IT problems.

Previously, everyone on the team knew automated testing would be ideal and the benefits great, but to reach that state, we had to overcome many obstacles. However, just as automating tests pays off with every subsequent cycle of testing, changing the team's mindset to accept automated testing as the preferred way to test improves overall productivity and velocity of the team.

Biography

Alan Leung is a senior consultant with Sierra Systems. He is an expert systems integrator with architecture skills that has been a leader on numerous initiatives. A former member of IBM Global Services, his engagements have included health (Alberta Health), education (Athabasca University), and financial industry (Bloomberg) clients in Canada and the USA. Having been a member of a troubled project, written poor code early in his career and code reviewed a lot of suspect source in years past, he is now a code quality evangelist and a best practices advocate. Previously, Alan has presented at Edmonton (Alberta) Java user group events and at an internal IBM conference (Blue Horizon). He is Sun Java Certified and ITIL certified.

1 Challenges of testing S2S messaging

Unlike testing web-based GUIs or other graphical user interfaces, testing System-to-System (S2S) messaging interfaces requires tooling as well as a high degree of technical competence. To test S2S messaging, the tester must simulate an external system sending messages in the correct protocol. Constructing a valid request with proper security tokens, message IDs, and timestamps is much more complicated than merely interacting with a web page or desktop window. There is little to no room for improperly constructed messages: Not copying the entire security token string obtained from a secondary web service, neglecting to include a unique message ID, or missing a double quotation mark when building a test message will result in an error, and often the error response is more suited to a developer than the typical test analyst. Nearly every single detail needs to be correct over hundreds of executions of submitting test messages. Quickly verifying that the SOA interface's response fits expectations is another problem, as the result is not formatted for human readability: for example, checking for the response code and returned error message in an XML blob. The key pieces of the response that need verification are buried amongst other XML clutter that lacks whitespace, unlike a web page.

Our project team develops and supports seven interfaces that consume and return patient demographic data, for a provincial health ministry. Our interfaces are on the receiving end of S2S messaging from Electronic Medical Record applications (software used in physician clinics), Pharmacy Management System applications (software used in pharmacies), and internal ministry applications that rely on our registry. Five of these interfaces are SOAP-based and REST-based web services, and XSL transforms and an Enterprise Master Person Index (EMPI) data source are some of the technologies involved in the mix. Manually testing these interfaces was barely manageable when there was only five interfaces; the testing schedule was often exceeded. With two additional interfaces, either more resources and/or an increased testing schedule was necessary, but I believed that it was necessary to attempt automation of the testing.

2 Tool Selection

SOA interface test automation requires good tooling and when choosing a testing tool (or for that matter, any software tool), we should consider the following checklist:

- Mature product - to be confident that most tool functionality is free of defects
- Feature-rich and powerful – to satisfy scope of testing and be able to do it in an automated manner
- Good support - in the (likely) event that questions/problems arise when using the tool. Accessing good support through third-party channels such as StackOverflow can occur if the product is widely used and/or is a Mature product
- Extensible – when there is no built-in functionality that aids with testing an aspect of an SOA interface

Following is a summary of several tools in use (SoapUI, PIN S2S Tool, RESTClient, other home-grown tools) or considered (SoapUI Pro) at our health ministry.

	SoapUI	SoapUI Pro	PIN S2S Tool	RESTClient	Other home-grown tools
Description	Publicly available but not commercially supported tool for testing SOA interfaces	Commercial tool for testing SOA interfaces	In-house tool used by primary testing group for verifying and data loading one of ministry's mission-critical applications	Publicly available (http://rest-client.googlecode.com/) tool for testing HTTP/REST services	In-house tools created by project development team primarily for unit testing
Product Maturity	Developed since 2008 by a small team[1], at version 4	At version 4	Developed since approximately mid-2000s	Developed since 2008 by 1-2 developers[2], at version 3	The various tools have been developed and maintained by one developer over the course of approximately 4 years. User base is limited to the project team so the tools have not been tested as extensively as a widely available tool

[1] http://www.ohloh.net/p/soapui

[2] http://www.ohloh.net/p/freshmeat_restclient

	SoapUI	**SoapUI Pro**	**PIN S2S Tool**	**RESTClient**	**Other home-grown tools**
Features	Can construct requests for a variety of protocols (e.g. HTTP, REST, SOAP) and test cases can be configured with assertions for test automation	Features in addition to those offered by SoapUI (http://www.soapui.org/About-SoapUI/compare-soapui-and-soapui-pro.html)	Many features but primary goal is to support testing one particular application. Test cases can be configured with assertions for test automation	Limited to testing HTTP/REST services. Test cases can be configured with assertions for test automation, but built-in support is extremely limited such that a lot of custom code is necessary	Each tool is able to send messages for a specific protocol and sometimes limited to a specific SOA interface. Test scenarios cannot be configured with assertions for test automation
Support	Wide user community online, able to find documentation on SmartBear's web site as well as forums, blogs, etc.	In addition to support as per SoapUI, the Pro edition is commercially supported	Developers of tool have departed project	Documentation and user community is limited	Support is on a best-effort basis but communication of issues is easy
Extensibility	Test analysts can augment test automation at several points in a test lifecycle (test initialization, as explicit operations during test execution, during verification of the response, and test completion) by adding Groovy (a concise Java-compatible dynamic language) code. The SoapUI framework also has documented extension points that can be taken advantage of with Java code.		None, as far as known	Possible but built-in support is extremely limited such that a lot of custom code is necessary	Not possible by test analysts
Price	Free	$399 USD/year/license	No cost as in-house tool	Free	No cost as in-house tool

2.1 Free edition versus Pro edition

When I joined the project, SoapUI was the chosen testing tool, but was not being used to its full capability. The question is whether SoapUI Pro would have been a better choice. One of the practices recommended by Joel Spolsky (http://www.joelonsoftware.com/articles/fog0000000043.html) is to "use the best tools money can buy". However, due to policy issues, our project was never afforded the use of SoapUI Pro (http://www.soapui.org/About-SoapUI/compare-soapui-and-soapui-pro.html), but SoapUI Pro may be the better choice for a number of reasons:

- Ease of use, especially important for non-hardcore developers starting on the test automation journey. For example, "Coding Free Test Assertion", the ability to construct simple XPath assertions using a GUI rather than understanding XPath expressions.

- Less time developing scripts already available with the Pro edition ("Scripting Libraries"). The testers estimate 25-50% of scripting necessary for their testing is available in SoapUI Pro (based on looking at SmartBear web site documentation that indicates a feature is only available in the Pro edition). When comparing the Pro edition license cost versus the time savings for the development and test teams, it is evident why Joel Spolsky advocates spending money for the best tools.

- Team support so that test analysts can share a common SoapUI project file if testing the same SOA interface (http://www.soapui.org/Working-with-Projects/team-testing-support.html). Our testers resorted to having their own individual SoapUI project files per interface. This meant duplication of common scripts and properties which can be an issue for maintainability of the testing artifacts.

Note that SoapUI Pro would not have alleviated the need for some of the custom extensions made to SoapUI which were very particular to the SOA interfaces that we were testing or the applications that we were leveraging for verification. Also, one advantage of using the free SoapUI edition is that there were no issues transitioning SoapUI Projects to other teams that were not using the Pro edition as well.

Other commercial tools in the same space (SOA interface testing) include ITKO LISA and Rational Tester for SOA Quality but neither of these products was evaluated.

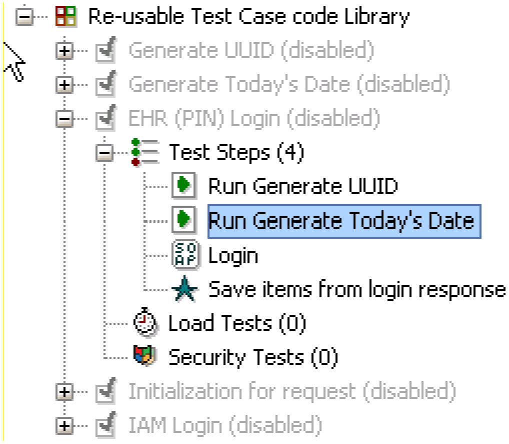
2.2 Piloting the tool

In order to determine whether SoapUI had the facilities to automatically test our SOA interfaces, I read the available online documentation from SmartBear and other web sites, but more importantly, I experimented with the tool by building a prototype against Google Maps REST APIs. Once I was relatively familiar with SoapUI's parameter substitution, assertion, and project organization capabilities, I was able to create an example for one of our interfaces, Active Query, to verify that SoapUI could be used on our project.

Due to commitments with primary tasks, there was little time allocated for one-on-one training. Instead my approach was to construct working examples of test cases against the Active Query interface, allowing the test team to experiment with SoapUI and the custom test automation framework. Having said this, it is important to note that not any tester can handle this approach. Testers without a development background would have had tremendous difficulty without constant one-on-one assistance. But the tester analysts on our team have a technical, development background, e.g., familiarity with SQL, relational databases, and procedural programming. While automating tests is not at the level of developing the application, "testing-scale" development/programming still requires many skills for which there existed gaps. How these gaps were addressed follows.

2.3 Custom framework

The customizations built on top of SoapUI consist of SoapUI Test Cases that can be executed from other Test Cases using the “Run Test Case” Test Step feature. This Test Suite of reusable Test Cases act as a library of subroutines heavily used for testing our SOA interfaces. The parameters and return values are transferred via Project-level Properties:

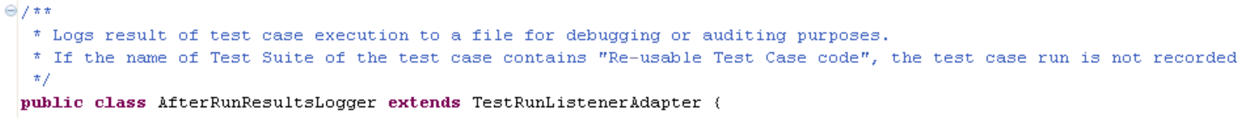


- Re-usable Test Case code Library
  - Generate UUID (disabled)
  - Generate Today's Date (disabled)
  - EHR (PIN) Login (disabled)
    - Test Steps (4)
      - Run Generate UUID
      - Run Generate Today's Date
      - Login
      - Save items from login response
    - Load Tests (0)
    - Security Tests (0)
  - Initialization for request (disabled)
  - IAM Login (disabled)

Figure 1: Reusable SoapUI Test Cases

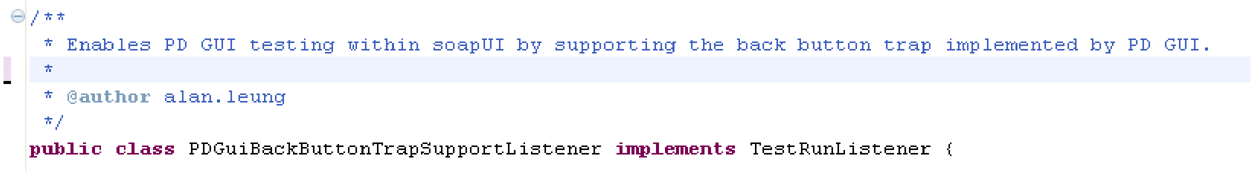
For example, the above figure shows the library of subroutines, implemented as a SoapUI TestSuite. The subroutines are TestCases that are all disabled as the TestSuite is not executed directly. To use this library, for example, the test analysts will include in their TestCases a “Run Test Case” Test Step to execute the “EHR (PIN) Login” TestCase. This TestCase will in turn execute other TestCases or TestSteps. For example, the “Run Generate UUID” TestStep will execute the “Generate UUID” TestCase. This TestCase contains a Groovy TestStep which executes some Groovy language (a concise Java-compatible dynamic language) code. The code modifies Project-level Properties as a way of “returning values” to the caller.

The second aspect of this SoapUI-based framework is Java code extensions that are either called directly from Groovy Test Steps or automatically executed by the SoapUI engine (http://www.soapui.org/Scripting-Properties/custom-event-handlers.html). Our testing eventually required three custom event handlers which I developed due to their complexity. Below are screen-shots of the beginning of the source code for each handler. The code comment describes the handler and the class from which the custom handler derives illustrates how it is implemented.



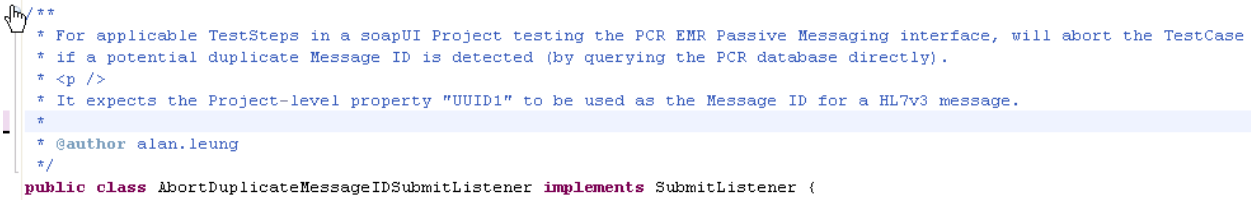
```
/**
 * Logs result of test case execution to a file for debugging or auditing purposes.
 * If the name of Test Suite of the test case contains "Re-usable Test Case code", the test case run is not recorded
 */
public class AfterRunResultsLogger extends TestRunListenerAdapter {
```

Figure 2: AfterRunResultsLogger event handler



```
/**
 * Enables PD GUI testing within soapUI by supporting the back button trap implemented by PD GUI.
 *
 * @author alan.leung
 */
public class PDGuiBackButtonTrapSupportListener implements TestRunListener {
```

Figure 3: PDGuiBackButtonTrapSupportListener event handler



```
/**
 * For applicable TestSteps in a soapUI Project testing the PCR EMR Passive Messaging interface, will abort the TestCase
 * if a potential duplicate Message ID is detected (by querying the PCR database directly).
 * <p />
 * It expects the Project-level property "UUID1" to be used as the Message ID for a HL7v3 message.
 *
 * @author alan.leung
 */
public class AbortDuplicateMessageIDSubmitListener implements SubmitListener {
```

Figure 4: AbortDuplicateMessageIDSubmitListener event handler

Having this mainly unobtrusive framework in place meant that the test analysts had less work to construct new test scripts within SoapUI. That is, SoapUI automatically calls the handler code at defined points in a test execution life-cycle and therefore the test analyst does not need to explicitly add anything to their scripts. Instead, "it just works". Secondly, we reduced duplication of Groovy test code by moving such logic into a Java Archive (JAR) file and then having Groovy "one-liners" make calls to the Java library.

Unfortunately the "Re-usable Test Case code Library" SoapUI Test Suite needs to be present in each SoapUI Project, so the Test Suite can become inconsistent between SoapUI Project files if its contents are modified by the testers. This has not caused any issues yet but theoretically can cause the same problems as any duplication of code: code fixes or code enhancements in one copy need a way to be propagated and merged into other copies. On the other hand, the Java code extensions stored in a .JAR file is consistent across SoapUI Projects because it is shared across all Projects in a SoapUI application installation.

3 Process of championing test automation

Initially the test analysts had reservations about the degree to which to automate the tests: How much time would it take, how usable was this framework, how many roadblocks would be faced and could the roadblocks be overcome? As with any procedure, a test case can be broken down into many steps. Some steps could be automated and some could be performed manually. My push was for complete hands-off automation but at the beginning, because of the high learning curve, there was some trepidation on how much could truly be automated. In the face of the uncharted territory and frankly, the large amount of upfront work necessary to automate tests, it was easy to consider falling back on manual one-off steps in certain situations.

3.1 Tasks in testing SOA interfaces to automate

There are many tasks involved in testing SOA interfaces and therefore many tasks to consider automating. At a high-level these tasks are defined below and our particular context is described:

1. **Use full capabilities of testing tool(s)** - in our case, the primary tool is SoapUI, and facilities it offered (that is available in the best testing tools) included:
    - Parameter substitution into the web service requests, automatic assertions, and endpoints per environment
    - Reading from data pool, e.g. from properties file or a relational database
    - Repeatable test execution - construction of test steps so that data is set up, test is executed, and clean-up of execution results occurs
2. **Construct valid web service requests** – our tested applications communicate with XML so it is important to understand what is valid XML. In particular, our health IT domain uses the HL7v3 messaging specification, which is XML-based with its own namespaces and XSD schemas
3. **Generate/manipulate test data** – SoapUI supports user-developed Groovy scripts to construct sections of the web service request based on reading information from a data pool, for example
4. **Interpret error messages returned by the application** – some of our SOA interfaces lie behind a Health Information Access Layer (HIAL), essentially an Enterprise Service Bus (ESB), which can return XSD schema validation errors. Often these error message are cryptic. For poorly constructed requests, the HIAL sometimes returns misleading or unfathomable messages
5. **Validate that web service responses match expectations** – SoapUI supports, among others, XPath expression assertions
6. **Validate that request to web service was actioned** - for some S2S requests, we could make a second request to confirm the execution of the first request. And although SoapUI has the JDBC Test Step to execute SQL against a relational database, our system uses a specialized data source, the Initiate EMPI; queries against the underlying relational database is generally frowned upon. However, often the most tell-tale way to verify the web service performed correctly is to use SoapUI to execute HTTP requests against Java Servlets that form the Initate EMPI end-user interface. The resulting web page can validate that, e.g., an Add request was stored into the EMPI.
7. **Logging of test efforts** - to automatically record evidence of testing and the results. The test analysts need to sign off the application before it can proceed to the next environment (i.e. “system test” to “user acceptance test” environments) and the logged test results provide evidence that the application performed properly, from an interface consumer’s perspective, when it was tested. The logs can also be used to show how the application behaved historically.

Sometimes a background in related technologies confused matters. For example, Selenium IDE behaves differently than HTTP testing with SoapUI. For another example, Oracle PL/SQL date function syntax differs from Groovy date function calls.

3.2 Building the confidence and ability to automate tests in the test analysts

For each of the tasks in the previous section, the test analysts needed to learn and become comfortable with constructing test scripts to perform the task in an automated way. On the other hand, we wanted to minimize the amount of assistance necessary from the development team, since this was not time allocated in project plans.

Following is a description of the particular process used to transfer knowledge for each of the tasks previously described:

1. **Use full capabilities of testing tool(s)** - testers started from an example SoapUI Project I carefully defined. I also authored a dozen-slide PowerPoint deck explaining the major intricacies of the Project and how to go forward. From there, the testers could conduct online searches for more information now that they knew what keywords and phrases to search for

2. **Construct valid web service requests** - for XML, the testers easily learned by example. However, for the specialized HL7v3 message specification, I put together a hand-out summarizing the scope of the specification that was important to our team, stepped through the official pan-Canadian Infoway documentation, and noted where to find relevant information.

3. **Generate/manipulate test data** - because of the testers' backgrounds, I did not need to give a Groovy programming tutorial. For more advanced constructs, the testers searched online for Groovy/Java examples and occasionally I fielded a question

4. **Interpret error messages returned by the application** - for schema validation error messages and HIAL error messages, both of which were only occasional, it was more difficult educating the testers on the reasons behind these messages because we did not catalogue these messages (i.e. recording exception text and cause). Thus, for some error messages, seeing them for the second time was like seeing them for the first time

5. **Validate that web service responses match expectations** - although my example SoapUI Project had some simple XPath expressions to perform assertions, I also referred the testers to online resources (http://msdn.microsoft.com/en-us/library/ms256086.aspx). However, for more complex assertion requirements, I needed to start off the testers with custom examples on a case-by-case basis

6. **Validate that request to web service was actioned** - our method of using SoapUI to behave like a web browser required some explanation to the testers so I authored a PowerPoint deck with diagrams to show the difference between a web browser, SoapUI, and Selenium, as well as how to log into the Initiate EMPI web application and use it to perform a search. The testers and I walked through an example SoapUI Project demonstrating this capability and then they had the PowerPoint slides as reference. A couple of weeks passed between the time of this demo and when this capability was tried. One test analyst was struggling to extend the example SoapUI project with his own verification steps. As I sat with him and spent a few minutes re-explaining how to use SoapUI like a web browser in an automated manner, switching between the PowerPoint and SoapUI application, he began to recall what we had walked through weeks earlier. The expression of understanding was gratifying.

   Obviously, the lesson is to deliver training in a just-in-time manner whenever possible, as even a carefully-authored PowerPoint reference may not be enough for a complex topic.

7. **Logging of test efforts** – in fact, this functionality was initially implemented as a Groovy script by one of the testers. But in order to alleviate the need to copy and paste this script into each Test Case, I extended SoapUI with a Java-based Event Handler that was more or less the original Groovy script but "Java-fied".

Much like automating tests, a large investment of developer time and effort was necessary upfront. For a couple of weeks, I estimate approximately two-thirds of my time was spent on a framework within SoapUI (organization of a SoapUI Project structure to support code re-use, Java extensions of SoapUI interfaces) and to a lesser extent, one-on-one assistance with each of the testers. This time commitment eased off thereafter with occasional bouts of activity with the framework or test analysts. After the initial investment of time and effort, the key to the testers' current self-sufficiency are:

- **Testers having a technical background** – giving them the confidence and experience to work towards automating the tests

- **Expanding on working examples** - because of their technical background, the test team could experiment with and expand on the working examples

- **Available online knowledge** - because SoapUI is an industry-wide tool with many users, and Groovy has many online resources, etc. the test team found many answers by doing an online search. However, the key is to establish a foundation in order to know what to search for

- **Just-in-time training sessions -** delivered from developer to test team for new testing capabilities that will be implemented by testers within days. These sessions might involve emailing the test analysts a sample SoapUI project demonstrating the new capability and walking through the sample as a group at someone's desk. Or it might involve booking a meeting room and going through a PowerPoint presentation together. However, the key is that as little time transpires between this education session and when the test analysts actually implement the new test capability in their work.

I've been on teams where the business analyst team, test team and development team are segregated by an invisible barrier: The development team is reluctant to clarify requirements with the business analyst team, thus incorrectly building aspects of the application; the test team doesn't ask the development team for assistance to test more challenging parts of the application, thus insufficiently testing the application or taking more time than necessary; and the business analyst team has not verified the test team's test scenarios for completeness, thus defects are not detected until much later.

While I offered assistance to the test team to automate their tests, I tread a fine line so as not to intrude on their day-to-day work. However, when I thought there were gaps in the test verification, e.g. using a simple but fallible Contains assertion versus a complicated XPath assertion, I intervened to introduce a failsafe assertion.

3.3 Train the trainer and sharing automated test scripts

In our software development lifecycle, testing takes place by the development team, then the system test team (the team I assisted), and followed by the user acceptance test team (which is sometimes the "business" team that gives direction on what applications to build and manages production operations). Armed with their knowledge of the testing framework, the system test team transitioned their SoapUI Projects to the UAT (User Acceptance Test) team as a basis for more test scenarios during the UAT phase.

The transition from the system test team to the UAT team highlights that a technical background is highly recommended to automate new test scenarios and to even work with existing automated test scripts. Sometimes the UAT team consisted of business subject matter experts who did not have a development background. Then the UAT team consulted the system test team for one-on-one assistance frequently.

Sometimes a UAT team consisting of business SMEs did not use their time to develop new automated test scripts but instead analyzed test results with a big picture understanding of the application within a larger overall system. Often this was valuable and only possible because the system test team delivered essentially an executable product, the SoapUI Projects, to the UAT team. Also, feedback from the UAT team sometimes augmented system test scripts.

4 Quantifiable results and subjective results

Was it worth the time and effort to convert the testing to be automated rather than manual? Our team firmly believes so:

- Considering one SOA interface, due to automation, we increased our test coverage by increasing our test scenarios from 110 to 480. Considering another interface, test scenarios increased from 88 to 125.

- Regression testing during infrastructure upgrade projects (DB2, Initiate EMPI, etc.) only takes 7 minutes of unattended test execution rather than approximately 7 days of tedious manual labor. For another interface, automated execution takes 111 minutes versus an estimated 475 minutes of manual execution.

- UAT team can re-use automated test scripts (although making changes for data available in UAT environment)

- Automated execution, when it needs to be extremely stringent as per S2S testing, eliminates typo type errors (after the test is established once) and does not inadvertently skip steps

- For ongoing support, the application is documented based on the behavior of tests. With automated tests, inquiries about the application are handled much quicker, and there is no need to rely on stale documentation (source code comments, requirements documentation, operation manuals, etc.). Some developers may say that the source code is truth, but it is not the human interpretation of source code that is truth, but rather the "ultimate semantics of a program is given by the **running** code" (http://programmer.97things.oreilly.com/wiki/index.php/Only_the_Code_Tells_the_Truth)

- Automating tests and the ease with which to create and execute more variations of a test scenario helps to update stale documentation to match the behavior of the application and makes requirements documentation more detailed

- Automated tests quickly detect defects that are often created when fixing another defect (http://brockreeve.com/post/2011/03/20/Bug-Fixing.aspx). Because of the intricacies of medium to large software systems, in the rush to correct an urgent defect, a developer can introduce a new defect while fixing the original one. Without a set of automated unit tests, these errors are overlooked and the application is deemed ready for the test analysts. But a comprehensive set of tests can quickly uncover these shortcomings. This improves the team's overall velocity (http://en.wikipedia.org/wiki/Velocity_%28software_development%29).

- When developing automated tests, by serendipity, sometimes defects are uncovered, and the half-developed test was not part of the official test scripts

With so many benefits to automating testing, it is easy to forget the initial hard work required to arrive at a state of high test automation. Therefore, management has to understand and buy-in to the up-front time/cost with automating testing, realizing the longer-term benefits. Some time prior to the official start of the testing phase may be needed to become familiarized with any new automated testing framework capabilities. Subsequent to the testing phase, we need to also consider maintenance efforts, to update the automated test suite/framework when the application's behavior changes or to re-factor the test harness for ease of future use (e.g. additions of test scenarios, less need to copy-and-paste).

Our team took months to learn the entire testing framework based around SoapUI. They acquired many skills during this time, including SoapUI, Groovy/Java, JDBC, XML, XPath, REST, and SOAP. However, for projects of a shorter duration (those measured in weeks), our approach would not be viable. Instead, either more capable tools are used or dedicated testing framework development and test analyst training time needs to be allocated.

5 Test automation Best Practices

Several best practices we learned from this experience:

1. **Automatically log test execution** - this alleviates the need for screen prints and records a history of how the application was behaving in the past

2. **Parameter substitution for uncertain aspects under development** – as an example, we parameterized two code values that required the Canada Infoway standards organization to approve. When the standards organization decided on other values than we were using ("MDATA" rather than "MERGEDATA"), it was straightforward to alter the test properties.

3. **Eliminate duplication** – for example, of Groovy scripts. SoapUI offers several methods to reduce duplication: the ability to call any code from a Java library (.jar) placed into its program extension directory (bin/ext) and execution of customized SoapUI Event Handler extensions. Also, testers should write more sophisticated Groovy script to eliminate simpler but copy-and-pasted Groovy script Test Steps. Duplication causes problems when for example, a defect needs to be fixed in each copy of the code

4. **Verify validity of assertions** - while it is satisfying to see a Test Case run "green" by passing all its assertions, it is imperative to double-check that the configured assertions only verify the expected result and do not pass a "failure" result. For example, a loose Contains assertion may pass even when a stricter XPath assertion fails (i.e. the matching string is appearing in an unexpected location in the response).

5. **Treat testing artifacts like application source code** – due to the complexity of automated test scripts, they can stop working unexpectedly and it is useful to compare with a known working version. Therefore testing artifacts should be stored in a version control system. A different branch of the testing artifacts should be created when application source code is branched.

6 Conclusion

For a test team accustomed to manual testing on the project, achieving test automation as the general mode of testing requires support from the development team as well as commitment from the test team. Obvious benefits are increased testing productivity and overall development velocity, but a less-obvious benefit is the valuable skills gained by the test team. However, management must realize that automating testing needs sufficient time to establish but the return on investment is realized with every execution of the test suites.

References

Black Duck Software, 2013. SoapUI open source code analysis, http://www.ohloh.net/p/soapui (accessed July 31, 2013).

subwiz (a rest-client author). 2013. rest-client Google Project Hosting web site, http://code.google.com/p/rest-client/ (accessed July 31, 2013).

Black Duck Software, 2013. rest-client open source code analysis, http://www.ohloh.net/p/freshmeat_restclient (accessed July 31, 2013).

SmartBear Software, 2013. “Compare SoapUI and SoapUI Pro”, http://www.soapui.org/About-SoapUI/compare-soapui-and-soapui-pro.html (accessed July 31, 2013).

Spolsky, Joel, 2000. “The Joel Test: 12 Steps to Better Code,” Joel on Software blog, entry posted August 9, 2000, http://www.joelonsoftware.com/articles/fog0000000043.html (accessed June 10, 2013).

SmartBear Software, 2013. “Team Testing Support”, http://www.soapui.org/Working-with-Projects/team-testing-support.html (accessed July 14, 2013).

SmartBear Software, 2013. “Custom Event Handlers”, http://www.soapui.org/Scripting-Properties/custom-event-handlers.html (accessed June 10, 2013).

Microsoft, 2013. “XPath Examples,” Microsoft Developer Network, updated August 2, 2012, http://msdn.microsoft.com/en-us/library/ms256086.aspx (accessed June 10, 2013).

Sommerlad, Peter, 2008. “Only the Code Tells the Truth,” 97 Things Every Programmer Should Know, entry updated October 16, 2008, http://programmer.97things.oreilly.com/wiki/index.php/Only_the_Code_Tells_the_Truth (accessed June 10, 2013).

Reeve, Brock, 2011. “Bug Fixing,” blog entry posted March 20, 2011, http://brockreeve.com/post/2011/03/20/Bug-Fixing.aspx (accessed June 10, 2013).

Wikipedia, 2013. “Velocity (software development),” Wikipedia, entry updated March 6, 2013, http://en.wikipedia.org/wiki/Velocity_%28software_development%29 (accessed June 10, 2013).

The Perfect Couple: Domain Models & Behavior-Driven Development

Dan Elbaum
dan@danelbaum.com

Carlin Scott
carlin.q.scott@gmail.com

Abstract

Behavior-Driven Development entails a unified set of practices for expressing requirements in business-readable language, and binding them to automated test cases. Unfortunately, the Agile community has been so focused on the test automation aspects of the methodology that they have overlooked the challenges involved in the initial step of BDD: expressing requirements in business-readable language.

While business-readable language simplifies communication with business stakeholders, it also complicates the process of specifying requirements for the implementation team. The informality of business language makes it difficult to write requirements that are accurate, consistent, and unambiguous. How can practitioners specify technical requirements with business-readable language?

This paper explores how domain models help to reconcile the looseness of business language with the exacting requirements of engineering. By systematically representing key concepts in the business domain with visual diagrams and operational definitions, we can construct a domain model that allows us to unambiguously specify implementation requirements in business-readable language.

Biography

Dan Elbaum is a Senior IT Business Analyst at Con-Way Enterprise Services in Portland, Oregon. Over the past 6 years, he has focused on the development of web-based information systems. Dan holds an M.S. in Management from the University of Florida and is an IEEE Certified Software Development Professional, as well as an INCOSE Associate Systems Engineering Professional.

Carlin Scott is a Software Development Engineer in Test at Intel Corporation and previously worked for Hewlett-Packard. He has worked in QA as an SDET for the past 4 years and holds a BS in Computer Engineering from Oregon State University.

1 Introduction & Motivation

Requirements hand-off from business stakeholders to implementation teams is a notoriously weak point in commercial software development due to overlapping technical and social factors. At the technical level, the intrinsic complexity of software makes it difficult to conceptualize and communicate requirements. That technical layer of difficulty is compounded by the organizational context of software development, where requirements must be coordinated with multiple stakeholders who come from different disciplinary backgrounds, think about software at different levels of abstraction, and express their ideas in different terminology. Commentators often suggest that this “communication gap” is the main source of our requirements problems.

Behavior-Driven Development aspires to bridge the communication gap by framing requirements at a level of abstraction that is mutually understandable to business and technical stakeholders alike. Unlike traditional imperative specifications that describe how code operates procedurally, BDD specifications describe an application’s intended behavior from the perspective of a domain expert, using domain-specific terminology (also referred to as “ubiquitous language”). There is wide consensus that the vocabulary of a ubiquitous language should be derived from a model of the application domain[1], yet few practitioners bother to create that domain model. This is problematic. To skip domain modeling is to miss the point of domain-specific language: it is intended to shift our focus to a different level of abstraction, not to relax our level of precision. Domain modeling deserves broader consideration from Agile practitioners because it is a powerful technique that allows us to structure our conceptions of an application domain, disambiguate ubiquitous language, and ultimately bridge the communication gap.

The purpose of this paper is to explain how domain models augment ubiquitous language, and demonstrate a general method for how to apply them in practice. The goal of this method is to enable Agile teams to comprehend specifications faster and more accurately. The method uses visual diagrams and operational definitions to illustrate and define domain-specific concepts. The result is a richer ubiquitous language that allows us to communicate application requirements with greater clarity and consistency.

This paper is divided into five sections. The first section describes how BDD carries requirements throughout the application development lifecycle. Section two evaluates the strengths and weaknesses of ubiquitous language in different areas of requirements engineering. Section three explains how domain models enhance ubiquitous language. Section four presents our method for building a domain model, deriving a ubiquitous language from it, and using those terms as a controlled vocabulary to write scenarios. Section five addresses anticipated criticisms of domain modeling as anti-Agile.

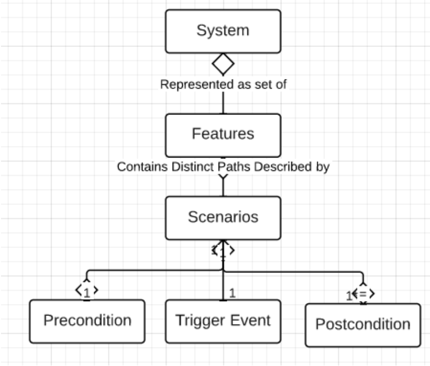
2 The Life of a Requirement in Behavior-Driven Development

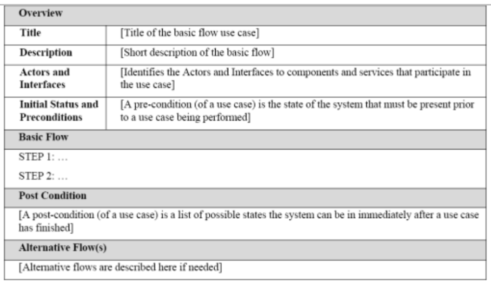
Behavior-Driven Development is an emerging agile methodology that carries requirements throughout the entire value stream, from analysis, through to development and testing. The methodology was first introduced by Dan North, who defines it to mean "implementing an application by describing its behaviour from the perspective of its stakeholders”.[2] In what follows below, we provide an overview of how requirements are formulated and carried throughout the development lifecycle in BDD.

Feature-centric. As a starting point, BDD organizes development by partitioning an application into modular features that can be independently built. Requirements are organized around individual features.

Use Case Scenarios. Each feature’s intended behavior is described with a type of use case scenario called as a Given-When-Then scenario, a one sentence narrative that describes the set of possible interactions that a user can have with the feature. The general format is: Given the application’s initial state, When a trigger event occurs, Then the application should exhibit some behavior in response.

Below are representations of use case scenarios. The meta-model is on the left, the traditional tabular format is in the middle, and the BDD format is on the right.





Overview	
Title	[Title of the basic flow use case]
Description	[Short description of the basic flow]
Actors and Interfaces	[Identifies the Actors and Interfaces to components and services that participate in the use case]
Initial Status and Preconditions	[A pre-condition (of a use case) is the state of the system that must be present prior to a use case being performed]
Basic Flow	
STEP 1: … STEP 2: …	
Post Condition	
[A post-condition (of a use case) is a list of possible states the system can be in immediately after a use case has finished]	
Alternative Flow(s)	
[Alternative flows are described here if needed]	

Given Precondition

When Trigger Event

Then Postcondition

Ubiquitous Language. We write use case scenarios in terms of domain-specific abstractions elicited from domain experts. Collectively, these terms constitute a “Ubiquitous Language”. Framing requirements at this level of abstraction enables domain experts to participate in requirements validation, and confirm that a specification satisfies their requirements. The table below illustrates how different levels of abstraction could be used to frame a scenario where a user logs into Netflix and sees on their landing page a set of recommended movies.

Level of Abstraction	Use Case Scenario
Domain	Given that I am a Netflix Member When I submit correct login credentials Then I should see my Movie Recommendations
User Interface	Given that I am on the Netflix login page And I have typed my email address into the username field And I have typed my password into the password field When I click on the login button Then I should see my homepage
Application	Given that the text input posted contains no dangerous characters When the user’s email address matches a row in the customer table And they successfully authenticate Then their session should be initiated And the movies in the recommendation table ranked 1-8 should be retrieved

In the example above, at the domain-specific level of abstraction, “Movie Recommendations” would be considered a domain-specific concept. Suppose it refers to the output of a learning algorithm which aggregates members’ viewing history to construct a model of their preferences and recommends films they are likely to enjoy. That term would be a formal construct within the application domain. In BDD, all participants in system development are supposed to adopt the same terminology for referring to such abstractions when communicating with one another about the application, making a single set of terminology effectively “ubiquitous”. The hypothesis behind this approach is that cross-functional teams can communicate more effectively when they speak a common language.

Use-Case Testing. To verify that a feature satisfies its requirements, we implement each of its Given-When-Then scenarios as executable tests. Each step in a GWT scenario is parsed into an executable block of code that directly exercises the system-under-test. When run, the Given step sets up the precondition, the When step executes the trigger event, and the Then step checks to see if the post-condition has occurred. If so, then the entire test passes. This testing technique is known as use-case testing because the tests are intended to simulate how a real user would interact with the system.

Because GWT scenarios serve a dual-purpose in BDD as both a functional requirements specification and a test case, these are often referred to as “executable specifications.”

Test-Driven Development: These “executable specifications” are the tests that drive BDD’s test-driven development lifecycle. The focus of development is to code all features so that they pass all of their Given-When-Then tests. Once those tests pass, development of a feature is considered complete. The philosophy behind this approach is that writing tests before development helps to build-in quality from the outset, which is more effective than “inspecting it in” through post-development quality assurance activities.

Thus, every step in the development process is glued together by Given-When-Then scenarios written in domain-specific terminology. Domain-specific terms are elicited from stakeholders, and used to write Given-When-Then scenarios that specify the application’s intended behavior. Then, these scenarios are translated into test cases which guide implementation in a test-driven lifecycle.

3 Ubiquitous Language: Strengths & Weaknesses

We have seen that domain-specific scenarios are the singular artifact that BDD uses to carry requirements throughout the entire development lifecycle. A domain-specific level of abstraction is used to elicit requirements from upstream business stakeholders, and to specify and verify requirements with the downstream implementation team. Unfortunately, ubiquitous language is not equally effective for all of these requirements engineering activities. Ubiquitous language is convenient for requirements elicitation because it is informal. However, this informality makes it less suitable for requirements analysis and specification. In this section, we review ubiquitous language’s strengths and weaknesses for different requirements engineering activities.

3.1 Strengths of Ubiquitous Language: Requirements Elicitation & Validation

The primary strength of ubiquitous language is that it simplifies requirements elicitation and validation. It provides a common language that enables business and technical stakeholders to communicate more effectively. Speaking in terms of the application domain enables stakeholders to focus on their business requirements, and avoid conflating them with implementation details. The black-box perspective of ubiquitous language is beneficial in that it helps to separate “the specification of requirements from design, simplifying the model and making the requirements model easier to construct, review, and formally analyze.”

A secondary strength of ubiquitous language is its ability to provide the development teams with cognitive benefits that enable them to deliver better-working software, faster. In using ubiquitous language to collaborate with businesspeople, development teams gain insight into the underlying purpose that an application is intended to fulfill, and that helps them to comprehend business requirements. Comprehending complex requirements specifications has been shown to depend crucially on understanding the rationale behind them.[3] Nancy Leveson, a software researcher specializing in software specifications, has extensively documented the phenomena whereby software engineers have difficulty comprehending complex specifications if the underlying rationale and intent is not explained. Another principle which would suggest that ubiquitous language may have a beneficial psychological impact on implementation teams’ understanding of an application is the Sapir-Whorf hypothesis. The Sapir-Whorf hypothesis posits that the language that we speak influences the way that we conceptualize the world. As applied to software development, adopting the terminology of the business domain may enable the development team to better understand the perspective of business stakeholders.

3.2 Weaknesses of Ubiquitous Language: Requirements Analysis & Specification

Ubiquitous language lacks the degree of exactness required to analyze requirements and specify scenarios clearly. Because scenarios directly influence implementation in BDD, precision is critical, and errors are costly. In contrast to user stories, which can be written loosely because they only convey the general purpose of a feature, BDD scenarios must be expressed more precisely because they prescribe how software is intended to function, and serve as the basis for implementation. Heightened attention to accuracy is necessary, because errors in ubiquitous language can directly introduce defects into code and, as everyone knows, defects introduced in the requirements stage are often the most difficult and expensive to correct. Consider, for example, how inexact usage of ubiquitous language might impact the development of a college course registration application. Suppose that we're building a test suite which contains scenarios that refer to the instructor of a course in various ways such as a lecturer, professor, and teacher. It may be unclear whether these terms are synonyms, or different domain-level concepts that should be implemented as separate classes. If such a misunderstanding weren't detected early upstream, it would directly introduce cascading defects into the product. Such a possibility is not at all unlikely because technologists often perceive non-technical sounding business terms as fluffy, vague, and not worth paying attention to.

Another disadvantage of ubiquitous language is that it cannot express all aspects of a domain that must be understood to analyze elicited requirements, and form a specification sufficiently detailed to serve as the basis for development. Much of the information that software engineers need to build a system is difficult to convey in non-technical terms. If we use only business-readable language in requirements specifications, how are software engineers supposed to get the information they need? This is a problem which must be resolved in order for domain-specific language to effectively carry requirements throughout specification and verification.

4 Domain Models Improve Analysis & Specification

Domain models can ameliorate the shortcomings of domain-specific language that we have just described in regard to requirements analysis and specification. In the context of software development, a domain model is an abstract representation of the elements of the business domain that the software-under-development is intended to embody. It "collects all the information about the problem domain that must be known and understood to allow capturing requirements for the system, specifying them, implementing and verifying the system."[4]

The general purpose of domain models is to organize domain-specific concepts into a coherent representation. This allows us to make technical inferences about the domain without creating a separate set of "technical specifications". Structuring information about the domain into a formal or semi-formal model allows us to analyze requirements for completeness, consistency, and correctness, as well as to communicate requirements to the implementation team more effectively. In this regard, domain modeling aligns with our overarching goal in requirements specification: to support the implementation team's ability to understand and reason about the application domain effectively and efficiently. In this section, we provide an overview of the benefits conferred by the two key parts of our domain models, visual diagrams and operational definitions.

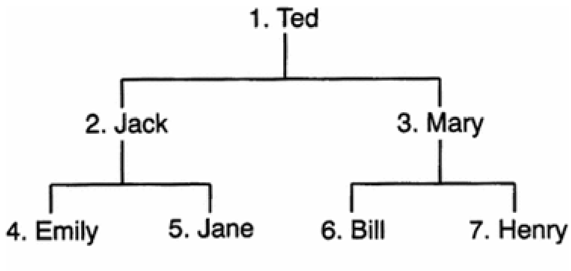
4.1 Visual Diagrams Provide Cognitive Benefits

Visual diagrams are useful in domain modeling because they present important cognitive benefits that enable people to learn certain types of information more effectively. Text and pictures can represent equivalent information, but pictures can be processed faster in many instances. While textual requirements are useful in some phases of software development, visual representations are an effective tool for aiding developers' understanding of complex specifications. A large body of cognitive psychology research has proven that supplementing or replacing text-based representations of information with visual diagrams can help learners to process information more effectively, with less time and mental effort.[5] This directly supports Agile's aim to deliver working software rapidly. It attacks the limiting factor on the speed of software delivery: the rate at which developers can process information about the system-to-be.

One important psychological advantage that graphical diagrams confer is known as the "gestalt effect". The gestalt effect states that humans perceive objects by trying to grasp them as a whole. When equivalent information is represented in both text and diagrams, people can gain a high-level overview faster from the diagram. Depicting a high-level overview of a system helps development teams to contextualize and comprehend its lower-level details.

Another important psychological advantage of visual diagrams is that they support alternate search strategies, or ways of finding information. Viewers can zoom into or out of different sections in a diagram, or focus on the relationship between two parts of a diagram. Text provides must generally be read in a linear order. Visually displayed of information can be perceived in a single glance.

The below diagram of a family tree is from a classic cognitive psychology paper that describes the advantages of visual representations of information over text.[6] Both the text and the diagram contain equivalent information. Try using the text to infer cousin and sibling relationships, and then try it again using the diagram. Which is easier to use?

TEXT-BASED REPRESENTATION	**DIAGRAMMATIC REPRESENTATION**
1. Ted is Jack's parent. 2. Ted is Mary's Parent. 3. Jack is Emily's parent. 4. Jack is Jane's Parent 5. Mary is Bill's Parent 6. Mary is Henry's Parent	 1. Ted 2. Jack 3. Mary 4. Emily 5. Jane 6. Bill 7. Henry

In section 5, we will review some diagrammatic notation which can be useful for domain modeling.

4.2 Operational Definitions Disambiguate Domain-Specific Terms

Operational definitions are the second key part of domain models. Domain-specific concepts must be precise in order to be useful in specifying an application's behavior. In order to get the necessary level of precision, we must formalize the semantics of domain-specific terms by operationally defining each one. The collection of operational definitions serves as a controlled vocabulary which we use to describe the application's intended behavior in Given-When-Then scenarios. This technique, known as Vocabulary Management, is a way to "improve the effectiveness of information storage and retrieval systems. The primary purpose of vocabulary control is to achieve consistency in the description of content objects and to facilitate retrieval".[7] These definitions are crucial because they disambiguate domain terminology and reduce opportunities for misunderstanding. Teams working on complex projects may need to re-review the specifications multiple times throughout the development cycle in order to refresh their memory or clarify a term. A glossary of operational definitions serves as an authoritative source that people can refer back to at any time.

5 Building Domain Models for Usage in BDD

This section describes our general method for creating ubiquitous language from domain models. The approach consists of three steps: 1) graphically model the business domain, 2) operationally define its constituent elements, and 3) use those definitions as a controlled vocabulary for writing BDD Scenarios.

5.1 Step 1: Build a Visual Diagram of the Domain

The first step in our approach is to visually diagram two key aspects of the business domain: static structure, and dynamic behavior. The static structure of a domain is a point-in-time snapshot of the key elements in the domain. Often referred to as the 4 W's of Who, What, Where, When, this is simply the key people, things, places, and times involved in the domain. The dynamic behavior is the set of processes, events or transactions that occur over time which effect or involve its static parts listed above.

Our recommendation is to start by modeling the static aspects before the dynamic aspects. The logic behind this decision can be illustrated through analogy. Just as the motion of a mechanical system is easier to understand after examining the system at rest and seeing how its parts fit together, the behavior of an information system is also easier to learn after we understand its structure.

5.1.1 Capturing Key Domain Concepts in a Conceptual Data Model

In developing information systems, as most of us in commercial application development are, the static elements of the domain are represented within the application as persistently stored data. The field of information science has already developed mature data modeling techniques that are useful not only for implementing databases, but also for illustrating the structural organization of abstract information. That is our purpose here. We will depict the static aspects of the application domain using a "conceptual data model". The Data Management Body of Knowledge defines a conceptual data model to mean "a visual, high-level perspective on a subject area of importance to the business. It contains only the basic and critical business entities within a given realm and function, with a description of each entity and the relationships between entities. Conceptual data models are intended to convey a conceptual overview of domain concepts, not to specify the implementation of a database. We omit implementation-related details that are not relevant to a conceptual overview. For example, if our diagram of choice for a conceptual data model were an entity-relationship diagram, we would not normalize the tables or enforce referential integrity.

One proposed approach to creating a conceptual data model within the BDD context is to start with an unstructured "mind map" of key domain-specific terms, and transform it into a UML class diagram.[8] Whether the notation of choice is a class diagram or an extended entity-relationship diagram, the following concepts can help to organize the structure of domain concepts.

5.1.1.1 Entities & Attributes

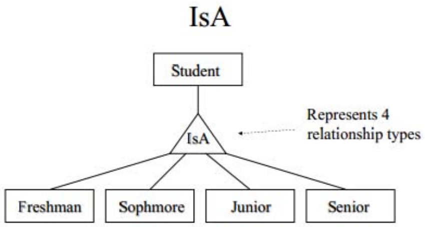
At the most basic level, our conceptual data model must capture the key people, places, and things as atomic entities, and indicate their key properties as attributes.

5.1.1.2 Associations & Their Multiplicities

The relationships between the entities identified above should be noted, and characterized in terms of their "multiplicity". Multiplicity is the number of instances of some entity that can exist for each instance of a related entity. To illustrate the importance of multiplicities, consider the following issue that could arise in the context of a college class registration application. Suppose that a lazy student attempts to enroll in two classes held during the same time period because he plans to skip all lectures anyways, and only come to class on exam days. Whether the student is permitted to double-enroll for a timeslot is not merely an implementation detail, it's a business rule. This information can be concisely expressed using the notation for multiplicities.

5.1.1.3 Type Hierarchies

Type hierarchies are a generic type of relationship between entities that we can express precisely with diagrammatic notation. Identifying subtypes of a superclass, or supertypes of a subclass, is useful for illustrating classification-related aspects of information organization. The following diagram contains an example of a type hierarchy which illustrates the four subtypes of high school student: Freshman, Sophomore, Junior, Senior.



5.1.1.4 Aggregation

Aggregation is another useful concept for illustrating the organization of information. For example, suppose we're looking at population by geographical areas. The population of a continent is the aggregated population of the countries it contains, which is the aggregate of its state populations, etc.

5.1.1.5 Composition

Composition is another generic concept that illustrates how a higher-level object emerges from the composition of its constituent parts, i.e., humans are more than the sum of their anatomical parts. UML's class diagrams provide support for expressing this type of relationship.

5.1.2 Diagramming the Dynamic Processes

Next, we must identify the processes that the application supports, such as ordering a pizza or executing financial transactions. BDD conceptualizes processes as state machines, where a trigger event causes the system to transition from an initial state to a final state. Along these lines, BDD scenarios can be represented as a graph, where the initial state is a node, actions required to transition to another state is an edge, and the final state is another node. Each scenario specifies a possible state transition. There has been some work published that provides guidance on how BDD scenarios map to constructs in Business Process Modeling Language.[9] We provide a graphical depiction of a simple BDD scenario related to grocery store checkout at the end of this section.

5.2 Step Two: Create a Controlled Vocabulary of Operational Definitions

The second step in our approach is to label and operationally define key elements of the domain. An operational definition is "a definition that identifies a concept as a variable that can be used in a hypothesis".[10] These terms can be thought of as operational definitions in the sense that the BDD scenarios are testable hypotheses about how the system should behave. Together, these terms serve as a controlled vocabulary for writing BDD statements.

5.3 Step Three: Apply the Controlled Vocabulary to Write Specifications

The idea here is to use the controlled vocabulary derived from our domain model to write our Given-When-Then scenarios. Given that this specifies exact terminology for key roles, objects, transactions, etc, we can express most interactions with the system these terms. Consistent terminology here will enable the software delivery team to look up the definition, or refer back to the graphical models to refresh their memory. Below we provide a sample scenario for a section of a feature involving grocery store self check-out. It includes a process diagram and operational definitions for the relevant ubiquitous language.

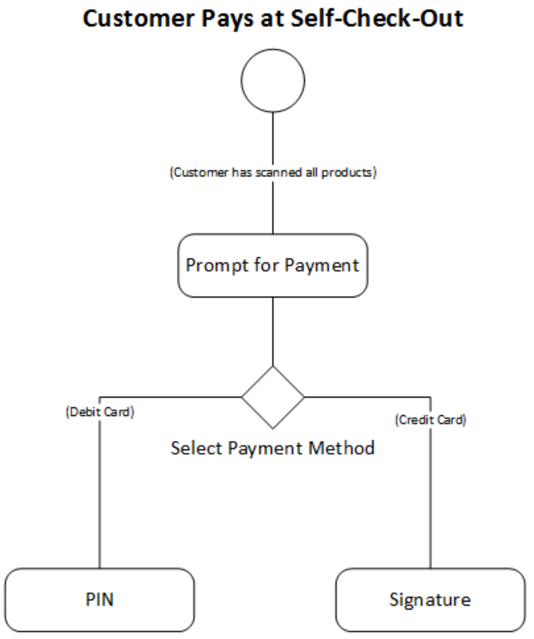
Operational Definitions

Customer – Person Purchasing of Product(s).

Product – Store inventory item available for purchase

Form of Authorization – Information Required to Complete Purchase with non-cash payment method.

```
Feature: Self-Checkout

 Scenario: Payment Authorization

 Given Customer has been prompted for payment

 When Customer pays with <Payment Method>

 Then Customer should be prompted for <Form of
Authorization>

       |Payment Method| Form of Authorization |

       |Debit Card    | PIN                   |

       |Credit Card   | Signature             |
```



6 Does Domain Modeling Violate Agile Principles?

The approach suggested in this paper may meet some resistance from practitioners who perceive domain modeling as anti-Agile. Does domain modeling enhance or detract from the fast delivery of working software?

As a starting point, we must note the obvious fact that a domain model takes time to create. Thus, the domain model will delay delivery unless it accelerates other activities in the development process. Luckily, there is reason to believe that domain models may actually accelerate delivery. Domain models could allow the team to understand the requirements faster, and more accurately. Learning the specification faster produces a small gain in speed. The major gain in delivery speed would be related to understanding the specification more accurately. This would enable the team to correctly implement quality code from the outset, reducing the need for re-work downstream.

We suspect that the benefit of domain models is directly proportional to the complexity of the application at hand. As the application domain grows in complexity and becomes less intuitive, the quality of specifications plays a larger role in enabling a quality implementation. As a result, domain modeling may not be a valuable technique to apply to simple applications. However, we believe it's an extremely worthwhile approach for more complex domains such as health informatics, financial systems, or supply chain management.

A secondary anticipated criticism of domain modeling is its alignment with Agile's aim to reduce comprehensive documentation. Domain modeling is not a comprehensive form of documentation. A domain model can be continuously iterated to provide just-in-time detail, and it does not require a "big-design up front" approach where the complete application needs to be modeled in its entirety before development begins.

Given that domain modeling employs lightweight models to accelerate delivery of working software, we argue that it advances the Agile principles.

7 Conclusion

Domain models enable better analysis and specification of requirements written in domain-specific terms. Constructing a domain model helps us to decompose stakeholders' descriptions of the domain into a set of primitive semantic constructs (the vocabulary of the ubiquitous language) that we can then combine to compose systematic descriptions of how the domain behaves (using Given-When-Then scenarios). Domain modeling also precludes the need for separate "technical specifications" to some extent by illustrating the structural organization of domain-level abstractions with diagrammatic notation. These diagrams exploit powerful psychological principles to enable people to process specifications more effectively. The approach to domain modeling presented in this paper is intended to serve as a basic explanation of a general technique. Our hope is that this paper has made readers more aware of the importance of domain modeling to BDD.

8 References

[1] Solís, Carlos and Wang, Xiaofeng. 2011. "A Study of the Characteristics of Behaviour-Driven Development." *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications.*

[2] North, Dan 2006. "Introducing Behaviour-Driven Development" http://dannorth.net/introducingbdd. (Accessed July 1st, 2013)

[3] Leveson, Nancy G. 2000. "Intent Specifications: An Approach to Building Human-Centered Specifications", *IEEE Transactions on Software Engineering*.

[4] Broy, Manfred. 2013. "Domain Modeling and Domain Engineering: Key Tasks in Requirements Engineering". *Perspectives on the Future of Software Engineering*.

[5] Larkin, Jill and Simon, Herbert. 1987. "Why a Diagram is (Sometimes) Worth a Thousand Words", *Cognitive Science* 11, 65-99.

[6] Winn, William, Li, Tian-Zhu,Schill, Donna. Diagrams as Aids to Problem Solving: Their Role in Facilitating Search and Computation, *Educational Technology Research and Development*. 1991, Volume 39, Issue 1, pp 17-29

[7] ANSI/NISO Z.39.19.2005. "Guidelines for Construction Format, Mgmt of Monolingual Controlled Vocabularies"

[8] Wanderley, Fernando, and Silva De Silveria, Denis. 2012. "A Framework to Diminish the Gap between the Business Specialist and the Software Designer." *Eighth International Conference on the Quality of Information and Communications Technology.*

[9] Carvalho, Silva, and Manhaes. "Mapping Business Process Modeling Constructs to Behavior Driven Development Ubiquitous Language." (arXiv:1006.4892)

[10] Svenonius, Elaine. 2000. *The Intellectual Foundation of Information Organization.* Cambridge, MA: MIT Press

The Far Away BA: Business Analysis Tips For Pulling It All Together Without Being There At All

Jourdan Arenson

arensonj@comcast.net

Abstract

For Business Analysts who help build in-house IT systems at mega-global corporations, system requirements are usually not simply there for the "gathering." Often the Business Analyst (BA) has to lead an expedition of requirement discovery across the continents to drive out the needs of stakeholders in Europe, Asia and the Americas.

If the organization has a chaotic history of mergers and hasty system integration, it is likely business partners and users are overworked and overwhelmed by the complexity of their own systems. The enhancement list may include more projects than the Dev team can possibly look at, much less build. Yet the BA may need to track enough details on all projects in the pipeline, so the Dev team can get working on any one.

In this fragmented, distracted corporate environment somebody needs to organize the vision, keep the focus, root out the issues, and drive the design forward. How can the BA provide this kind leadership without any official management authority? How can the BA stay top of all the dynamic complexity? How can the BA guide a distributed team toward the consensus and collaboration needed for successful, efficient software engineering?

This paper gives observations from a Business Analyst who has faced all these challenges, with a couple of twists not typical of BAs at large corporations: He's worked from home full time for the past 12 years. And, for the past three years, he's worked in an Agile Dev team.

Biography

Before getting into IT, Jourdan worked as a journalist and marketing copywriter. In 1995, he switched to tech writing ("because you never need to lie in tech writing") and got a job documenting foreign exchange trading systems at Bank of America in San Francisco.

After taking a course in Unified Modeling Language, he decided to become a Business Analyst. And after a couple mergers where his business partners and Dev teams were broken up and distributed around the globe, he was able to convince his managers that he could move to Eugene, Oregon and work from home as a Business Analyst.

He's spent the past 12 years on conference calls helping colleagues in the global market centers of Chicago, New York, Charlotte, London and Bangalore build the Bank's web-based foreign exchange trading applications. During those calls, nobody else suspected he was looking out the window at the chickens in his backyard.

Why I Like Engineering

I like the term "software *engineering.*" I think engineering is a wondrous human ability. Engineering combines our talents of social collaboration and technical ingenuity. Humans are good at working together to design and build complex systems to meet our goals.

In an engineering project, people work together to build a system designed to achieve a stated purpose. The system could be a spacecraft, a computer chip, a bridge, or application software for a global corporation. The complexity exists on many levels: requirements management, business algorithms, technical design, logistical coordination, and much more. In the final system, all these diverse elements must function together to meet the stated need.

As a Business Analyst (BA) in software engineering I am on a team that builds systems to meet business goals, but as the BA, I don't have to build the systems, and I don't have to come up with the goals. Instead, I support the Business Team (Biz)—who decides what needs to be built—and I guide the Development Team (Dev)—who decides how to build it. My responsibility is to help the teams articulate and agree on what we are going to build.

Software engineering is about managing complexity and collaboration. As the BA, I want to: 1) make the collaboration as efficient and pleasant as possible, and 2) make the required complexity as clear and simple as possible.

While I work on technically complex systems, managing collaboration is the higher priority for me. This is because I work at a large global corporation where my team members are spread out all over the world. Because team members are not physically together, it can be hard to build good relationships. With my unique role between the two sides, I have the opportunity and duty to foster collaboration.

In this paper, I would like to share some Business Analysis tips I've found on managing the complexity and collaboration of software engineering projects.

Engineering Consensus

Since engineering is about collaboration, how smoothly a software engineering project goes largely depends on the quality of relationships between the teams. When team members are not together in one location, it is easy for relationships to be weak. It is easy to mistrust others. It is easy to avoid problems, point fingers and be defensive.

When the Biz-Dev relationship is poor, there is usually some power differential going on. If the business has the upper hand, the BA and Project Manager (PM) may be openly aggressive, intimidating Dev into agreeing to deliver more software, faster than they can handle.

Other times, the Dev, BA and PM may be passively aggressive, forcing the business to follow excessive bureaucratic process which slows down the rate at which they deliver software.

I have experienced both and I have found that the worst thing I could do was this:

1. Work exclusively with Biz to define all the system requirements in all their complexity.
2. Write it up in a big document.
3. Set up a meeting with Dev and show them the document.
4. Say "Build this."

Whenever I've done that, the Biz-Dev relationship took a dive toward one of those extremes.

The Agile mindset definitely helps, but prior to moving to Agile I followed a principle during requirements gathering that always seemed to result in better relations between the Biz and Dev. The principle is:

> **Build consensus step by step, moving from the general to the specific.**

Let me illustrate this in reverse terms. When I described the worst thing I could do, it was going from the general to specific all on my own, *without* building consensus. I would have all the system requirements in my head or on paper. Even if I had it right—which I usually did not—others would challenge my model, if only because they didn't have input into its evolution. I discovered that it is very difficult to get buy-in on "fully baked" requirements.

The best way to get buy-in is to first establish it at a general level, and then establish it again and again, at each more detailed level. There is often a bit more running around and chasing people down—but it is less time consuming than trying to sell fully-baked requirements because people participate in the evolution of the vision of the final result.

It starts with building consensus on the "general." Let's say we're building a new system (as opposed to enhancing an existing one). Building consensus on the general means getting the Biz and Dev to agree on what is the general scope of the initial release.

Note that this is *my* first goal: the general scope of the *initial* release. It may not be the first goal of others. Often the business wants to communicate a grand vision for the system. So I don't press too hard at first. I document a high-level features list, adding whatever is proposed, but in the back of my mind I'm thinking: "What are we going to build first?"

Eventually I start to press the point and say something like: "While it is good to discuss everything we eventually want to build one day, if we don't agree on what the Dev team should build first…they won't build anything."

Here I take the feature list and use that as the basis for a Scope Document. I keep it pretty high-level: a listing of screens, features, interfaces to outside system, workflow, and user roles. I want to make sure I have every major feature and dependency on the table and up for discussion.

Dev is going to use this to decide what they are comfortable taking on for the first or next release. Biz can use it to determine if the proposed set of features meets their business goals.

Once I have a comprehensive list of features for a release, I press for consensus at this more general level *before* moving to specifics. I want to hear Biz and Dev agree: "Yes, this is what we want to put our arms around in our release."

I don't want to be too rigid in this, because there are invariably features that Biz would like in the next release that will not be built in time. So I always have sections where I note features that are "nice to have for first release" or "to be prioritized for future release." I try my hardest to never say "no" to Biz. Instead, I want to say "yes, for a future release."

Eventually I get consensus at the general level, and move down to build consensus on specifics. I try to start with the feature that has the most risk of being more complicated than everybody thinks. It could be due to missed or poorly defined requirements. Or it could be something that the Biz wants from the system that might be difficult to implement.

In this, I'm looking for features that may impact the "scope consensus" we've established. This is how I try to stay in front of scope creep. It's not that scope creep is always bad, it is often necessary to expand scope to address some unexposed complexity that you *have* to account for. But I try to be the first one who spots it, so I can proactively manage it as pleasantly as possible.

Let's say I uncover a feature with newly exposed complexity. The fact that we previously had consensus on scope helps, because I can loop back to that. I say something like, "Gosh this is not what we had in mind when we agreed on scope. Let's go back and re-confirm our consensus on the scope." The memory that we once agreed seems to make it easier to negotiate scope creep.

In addition to managing scope creep, the Scope Document helps when the team is lost in the weeds of an individual feature. In discussing an individual feature, we may lose track of what the system is trying to accomplish. Going back up to a more general level helps set the context for what details are the most important.

The Scope Doc is also a tool for expectation management. In many cases, business folk don't have a sense of the amount of time and effort needed to build a system, so it can be hard for them to have realistic expectations. This document establishes a fixed point that everybody has agreed will be a useful and realistic accomplishment. If the Dev meets or surpasses what is outlined in the Scope Doc, the business will have their expectations met.

"Build consensus step by step, moving from the general to the specific" works in other situations. It is a great way to organize a meeting agenda and it is a good idea to have in the back of your mind as you run a meeting. I admit it is a fairly obvious, common sense principle, but I have found that if I use this principle when I get rushed or flustered it makes it clearer where and how I should be guiding the team.

Advocate for the Business Team

As mentioned, poor Biz/Dev relationships stem from a power differential. When Biz has the upper hand, Dev can be forced to take on more than it can handle. And sometimes Dev can use unnecessarily complex processes to fight back. It is the role of the BA to advocate for each side. Let's begin with the role of advocate for Biz.

The best thing I can do for my business partners is to elicit reasonable, buildable, coherent, comprehensive, system requirements that will efficiently meet their business needs.

- "Buildable" means that what they are asking for has a reasonable chance of being built, given the resources and the present condition of the system they are enhancing.
- "Coherent" means that there is a logical consistency to the requirements. I want to show that the system can work on paper, and there are no logic flaws, contradictions or conflicting requirements.
- "Comprehensive" means that we are asking for everything that needs to be done. This is often the hard part. Biz can easily give requirements for that sweet feature that they want, but they forget the boring supporting features that are needed to achieve them such as, user administration or static data entry functions.
- "Efficiently meeting business needs" means that we specify a system that meets their needs and can be built in the most cost effective way: not too much and not too little.

If it all comes together well, I want to help Biz present the Dev team with elegant requirements. We want the Dev team to look at the requirements and think: "That's cool, we want to build this."

I try to do this with all business partners, and over the years I have had great ones and I have had terrible ones. The good ones know their systems, understand the level of effort for enhancements and are reasonable negotiators. My business partner over the last couple years has been the best, and I am very grateful.

Not everything is perfect, however. We sometimes have "off-site" requirement conferences where lots of stakeholders gather in a conference room and lock themselves in for multi-day requirements sessions. Even if all the folks are smart and reasonable, people from different departments with different responsibilities will have competing priorities, and the discussion can often get heated. The principle for this situation is:

Wait for all the hot air to condense, show what's left, and build on that

I don't mean to imply that everything my colleagues say is hot air, but a lot of hot air gets blown around in those off-sites. The principle means that I don't try to force all comments into structured format. I certainly do *not* type "1.2.19.1.10 The system shall…" every time somebody says something.

The hot air has to blow and it may be chaotic at times. I let them talk and talk and talk—maybe even the entire first day—but when the time seems right I try to begin the process of condensation. How does it condense? Hot air will always condense under the pressure of having to specify pertinent details.

It's hard to describe when to intervene, but experience helps. There are times when I can tell the conversation is going in circles because there is no clarity or consensus on one particular aspect of the system. I can tell that if folks defined and agreed on the specifics of that part of the design, they'd have a more solid basis on which to continue discussion.

This is the time to pull out a BA modeling technique from the toolbox, something unambiguous that will crystallize and distill the aspect of the system everyone needs to agree on.

Let's say the team is talking about a workflow where different roles have different functions. Most folks will be focused on defending the processing responsibilities of their own group. The conversation bogs down or starts going in circles, and I realize that discussion will improve if we agree on the role names and the logic of who can have what roles.

I ask folks to help me mock up the user entitlement screen that will be built to assign roles. We write out the role names and draw in check boxes and radio buttons to illustrate the logic of what roles one person can have. It's not about designing a screen, but about agreeing on common terms for the roles and the logic for assigning them.

With this crude mockup of the screen on the whiteboard, we now have a concrete hypothesis that can be tested as we continue to discuss who does what in the workflow. It provides a tentative consensus on some specific details, and anchors the conversation to those details. "OK, if these are the roles we have, then how would the workflow go?" Maybe the original hypothesis on the roles is all wrong but at one point we agreed on them, and we can go back together and agree on how they have to change.

The BA modeling techniques I use most for this are simple things I can draw on a white board: screen mockups, business domain class diagrams, workflow diagrams, and sequence diagrams. Sometimes I only need to model one aspect and get consensus on that to advance the discussion. Often this is a critical but mundane aspect of the system, such as agreeing on the user role names.

Personally, I feel I provide more value to my business partners this way, as compared to mindlessly writing down everything everybody says like a stenographer.

I mentioned some bad business partners. The worst was one who didn't understand the system he had, would give me requirements one day, completely forget about them the next, and was the kind of negotiator who would literally pound his fist on the table in IT planning meetings. (I do mean literally. I was on the conference call at home and could hear it pounding on the table over the speaker phone.)

But there's another business partner that can be more dangerous than the fist-pounding idiot who everybody knows is an idiot. This is the business partner who isn't as smart as he thinks he is, but still has it right half the time. This is a dangerous combination. A person like this is wrong half the time, but he thinks he is right *all* the time.

If they are ambitious, people like this can be especially dangerous, because they can be very convincing to senior managers, even though what they are saying can be dead wrong.

With this kind of person, it's best to make it look like everything is their idea, and every idea is a good idea…until they themselves see it is a bad idea. Maybe they have a requirement that is logically impossible, will lead to the users into a dead-end in the system, or will cause a huge data maintenance nightmare.

I let them explain it me, and then I model it on my screen or on a whiteboard so they can see what would happen if we were to actually build that. They have to see for themselves that the program they are asking for will not give them the output they want. They have to see how garbage going in leads to garbage coming out. When they see that, they will change their mind.

It can take a lot of patience and extra work. But it keeps the focus on collaboration and the relationship on good terms, so it is worth it.

Advocate for Development Team

I have always considered myself more of a member of the Dev team and identify with them more than the business or project management side. I think I have had very good relations with my Dev partners. The few exceptions were when I was under the thumb of a nasty Project Manager who acted as a drill sergeant and used me to try to beat up or get evidence against the Dev team. But even in a hostile political environment, the BA has so many ways to help Dev be successful that they generally have seen me as an ally. Here are some tips I've learned over the years.

Learn the level of prep your lead developer wants

The first time I work with a lead developer I spend time questioning him on the level of detail they want to see from me. I do this *before* I spend too much time preparing anything. It's surprising how much variation there can be among developers on this point.

For example, one lead developer looked at my requirements doc and turned me away because I had "already designed the system." He said: "You need to tell me the 'what' not the 'how.'"

Later on another project with a different lead developer, I was turned away because I had *not* designed the system, at least to the level that he wanted. He said: "You need to go back and find out what they want."

In the first case, the developer was something of know-it-all control freak. He didn't think anybody should make decisions besides him. In the second case, the developer was simply over-worked. He didn't have time to take on the project and so he was sending me away to do more leg work. In both cases I could have had a talk with them to figure what level of prep they wanted, before going to them with any specification. Once we established this, things went fine.

Like a tourist in a foreign land, learn enough programming language to get by

As a BA, I sometimes feel like a tourist who travels between two foreign countries: the land of business and the land of developers. I have always been interested in foreign languages, so I've found it fun, interesting and rewarding to learn about programming languages.

Note that I didn't say "to learn a programming language," I said "learn *about* programming languages." I never expect to write production code, but—like a knowledgeable tourist—I have learned enough to make my life easier, and to communicate more precisely with the developers in their language. I bought a Java book to learn about the basics of object oriented-programming: how classes have attributes and methods, how methods take in some data, process it, and return some other data. I learned the basics of relational databases and SQL: tables, columns, rows, views, and queries. I have learned enough about Perl and Visual Basic to allow me to search the internet and find little programs to help me automate tasks. I can't write the programs from scratch, but I know how to steal somebody else's code and make it work for me.

This has given me confidence when talking with developers about their production code. When developers talk with me, they know they don't have to go into programming basics to explain what they are doing. And when I give them requirements, I have a good idea of what they face when coding. This has helped me advocate for Dev while working with business partners on requirements. I have a better sense of what Dev has to do to build a feature, what is hard, what is easy, and what is impossible.

Now if I think something is easy I'm careful not to promise. I'll say: "I'm not a software engineer, but I think that should be easy." If something seems hard or impossible I can help Dev by setting business expectations.

At the same time, like a savvy tourist who knows a little bit of the language, I won't be fooled so easily. If I find a developer giving huge estimates for something I recognize as a trivial coding change, I can stand up to them with some confidence.

Plus, learning about programming is fun. There is a pleasure to reading simple, elegant code. A programming task is like a brain teaser. In most cases, I could never solve the problem myself but it's cool to see how somebody else solved the problem with a few clever lines of code.

Faire La Mise En Place

Mise en place is a technique that comes from the culinary arts. It translates to "put in place." I've found developers love it if I can help set the *mise en place* for their coding.

Julia Child explains: "all well-trained chefs prepare their *mise en place,* the professional culinary term for setting all the food and equipment needed out on the work space and prepping them before beginning to cook. It is absolutely necessary in the fast and hectic atmosphere of a restaurant kitchen…When everything is there, it makes cooking so much faster and easier—and you don't forget to add any special ingredients" (Child 1995, 278).

In a software engineering context we might say: "When everything is there, it makes *coding* so much faster and easier—and you don't forget to add any special *requirements.*"

As a BA, I am always on the look-out for tasks I can do to set the developer's *mise en place.* Generally, these are not the normal requirements artifacts, but more specific analysis tasks that I can see the developer would have to do before actually writing code.

A recent example was a requirement for our system to automatically display a bank's address when a client typed in an account number at that bank. The bank could be almost anywhere in the world. This data was provided by a vendor via giant fixed-width text files. The vendor's documentation was poor, but the information we needed was in there.

I could have left the requirement at that and let the poor developer figure out where the data was, what needed to be loaded into our database, and how to retrieve individual addresses. Instead, I sifted through the documentation and made an Excel file with annotated rows and columns that illustrated how the data was to be mapped. This was like washing, peeling and dicing all the onions, carrots and potatoes. The Dev could have done it himself, but with the annotated Excel file, he didn't have to. He could start cooking (coding) right away.

Agile means never having to say you're certain

At my job, I've been lucky to be in a group doing Agile development processes. Now, some may say that a huge bank could never be hip enough to do *real* Agile development. That may be true; we may not be doing real Agile, but whatever we are doing has made my team quicker and more responsive. Compared to other Dev teams in the bank that are tied down by grueling bureaucratic processes, our team delivers faster, more accurately, and with a fraction of the people.

The difference is obvious to everyone who works with us. One very senior (but not very tech-savvy) business executive asked if our team could develop a major enhancement to a high-profile online banking application owned by a completely different development team in a different division of the bank. My developers had no access to this application's development environments or code repository. There was no way we could change or do anything to this application. But this senior executive asked if my Dev team could "build the enhancement and then send it to him."

While I'm a big fan of Agile, I'm not an expert on Agile, so I'm not going to talk about techniques. Instead, I would like to highlight how Agile supports Biz-Dev collaboration and avoids the confrontation that often comes from a waterfall process.

First, let's look at how the waterfall process often leads to confrontation. It begins with what you might call "The "Waterfall" Conversation."

> **Business Partner:** *(After working with BA to draft massive requirements document)* This business requirements document represents what I want you to build.
>
> **Dev Team:** *(After filling in project plan)* This project schedule promises when I will deliver it.

Typically, the requirements document and project plan are seen as "contractual obligations" to which all parties "must be held accountable." But in reality, they are simply predictions about what needs to be built and how long it will take to build it.

If both sides made excellent predictions on what should be built and how long it will take, there's no problem. But too often the predictions are faulty and we move into the "finger-pointing" phase of the project:

> **Business Partner:** "It's taking longer than you said!"
>
> **Dev Team:** "Yeah, well, your requirements were not complete. You didn't tell me about everything I needed to build!"

The key difference with an Agile process is that there is no expectation that anybody will make excellent predictions. Rather, Agile processes embrace the *biological fact* that the human mind truly sucks when it comes to predicting how long it will take to complete a challenging and complex project, like a software engineering project (Kahneman, 2011).

I say "biological fact," because in the last decade an emerging scientific understanding of the human mind has shown that our "intuitive thinking" very naturally leads us to underestimate complexity.

Psychologist Daniel Kahneman's excellent book *Thinking, Slow and Fast* is one of the best compendiums of the scientific research on this subject. It's not a management book, but describes the latest scientific consensus on how the mind works—or in some cases doesn't work the way we think it should.

The waterfall process is often brought down by a mental glitch Kahneman calls the "planning fallacy." The planning fallacy results from a "cognitive bias" to forecast "based only on the information in front of us—what you see is all there is." (Kahneman 2011, 247).

In other words, we assume "a best-case scenario [when] there are too many ways for the plan to fail, and [we] cannot foresee them all." (Kahneman 2011, 254).

The science suggests we are naturally bad at planning complex tasks. And we are naturally blind to the "unknown unknowns." On top of this is our tendency to bolster ourselves with feelings of confidence at the beginning of projects when the devil in the details is still unknown. Kahneman says:

> "Neither the quantity nor the quality of the evidence counts for much in subjective confidence. The confidence that individuals have in their beliefs depends mostly on the quality of the story they can tell about what they see, even if they see little. We often fail to account for the possibility that evidence that should be critical to our judgment is missing—what we see is all there is. Furthermore, our associative system tends to settle on a coherent pattern of activation and suppresses doubt and ambiguity." (Kahneman 2011, 87).

The difference with Agile is that we are frank about our cognitive weakness in predicting what is needed or how long it will take. The Agile mindset focuses on providing the highest value features as early as possible, and assumes that any predictions about the future are likely incorrect. Let's look what you might call "The Agile Conversation" and see how it avoids certainty and over-confidence.

> **Business Partner**: (*After brainstorming requirements on whiteboard) "*This is the kind of thing I need you to build. Features A, B and C are the most important. D, E and F are still important but they can come later."
>
> **Dev Team**: "That's cool. We'll spend the next iteration building A. After that we can see whether you want to change your mind.
>
> **Business Partner**: "Cool."

In the Agile conversation folks don't speak with any certainty about requirements or schedules. More importantly, they don't demand certainty from the other side. However, there is great confidence that they can *deal* with uncertainty, and it really has worked this way for our team. We no longer waste time preparing for the "finger-pointing" phase with the business. We don't pretend we know exactly how things will go, but we do know we are agile enough to deal with it. And when problems come up, we don't confront each other like opposing lawyers. Instead, we keep on collaborating.

Advocate for Myself

I work from home, so I rarely see people face-to-face. Nobody reports to me, so I have no management authority. Somehow, I need to guide and influence my colleagues toward pleasant collaboration around staggering complexity for multiple projects, all running at the same time.

The only way to pull this off is to maintain a certain kind of reputation. I need my colleagues to see me as the guy who:

- Is fluent in the details of the required complexity
- Raises, tracks and brings issues to resolution
- Updates documents on a timely basis
- Is methodical and comprehensive while remaining *relaxed*

When coordinating lots of people, requirements and tasks, there's a tendency to go a little manic, a tendency to impress others with the amount of effort needed to stay on top of things. I try to avoid this. A BA can best serve as the calm face of the Dev team. As spokesman for the Dev team, the message I want to send is: “We can do this. But we are going to do it right. We are not going to be hasty and frantic. We are going to be methodical and comprehensive.”

The key to giving this impression is to demonstrate that I can stay on top of all the changing complexity without getting stressed about it. A book that helped me pull this off is David Allen’s *Getting Things Done: The Art of Stress-Free Productivity.*

Allen offers a fairly simple system for tracking and prioritizing all the things you have to do in life, be it software engineering or anything else. His system is platform agnostic. It can be implemented on paper or electronically, using any software that makes sense. The goal is to capture and process details in an external system, so you don’t have to think so much about it.

Allen calls it a “logical and trusted system outside your head and off your mind” (Allen 2001, 3). He goes on to explain the impact of such a system on your reputation: “When people with whom you interact notice that without fail you receive, process and organize in an airtight manner the exchanges and agreements they have with you, they begin to trust you in a unique way.” (Allen 2001, 225)

I have followed Allen’s advice to implement my own version of his tracking system. As a result, I’m confident I will follow through on what I have to do, and because I don’t have to *consciously* remember the details, I do not worry about what I might be forgetting, which helps, especially when tracking multiple projects.

An even greater benefit (one that Allen doesn’t mention in his book) is that while those details are off my *conscious* mind, they are sorting themselves in my *unconscious* mind. This may sounds like new-age hooey but it is backed up by recent research in neuroscience.

David Eagleman’s book, *Incognito: The Secret Lives Of The Brain* explains what this research has shown: “The first thing we learn from studying our own circuitry is a simple lesson: most of what we do and think and feel is not under our conscious control. The vast jungles of neurons operate their own programs. The conscious you — the *I* that flickers to life when you wake up in the morning — is the smallest bit of what's transpiring in your brain” (Eagleman 2011, 4).

While the conscious mind plays the smaller role, it is an ego-maniac, taking credit for the work done by the subconscious. Eagleman explains that when you suddenly see a solution to a problem, “your brain performed an enormous amount of work before your moment of genius struck. When an idea is served up from behind the scenes, your neural circuitry has been working on it for hours or days or years, consolidating information and trying out new combinations. But you take credit without further wonderment at the vast, hidden machinery behind the scenes” (Eagleman 2011, 7).

This underscores the benefit of “keeping the details off my mind” in a new way. I track all the details in my external system, so my conscious mind doesn’t worry so much, but I also do it to prime my unconscious so it can carry on with these details while I go on to the next thing.

The key to the system is to express details in the clearest, most precise language I can muster. These are, in a sense, notes to my unconscious. They may be disconnected. I may not see how they should be organized, and I may not see the solution yet. I don’t worry about that. I just want to explain to myself the issue that needs to be addressed.

It is important to do this in real time, as I discover the details. I do not let them pile up, because that puts too much pressure on the conscious mind to remember them. When a new task, requirement or issue comes up, I find the appropriate place to document it as soon as I can. I may spend time re-reading the old details, and muse about a new organizing principle. I toy with it a little to seed the problem in my unconscious, but then I try to forget about it and move on to the next thing.

This method works for major deliverables, like requirements documents, and it also works for smaller project tasks, like writing emails or meeting agendas. I take the opportunity to lay out the problem or action as clearly as possible when I’m thinking about it, which is when I’m drafting the email or scheduling the meeting. This allows me to forget about the action so I can move on to the next thing, and gives my unconscious something to chew on.

People may not read these emails or agendas carefully, but that doesn't matter, because I am doing this partly for myself. When the issue or meeting comes up, the details are clearly documented, so I quickly remember what we need to address. And sometimes, I find an insight or solution I didn't see before.

One way I remind myself to do this is to pretend I am leaving for vacation the next day. When people go on vacation, they usually think: "I know I won't remember all this stuff after I get back from vacation, so I better write it down before I go."

I try to do this every step of the way. Whenever a pertinent detail comes up I think to myself: "I don't want to have to remember this, so I better capture it as clearly as possible."

Then, at the end of the day, I go on vacation until the next morning. But I keep a note pad by my bed, because I often get some pretty good ideas when I'm not thinking about work.

Conclusion

If you are a BA, think about actively managing your reputation with the Biz and Dev teams. You want to have a strong reputation as a good advocate for both sides.

If you are a developer, let your BAs know what level of detail you want from them *before* they go to work. Understand that they *must* advocate for the Business team, and use them to keep collaboration as effective and pleasant as possible.

If you work on software engineering projects, take a moment now and then to appreciate your team's ability to collectively manage complexity. Marvel at the complexity of the social interactions that somehow come together to define and build the complexity of IT systems. It may seem like a mess much of the time, but it is a wonder we can do it at all.

References

Allen, David. 2001. *Getting Things Done: The Art of Stress-Free Productivity.* New York: Penguin Books.

Child, Julia. 1995. *In Julia's Kitchen with Master Chefs*. New York: Alfred A. Knopf.

Eagleman, David. 2011. *Incognito: The Secret Lives of the Brain.* New York: Pantheon Books.

Kahneman, Daniel. 2011. *Thinking, Fast and Slow.* New York: Farrar, Straus and Giroux.

The Many Faces of a Software Engineer in a Research Community

Cristina Marinovici, Harold Kirkham

Pacific Northwest National Laboratory
Electricity Infrastructure
Richland, WA

cristina.marinovici@pnnl.gov, harold.kirkham@pnnl.gov

Abstract

The ability to gather, analyze and make decisions based on real world data is changing nearly every field of human endeavor. These changes are particularly challenging for software engineers working in a scientific community, designing and developing large, complex systems. To avoid the creation of a communications gap (almost a language barrier), the software engineers should possess an 'adaptive' skill.

In the science and engineering research community, the software engineers must be responsible for more than creating mechanisms for storing and analyzing data. They must also develop a fundamental scientific and engineering understanding of the data. This paper looks at the many faces that a software engineer should have: developer, domain expert, business analyst, security expert, project manager, tester, user experience professional, etc. In authors' opinion, the conclusions drawn while developing a power-systems scientific software could be extended to describe the process of any software development project, and thus guide all current and future developers.

Biography

***Cristina Marinovici** joined Pacific Northwest National Laboratory as a software engineer in August 2010. Before joining PNNL, she worked as software engineer at IBM, OR. She is responsible for the software testing and validation of projects modeling wind generation integration, power system operation and smart grid. She has a Ph.D. in Applied Mathematics from Louisiana State University, Baton Rouge, LA. She is a member of the IEEE, IEEE WIE, SWE and SIAM.*

***Harold Kirkham** worked at American Electric Power, NASA's Jet Propulsion Laboratory in Pasadena, CA, and in 2009 he joined the Pacific Northwest National Laboratory, where he is now engaged in research on power systems. His research interests include both power and measurements. He is a past chair of the PES Instrumentation and Measurements Committee, and a Fellow of IEEE. He received the PhD degree from Drexel University, Philadelphia, PA.*

1. Introduction

In the scientific and engineering research community, there are two groups of people: scientists and/or engineers who model the real word problem, and software developers who write software. Each group has its own way of operating and developing products, and finding the common ground represents a difficult task. Usually, while working on common tasks, the two culture clash. Each tries to impose its culture on the other: software engineers try to impose a “traditional” software engineering culture on scientists (for example by using change management procedures), while the scientist try to impose their culture on the software engineers (by deliberately avoiding such methods). In order to have finalized and successful products, a common ground and common strategies should be found and applied.

For a software engineer working in a scientific community the job required skills extend beyond the regular job description posted by human resources (HR). The posted job description covers the standard skills required from the employee (the standard coding and software engineering skills), but cannot cover all of the skills that define the ‘adaptability’ of the software engineer. In scientific development communities, the developer should possess communication skills, flexibility to work with diverse software and hardware platforms in interdisciplinary teams, know-how to deal with ambiguities, continuously changing requirements and, last but not least, some domain knowledge.

It takes a fair amount of time and practice to grasp the concepts and understand the technical terminology of any scientific field. One of the factors that make a scientific computing development different from the commercial software developments is the complexity of the domain. There are intricacies in problem formulation, and maybe dependencies on the environmental constraints. Even more, the commercial software is based on common sense and has roots in day-to-day life. In the scientific research environment it may be fairly straightforward to implement a weather gathering interface, but will not be obvious what is necessary for the implementation of an economic dispatch model.

2. Skill set

As in any other profession, in software engineering it is necessary to poses diverse skill sets and to continuously expand them. The standard skill set is attainable through traditional curriculum training, or through some more practical matters as research or projects work. The skills worth mentioning from this set include:

- Any software engineer should be able to write code in an engineered way, to account for the future needs of the provided solution and to assure flexibility of the final product.

- It is desired from software engineers to be able use quality assurance (QA) and software testing skills and techniques. The shifts toward customer/user friendliness are changing the role of the software engineer, forcing him to be also the tester and the user of its product.

- The use of scripting languages represents one of the important skills. As John Ousterhout mentioned in his article, “scripting languages are designed for ‘gluing’ applications; they use typeless approaches to achieve a higher level of programming and more rapid application development than system programming languages. Increases in computer speed and changes in the application mix are making scripting languages more and more important for applications of the future.” (Ousterhout, 1998).

- The ability to process and clean data is another important skill. The operation of cleaning real-life data it is not an easy process, and it is the most important step since incorrect or inconsistent data can lead to false conclusions. Data cleansing is an iterative, expensive and time-consuming process.

- The software engineer is required to have the ability to apply numerical linear algebra, computational and statistical methods to model data in order to analyze and interpret the real life collected data.

- In recent years, the software engineer should scientifically understand the results of the analyzed data and employ these results in the decision making process.

To thrive in the scientific software development world, any software developer needs to learn continuously. This learning may come from formal degree programs with supplementary courses, or in seeking certifications in in-demand technologies. Lately, evolving your skill in response to the market changes will require blending of business and leadership courses with IT training. As developers are becoming more important to the business, they should be able to interact more with the customers and influence the way in which business value is delivered.

The science and engineering research community does not abide by the same criteria as the software engineering community. The rawness of software engineering and the complexity of scientific problems make collaboration between software engineers and scientists difficult. Typically, scientists' formal education in software development is limited; their knowledge of software development has been gathered informally from books, websites, from their colleagues, and often from the working code-sources they have encountered (Segal, 2011). Frustration due to miscommunication and misunderstanding lead scientists to perform the software design and development in isolation, writing clever code for specific architecture and forsaking the slower, more methodical approach to development favored by the software engineers. For scientists the emphasis may be on creating papers. Before publication, papers are peer reviewed and thoroughly scrutinized, but the review does not extend to the software used. The software can escape a lot of testing, and does not need to have flexibility to apply to other problems.

The goal is to unify the two cultures. The hope is to move scientific software development towards methodologies and techniques that have proved valuable for the software engineering community. The way to go is to impose that culture on the scientists, rather than the other way round.

Scientists are not generally early adopters of the software engineering techniques, They want proof that adoption of new methodologies will do no harm to their scientific product. It is desired to conduct the scientific development process in an iterative way, to work in small increments, and to define work-load for shorter iteration. Straight application of technological methodologies (i.e., Agile, lean, XP) may not be suitable for the scientific software developments, but modifying them to tailor every project needs may improve productivity and quality of the delivered products. Application of Agile methodologies, pair programming and creation of an open communication environment help one of the power-system projects in which the author worked on to achieve a usable and deliverable phase (Cristina Marinovici, HICSS 2013).

In scientific research communities, the work is partitioned per projects and pools of money. All the team members (scientists, engineers, software developers, analysts, testers or management) are paid from the same pool of money. The work will be carried on until the task is finished or the financial support runs out, whichever comes first. In this milieu, another good skill to have for a software engineer is the ability to estimate accurately. People are afraid to make estimations, since they are perceived as a prediction of the amount of effort and time required to deliver something specific. It is true that every estimate has risk, but providing accurate estimates helps management. They need good estimates for financial planning.

3. Domain Knowledge

Software development for scientific communities is challenging. Domain knowledge is not optional; it is essential. Software engineers who possess domain knowledge are valued. It is not easy to gain knowledge in scientific fields and to be able to "speak" the same language as the scientists and engineers. In order for software engineers to gain and grow their domain knowledge, a lot of practice and experience on dedicated projects is required, and continuous question asking is involved. Developers need to learn how to see through an end- user's eyes and how to employ the gathered information in the correct way. As it can be seen, the gain of domain knowledge is based on communication between the interdisciplinary team members. Many projects fail due to lack of proper communication between the scientists and developers.

This sort of difficulty has ancient roots. James Clerk Maxwell wrote in an 1877 report to Cambridge University (Maxwell, 1877): "It has been felt that experimental investigations were carried on at a disadvantage in Cambridge because the apparatus had to be constructed in London. The experimenter had only occasional opportunities of seeing the instrument maker, and was perhaps not fully acquainted with the resources of the workshop, so that his instructions were imperfectly understood by the workman. On the other hand the workman had no opportunity of seeing the apparatus at work, so that any improvements in construction which his practical skill might suggest were either lost or misdirected. During the present term a skilled workman has been employed in the Laboratory, and has already greatly improved the efficiency of several pieces of apparatus." Maxwell described a two-way communication gap similar in essence to those we still have. These days, our "instrument makers" are software developers, but the effect of the communication gap is similar.

There is, between the user and the software developer, an excellent chance of a failure to communicate. Here is an example reported at a meeting of the IEEE Power and Energy Society recently (Berrisford, 2012). A power company supplying a large consumer (in Canada) changed out the electric meter. Afterwards, the consumer complained that the electric bill had increased, but they asserted that the load had not. The puzzle was that both the new meter and the old were digital, and there was no reason to think there was anything wrong with either. Indeed, investigation showed that both meters were performing accurately. It is a reasonable question to ask how this could be. In fact, the bill had increased not because the customer was being billed for greater energy consumption, the bill increased because the installation of the newer meter resulted in a penalty for the customer based on the power factor. The two meters did different calculations that (one would think) would give the same result, since both were (ostensibly) measuring the same thing: the power factor. The result of the measurement would have been the same had the waveforms been sinusoidal, but they were not. Distortion caused the difference.

The scientific field that the authors work in is power systems. Getting familiarized with the terminology and the physical meaning of the concepts is not simple. In many cases, one understand the words, but does not understand the engineering or scientific meaning. During one of our scientific software development projects, it became clear to the project team (the term project team is intended to means the whole group, including the developers, testers and power-system engineers), that there were communication problems. Team members did not have a common vocabulary and did not have a shared perspective on the software development process. Hence, misunderstandings arose related to the software's feature set, priorities, architecture and structural requirements. These misunderstandings caused several false starts, and unproductive project tasks. Clearly, team internal communication affected the productivity of the project and the knowledge exchange process was reduced.

In Agile methodologies, the willingness of the team to assume a more cross-functional role when it comes to testing the built product is vital (Lee Henson, 2012). In our scientific development project, adoption of appropriate methodologies improved the team dynamic and its communication: the power engineers worked together with the developers and testers to properly translate the requirements into software. This type of knowledge sharing can be viewed as a way of saving everyone's time. Migration from the initial system to the second one also helped individual knowledge to be transferred to the whole group.

In many cases, the power-systems engineers discussed the problem to be modeled with the team, helping the selection of the optimal approach. Since the designed simulator was solving problems that could not be verified by hand, in many cases the system testing required pairing with a power system engineer. For example, for a system with G generators considered as ways to meet the load in T time steps, there are $(2^G)^T$ combinations to examine. For any realistic situation, that is a large number; even 5 generators and 24 hours (a trivial system size) generates more than 10^{36} combinations. Also, there are many constraints that must be satisfied in the same time, and their combined effect cannot be easily predicted (M.C. Marinovici, ISGT 2013).

Analyzing the output required a lot of domain knowledge (i.e., power systems, optimization, distributed system behavior modeling, etc.), knowledge that only experienced engineers could have. In many cases little was known about the proper and desired behavior of the software product and the well defined software testing and software quality strategies weren't enough. At that point, exploratory testing was performed. It helped us to define the next steps in testing and go beyond the obvious. Simultaneous learning, test design and test execution were incorporated in our testing strategy. This approach revealed gaps in the implementation that weren't evident from visual-inspection of the results and graphs.

4. Developing 'Adaptive' Systems

The need to embrace change becomes an accepted fact of life in software development environments. Any software project is subjected to continuous changes as it supports evolving user needs. Looking at the motivations behind the software changes during our software simulator life cycle, the following can be identified:

- Dynamic marketplace - changes required because of company changes, such as management changes or restructuring;
- Variety of environments - changes required to meet some external environment or outside-the-organization needs;
- Technological evolution - required to meet new technical demands and learning changes (due to gained knowledge as a result of individuals or group learning) (Cristina Marinovici, HICSS 2013).

The above changes also determine different perceptions of the system's value. With so many ruling factors involved in the software development lifecycle, the goal of system design is not robust systems, but rather the delivery of value to system stakeholders. In order to maintain value delivery over a system lifecycle, "either the system must change, or at least the perception of the system must change, in order to match new decision maker expectations" (Adam Ross, 2008).

As a team develops the system architecture, they must consider the inevitability of change in the future system. Robustness of the software can be added later on, in the development stages – in the design or implementation phase. In the development stages, a must have feature of the developed system is modularity. The software modularity may reduce change cost, though perhaps at a higher initial cost. Due to knowledge gained as a result of individual or group learning, a second system development may occur. This represents an improved and better engineered version of the old system. Usually, once the development of the smarter system started, many fixes and improvements, not included in the old system will be forced in, and the chain of changes will continue.

Embracing the changes and not fighting them will help progress, and will increase team competence and ability to assure product quality. Making mistakes is human, but learning from them will help us progress. The team will gain experience in assessing risk and preventing failures by thinking ahead in the development process. As system complexity increases, risks grow proportionally, but some of the failure cases can be caught earlier if knowledge is made available.

Most of the software products, developed for scientific research communities, are embedded in a network of complex dependencies (i.e. inter-products, inter-services and inter-suppliers). In this type of architecture, small changes in the system's input can produce unpredictable changes in the overall system behavior. Therefore, the software design problem becomes one of designing an 'ecosystem' where software components, project artifacts (i.e. design documents, requirements documents, bug tracking reports, schedules, etc.) and human resources are considered agents. The development of engineering models and methods for assembling software systems that can dynamically adapt to context and account for themselves is a difficult task and should be treated on project by project basis.

5. Roles and Responsibilities

As software practitioners, understanding of the roles and responsibilities within the team framework can affect the team productivity and communication. Trying to develop a solution outside the carefully monitored framework may lead to disaster. In the scientific research projects staffed by multi-disciplinary teams the roles and attributions are a bit different, displaying deviation from the standard. The team is governed by a sense of product ownership – every team member contributes creatively and has some control over the final product. All the efforts should be focused towards maximizing customer value.

The success of any team is based on communication, team-feedback, information exchange, cooperation, and self-organization. In our scientific development project, the team grown into a great organic team that is self-organized, established and assured an environment of good communication:

- The software engineer was not only the developer, but also the business analyst, the tester, the architect. Real life data are often noisy, corrupted by error, or awkwardly formatted. To work with these data, the software engineer needs not only to translate data from one format to another, but to do some amount of cleaning (or scrubbing) before usage. In many cases, the software engineer was also the product advocate. Incorporating customers' and teammates' feedback in the development process allowed the team to gain experience in assessing risk and preventing failures. By becoming the product user, the software engineer redefined the friendliness of the product, made it more intuitive and easy to use.

- The scientist was not only the field expert, but also the tester, the end-user and the customer. By the decisions that the scientist was making, his role was more of a lead. The scientist paired with the software team and helped in the modeling, optimization and validation of the software. This type of knowledge sharing can be viewed as a way of saving other's time and a direct way of improving software quality. The quality of the code improved, as a result of multiple peer reviews, and corner-cases testing.

- The manager was not only the manager, but the person who kept the team focused, who communicated a vision between the team and the customers. Management also determined metrics around value in the form of test coverage, team velocity, Agile adoption assessment, defect counts, and definition of done and so on. In this effort, the management tried, but was not always successful in avoiding the project management traps.

The size of the team is also important. In our scientific project development, the team has a small size, and adoption of new methodologies did not constitute a draw-back factor. There was a point in the product life-cycle when management tried to speed up development by adding more manpower. It is well known that a bigger team will require more channels of communication; people need time to get up to speed and their productivity is different from person to person. There is no linear dependency between the number of workers and their productivity. In the case of this simulator, the increased manpower represented a "surge," and the long-term picture was one of decreasing manpower.

Not in every project the application of standard methods and methodologies will work. Trying to find a better fit between a team and the methods that they are adopting is a good thing to consider. The team might experiment in adjusting the methods until they find what works best for them; there is no pre-can methodology to apply for every project. At this point in the product life-cycle, the knowledge gained can be utilized to improve the project's state of art, practice, and the team's education and training.

In any project, scientific or not, the adoption of dynamic methodologies gives team members the opportunity to experience different roles (scrum master, tester, user, developer, analyst, architect or manager). This change in roles provides a better perspective and understanding of the task(s), as well as a better estimation of the work-effort. In the multidisciplinary team the *us versus them* mentality is a fact of life, and therefore adoption of methodologies that provide one with the chance to experience work from the point of view of the other group is very valuable.

In our project, adoption of Agile methodologies and methods created an environment of camaraderie, increased team's respect on each other work and provided a better understanding of the difficulties and project hurdles. The learning process was bi-directional – scientists were learning from software engineers and vice versa. All the team members were familiar with the state and the direction of the project.

Good communication between the team members transformed the software engineers into product advocates, participating in discussion and demonstration of the product to clients and users. Usage of version control tools made the software product easy to deliver to interested parties with different modules incorporated. In this way, the team scientific staff has been convinced that reusing existing frameworks save time and effort, and it is more efficient than building their own from scratch. Therefore, for the whole project team, it was clear that understanding and implementing software engineering principles and other best practices is just as important as specialized knowledge.

6. Conclusions

Scientific software systems are growing larger, and their level of complexity is increasing. Software implementation of such systems requires proper software engineering and collaboration between scientists, engineers, developers and customers. Interaction among the computational-science and software engineering communities becomes more visible as software engineers become product advocates, participating in discussion and demonstration with clients and users. More dialogue and studies will consolidate this process. The two cultures have a lot to learn from each other.

References

Book:

Segal, Judith A. and Morris, Chris (2011), “Developing software for a scientific community: some challenges and solutions”, In: Leng, Joanna and Sharrock, Wes eds. Handbook of Research on Computational Science and Engineering: Theory and Practice. USA: IGI Global, pp. 177–196.

Journal Articles:

John Ousterhout, “Scripting: Higher Level Programming for the 21st Century”, IEEE COMPUTER, 1998

Cristina Marinovici, Harold Kirkham, Kevin Glass, and Leif Carlsen, “Engineering Quality while Embracing Change: Lessons Learned”, 46th Hawaii International Conference on System Sciences (HICSS), 2013

Berrisford, A.J., “Smart Meters should be smarter”, IEEE Power and Energy Society General Meeting, 2012 Digital Object Identifier: 10.1109/PESGM.2012.6345146

M.C. Marinovici, H. Kirkham, K.A. Glass, L.C. Carlsen, “Modeling Power System Operation with Intermittent Resources”, IEEE Innovative Smart Grid Technologies Conference, 2013

Adam M. Ross, Donna H. Rhodes, and Daniel E. Hastings, “Defining Changeability: Reconciling Flexibility, Adaptability, Scalability, Modifiability, and Robustness for Maintaining System Lifecycle Value”, Systems Engineering, 2008

Web Sites:

Hilary Mason and Chris Wiggins, “A Taxonomy of Data Science“, September 25, 2010, http://www.dataists.com/2010/09/a-taxonomy-of-data-science/ (accessed April 9, 2013)

Maxwell, J.C., Report on the Cavendish Laboratory for 1876 from the Cambridge University Reporter, ca May 1877) In A history of the Cavendish laboratory, 1871-1910, https://www.google.com/search?q=Report+on+the+Cavendish+Laboratory+for+1876+&ie=utf-8&oe=utf-8&aq=t&rls=org.mozilla:en-US:official&client=firefox-a (accessed April 20, 2013)

Lee Henson, “Empowering Agile Teams”, 2012, http://agiledad.com/index.php/downloads/ (accessed June 27, 2013)

Games in the Workplace: Revolutionary or Run-of-the-mill?

Chermaine Li, Shilpa Ranganathan & Sidharth Vijayachandran

Abstract

Quality means very different things to different people – and, when it comes to software, all perceptions of quality are valid and all approaches must be tested. So to produce high quality software in today's networked universe, you either need to test all possible combinations (which isn't practical), or you need to test differently. When we run our tests, we essentially train the software to respond to a specific set of user actions. Traditional test strategies focus on feature or component level testing through either manual or automated means, but may not account for end-user scenario-based[1] testing or usability related aspects of the product as a whole.

On the Lync team, we use productivity games[2] specifically designed for our features and signature scenarios as a way to complement traditional workplace testing methodologies, and to expand how we test software. This paper discusses how we used productivity games to enhance the quality of our products by improving team collaboration, employee engagement, and cost savings. We illustrate the use of productivity games in the workplace as a way to involve many faces of quality into the software development lifecycle by including all major stakeholders into the gaming experience and targeting multiple platforms. We also show examples of how structured productivity games improve software quality, ensure a highly productive workforce and transform an often mundane task of software testing into a fun activity.

Biography

Chermaine Li (chermali@microsoft.com) *is an engineer on the Lync team at Microsoft working on products for Mobile and Slate platforms. She is passionate about finding innovative ways in which customer focused engineering can be adapted into engineering processes.*

Shilpa Ranganathan (shilpar@microsoft.com) *is a Senior Test Lead at Microsoft. She is passionate about working on innovative techniques to improve software testing and helping create and ship high quality products.*

Sidharth Vijayachandran (sivijaya@microsoft.com) *is an engineer on the Lync team and is passionate about building great software. He is currently working on the Lync store app for Windows 8 and learning the art of Scrum.*

1 Introduction

What's driving the use of productivity games in Lync?

The world of technology and the demographics of a global workforce are dramatically changing the way we work. The lines between work and "life" are being blurred. New communication and collaboration tools allow employees to connect in different ways. Enterprise social tools, such as Lync, Skype, Yammer, and Salesforce.com allow collaboration across organizational boundaries.

Microsoft Lync is communications software that runs on many platforms, such as Windows PC, Windows Phone, iOS, and Android, and enables people to collaborate across the globe. As with any mode of collaboration and communication, consumers want access to their apps on all their devices, platforms and services. Users can choose not only to participate in meetings using different Lync clients, but they can also switch platforms while in a conversation. Lync clients on each platform must seamlessly interoperate with each other throughout the entire conversation. For example, Lync enables users to join a meeting on the go with their mobile device and then seamlessly transition the meeting to their work computer. As such, end-user scenarios, interoperability and integration testing[3] of all Lync features and usability testing across all Lync clients is a challenge.

Feature or component level testing, done through manual and automated means, may not identify all defects in the product. When working toward a typical milestone, a variety of people like designers, program managers and customers also influence the software. These diverse perspectives help uncover interaction and usability issues. As we move towards the end of the product cycle, the cost of addressing these issues is greater, and hence it makes sense to involve these people throughout the creation process to get continuous feedback.

Additionally, testing is traditionally done in a monotonous and isolated environment. The same tests and techniques become less effective over time, and it is progressively harder to improve the quality of the product while keeping the team engaged. As such we needed to invest in a strategy to help uncover these defects early on during the product cycle – productivity games (Smith, 2012).

Productivity games can be a cost effective way to improve the quality of a product by providing the necessary engagement and incentive to all stakeholders to participate in the testing process. Productivity games also motivate these stakeholders to uncover issues and provide feedback, while offering additional benefits like improving employee morale, facilitating collaboration and team-work. (Smith, 2012) When we have a diverse set of users looking at the product across different deployments and environments, we get continuous feedback that help catch issues immediately.

2 Productivity games in the Workplace

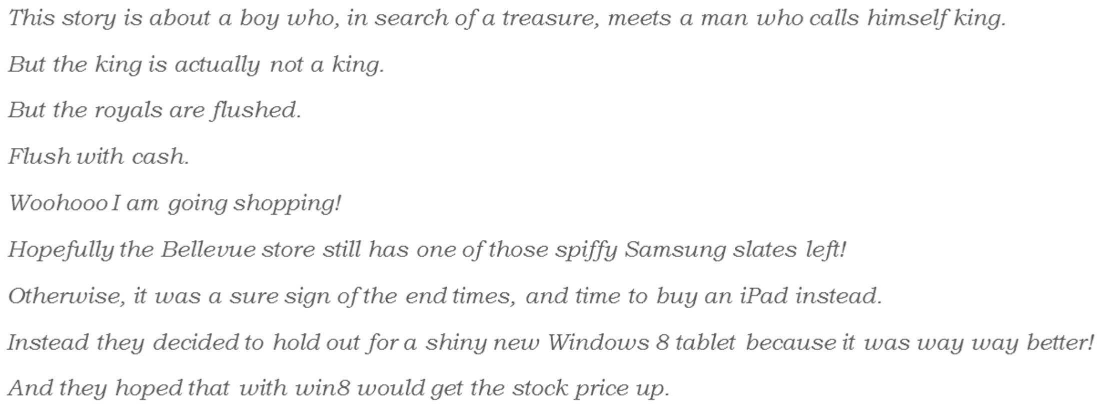
To validate our initial hypothesis that productivity games are an effective and novel way to improve software quality we structured our efforts in two directions – across Microsoft, and within the Lync organization.

2.1 Games exclusive to the Lync organization

Significant thought was put into how we could incorporate productivity games into our day to day test lifecycle. Analysis of our test processes indicates that a majority of the product's features are tested in silos by individuals with the occasional bug bash[4] organized to bring the team together to test a fully integrated feature set. It is also clear that the defect density tends to increase towards the end of a product cycle as members from different teams begin integration, make design tweaks and respond to customer feedback. By organizing productivity games for the Lync organization our goal was to create an experience that would motivate all our team members to participate, and to specifically involve individuals who do not usually participate in software testing. Described below (in no specific order) are the productivity games that were conducted within the Lync organization.

2.1.1 Using Lync's instant messaging capability to build crazy stories

The goal of our testing was to uncover reliability issues in the Instant Messaging (IM) feature in Lync where messages were being exchanged across multiple user accounts. To address this we created a game that required all participants to send instant messages in a round-robin manner. The organizer begins the game by sending any random sentence to the person on his/her right. That person would include the person to his/her right into the conversation and then add a new sentence. Each person added to the conversation only sees the previous sentence (not the whole conversation) and has to add a meaningful sentence of their own. Only the organizer sees the entire discussion. At the end of the game the organizer reads out a completely garbled and nonsensical story. This game exposed a few issues around sequencing of the messages when sent within a very short span of time, and allowed us to debug issues around missing messages. In particular we were able to isolate a hard to reproduce problem and identify mitigations for it.



This story is about a boy who, in search of a treasure, meets a man who calls himself king.

But the king is actually not a king.

But the royals are flushed.

Flush with cash.

Woohooo I am going shopping!

Hopefully the Bellevue store still has one of those spiffy Samsung slates left!

Otherwise, it was a sure sign of the end times, and time to buy an iPad instead.

Instead they decided to hold out for a shiny new Windows 8 tablet because it was way way better!

And they hoped that with win8 would get the stock price up.

2.1.2 Using Lync's sharing capability to identify landmarks

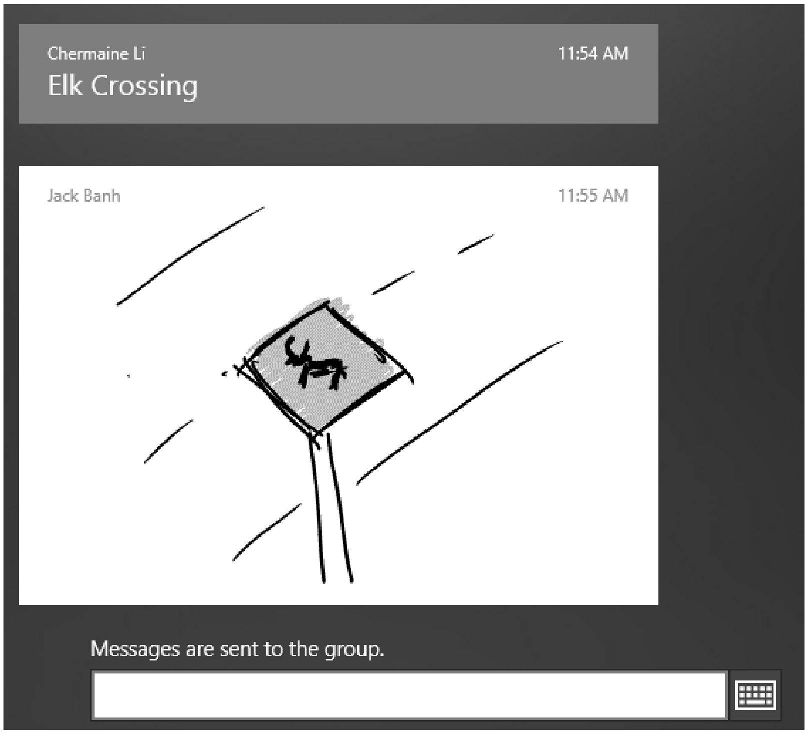
With the launch of Windows 8 the test team had to face the epic challenge of testing the performance of the Lync app for Windows 8 store on a variety of form factors. Tablets in particular had stricter performance criteria and could pose a risk to the perceived performance of Lync features. We designed a game where users were required to navigate between the various screens to complete certain tasks in order to identify the responsiveness of Lync, and to discover how accessible it is for a user to navigate by swiping or using other in-app shortcuts.

For this game, the moderator used Lync to share images of famous landmarks. The fastest person to send the name and location of the landmark using the IM feature won that round. This game proved to be a smart way to identify the device configurations that were optimal for Lync features. It also exposed some configurations for which we needed to optimize performance or provide messaging to the users on long-running tasks. It also served to test new features that were slated to be released as part of the Lync app for Windows 8 store.

2.1.3 Ink on Lync and safe driving

With the first version of the Lync app publicly available, a new feature called Ink (support for drawing instead of typing messages) was introduced for the next release. Our challenge was to validate the design of this feature as well as get usability feedback from different users about whether they were able to use Ink to complete their tasks, and verify if users were able to exchange accurate ink content using touch, mouse or pen, between the various older versions of Lync clients. We needed to ensure that the messages were displayed correctly and could be viewed by all users. To motivate our colleagues to contribute towards using the app we proposed a game that would reinforce Washington State's traffic signs in the user's minds and simultaneously test the Ink feature. During the game the organizer calls out the name of a particular traffic sign and all users are expected to draw that particular sign using the Ink feature and send it to the Lync meeting. Points were awarded for the best "Ink artists" and the winners took home goodies and a grand prize making it a fun testing experience.

During the meeting we realized that some devices were more optimized for a touch experience. It was evident that the drawing experience was much better using a pen rather than a finger which favored Microsoft’s Surface Pro over other non-pen input devices. It also highlighted subtle user experience issues of the Ink feature that we could improve upon. For instance, a common complaint during the game was that the thickness of the ink stroke needed to be increased to enable more fluid drawing. This game also brought about widespread awareness of the new feature on the Lync for Windows 8 store app.



Chermaine Li 11:54 AM

Elk Crossing

Jack Banh 11:55 AM

Messages are sent to the group.

2.2 Involving Microsoft's employees

Dogfooding[5] is a common practice where a software company uses its own product to demonstrate the quality and capabilities of the product. Microsoft uses dogfooding as one of the ways to gather feedback on a product in real-life scenarios. Since the process is voluntary, it is often difficult to keep users motivated and actively engaged in using the product. Collecting dogfood data on a mobile platform presents an additional challenge due to the restrictions that the mobile platform may bring. For example, a mobile device will have a smaller keyboard than a desktop computer, making it harder to type a message. When a user receives a new IM, they are more likely to respond from a computer than a mobile phone. This is a common feedback from Lync dogfood users when asked what barriers prevent them from using Lync on a mobile phone. The Lync Mobile team delivers the app on Windows Phone, iOS, and Android platforms and needed feedback with respect to usability and reliability of these features of Lync on all supported platforms. To encourage dogfooding, the team decided to use productivity games.

2.2.1 Lynctober MobileFest

The Lync Mobile team held a month-long challenge during October 2011 to gather dogfood data. Each week, a representative from the Lync team would send a list of tasks and survey forms to dogfood users. Users could complete as many tasks as they chose and respond to the survey form. We handed out points for any issues, frowns, smiles, or ideas reported, and gave out weekly prizes. In addition, the grand prize of a mobile device was given to the person with the highest score at the end of the challenge. For the conclusion of the MobileFest, we held a Mobile Beta Bash and invited all dogfooding users to spend an afternoon participating in productivity games with the Lync team. This also gave the users a quick way to accumulate points and get on the leader board. Some of the productivity games at the Mobile Beta Bash included:

- IM Taboo - The first person to guess the word using hints sent over IM wins.
- Story Station - Build a story using IM conference by only seeing the sentence from the previous person. The result was a fun, nonsensical story that drew a lot of laughter.

- Solve Me A Riddle - Solve a riddle posted on a wall in the room and call in your answer.
- Rock, Paper, Scissors with Presence - This is a remake of the popular children's game played by changing a user's presence status.

Each game was hosted at its own station and participants could travel to as many stations as they liked. The Mobile Beta Bash also gave participants the opportunity to interact directly with Lync engineers. We found that participants really enjoyed the events. They shared feedback, and felt encouraged by the fact that the engineering team was really responding to their comments and concerns. Participants who attended the event tended to be more engaged in dogfooding even after the event ended. Conducting productivity games across the company opened up a whole new avenue for increasing brand loyalty and product awareness whilst allowing employees across Microsoft to have fun and derive a certain satisfaction that they had contributed to the release of another Microsoft product. The Lynctober MobileFest was a huge success with dogfood users, but it came with a price tag. The team dedicated a lot of resources to planning and coordinating the event, creating a system to handle surveys and feedback, and handing out prizes during the event.

2.2.2 Lync Mobile Dogfood Bash

Following the success of the Lynctober MobileFest, the Lync Mobile team decided to use productivity games again in a subsequent release to gather dogfood data. This particular release had an additional challenge. We had limited test resource available and needed program managers to step in and help coordinate the event. Instead of a month-long challenge, this event was a single-day online meeting version of the games. Users collected "badges" during the day for completing tasks, and the person with the most badges won. Knowing that participants would drop in and out of the event at various times during the day, we tailored the tasks to require a minimal amount of people and allowed interactions to be asynchronous. Badges were collected for activities such as:

- Making a video call to a Bot[6] and identifying a random object shown by the Bot.
- Leaving a voicemail to a Bot.
- Receiving a surprise call.
- Joining a conference at a certain time.

Again we heard feedback from users that they enjoyed the productivity games and felt more encouraged to provide feedback in the future. Talking to a real person rather than emailing a large distribution group gave users the confidence that someone was listening to their feedback.

However, there was less effort put into organizing this event and it reflected in the results we received. The initial MobileFest had weekly challenges that helped advertise the event and created enthusiasm amongst the players. Without such benefit for the Dogfood Bash, we saw that the number of people who attended the event was significantly less and users tended to stay for a shorter period of time. Since this was a one-day event, people who could not attend missed out completely. In addition, we found it harder to keep people motivated and interested through online interaction. Though the data collected was useful, we did learn that the quality of the data we received was directly proportional to the effort we put in to organizing the productivity games.

3 Lessons learned

Productivity games are a fun and cost effective way to uncover issues that traditional test methodologies may have overlooked early in the product cycle. Some of the lessons we learned by using this test strategy are detailed below.

3.1 Reduce risks by encouraging a diverse user base to participate

Most products have a huge test matrix that needs to be validated across various deployments, platforms and configurations. The typical solution employed by most test teams is to simulate these environments and run all possible tests against them. However, in many cases it is extremely difficult to simulate such complex real world environments in a test lab. Also, this is clearly an expensive task. For example a simple test to validate the quality of Ink messages requires that the test be performed on different platform architectures, against multiple Lync server topologies, on all applicable form factors like tablets and desktops, and using multiple ink input modes such as pen, mouse and finger. Multiply these permutations by the number of Ink scenarios and the insurmountable nature of the task becomes obvious.

Crowdsourcing is a very cost effective technique in such situations. Users bring different perspectives towards using our product and exercise functionality on various platforms, devices and environments. People also tend to exercise the same scenarios in different ways to uncover distinct issues. The goal is to make sure we are able to incentivize and engage the diverse user base to help validate our scenarios. As such we need to create experiences that appeal to a broad set of users. Productivity games that rely on unique player skills limit the number of capable participants which then limits the various user configurations. For example, the “Ink on Lync and learn safe driving” game was about a user’s knowledge of traffic signs. This made the game generic enough for everyone to participate and did not rely on the technical knowledge of the intricacies of the Ink feature.

In the end, not all users have the same incentives. Some are motivated by leader board scores, others by prizes and yet others by contributing towards charitable causes. (Smith, 2012) By designing productivity games that cater to diverse cultural and personal motivations, we can provide incentives to engage a wider audience.

3.2 Engage people in different roles to improve quality – not just testers

Improving the quality of a product is not just a tester’s job. Other individuals like designers, program managers, and security and localization experts can contribute significantly towards improving the product quality by uncovering various issues based on their subject matter expertise. In a typical product lifecycle these individuals are not part of the testing process and tend to review and provide feedback on the non-functional aspects of the product very late in the cycle. This introduces the risk of destabilizing the product due to the changes from their feedback. Encouraging these individuals to validate and provide immediate feedback on the product will improve product quality.

3.3 Increasing brand awareness and product knowledge

Organization-wide productivity games clearly lead to an increase in brand awareness of Lync and customer perception of symbiosis with the product team. With multiple types of Lync clients (e.g. mobile clients, desktop clients, web browser clients) available to consumers, the use of productivity games served to educate people on scenarios where one client was more relevant. It also helped promote new Lync clients (e.g. Lync app for Windows 8 store) in cases where consumers were entrenched in the habit of using another client. The use of niche features was encouraged during these productivity games.

One piece of feedback from participants in our mobile events was that the productivity games helped them learn features that they never knew existed in the Lync client for a particular platform. After discovering the richness of the feature set of Lync on mobile devices, these users claimed that they are more likely to use Lync on their phones in the future. In addition, using games to teach makes training more effective since the learning process is attractive and rewarding (Smith, 2012). While teaching Lync features was not an intended goal of these events, it is an interesting avenue that the Lync team can explore when considering future events.

3.4 Gathering consumer feedback

Without a doubt, it is clear to us that productivity games yield benefits that routine test approaches struggle to provide. During and after each of the games we conducted, we received very valuable feedback from our dogfood users (gamers) that illustrated how Lync customers interact with our product on a daily basis. Once our product ships, we have very limited ways to contact customers to get their feedback. Productivity games provided an incredibly important avenue to receive such feedback early on. Throughout these games, engineers also get insights into the usage patterns of customers. As Lync engineers working on specific features, we get used to the way our features behave. We learn tricks and shortcuts that we unfairly expect our dogfood users and customers to grasp the first time they use our product. Watching and talking directly to gamers creates empathy and helps us bring the end user's point of view into the testing process.

3.5 Identifying feature requests for the next version of the product

In addition to improving brand awareness, we found that the Lync productivity games that we designed helped us identify feature requests for upcoming releases of the product. We notice that around 10% of the feedback received is usually in the form of wish lists from users. This gives us a broad perspective of the types of features desired by customers. It also allows us to predict market trends and helps us prioritize work for our future releases.

3.6 It's Fun!

We found that productivity games evoke a feeling of fun and served to boost team morale. The team enjoyed participating, and, at the same time felt a sense of accomplishment for completing productivity tasks. As a result, we were able to increase employee engagement, collaboration, trust, and employee retention. In the past, our participation rate for bug bashes was low because of the repetitive nature of the tests. With productivity games, people enthusiastically participated in these new forms of bug bashes. The benefits when a group of employees leave a conference in laughter and high spirits, knowing that they have each played a crucial part in improving the quality of our product, are immeasurable.

The competitive nature of some productivity games and the additional incentive of having rewards ensured that gamers were more focused and motivated. Competition in particular led to higher volumes of feedback and an increase in the fun quotient of the events we conducted.

4 Cost and Disadvantages

Despite all the benefits of using productivity games there are a few costs and disadvantages to keep in mind. Productivity games take a lot of resources. Time and effort is required to design productivity games with quality results. In addition, we needed additional infrastructure to support the increase in feedback that we got. A huge event like the Lynctober MobileFest also requires monetary support.

Not all scenarios are suitable for productivity games. It is important to understand the target audience before designing a game. For instance, productivity games targeting the internal product teams need to be more challenging since the teams are often aware of the subtleties of the application. Yet, the value of designing productivity games for internal product teams is very high since feedback is highly targeted and well-structured. For productivity games targeting a wider audience, we try to focus on mainline scenarios and showcasing new features.

We use productivity games to transform the mundane task of software testing into a fun activity. As such, we also need to change the games from time to time to ensure our participants remain engaged and challenged. Each of our productivity games is designed with a particular feature in mind, to help build awareness and get more eyes on the feature early on. Once a particular feature is tested via crowd sourcing, we can fall back on traditional testing strategies such as automation to ensure the quality of the feature in the future. By focusing on a different feature area each time, this also ensures that our games vary.

Finally, it is hard to quantitatively measure the success of productivity games. It is difficult to prove that the same benefits cannot be achieved through more traditional test techniques. However, productivity games provide a means of deriving value through a fun and inclusive way of testing.

5 Future plans

Until now we’ve restricted the scope of our productivity games to Microsoft employees. We plan to expand this to a wider audience. The Microsoft Lync team has an extensive Technology Adoption Program, where external companies try out a Beta version of Lync. We've heard internally how productivity games help dogfood users learn new features and build brand awareness. Perhaps the same can be used with Beta users outside of our organization to showcase new Lync features, encourage usage, and gather feedback. Games can make learning a product more attractive and rewarding for our Beta testers in other organizations.

Productivity games are successful in increasing organizational citizenship behavior and expanding in-role behavior (Smith 2010). Thus far, we've focused on using games that encourage good corporate behavior by incentivizing dogfooding and cross-feature testing. The next step for the Lync team may be to use productivity games to dissolve the boundaries between different roles. With the Lync team moving to a more agile development[7] model, it is beneficial for everyone to be capable of interchanging between our traditionally defined roles. We can design productivity games to teach, and users can learn within the context of play. Through productivity games, we can continue to increase the quality of our products by improving team collaboration, the skillset of the team, and employee engagement while reducing cost.

References

Games as a way to test products. (n.d.). Retrieved from Code In Heels: http://codeinheels.tumblr.com/post/28472885185/games-as-a-way-to-test-products

McDonald, M., Smith, R., & Musson, R. (2008). *The Practical Guide to Defect Prevention.* Microsoft Press.

Smith, R. (2010). *Communicate Hope: Using Games and Play to Improve Productivity - 42projects*. Retrieved from Management Innovation eXchange (MIX) : http://www.managementexchange.com/story/communicate-hope-using-games-and-play-improve-productivity-42projects

Smith, R (2011). Productivity Games – Ross Smith.

http://productivitygames.blogspot.com/

Smith, R. (2012). *How Play and Games Transform the Culture of Work*. Retrieved from journalofplay.org: http://www.journalofplay.org/sites/www.journalofplay.org/files/pdf-articles/5-1-interview-how-play-and-games-transform-the-culture-of-work.pdf

Smith, R., & Williams, J. (n.d.). *Score One for Quality!* Retrieved from 42Projects: http://www.42projects.org/docs/GTAC_LQG.PDF

[1] **Scenario-Based Testing:**

A test methodology where test cases are derived from real user scenarios.

[2] **Productivity games:**

A sub-category of games designed to improve productivity and morale of people in the workplace.

[3] **Integration testing:**

A stage in software testing where all components of a product come together and the system is verified as a whole.

[4] **Bug bash:**

A coordinated event where multiple people come together to explore and find bugs in a product within fixed amount of time.

[5] **Dogfooding:**

Common term in software where a company releases a pre-Beta build of software to be used by its own employees. The software builds are usually unrefined and may contain bugs. The goal of dogfooding is to demonstrate confidence in the software that a company develops by encouraging employees to use the software first.

[6] **Bot:**

An automated program that simulates the behavior of a Lync user.

[7] **Agile Development:**

A software development methodology that focuses on short and incremental iterations to produce working output, and to incorporate customer feedback on an ongoing basis.

Administering Quality Assurance for Large IT Programs

Sreeram Gopalakrishnan

sreeram.gopalakrishnan@cognizant.com

Abstract

Large IT programs, such as ERP implementations, are massively impactful long-term initiatives that pervade multiple departments of the organization including IT, business, infrastructure, security, training, PMO, third-party services and the like. Large programs are unique in several other ways too, such as the high amount of executive visibility they carry, the number of different vendors and the consequent dynamics involved, the sheer amount of interdependencies among the different parts of the program, and the vast amount of changes that the interdependencies spawn. In short, a large program is much bigger than the sum of its parts.

Administering Quality Assurance (QA) for such large programs calls for a vastly different approach vis-à-vis for medium or small projects. The QA Program Manager, as the person responsible, will need to be well-versed with the uniqueness of large programs. Despite QA fundamentals remaining the same, the integrated nature of large programs are bound to stretch the QA processes to their limits. Hence, any attempt to straightjacket QA as a bundle of processes, methodologies, tools and strategies is most likely to fail.

The key to QA's success, will be how well the QA Program Manager can manage the integration aspects of QA, over and above managing the customary technical aspects of QA. The integration aspects cut across areas such as team organization, team building, communication, test processes, test planning and test management. Examples of a few integration items would include (1) org structure balancing QA's integration and independence needs (2) integrated dependency planning, tracking and reporting (3) common set of testing processes and nomenclature that merge the vendor-specific differences (4) strong governance model to control process adherence and deviations (5) framework to manage multivendor dynamics and promote a badgeless team (6) integrated test environment landscapes, their conflicts, and data refresh strategies (7) defect fix turnarounds and defect aging in a multi-vendor / multi-team scenario (8) Regression scope and ownership and (9) Performance testing integration.

The paper elaborates on the management of the above listed integration aspects for large programs, and discusses some of the better practices to deal with them effectively.

Biography

Sreeram Gopalakrishnan is a passionate and experienced QA professional with 14 years of industry experience, currently working as QA Senior Manager at Cognizant Technology Solutions. He has managed QA for large system integrations, technology migrations, new application developments etc. He has an MBA in Systems, and also holds PMP and CMST certifications. He has presented / published papers in international conferences / magazines, notable ones being IGNITE in 2010, Indian Business Academy in 2009, and Software Test & Performance magazine in 2008.

1. Introduction

As described in the abstract, large IT programs have several unique characteristics compared to small and medium IT projects. These characteristics influence the application and management of QA activities for large programs in equally unique ways.

In typical IT projects, QA's involvement ranges a wide variety of activities from reviews, walkthroughs and inspections, to the most well-known role of testing, and also extend to specialized roles such as governance and consulting. Even though the underlying fundamental principles for these activities remain the same across all types of projects, it is the size and type of projects that influence the activities with respect to their scope, strategy or approach, rigor, situational significance, and the like.

The objective of this technical paper is to review these distinct aspects of administering QA for large IT programs.

Due to the vastness of the subject, this paper will limit the discussion to what is different compared to the standard QA practices or methods.

The paper starts with a discussion of the characteristics of large IT programs and their influence on the QA processes.

It then discusses the top QA areas which in the author's experience have the maximum impact in the context of large programs, namely: (1) QA org structure, (2) risk and dependency management, (3) test nomenclature, (4) flexibility of strategy and governance, (5) status reporting cadence (6) environment management (7) defect management (8) Regression model and (9) Performance testing.

2. Characteristics of Large IT Programs

Let us start by observing some of the characteristics of a large IT program that set it apart from a medium or small IT project.

2.1 Span

Typically, large programs span multiple years and multiple releases, with incremental solution capabilities delivered in each release. The programs involve and impact several business areas within the organization, often concurrently, over an extended period of time.

2.2 Integration

Over the course of the program, integration happens not just of the IT applications, but also of the business areas as a whole. Prior to integration, the IT work starts off as small individual projects within each of the business areas, mostly at an application level. During this phase, each project goes through its own SLDC of requirements-design-build-test-deploy. As the program rolls on, the integration starts and the individual projects start to come together. The integration grows, and so do the complexity and cross-impacts across the the projects or applications or business areas.

2.3 Vendor Methodology Differences

Each business or application area may have several IT vendors offering their products and services. Each vendor would bring its own IT and QA methodologies and tools. At the initial stages of the program, prior to integration, the differences in methodologies and tools wouldn’t matter much, however once the integration starts, they surely pose problems if they continue to remain distinct.

2.4 Multi-Vendor Dynamics

The multi-vendor scenario in large IT programs can potentially cause considerable amount of competitiveness among the vendors, which if unchecked, will adversely impact the program, because of mutual dependencies that exist among the vendor teams.

The competitiveness, sometimes even friction, is not limited to the vendors only. Even the IT and business units within the customer organization may have misalignments and conflicts, as they too would have operated as independent units prior to their collaboration as a part of the program. The vendor organizations getting caught in the cross fire between the departments is not uncommon.

2.5 Top-Management Visibility

Yet another aspect worth drawing our attention to is the high amount of top management visibility that large programs carry. The high executive visibility is due to the large budgets and spends involved, as well as due to the potential stock price impact and market expectations from such high visibility programs.

The above listed unique characteristics, though are not QA-specific, they impact and influence QA activities as much as they do for the rest of the program. They introduce numerous parameters and challenges that force changes to the way QA is administered in a normal project. It is discussed in the following sections.

3. Org Structure for QA Integration and Independence

Given the complexity of a large program, QA’s success in meeting its objective of assuring quality will depend on how QA is structured within its own organization, as well as on how QA is integrated with the rest of the program.

The key factors to bear in mind when defining the organizational structure would be QA’s needs of efficiency in operations, integration with the program, and independence in its operations.

3.1 Efficiency within QA

Efficiency looks at the best ways of grouping QA’s tasks for maximum collaboration and productivity gains. In addition to project delivery which is at the core of QA activities, there are several other tasks typical of large programs that consume a considerable amount of time and effort. Unless if efficiently grouped and managed, these tasks can prove to be quite costly. They include change request management, budget and resource management, process management, metrics and measurement, environment planning and governance.

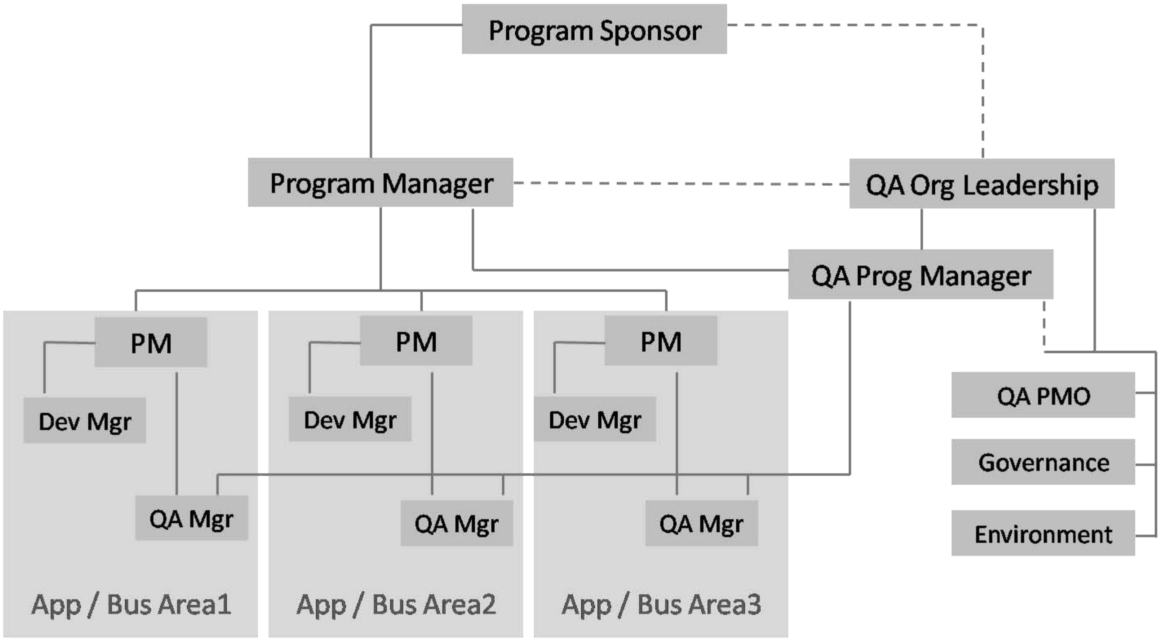
3.2 QA-Program Integration

In large programs, due to their size and speed, things are constantly in flux and not many people get an end-to-end view of the program. It becomes too much of changes to keep up with, even with the best connected of teams. Without QA well integrated with the rest of the program, and especially with QA being at the end of the SDLC, it often suffers due to late change notifications, late discovery of dependencies and compression of timelines.

3.3 Independence of QA

Independence stems from the fact that QA has to be objective in its assessment of quality and risks to serve the best interests of the program. This cannot be done effectively, unless if the org structure allows QA sufficient independence and authority. It needs independence to call out the status and risks as it sees, and needs authority to secure the support of other teams to action on the risks and issues.

The diagram below illustrates an organization that supports the efficiency, integration and independence needs of QA.



4. Managing Risks and Dependencies

Now that we have the right org structure in place that takes care of QA's integration and independence, it provides the launch pad for dealing with the project risks and dependencies in a proactive manner. The intent should be to get visibility as early as possible into the changes, risks and dependencies that impact the program, and also to provide QA the ability to influence the stakeholders positively to remedy a risk or an issue.

The requirement then is to set up the right forums for interactions and communications where QA is a part of all key discussions on program status, changes and risks.

Some of the key tools and mechanisms that have been found to be most effective for this are Interdependency Meetings with Program Team, Weekly Status Meetings with Program Leadership, Risk and Issue Logs.

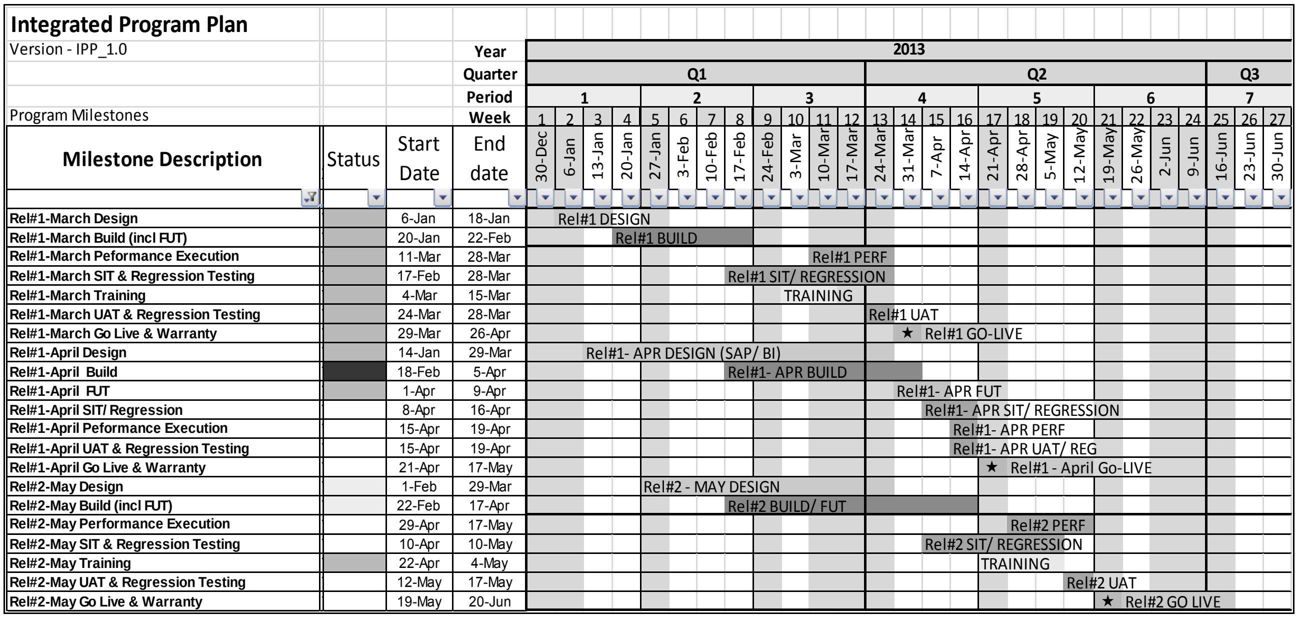
4.1 Interdependency Meetings

Interdependency meetings are where the different teams involved in the program come together, preferably on a weekly basis at least, and discuss the developments and changes happening in their respective areas. The meetings typically involve the Program Manager as the chair, and the IT Project

Managers, Solution Leads of each application area, and sometimes the Business Leads as well. QA Program Managers must be a part of this meeting. The QA Program Manager can be supported by the QA Leads of individual application areas, but it is not mandatory, as the intent of these meetings are not to analyze the technical solution details or make instantaneous decisions, but more to discuss the plan, schedule and dependencies, changes so that each team can take them away for its own impact assessment.

The interdependency meetings are driven out of an Integrated Program Plan (IPP), the high-level project plan that summarizes the key milestones from individual teams' detailed project plans (dev, QA, business, deployment, training etc). These meetings, notwithstanding the challenges of maintaining the IPP up-to-date, are vital for communicating and analyzing project interdependencies and cross-impacts.

Shown below is the Gantt view of a sample Integrated Program Plan that shows its key milestones across multiple releases.



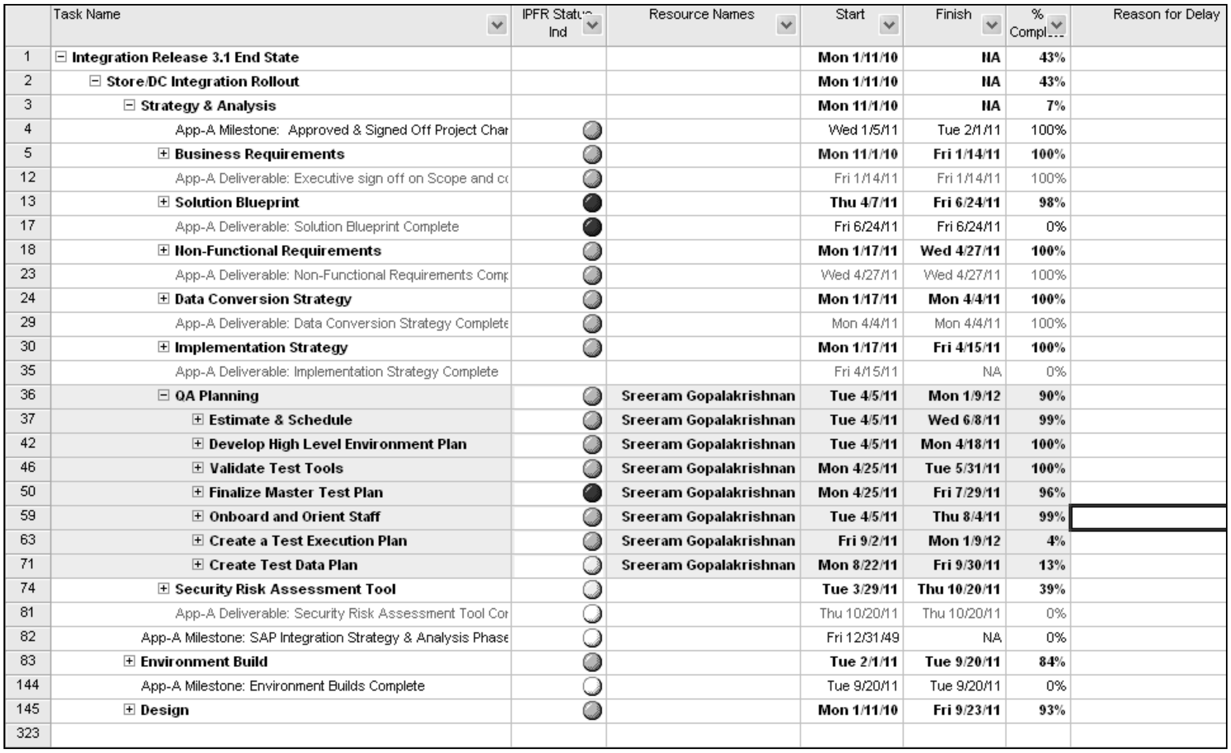
Integrated Program Plan

Version - IPP_1.0

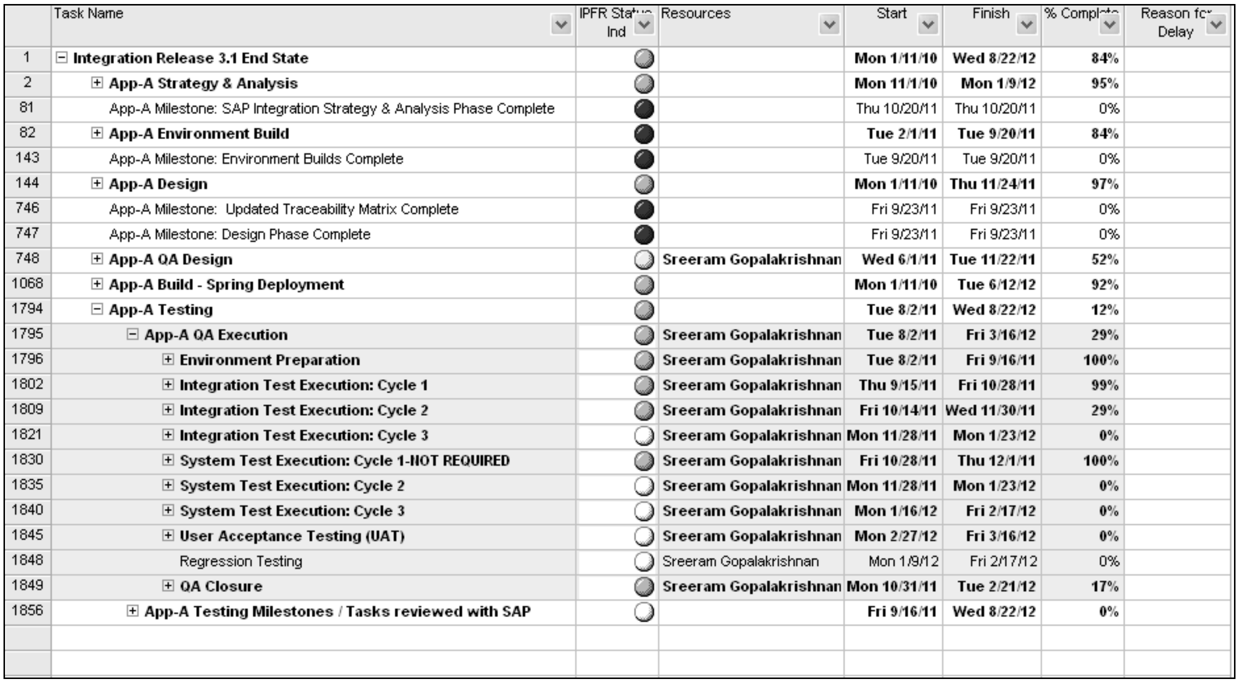
Program Milestones

Milestone Description	Status	Start Date	End date	
Rel#1-March Design		6-Jan	18-Jan	Rel#1 DESIGN
Rel#1-March Build (incl FUT)		20-Jan	22-Feb	Rel#1 BUILD
Rel#1-March Peformance Execution		11-Mar	28-Mar	Rel#1 PERF
Rel#1-March SIT & Regression Testing		17-Feb	28-Mar	Rel#1 SIT/ REGRESSION
Rel#1-March Training		4-Mar	15-Mar	TRAINING
Rel#1-March UAT & Regression Testing		24-Mar	28-Mar	Rel#1 UAT
Rel#1-March Go Live & Warranty		29-Mar	26-Apr	★ Rel#1 GO-LIVE
Rel#1-April Design		14-Jan	29-Mar	Rel#1- APR DESIGN (SAP/ BI)
Rel#1-April Build		18-Feb	5-Apr	Rel#1- APR BUILD
Rel#1-April FUT		1-Apr	9-Apr	Rel#1- APR FUT
Rel#1-April SIT/ Regression		8-Apr	16-Apr	Rel#1- APR SIT/ REGRESSION
Rel#1-April Peformance Execution		15-Apr	19-Apr	Rel#1- APR PERF
Rel#1-April UAT & Regression Testing		15-Apr	19-Apr	Rel#1- APR UAT/ REG
Rel#1-April Go Live & Warranty		21-Apr	17-May	★ Rel#1 - April Go-LIVE
Rel#2-May Design		1-Feb	29-Mar	Rel#2 - MAY DESIGN
Rel#2-May Build (incl FUT)		22-Feb	17-Apr	Rel#2 BUILD/ FUT
Rel#2-May Performance Execution		29-Apr	17-May	Rel#2 PERF
Rel#2-May SIT & Regression Testing		10-Apr	10-May	Rel#2 SIT/ REGRESSION
Rel#2-May Training		22-Apr	4-May	TRAINING
Rel#2-May UAT & Regression Testing		12-May	17-May	Rel#2 UAT
Rel#2-May Go Live & Warranty		19-May	20-Jun	★ Rel#2 GO LIVE

The illustrations below are those of the Integrated Program Plan and detailed QA project plan in Microsoft Project. It shows how the detailed QA project plan is tied into the Integrated Program Plan.



	Task Name	IPFR Statu Ind	Resource Names	Start	Finish	% Compl	Reason for Delay
1	⊟ **Integration Release 3.1 End State**			**Mon 1/11/10**	**NA**	**43%**	
2	⊟ **Store/DC Integration Rollout**			**Mon 1/11/10**	**NA**	**43%**	
3	⊟ **Strategy & Analysis**			**Mon 11/1/10**	**NA**	**7%**	
4	App-A Milestone: Approved & Signed Off Project Char	◍		Wed 1/5/11	Tue 2/1/11	100%	
5	⊞ **Business Requirements**	◍		**Mon 11/1/10**	**Fri 1/14/11**	**100%**	
12	App-A Deliverable: Executive sign off on Scope and cc	◍		Fri 1/14/11	Fri 1/14/11	100%	
13	⊞ **Solution Blueprint**	●		**Thu 4/7/11**	**Fri 6/24/11**	**98%**	
17	App-A Deliverable: Solution Blueprint Complete	●		Fri 6/24/11	Fri 6/24/11	0%	
18	⊞ **Non-Functional Requirements**	◍		**Mon 1/17/11**	**Wed 4/27/11**	**100%**	
23	App-A Deliverable: Non-Functional Requirements Comp	◍		Wed 4/27/11	Wed 4/27/11	100%	
24	⊞ **Data Conversion Strategy**	◍		**Mon 1/17/11**	**Mon 4/4/11**	**100%**	
29	App-A Deliverable: Data Conversion Strategy Complete	◍		Mon 4/4/11	Mon 4/4/11	100%	
30	⊞ **Implementation Strategy**	◍		**Mon 1/17/11**	**Fri 4/15/11**	**100%**	
35	App-A Deliverable: Implementation Strategy Complete			Fri 4/15/11	NA	0%	
36	⊟ **QA Planning**	◍	**Sreeram Gopalakrishnan**	**Tue 4/5/11**	**Mon 1/9/12**	**90%**	
37	⊞ **Estimate & Schedule**	◍	**Sreeram Gopalakrishnan**	**Tue 4/5/11**	**Wed 6/8/11**	**99%**	
42	⊞ **Develop High Level Environment Plan**	◍	**Sreeram Gopalakrishnan**	**Tue 4/5/11**	**Mon 4/18/11**	**100%**	
46	⊞ **Validate Test Tools**	◍	**Sreeram Gopalakrishnan**	**Mon 4/25/11**	**Tue 5/31/11**	**100%**	
50	⊞ **Finalize Master Test Plan**	●	**Sreeram Gopalakrishnan**	**Mon 4/25/11**	**Fri 7/29/11**	**96%**	
59	⊞ **Onboard and Orient Staff**	◍	**Sreeram Gopalakrishnan**	**Tue 4/5/11**	**Thu 8/4/11**	**99%**	
63	⊞ **Create a Test Execution Plan**	◍	**Sreeram Gopalakrishnan**	**Fri 9/2/11**	**Mon 1/9/12**	**4%**	
71	⊞ **Create Test Data Plan**	○	**Sreeram Gopalakrishnan**	**Mon 8/22/11**	**Fri 9/30/11**	**13%**	
74	⊞ **Security Risk Assessment Tool**	○		**Tue 3/29/11**	**Thu 10/20/11**	**39%**	
81	App-A Deliverable: Security Risk Assessment Tool Cor	○		Thu 10/20/11	Thu 10/20/11	0%	
82	App-A Milestone: SAP Integration Strategy & Analysis Phase	○		Fri 12/31/49	NA	0%	
83	⊞ **Environment Build**	◍		**Tue 2/1/11**	**Tue 9/20/11**	**84%**	
144	App-A Milestone: Environment Builds Complete	○		Tue 9/20/11	Tue 9/20/11	0%	
145	⊞ **Design**	◍		**Mon 1/11/10**	**Fri 9/23/11**	**93%**	
323							

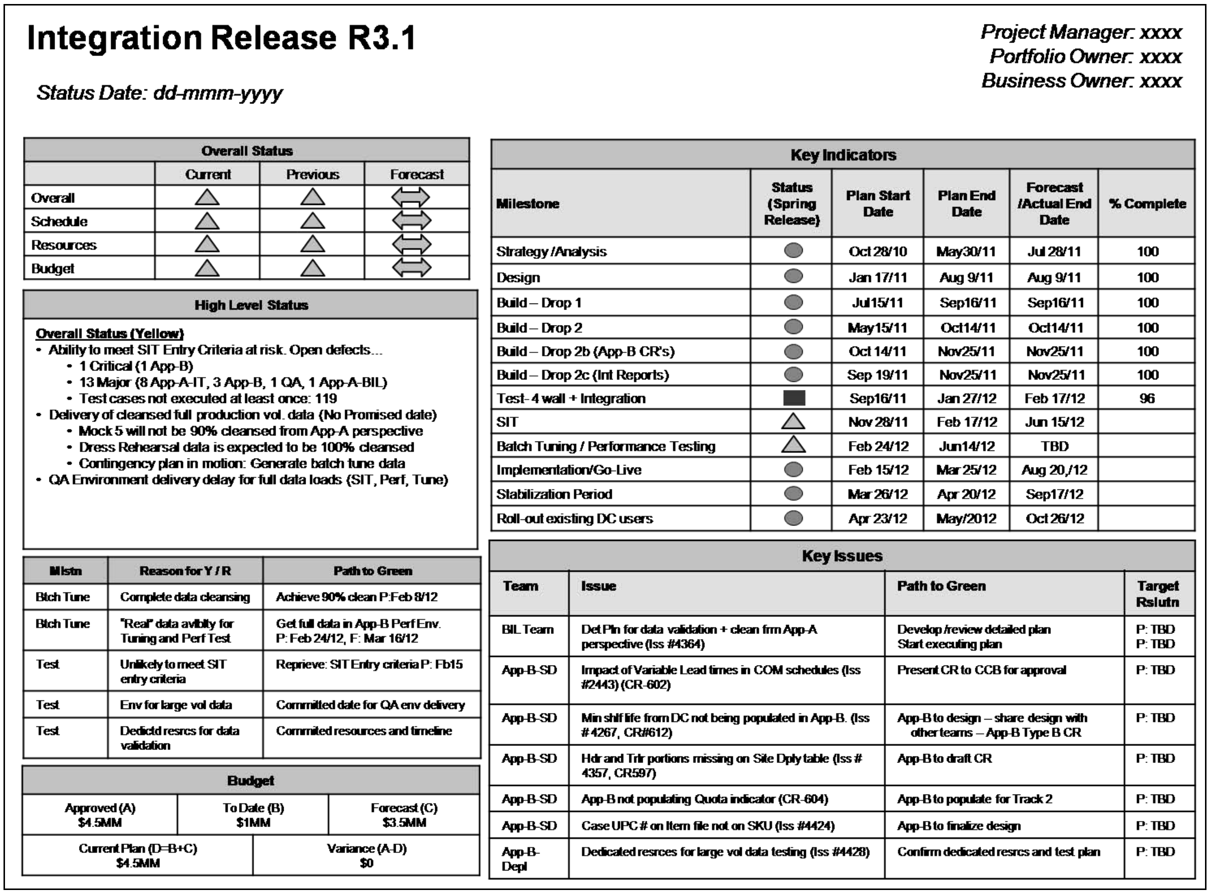


	Task Name	IPFR Sta Ind	Resources	Start	Finish	% Compl	Reason fc Delay
1	⊟ **Integration Release 3.1 End State**	◍		**Mon 1/11/10**	**Wed 8/22/12**	**84%**	
2	⊞ **App-A Strategy & Analysis**	◍		**Mon 11/1/10**	**Mon 1/9/12**	**95%**	
81	App-A Milestone: SAP Integration Strategy & Analysis Phase Complete	●		Thu 10/20/11	Thu 10/20/11	0%	
82	⊞ **App-A Environment Build**	●		**Tue 2/1/11**	**Tue 9/20/11**	**84%**	
143	App-A Milestone: Environment Builds Complete	●		Tue 9/20/11	Tue 9/20/11	0%	
144	⊞ **App-A Design**	◍		**Mon 1/11/10**	**Thu 11/24/11**	**97%**	
746	App-A Milestone: Updated Traceability Matrix Complete	●		Fri 9/23/11	Fri 9/23/11	0%	
747	App-A Milestone: Design Phase Complete	●		Fri 9/23/11	Fri 9/23/11	0%	
748	⊞ **App-A QA Design**	○	**Sreeram Gopalakrishnan**	**Wed 6/1/11**	**Tue 11/22/11**	**52%**	
1068	⊞ **App-A Build - Spring Deployment**	◍		**Mon 1/11/10**	**Tue 6/12/12**	**92%**	
1794	⊟ **App-A Testing**	◍		**Tue 8/2/11**	**Wed 8/22/12**	**12%**	
1795	⊟ **App-A QA Execution**	◍	**Sreeram Gopalakrishnan**	**Tue 8/2/11**	**Fri 3/16/12**	**29%**	
1796	⊞ **Environment Preparation**	◍	**Sreeram Gopalakrishnan**	**Tue 8/2/11**	**Fri 9/16/11**	**100%**	
1802	⊞ **Integration Test Execution: Cycle 1**	◍	**Sreeram Gopalakrishnan**	**Thu 9/15/11**	**Fri 10/28/11**	**99%**	
1809	⊞ **Integration Test Execution: Cycle 2**	◍	**Sreeram Gopalakrishnan**	**Fri 10/14/11**	**Wed 11/30/11**	**29%**	
1821	⊞ **Integration Test Execution: Cycle 3**	○	**Sreeram Gopalakrishnan**	**Mon 11/28/11**	**Mon 1/23/12**	**0%**	
1830	⊞ **System Test Execution: Cycle 1-NOT REQUIRED**	◍	**Sreeram Gopalakrishnan**	**Fri 10/28/11**	**Thu 12/1/11**	**100%**	
1835	⊞ **System Test Execution: Cycle 2**	○	**Sreeram Gopalakrishnan**	**Mon 11/28/11**	**Mon 1/23/12**	**0%**	
1840	⊞ **System Test Execution: Cycle 3**	○	**Sreeram Gopalakrishnan**	**Mon 1/16/12**	**Fri 2/17/12**	**0%**	
1845	⊞ **User Acceptance Testing (UAT)**	○	**Sreeram Gopalakrishnan**	**Mon 2/27/12**	**Fri 3/16/12**	**0%**	
1848	Regression Testing	○	Sreeram Gopalakrishnan	Mon 1/9/12	Fri 2/17/12	0%	
1849	⊞ **QA Closure**	◍	**Sreeram Gopalakrishnan**	**Mon 10/31/11**	**Tue 2/21/12**	**17%**	
1856	⊞ **App-A Testing Milestones / Tasks reviewed with SAP**	○		**Fri 9/16/11**	**Wed 8/22/12**	**0%**	

4.2 Weekly Status Meetings with Program Leadership

Weekly Status Meetings with Program Leadership is another important forum that serves a very important function. Unlike Interdependency Meetings where the focus is on cross-project dependencies and impacts of plan or schedule changes across different teams, the weekly status meetings are to discuss progress against plan. Actual progress against plan is discussed and the top factors slowing down the progress are either resolved or marked for escalation. Due to the size of the program, the Weekly Status Meetings are conducted out of Summarized Program-Level Weekly Status Reports, that are rolled up versions of the Detailed Weekly Status Reports of individual project teams.

The below two illustrations are those of program-level weekly status report, and QA-level weekly status report. You can notice how the summarized QA milestones in the program-level report are blown up into low-level milestones in the QA-level report.



Integration Release R3.1

Status Date: dd-mmm-yyyy

Project Manager: xxxx
Portfolio Owner: xxxx
Business Owner: xxxx

Overall Status			
	Current	Previous	Forecast
Overall	△	△	⇔
Schedule	△	△	⇔
Resources	△	△	⇔
Budget	△	△	⇔

High Level Status

Overall Status (Yellow)

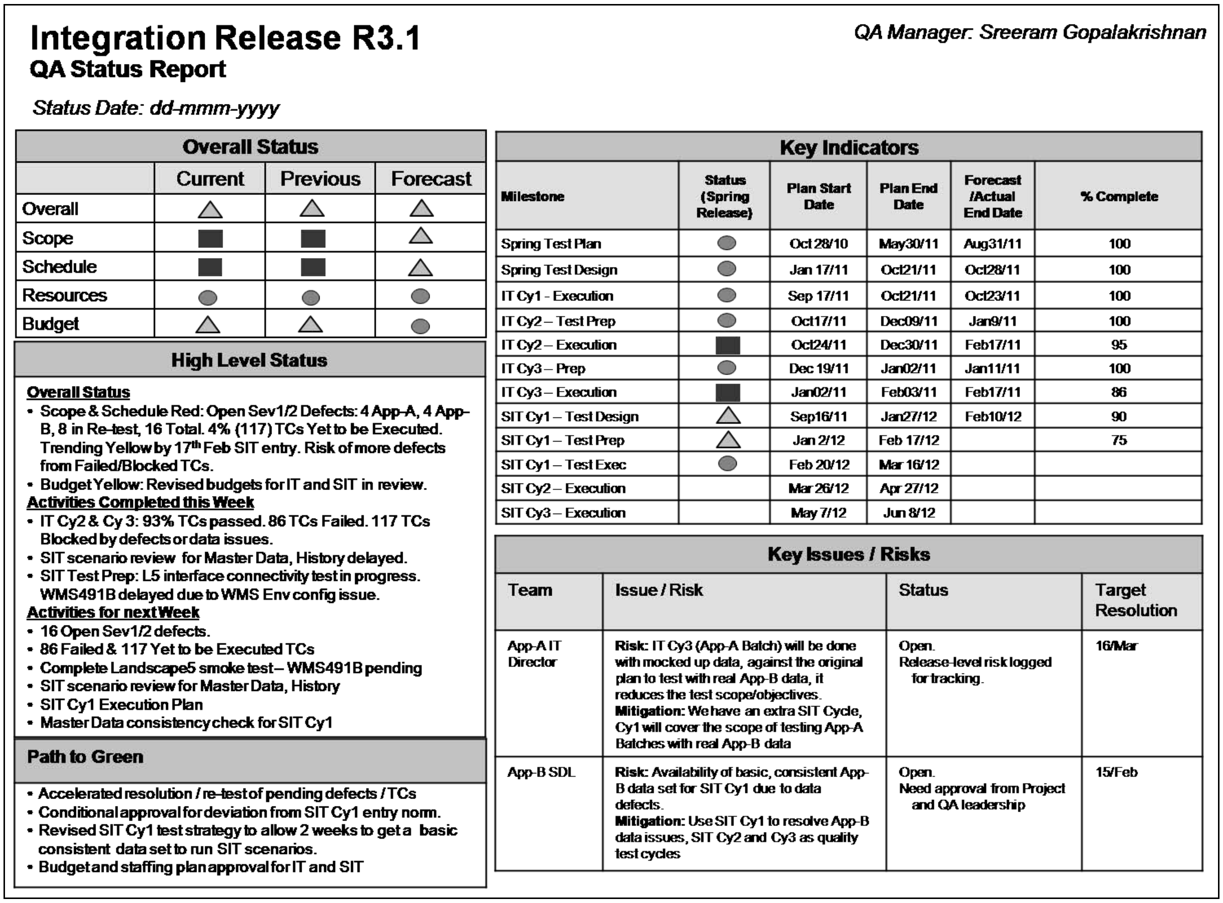
- Ability to meet SIT Entry Criteria at risk. Open defects…
  - 1 Critical (1 App-B)
  - 13 Major (8 App-A-IT, 3 App-B, 1 QA, 1 App-A-BIL)
  - Test cases not executed at least once: 119
- Delivery of cleansed full production vol. data (No Promised date)
  - Mock 5 will not be 90% cleansed from App-A perspective
  - Dress Rehearsal data is expected to be 100% cleansed
  - Contingency plan in motion: Generate batch tune data
- QA Environment delivery delay for full data loads (SIT, Perf, Tune)

Mlstn	Reason for Y / R	Path to Green
Btch Tune	Complete data cleansing	Achieve 90% clean P:Feb 8/12
Btch Tune	"Real" data avlbity for Tuning and Perf Test	Get full data in App-B Perf Env. P: Feb 24/12, F: Mar 16/12
Test	Unlikely to meet SIT entry criteria	Reprieve: SIT Entry criteria P: Fb15
Test	Env for large vol data	Committed date for QA env delivery
Test	Dedictd resrcs for data validation	Commited resources and timeline

Budget		
Approved (A) $4.5MM	To Date (B) $1MM	Forecast (C) $3.5MM
Current Plan (D=B+C) $4.5MM	Variance (A-D) $0	

Key Indicators					
Milestone	Status (Spring Release)	Plan Start Date	Plan End Date	Forecast /Actual End Date	% Complete
Strategy /Analysis	●	Oct 28/10	May30/11	Jul 28/11	100
Design	●	Jan 17/11	Aug 9/11	Aug 9/11	100
Build – Drop 1	●	Jul15/11	Sep16/11	Sep16/11	100
Build – Drop 2	●	May15/11	Oct14/11	Oct14/11	100
Build – Drop 2b (App-B CR's)	●	Oct 14/11	Nov25/11	Nov25/11	100
Build – Drop 2c (Int Reports)	●	Sep 19/11	Nov25/11	Nov25/11	100
Test- 4 wall + Integration	■	Sep16/11	Jan 27/12	Feb 17/12	96
SIT	△	Nov 28/11	Feb 17/12	Jun 15/12	
Batch Tuning / Performance Testing	△	Feb 24/12	Jun14/12	TBD	
Implementation/Go-Live	●	Feb 15/12	Mar 25/12	Aug 20,/12	
Stabilization Period	●	Mar 26/12	Apr 20/12	Sep17/12	
Roll-out existing DC users	●	Apr 23/12	May/2012	Oct 26/12	

Key Issues			
Team	Issue	Path to Green	Target Rslutn
BIL Team	Det Pln for data validation + clean frm App-A perspective (Iss #4364)	Develop /review detailed plan Start executing plan	P: TBD P: TBD
App-B-SD	Impact of Variable Lead times in COM schedules (Iss #2443) (CR-602)	Present CR to CCB for approval	P: TBD
App-B-SD	Min shlf life from DC not being populated in App-B. (Iss # 4267, CR#612)	App-B to design – share design with other teams – App-B Type B CR	P: TBD
App-B-SD	Hdr and Trlr portions missing on Site Dply table (Iss # 4357, CR597)	App-B to draft CR	P: TBD
App-B-SD	App-B not populating Quota indicator (CR-604)	App-B to populate for Track 2	P: TBD
App-B-SD	Case UPC # on Item file not on SKU (Iss #4424)	App-B to finalize design	P: TBD
App-B-Depl	Dedicated resrces for large vol data testing (Iss #4428)	Confirm dedicated resrcs and test plan	P: TBD



Integration Release R3.1

QA Status Report

QA Manager: Sreeram Gopalakrishnan

Status Date: dd-mmm-yyyy

Overall Status

	Current	Previous	Forecast
Overall	△	△	△
Scope	■	■	△
Schedule	■	■	△
Resources	●	●	●
Budget	△	△	●

High Level Status

Overall Status

- Scope & Schedule Red: Open Sev1/2 Defects: 4 App-A, 4 App-B, 8 in Re-test, 16 Total. 4% (117) TCs Yet to be Executed. Trending Yellow by 17th Feb SIT entry. Risk of more defects from Failed/Blocked TCs.
- Budget Yellow: Revised budgets for IT and SIT in review.

Activities Completed this Week

- IT Cy2 & Cy 3: 93% TCs passed. 86 TCs Failed. 117 TCs Blocked by defects or data issues.
- SIT scenario review for Master Data, History delayed.
- SIT Test Prep: L5 interface connectivity test in progress. WMS491B delayed due to WMS Env config issue.

Activities for next Week

- 16 Open Sev1/2 defects.
- 86 Failed & 117 Yet to be Executed TCs
- Complete Landscape5 smoke test – WMS491B pending
- SIT scenario review for Master Data, History
- SIT Cy1 Execution Plan
- Master Data consistency check for SIT Cy1

Path to Green

- Accelerated resolution / re-test of pending defects / TCs
- Conditional approval for deviation from SIT Cy1 entry norm.
- Revised SIT Cy1 test strategy to allow 2 weeks to get a basic consistent data set to run SIT scenarios.
- Budget and staffing plan approval for IT and SIT

Key Indicators

Milestone	Status (Spring Release)	Plan Start Date	Plan End Date	Forecast /Actual End Date	% Complete
Spring Test Plan	●	Oct 28/10	May30/11	Aug31/11	100
Spring Test Design	●	Jan 17/11	Oct21/11	Oct28/11	100
IT Cy1 - Execution	●	Sep 17/11	Oct21/11	Oct23/11	100
IT Cy2 – Test Prep	●	Oct17/11	Dec09/11	Jan9/11	100
IT Cy2 – Execution	■	Oct24/11	Dec30/11	Feb17/11	95
IT Cy3 – Prep	●	Dec 19/11	Jan02/11	Jan11/11	100
IT Cy3 – Execution	■	Jan02/11	Feb03/11	Feb17/11	86
SIT Cy1 – Test Design	△	Sep16/11	Jan27/12	Feb10/12	90
SIT Cy1 – Test Prep	△	Jan 2/12	Feb 17/12		75
SIT Cy1 – Test Exec	●	Feb 20/12	Mar 16/12		
SIT Cy2 – Execution		Mar 26/12	Apr 27/12		
SIT Cy3 – Execution		May 7/12	Jun 8/12		

Key Issues / Risks

Team	Issue / Risk	Status	Target Resolution
App-A IT Director	**Risk:** IT Cy3 (App-A Batch) will be done with mocked up data, against the original plan to test with real App-B data, it reduces the test scope/objectives. **Mitigation:** We have an extra SIT Cycle, Cy1 will cover the scope of testing App-A Batches with real App-B data	Open. Release-level risk logged for tracking.	16/Mar
App-B SDL	**Risk:** Availability of basic, consistent App-B data set for SIT Cy1 due to data defects. **Mitigation:** Use SIT Cy1 to resolve App-B data issues, SIT Cy2 and Cy3 as quality test cycles	Open. Need approval from Project and QA leadership	15/Feb

5. Test Phase / Type Definitions and Nomenclature

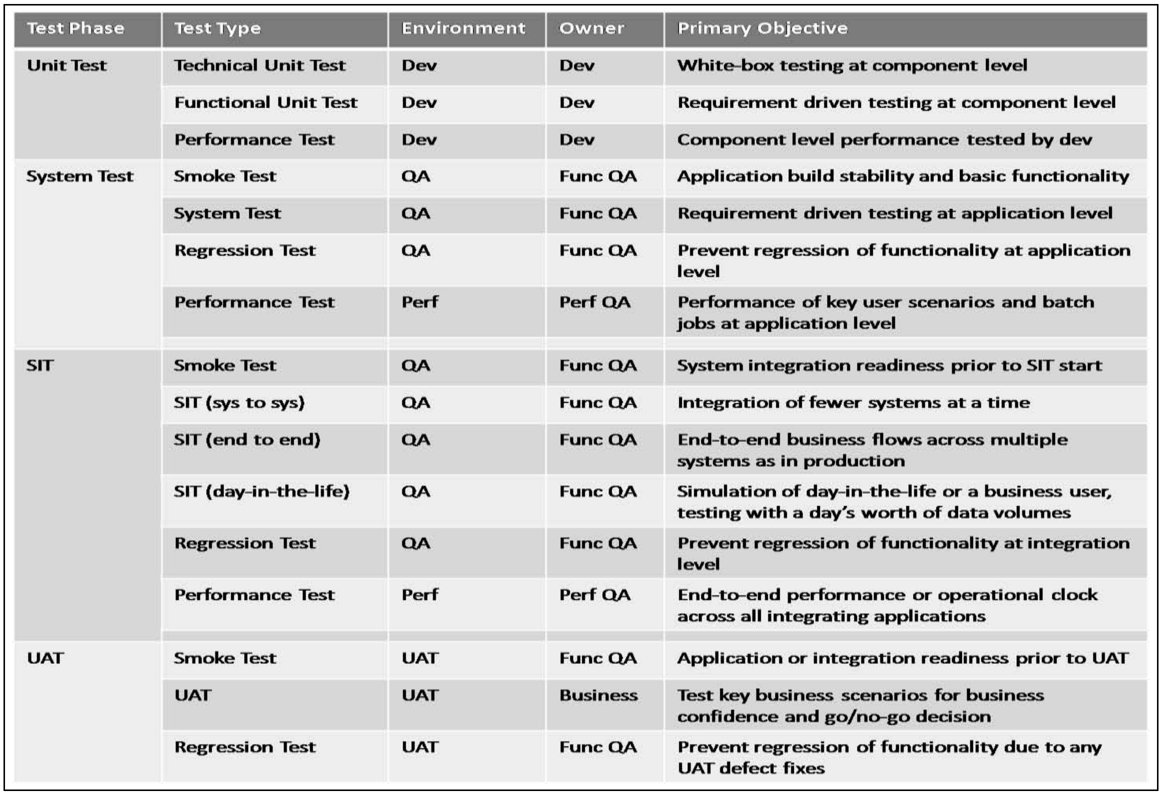
One of the common problems in large programs is the difference in the ways in which the vendors involved refer to the test phases and test types. Usually, each vendor brings its own QA methodology and nomenclature. While this is not a problem during the initial phases of the program where they operate in silos, this starts to pose a problem once the integration starts. With each vendor referring to a particular phase or type of testing in different terms, it leads to confusion and debates.

To quote a few examples, a certain vendor might refer to the QA phases as Functional Unit Testing, Integration Testing, System Testing, User Acceptance Testing, whereas some other might refer to them as System Testing, System Integration Testing, User Acceptance Testing. In the above example, the Functional Unit Test of the first vendor would be the near equivalent of System Testing of the second vendor and the Integration + System Testing of the first vendor would be the equivalent of System Integration Testing of the second vendor. Such differences cause a lot of confusion and arguments during the course of a program. Hence is imperative that the QA organization comes up with clear definitions of the test phases and their nomenclature.

Another issue commonly observed with test phase and test type definitions is that not everyone in the program would be clear about the differences between Test Phases and Test Types, particularly the non-QA teams. Test Phases and Test Types get used interchangeably adding to the confusion.

A third issue related to test phase and test type nomenclature is the large number of non-standardized test types that each vendor QA team brings in as a part of its methodology. String testing, Assembly testing, Gravy testing, Connectivity testing, Link testing, Process-oriented testing, Data model testing are some examples of the non-standardized test types. Even smoke and sanity testing are understood very differently by different teams. While there is no disputing the fact that the non-standardized test types have specific relevance to the program, they need to be clearly categorized or slotted into any of the standard test phases to prevent confusion about when or why they occur.

Given below is one of the more suitable ways of categorizing test phases and test types for large programs.



Test Phase	Test Type	Environment	Owner	Primary Objective
Unit Test	Technical Unit Test	Dev	Dev	White-box testing at component level
	Functional Unit Test	Dev	Dev	Requirement driven testing at component level
	Performance Test	Dev	Dev	Component level performance tested by dev
System Test	Smoke Test	QA	Func QA	Application build stability and basic functionality
	System Test	QA	Func QA	Requirement driven testing at application level
	Regression Test	QA	Func QA	Prevent regression of functionality at application level
	Performance Test	Perf	Perf QA	Performance of key user scenarios and batch jobs at application level
SIT	Smoke Test	QA	Func QA	System integration readiness prior to SIT start
	SIT (sys to sys)	QA	Func QA	Integration of fewer systems at a time
	SIT (end to end)	QA	Func QA	End-to-end business flows across multiple systems as in production
	SIT (day-in-the-life)	QA	Func QA	Simulation of day-in-the-life or a business user, testing with a day's worth of data volumes
	Regression Test	QA	Func QA	Prevent regression of functionality at integration level
	Performance Test	Perf	Perf QA	End-to-end performance or operational clock across all integrating applications
UAT	Smoke Test	UAT	Func QA	Application or integration readiness prior to UAT
	UAT	UAT	Business	Test key business scenarios for business confidence and go/no-go decision
	Regression Test	UAT	Func QA	Prevent regression of functionality due to any UAT defect fixes

6. Flexibility of Test Strategy and Governance

As a part of defining the QA methodology for the organization or the program, the definition of test phases and test types involves not just the nomenclature, but also the definition of processes involved in the test phases and test types. Definition of the test processes typically follows the industry-wide model of ETVX, where the entry criteria, tasks and validations, and exit criteria are clearly defined for each test phase and test type. The definition and monitoring of ETVX will be a part of the QA governance activities.

In the case of large programs, where the ETVX model would put to test is in defining its tolerance limits. In other words, the maximum permissible deviations from the benchmarks should also be a part of the ETVX, along with the approval mechanisms to effect the deviations.

Defining tolerance limits thus is a critical element of the governance process, without which QA governance would run the risk of either degenerating into an ineffective control mechanism or be seen by the other program teams as too rigid and non-pragmatic to meet the objectives of the program. It is a fine line between ensuring effective process adherence and risking excessive rigidity.

A few examples of scenarios that will require guidelines to be published for permitting deviations are:

- Can SIT Cycle-2 be allowed to start when you have defects open in SIT Cycle-1

- Can UAT be allowed to start when SIT is not complete
- Can Performance testing overlap SIT, or should it wait till SIT is complete
- Should functional regression test be repeated if late Performance tuning changes get delivered

Such scenarios are very situational and it is very difficult to even come up with all possible scenarios. Hence we can't be prescriptive with the solution as well. The least that can be done is to define a governance framework that can quickly and effectively judge on such scenarios as they happen, avoiding the bureaucratic traps.

7. Setting Up a Status Reporting Cadence

High executive visibility automatically means high emphasis on quick, accurate and integrated status reporting. One of the most difficult aspects of a large program can be to get the status reporting right – that is, to get a single integrated view of the program status that is consistent with the status reported by the individual teams that constitute the program. Silly as it might sound, in reality it can be a daunting task. In a program, each team would create several status reports intended for different levels of audiences, with different levels of detail.

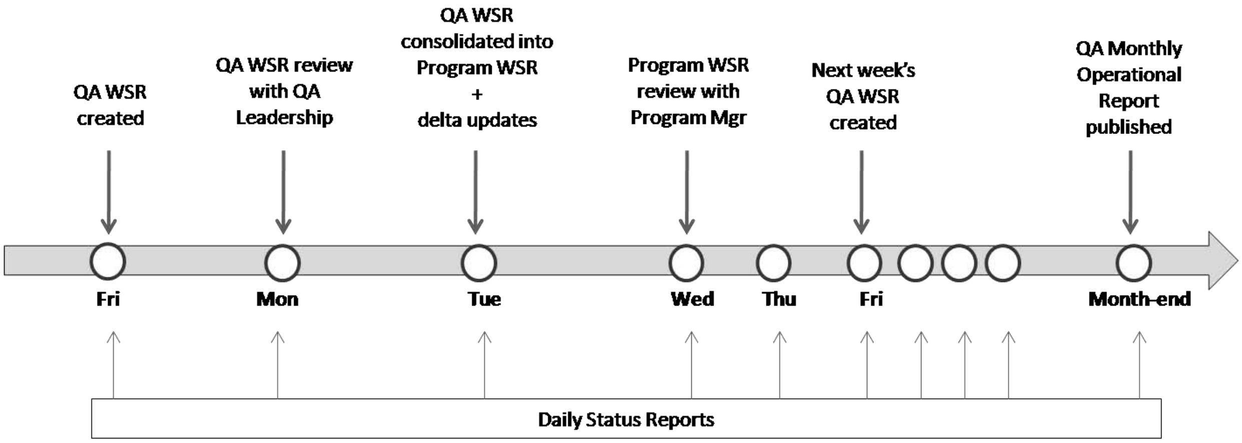
As much time as it takes to roll up the project status bottom-up through the different levels, the thick and fast churn of events in a large program shortens the shelf-life of the point-in-time reporting. It thus requires careful planning of the timing of the entire status reporting cycle, starting from report consumption, working backwards to report generation and roll up.

Each program is different and will need to define its own status reporting process. But there is no denying that it needs to be defined globally for the program and not left to the choice of the individual projects or teams.

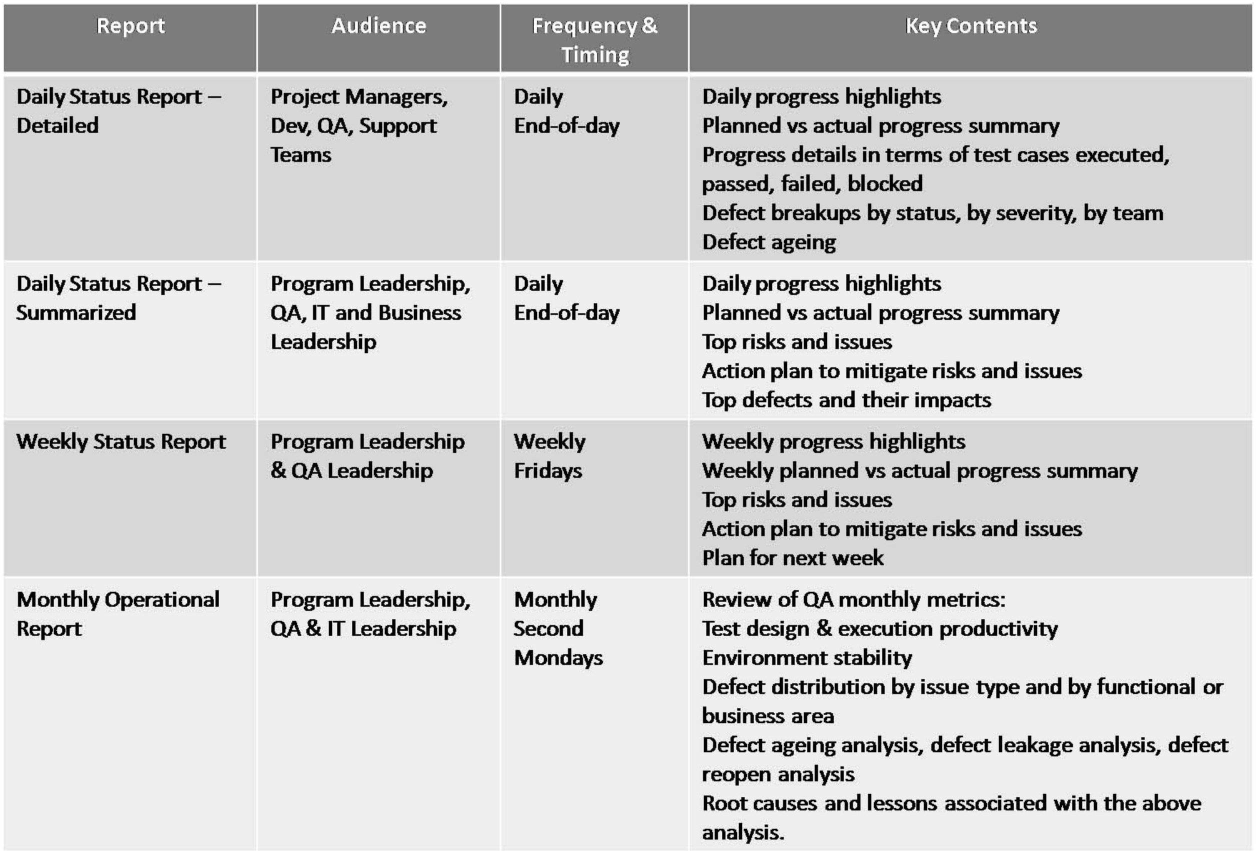
The status reporting process should clearly spell out the following:

- The levels of report that will be required and their intended audience
- The objective of each of the report and its content
- The frequency and specific timing of each of the report
- The data sources for the reports and the process of report generation and publishing

What can really help is a visual representation of the reporting cadence that shows the different reports with their expected timelines.


QA WSR created
QA WSR review with QA Leadership
QA WSR consolidated into Program WSR + delta updates
Program WSR review with Program Mgr
Next week's QA WSR created
QA Monthly Operational Report published
Fri
Mon
Tue
Wed
Thu
Fri
Month-end
Daily Status Reports

Given below is a suggested list of QA status reports optimized for maximum efficiency of reporting, with a mention of the intended audience, frequency and key contents.



Report	Audience	Frequency & Timing	Key Contents
Daily Status Report – Detailed	Project Managers, Dev, QA, Support Teams	Daily End-of-day	Daily progress highlights Planned vs actual progress summary Progress details in terms of test cases executed, passed, failed, blocked Defect breakups by status, by severity, by team Defect ageing
Daily Status Report – Summarized	Program Leadership, QA, IT and Business Leadership	Daily End-of-day	Daily progress highlights Planned vs actual progress summary Top risks and issues Action plan to mitigate risks and issues Top defects and their impacts
Weekly Status Report	Program Leadership & QA Leadership	Weekly Fridays	Weekly progress highlights Weekly planned vs actual progress summary Top risks and issues Action plan to mitigate risks and issues Plan for next week
Monthly Operational Report	Program Leadership, QA & IT Leadership	Monthly Second Mondays	Review of QA monthly metrics: Test design & execution productivity Environment stability Defect distribution by issue type and by functional or business area Defect ageing analysis, defect leakage analysis, defect reopen analysis Root causes and lessons associated with the above analysis.

What is equally important is to have a standard format for each of the reports. This goes a long way in representing information in a uniform way across all projects in the program, making them more actionable, and helping with the ease of consolidating the lower-level reports get into higher levels.

8. Test Environment Planning and Management

Talking about dependencies, the single-most biggest dependency for QA in any project is usually the availability and readiness of QA environments or landscapes (a cluster of connected application environments required to perform integrated testing for a program release).

Readiness of test environment covers multiple facets such as ensuring the availability of the physical application environments or servers, ensuring that the environments have the right application code and configuration versions, ensuring that the environments have the required quality and quantity of data, ensuring that the middleware and interfaces that form the plumbing between applications are connected and ready, ensuring that the test team members have the right accesses to the application environments.

In a large program, while the above fundamentals still hold good, the environment management effort and complexity gets accentuated as described in the following sections:

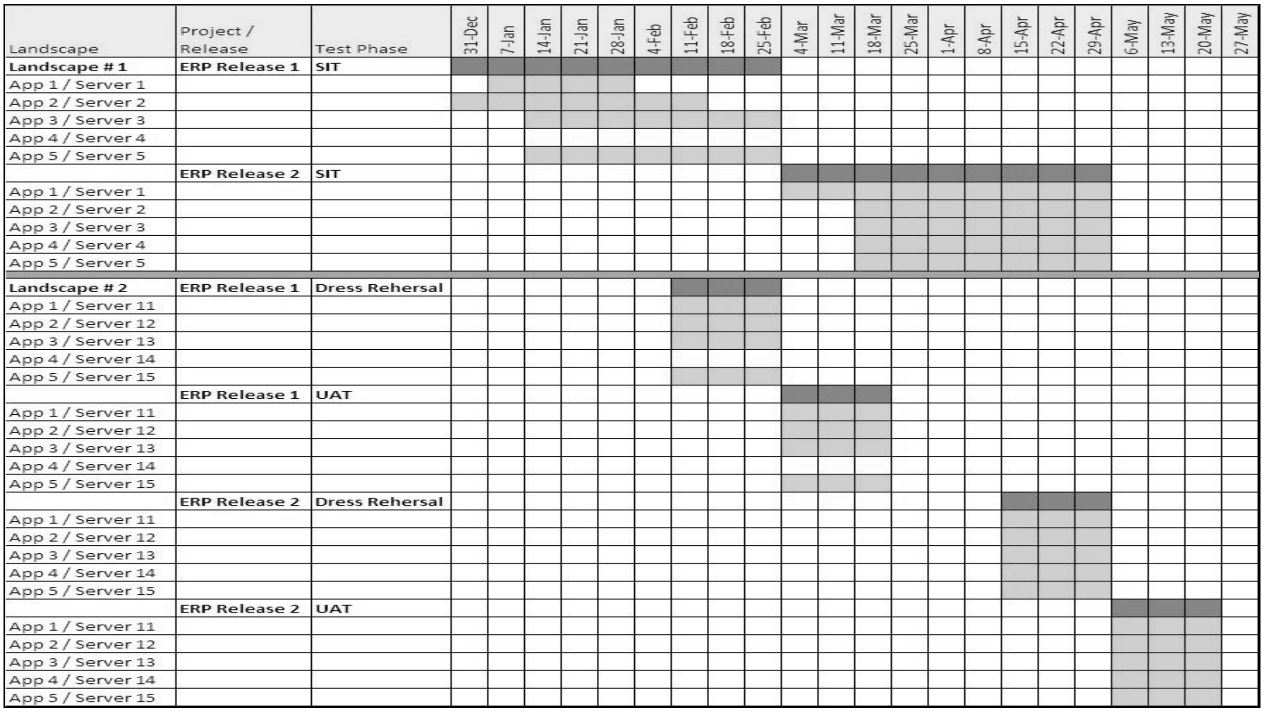
8.1 Integration with Release Management Function

Due to the size and complexity of the program, at any given point of time, multiple releases of the program would concurrently be in the testing phase, albeit not necessarily in the same phase of testing or in the same QA landscape. Ensuring that each QA landscape has the appropriate code versions for the release will be a critical QA environment management activity.

For this, the rules of collaboration between the QA environment management and release management functions will have to be clearly defined. Depending on the organization, the release management function could be at the organizational or program level. There could be several ways of collaboration, but the common two models are: (a) release management is completely owned and performed by the organizational/program level release management team for all environments including QA environments, and the QA environment team merely governs the process for QA environments (b) release management is owned by release management team, but performed by QA environment team for the QA environments.

8.2 Environment / Landscape Reservation

Often in large programs, due to cost reasons, not all environments can be dedicated or exclusive. In such cases, the available environments are shared across multiple releases within the program, or sometimes even with other projects outside the program. Hence, careful planning is necessary for reserving or blocking the specific shared environments for a particular release. The environment sharing complicates matters when an earlier release falls behind on schedule, impacting the environment availability for the subsequent ones. Despite the above limitation, environment reservation is a basic requirement in a large program. Given below is an illustration of the QA environment schedule that is owned and managed by the QA Environment team.



Landscape	Project / Release	Test Phase	31-Dec	7-Jan	14-Jan	21-Jan	28-Jan	4-Feb	11-Feb	18-Feb	25-Feb	4-Mar	11-Mar	18-Mar	25-Mar	1-Apr	8-Apr	15-Apr	22-Apr	29-Apr	6-May	13-May	20-May	27-May
Landscape # 1	**ERP Release 1**	**SIT**	▓	▓	▓	▓	▓	▓	▓	▓	▓													
App 1 / Server 1				░	░	░	░																	
App 2 / Server 2			░	░	░	░	░	░	░															
App 3 / Server 3					░	░	░	░	░	░	░													
App 4 / Server 4																								
App 5 / Server 5					░	░	░	░	░	░	░													
	ERP Release 2	**SIT**										▓	▓	▓	▓	▓	▓	▓	▓	▓				
App 1 / Server 1												░	░	░	░	░	░	░	░	░				
App 2 / Server 2														░	░	░	░	░	░	░				
App 3 / Server 3														░	░	░	░	░	░	░				
App 4 / Server 4														░	░	░	░	░	░	░				
App 5 / Server 5														░	░	░	░	░	░	░				
Landscape # 2	**ERP Release 1**	**Dress Rehersal**							▓	▓	▓													
App 1 / Server 11									░	░	░													
App 2 / Server 12									░	░	░													
App 3 / Server 13									░	░	░													
App 4 / Server 14																								
App 5 / Server 15									░	░	░													
	ERP Release 1	**UAT**										▓	▓	▓										
App 1 / Server 11												░	░	░										
App 2 / Server 12												░	░	░										
App 3 / Server 13												░	░	░										
App 4 / Server 14																								
App 5 / Server 15												░	░	░										
	ERP Release 2	**Dress Rehersal**																▓	▓	▓				
App 1 / Server 11																		░	░	░				
App 2 / Server 12																		░	░	░				
App 3 / Server 13																		░	░	░				
App 4 / Server 14																		░	░	░				
App 5 / Server 15																		░	░	░				
	ERP Release 2	**UAT**																			▓	▓	▓	
App 1 / Server 11																					░	░	░	
App 2 / Server 12																					░	░	░	
App 3 / Server 13																					░	░	░	
App 4 / Server 14																					░	░	░	
App 5 / Server 15																					░	░	░	

For the issues of environment conflicts and schedule dependencies quoted earlier, there are several innovative solutions available today. Environment virtualization, cloud, and on-demand environment provisioning tools are a few of them. The details of these topics are beyond the scope of this paper.

8.3 Test Data Synchronization and Refresh

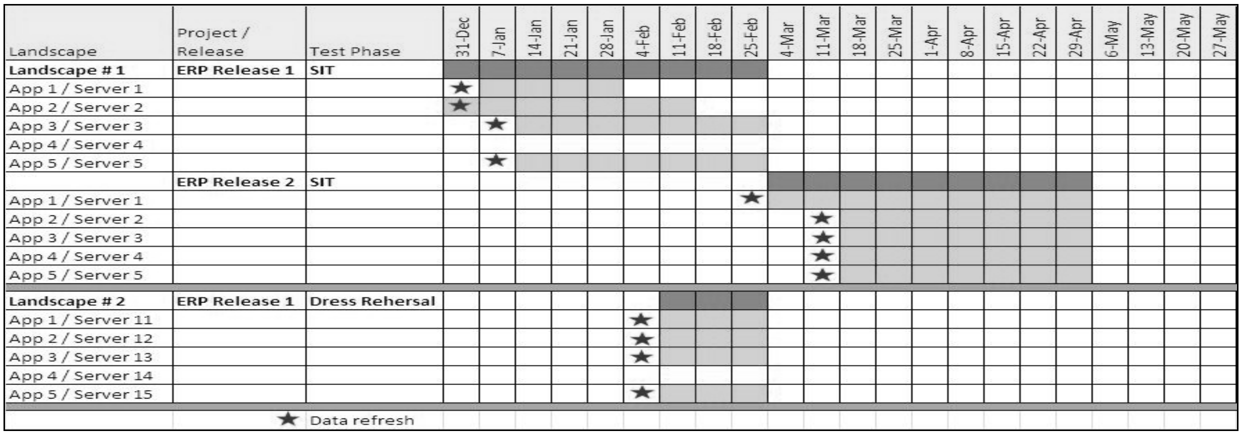
Yet another unique requirement of a large program is that SIT, and sometimes UAT too, requires the various applications in a landscape to have synchronized data, without which the integrated business scenarios will be difficult to test. In my experience, almost 40% of the effort involved in SIT, particularly for the end-to-end business flow tests, goes for data readiness. Thus, it is a significant activity. Let us examine how data readiness works in a program.

Most of the times, the base data in each application environment is set up by copying a slice of production data. Synchronization would mean that all the production copies should be of the same or nearly same date, so that the business flows between the systems do not fail for want of synchronized data. This requires meticulous data planning. Though there are data management tools available, one thing to remember would be that the tools can only simplify the mechanical and repeatable tasks of data management.

The real challenge is more in defining the data requirements, that is, in identifying the right data sets that fulfill the end-to-end business scenario needs. It will largely be a manual task, and will require a deep understanding of the data integration design. The trick here is to create an exclusive Data Lead position and staff it with the best person in the team, one who would have the understanding of the cross-system data flows. This person then would drive integrated data planning sessions among the participants of SIT, in which the data requirements are discussed and documented for each scenario. The Data Lead plays the key role of bridging the gaps in understanding of data among the individual teams.

Once the requirements are clear, this is where the data management tools can help. Tools can be explored for efficiency with the repeatable tasks of masking, sub-setting, archiving and so on.

Further, in the case of those application environments that are shared across multiple releases or projects, care should be taken not to refresh the data without checking the impacts to the dependent projects or releases. The environment schedule that we discussed earlier can be leveraged to indicate the data refreshes planned, helping teams to call out any adverse impacts well in advance. The below chart illustrates the data refresh schedule embedded into the environment schedule.



Landscape	Project / Release	Test Phase	31-Dec	7-Jan	14-Jan	21-Jan	28-Jan	4-Feb	11-Feb	18-Feb	25-Feb	4-Mar	11-Mar	18-Mar	25-Mar	1-Apr	8-Apr	15-Apr	22-Apr	29-Apr	6-May	13-May	20-May	27-May
Landscape # 1	**ERP Release 1**	**SIT**	▓	▓	▓	▓	▓	▓	▓	▓	▓													
App 1 / Server 1			★	░	░	░	░																	
App 2 / Server 2			★	░	░	░	░	░	░															
App 3 / Server 3				★	░	░	░	░	░	░	░													
App 4 / Server 4																								
App 5 / Server 5				★	░	░	░	░	░	░	░													
	ERP Release 2	**SIT**										▓	▓	▓	▓	▓	▓	▓	▓	▓				
App 1 / Server 1											★	░	░	░	░	░	░	░	░	░				
App 2 / Server 2													★	░	░	░	░	░	░	░				
App 3 / Server 3													★	░	░	░	░	░	░	░				
App 4 / Server 4													★	░	░	░	░	░	░	░				
App 5 / Server 5													★	░	░	░	░	░	░	░				
Landscape # 2	**ERP Release 1**	**Dress Rehearsal**							▓	▓	▓													
App 1 / Server 11								★	░	░	░													
App 2 / Server 12								★	░	░	░													
App 3 / Server 13								★	░	░	░													
App 4 / Server 14																								
App 5 / Server 15								★	░	░	░													
	★	Data refresh																						

8.4 Factoring Time for Smoke Test

A big mistake that several programs make is in not factoring sufficient time for smoke test of the environment prior to SIT start. If system testing has preceded SIT as it would normally do, then smoke test of individual applications would be less of an issue. However the middleware, that constitutes the data transfer mechanism between the applications, would surely have been an untested area thus far. Hence, smoke test of the data transfer mechanism would fit in right at the start of SIT, and the actual SIT cannot start until the smoke test is completed and data flow is established. One challenge with getting the integration to work would be the multiple layers of technology involved in the middleware, with each layer possibly owned by a different development team. Fixing middleware issues thus involves a lot of triage and analysis, with the issues often going around several times between the teams. A best practice successfully followed for smoke test of integration is a war-room with the testers and development teams seated around a table.

9. Defect Management

The importance of a well-defined defect management process for any project cannot be overemphasized. Once a project or a release enters the testing phase, defects and the various defect metrics become the central subject of almost all the project discussions and status meetings.

The different facets covered by the defect management process will include defect lifecycle, defect severity definitions, defect triage meetings, defect metrics, defect reports, defect management tool and the like.

With the premise that a robust defect management process is in place for a large program, there exist several unique challenges in a large program that would require special treatment from a defect management standpoint vis-à-vis a regular project.

9.1 Defect Triage during SIT

Defect triages during the system testing phase involve fewer teams, more focused discussions, and thus quicker resolution of defects. However, in the SIT phase of a large program, particularly when the big end-to-end business scenarios get tested, the defect triage meetings would require development and/or functional teams, in addition to the QA teams, from all the integrating application areas. The numbers become unwieldy and the teams don’t think of it as the best use of their time. The all-hands triage meetings thus become counter-productive.

A recommended best practice is to have one or more dedicated defect managers for the program who can drive and co-ordinate the defect resolutions across various teams, and an unambiguous method defined to identify the top priority defects for the day.

The defect managers conduct the daily defect triage where the defects assigned top priority are discussed. The triage is attended by the QA leads of all application areas, who in turn conduct focused defect-based discussions with the development and/or functional leads of the relevant application / business areas. The QA, development, functional and business leads are educated on the need to keep the defect information up-to-date in the central defect management tool. The defect managers do periodic follow ups during the course of the day with the assignees of the defect.

9.2 Defining Defect Turnaround SLAs

Defect aging is a vital metric that indicates the overall defect management efficiency in a program. At a granular level, it is a combination of the efficiency of the triage process itself, the efficiency of the development teams to turnaround fixes and the efficiency of the QA teams to re-test the fixes. While QA has control over triage and re-test, fix turnaround is not within its control. Nonetheless, fix turnaround time has a big impact on QA’s ability to get through the planned test coverage within its stipulated time.

Any attempt by QA to define organization-level or program-level defect turnaround SLA’s most likely would meet with a lot of resistance from the development teams involved. This is primarily due to the fact that SLA’s typically have contractual implications for the development vendors. Unless if the vendor contracts had defect SLA’s built into them, it would be hard for the QA organization to get the development teams to agree to SLA’s. This is where QA’s “independence” as per the program org structure would come to help. QA will require the Program Sponsor or Organization Leadership to exert its influence to come up with a solution that is mutually acceptable to both development and QA teams.

One approach would be for QA to publish the expected turnaround times as “guidelines”, instead of terming them “SLAs”. The ageing of the defects would be measured and team-wise ageing reports would be published periodically against these guideline values as the benchmark. Though this approach would lack contractual or legal muscle, it can prove to be quite effective in improving the ageing because of the transparent and actionable information it provides to the program management.

9.3 Usage of Severity and Priority

Generally, the usage of severity and priority of defects is not very well understood by QA and by the other teams in the program. By industry-standard definitions, severity refers to the impact to business, and priority refers to the impact to the test team. In the context of a large program, confusion often is over which parameter should drive defect resolution by development team – should it be priority or severity. A recommended best practice would be as follows. In the initial stages of the testing phase, in the interest of early defect detection, QA team’s aim would be to hit as many test cases as possible in the shortest possible time. Here the priority of defects should drive the defect resolution. In other words, in the early stages of testing, it is more important to fix defects blocking big chunks of test cases, thus paving the way for more functionalities to be flexed early. At this stage, fixing defects that would have a large business impact in production but a minor testing impact, would be second priority. As testing approaches the exit gate of a test cycle or a test phase, severity and priority would converge, that is, high severity is what would be high priority as well. In other words, to exit a particular test cycle or test phase, it is more important to close the defects that would have bigger business impact.

10. Regression Model

Quoting standard QA text, regression testing is testing done to ensure that the code has not regressed due to the changes made. In large programs, this generic definition will require to be made more specific with respect to the scope, ownership and methodology of regression – which we will collectively refer to as the regression model.

10.1 Scope of Regression

When there are multiple releases occurring in a program, there are two different views of regression possible. The first view considers the scope of regression to be all functionalities up to and including the just previous release. The second view expands the scope of regression to include even the functionalities of the current release, to be regression tested post SIT and UAT defect fixes. While the second view isn't technically incorrect based on what point in time used as reference, it is uncommon. Regardless, the scope of regression – as excluding current release functionalities or including current release functionalities – will need to be spelt out at the outset.

A reason why the scope needs to be delineated as above is because the scope under the two views may fall under two different QA teams, which could be two different vendors in some cases. In large programs, due to the large size and scope of regression, it is normally handled by a separate team, which is not the SIT team. The "dedicated" regression team thus would be responsible for the scope under view #1, whereas the SIT team would be responsible for the scope under view #2.

10.2 Ownership of Regression

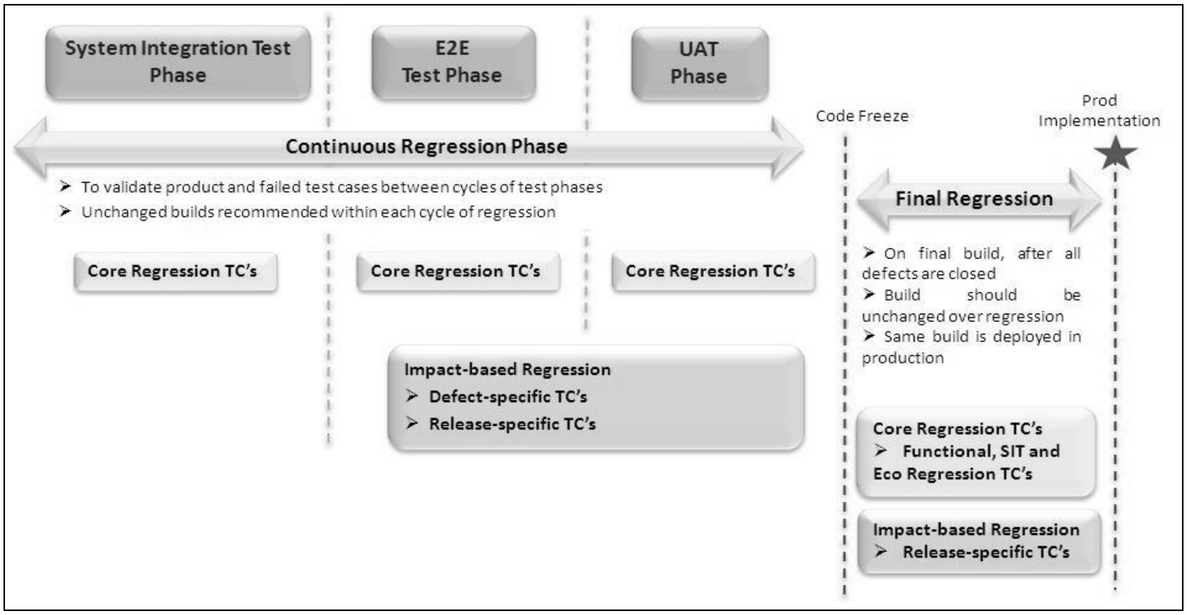
The above mentioned model is just one model of scope and ownership split. Under this model, there is surely more focus and coverage on regression, but at a proportionately higher cost, and sometimes with lesser agility due to knowledge transition required between SIT and regression teams for the newer functionalities.

There are alternate models possible, addressing the negatives mentioned above. For example, in some programs, there may not be a "dedicated" regression team. The SIT team themselves would handle regression responsibility. While this avoids the need for knowledge transition from SIT to regression team for newer functionalities, and results in cost efficiencies, the SIT priorities diminishing regression focus is a realistic risk to be weighed.

A hybrid of the above two models is the third alternative. The regression model thus is dictated by parameters such as focus, agility, cost and risk desired.

10.3 Methodology of Regression

In any project, best practice recommends that there ought to be a continuous regression phase and a final regression phase. Continuous regression phase runs parallel to SIT, UAT and Performance test phases. Final regression occurs after the SIT, UAT and Performance test phases have concluded, and a code freeze is enforced.


System Integration Test Phase
E2E Test Phase
UAT Phase
Code Freeze
Prod Implementation
Continuous Regression Phase
➢ To validate product and failed test cases between cycles of test phases
➢ Unchanged builds recommended within each cycle of regression
Final Regression
Core Regression TC's
Core Regression TC's
Core Regression TC's
➢ On final build, after all defects are closed
➢ Build should be unchanged over regression
➢ Same build is deployed in production
Impact-based Regression
➢ Defect-specific TC's
➢ Release-specific TC's
Core Regression TC's
➢ Functional, SIT and Eco Regression TC's
Impact-based Regression
➢ Release-specific TC's

The challenge is that in most programs code freeze remains idealistic, and never happens in reality, unless if it is a truly critical project. Due to time pressures SIT, UAT and Performance test phases slide and cut into the code freeze phase, in turn making final regression redundant.

The regression strategy in such a case will be to try and minimize the risk of late SIT, UAT or Performance defects. One approach here is to perform shorter impact-based regression cycles towards the end of regression. The impact-based cycles would focus on regression impacts of the SIT, UAT and Performance defects found during the intervals of regression.

11. Performance Testing

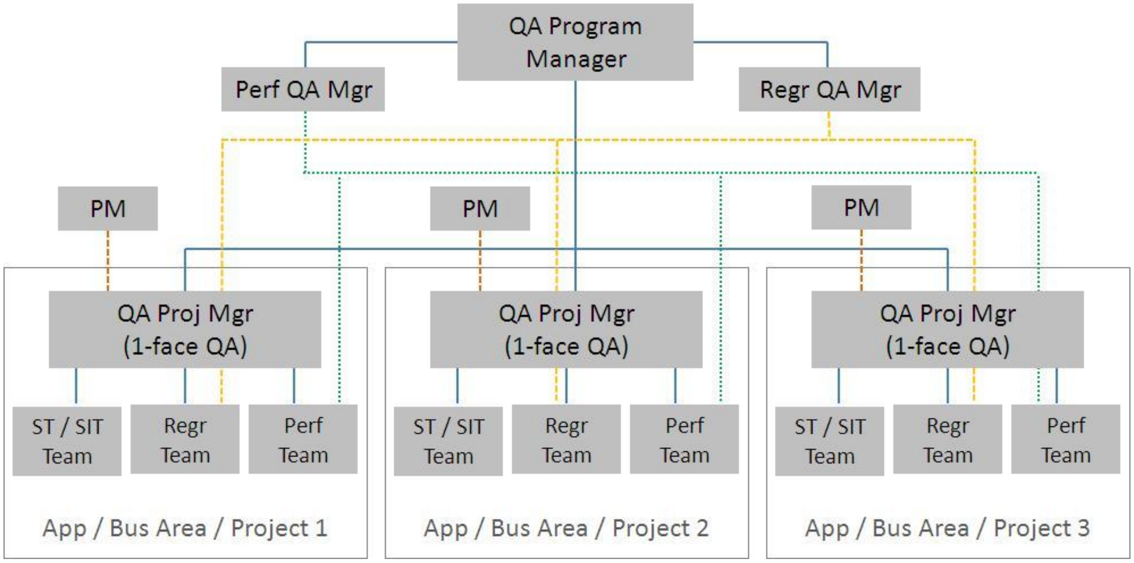
To wrap up this paper, we will now discuss the challenges associated with Performance testing in the context of a large program.

11.1 Integrating Performance

It is a well-known fact in QA circles that Performance testing is considered the poor cousin of Functional testing. Performance testing doesn't get the attention it deserves, at least until system testing and system integration testing are nearly completed, and Performance testing comes into the critical path of the project or release.

This can be a costly mistake in any project, and more so in a large program, because the large scale integration involved can compound the performance issues multifold. It requires Performance testing to be integrated fully with the program, just as the rest of QA is, right from the early phases. All the integration aspects mentioned in the earlier sections of this paper, namely, the org structure, interdependency meetings, integrated status reporting, test nomenclature, governance framework, environment planning and defect management would be as applicable to Performance testing as they are to the other test phases, although the environment requirements and test processes for functional and Performance testing are different.

To quote a successful best practice in this area, it is called the one-face-QA model. It is a specific org structure arrangement to ensure that the integration truly happens on the ground. In the model, for each project under the program, there will be a single overall QA manager, who would be accountable for the integration of all phases of testing happening in that project, even though different test phases could be the accountability of different teams or vendors. This will make sure that Performance does not get left out of the loop in any of the program changes or communications.



11.2 Performance Requirement Gathering

Yet another challenge confronting Performance testing in large programs is that performance requirement gathering process is often found to be rudimentary. The reasons are many. The relative lack of importance associated with Performance testing compared to Functional testing is surely a factor. There are inherent difficulties too. Performance requirements are harder to define, especially when the systems or their components are new, and thus lack historical data to provide a reference. Solution architects and business are unsure about the average, peak and breakpoints transactions loads and user volumes to be expected. Requirement definition in such cases turn into a trial-and-error, with the values being revised a few times based on the actual test results observed. This approach causes much rework and pushes the Performance test timelines very close to the deployment dates.

Performance requirement gathering process thus needs to be made more sophisticated. Workshops involving solution architects, business and performance engineers are a good way to avoid the uncertainties associated with understanding and defining of performance requirements. The workshops can be complemented by analysis of production performance data gathered through a pre-defined tool-based production monitoring process.

11.3 Application Performance vs. Operational Clock Performance

One thing unique to large scale integrations is that individual application performance is only half the battle. When several applications integrate, the end-to-end performance of the business process flows and batch job schedules (referred to as operational clock) are equally critical. The former does not automatically guarantee the latter. Operational clock performance requirements will have to be separately defined and tested for.

12. Conclusion

The paper is an elaboration of the unique challenges associated with administrating QA for large-scale system integration programs such as ERP implementations. The emphasis is on highlighting how the characteristics of a large IT program influence the application of QA processes and principles vis-à-vis small and medium IT projects.

Alongside the challenges, a number of the best practices and solutions have been discussed too, mostly based on the author's successful prior experience. However, the best practices and solutions are by not meant to be prescriptive. Each program is different, and there can be several alternate solutions possible for the challenges.

Several of the topics discussed would qualify as excellent materials for detailed analysis and study, and thus contribute to the expansion of the QA body of knowledge, particularly in the area of QA program management.

13. Glossary

DSR	Daily Status Report
ERP	Enterprise Resource Planning
ETVX	Entry Task Validation Exit
IPP	Integrated Program Plan
IT	Information Technology
PM	Project Manager
PMO	Project Management Office
QA	Quality Assurance
SDLC	Software Development Life Cycle
SIT	System Integration Test
SLA	Service Level Agreement
UAT	User Acceptance Test
WSR	Weekly Status Report

References

Black, Rex, and Dorothy, Graham. 2012. *Foundations of Software Testing: ISTQB Certification*.

Black, Rex. 2009. *Managing the Testing Process*.

Evolution of the Tester

Ben Williams

desertblade@gmail.com

Abstract

The Tester. Who are they, where did they come from, and where are they going?

Testers face different challenges than they did 10 years ago. The business and software landscape has changed dramatically. The modern-day tester needs to evolve or face extinction. Gone are the days of detailed requirements before starting development and of running regression tests after implementation. Often a new feature must be deployed as rapidly as possible, sometimes even the same day it was conceived.

The development world is quickly moving away from the waterfall model, replacing it with agile methodologies. Quality Assurance (QA) engineers find themselves trapped between the trusted but time-consuming ways of ensuring quality and the challenge of providing coverage in the fast-moving agile world. While most software developers have found their groove working in agile, quality engineers have struggled and find themselves lagging behind.

Automation has taken us a long way, but there are still shortcomings. Maintainability, reliability, and re-usability are issues faced in all automation suites. In many agile shops, there remains a high dependence on manual testing and drawn-out regressions, causing the all too common QA 'bottleneck.' The stereotype that QA slows down the process has created a tendency to limit the use or effectiveness of quality engineers or forgo testing completely; trading unnecessary quality risks for speed.

The pace of business has changed. In order to survive, QA must change with it. Reducing the need for manual regressions by expanding test automation, improving coverage reporting, simplifying manual regressions and providing continuous testing are solid first steps. Testers will also need to evolve their responsibilities to include understanding the product and end-goal; making sure that what they are working on is right for the end user and the company.

Biography

Ben Williams has over 10 years of experience working in QA. He started as a manual tester, spent time as a release/build manager and developer, and lead quality management teams for start-ups and Fortune 1000 companies. He is currently consulting as a Senior QA Project Lead for a Fortune 200 company, working in a cross-functional agile environment that relies on continuous integration and automation to provide faster quality turnaround.

Prior to his career in software development, Ben served in the US Army as a Combat Engineer. Along with demolition and bomb detonation he also learned the fundamentals of leadership, the importance of planning, and the ability to adapt to any given situation.

Ben has studied a number of disciplines including computer science, political science, psychology and law. He will graduate from George Fox University in December with a Bachelor of Science in Technology Management.

1. Introduction

A couple of years ago at Google's Test Automation Conference, in the opening keynote, “Test is Dead” Dr. James Whittaker said “If you could take a software tester from 1970 and move them forward in time to, say, 2010, they wouldn't have to learn anything new to test. That's a problem. That's a lack of innovation.” Dr. Whittaker’s statement bothered me. How could QA have changed so little over the past 40 years? His statement was a generalization, but he had a point.

Working in software quality for over a decade, I have experienced a large waterfall environment, a small agile start-up, and an agile development environment that still expected waterfall style processes for QA. While many software shops have embraced agile methodologies and continue to move forward, QA has had a difficult time adopting new methods. Today, the necessity and methods of testing are in question, resulting in our users and time lines paying the price. Testing is still just as important today as it was in 1970. Now is the time to reflect, learn from our past, and look for opportunities to change the future of quality through efficiency and innovation.

2. The Beginning

Software engineering first appeared in the late 1950s. Early on, applications were dependent upon the hardware they were created for, and were more designed than programed. In the late 1960s, hardware costs had fallen, but the cost of software development began to rise. Unfortunately many software projects did not scale, were over budget, late, or just plain failed. The “software crisis” arose out of poor performing applications. The idea of ensuring quality was born.

2.1 Software Development Life Cycle

The modern age of software testing began in the early 1970s. Dr. Winston Royce wrote, "Managing the Development of Large Software Systems", a paper highlighting his views on managing software development projects. The foundation of the waterfall model was based on this paper.

At first, almost all software projects used the waterfall model. Each phase of the model was a separate activity, requiring different teams to work on it. The phases had independent estimates and could not start until the application passed the previous phase.

Royce’s original waterfall phases included:

1. Requirements specification
2. Design
3. Construction
4. Integration
5. Testing and Debugging
6. Installation
7. Maintenance

There could be different phases, depending on the project. Typically testing did not start until the project reached the testing phase. At that time all the features of the application were fully specified, designed and programed. An official release candidate build was delivered to QA and testing began. The application would stay in this phase until testing was complete.

2.2 Testing Models

In their paper, “The Growth of Software Testing”, David Gelpering and Bill Hetzel classified software testing into different time periods. Each period had different goals and methods to define software quality.

Modern testing was built on a combination of these periods. We are now entering a new, not yet defined period.

2.2.1 Pre 1956 – Debugging oriented

The first software was dependent on the underlying hardware. The terms debugging and testing were interchangeable. Put simply, applications were written and then checked to ensure everything worked.

2.2.2 1957-1978 – Demonstration oriented

1957 was the first year that testing was distinguished from debugging by Charles Baker. Testing during this period was focused on problem-solving and demonstrating that the application behaved as expected.

2.2.3 1979-1982 – Destruction oriented

In his 1979 book, *The Art of Software Testing*, Glenford Myers defined testing as "...the process of executing an application with the intent in finding errors." In this period, the primary intent of testing was to cause fault in the application.

2.2.4 1983-1987 – Evaluation oriented

In 1983, the National Bureau of Standards published guidelines for federal information processing systems. The process integrated analysis, review and testing to evaluate the application.

2.2.5 1988-2001 – Prevention oriented

In 1985 a new methodology was written by the Institute of Electrical and Electronics Engineers (IEEE) unit, "Systematic Test and Evaluation Process" or STEP. It included test activities such as test planning, analysis, test design, and test plans with the goal of preventing defects.

2.3 Determining Quality

Manual test cases were—and are commonly still—the primary method in determining quality. Once product requirements were finalized and development began, QA started preparing test cases for the release. A handful of test case writers would crawl through all the product specifications. They would then create new tests that would validate new features and update any of the regression tests.

These tests were rolled into a test run that was used to validate the application. During the test phase, all the new feature tests cases, as well as the regression tests were executed. When all of the tests passed, the product was considered to have 'good quality'.

Sometimes requirements shifted during development for various reasons. This caused tests cases to become inaccurate. When executing the cases many testers would fail the case and file a defect. This created extra work in verifying the requirements and updating the test case. Another issue with executing tests was the expectation of following the steps verbatim. This limited the testers and resulted in missed bugs that were not on the specified testing path.

2.4 Waterfall Does Work

The waterfall model has been proven to work. It was the primary development method for over 30 years and many companies still depend on it.

If you work on projects with limited configurations, that are difficult to update (remember floppy disks?), or your industry requires checkpoints before moving into the next phase, then using a phased model like waterfall is the right approach. Some industries and companies will continue to use the waterfall model well into the future. Many software development projects today, however, do not fall into any of these scenarios. Waterfall is not always the right model.

3. The Evolution

We have entered into a new period in testing that is still being defined. Since 2001 the industry has begun shifting to new methods for development and different tools have appeared to help with testing.

3.1 Introducing Agile

The idea of incremental software releases can be traced back to the beginning of software development. Starting in the 1990s there were a few iterative methodologies proposed, such as eXtreme Programming (XP) and Scrum. In February 2001, 17 software developers met to discuss software development methods. They came up with the Agile Manifesto, which provided a method for a lightweight software development cycle to deliver software quicker than waterfall. Since then, multiple methodologies are now grouped under the agile method, including Scrum, Kanban, XP, and Lean.

These agile methods allow shorter delivery times and the ability to adjust to changing business demands. Developers and product managers have eagerly adopted this new approach to develop software. Instead of three, six, or 12-month turnaround times, agile cycles are typically two to four weeks. This shift to releasing more frequently means QA does not have the luxury of an independent test phase.

3.2 User Stories

Requirements no longer live in large unwieldy documents that are read once and forgotten. They are now captured in user stories, which are brief descriptions of a feature used to inform development. The story size is estimated to help the business understand the cost of the feature. Once the cost is estimated, the business can weigh the feature story against other stories to determine if it delivers appropriate return on investment.

The story should have acceptance criteria. It is usually captured in the story template, for example, "As a *role,* I want *goal* so that *benefit*" or as a “Done When.” User stories can be business-facing tests that address business requirements. Ideally each user story becomes an automated acceptance test, feeding into an automated regression suite that can run on a regular basis.

3.3 Exploding Configurations

Initially, most software testing was targeted to a specific hardware platform, for example, a mainframe computer. This limited the configuration matrix to a single axis. A tester only needed to test the application for the hardware platform it was designed to run on.

Then, in the 1990s, the home PC market exploded, with Microsoft-based operating systems leading the way. By the year 2000 there were five Windows lines; Windows 95, Windows 98, Windows 2000, Windows ME, and Windows NT. At that time Microsoft targeted their operating systems to the intended user; business or personal. Businesses mostly worked on Windows NT, with some migrating to Windows 2000. For personal PCs, Windows 95/98, both MSDOS based, were popular and Windows ME had a sliver of market share.

In 2001, Microsoft released Windows XP, with two flavors, Home and Pro. These operating systems were not MSDOS based and had slightly different targets. Pro was primarily focused for business use. Home was clearly targeted for personal use. Pro did include advanced networking, which attracted some personal users for in-home networks.

The Windows Vista release in 2007 was the first mainstream release that included both 32-bit and 64-bit versions. In 2009, with the release of Windows 7, 64-bit installs became more commonplace. The last release, Windows 8 in 2012, included the revamped interface called Metro and changes to the operating system that impacted the behavior of many existing applications.

Along with operating systems, Internet Browsers were also evolving. In the early 2000s, Internet Explorer accounted for 85% of all browsers, with Netscape Classic rounding out the last 15% (and declining.) At that time, testing a web-based product meant there were only one or two configurations to consider. Most sites where optimized for the more popular browser, Internet Explorer 6. Internet Explorer 6 was so popular that it took over 10 years for its usage to significantly decline.

Around 2005, alternatives to Internet Explorer started to gain popularity. By the end of that year, Firefox and Safari had gained noticeable share, at 23.6% and 1.56% respectively. Google Chrome, released late in 2008, rapidly became a popular browser. That was about the time Microsoft was releasing newer versions of Internet Explorers including 7, 8, 9 and 10. While Firefox, Safari and Chrome would auto-update, Internet Explorer required a manual update, causing many users to stay on the browser that came pre-installed on their PC.

In 2007 the modern era of mobile platforms was introduced with the iPhone, followed by Android, Windows Mobile, and Blackberry. All the devices supported native applications and used mobile-based web browsers.

All of the change occurring in a span of less than 10 years has created complexity for testing. Whether you are testing a native application or web-based application, there are now multiple configurations to consider. Today, we have six desktop browsers with over 4% market share, multiple mobile browsers, and many shops still support a few of the outdated browsers (looking at you IE6 and IE7).

Web-based testing can easily require over 10 different browser combinations that need to be tested with each release. Application testing can have six or more combinations. The idea of doing a full regression across all of the supported configurations can significantly impact time and resources.

And we haven't even covered the explosion of locales and languages. Today we work in a global economy meaning many applications support multiple regions. Each supported region adds another configuration that requires validation.

3.4 Manual and Automated Test Cases

In the last several years, we have seen an increase in automation test frameworks. Not all testers know or want to learn programming so many of the frameworks offer record and playback functionality to create automated test scripts. Unfortunately record and playback automation is far from what the vendor promises. While the tests are easy to create and will probably work, they are not typically reusable and if anything changes, they will have to be re-recorded. Record and playback does offer some advantages particularly for quick short-term use testing and as a stepping stone for detailed scripting.

Not all testing is suitable for testing through automation. Some testing, such as exploratory and user acceptance, are better suited for manual testing. Other testing, including component testing, API testing or performance testing, are easier with automation or specialized tools. Knowing what is suitable for automation and what is better to test manually will help testers choose the right approach and resources to test efficiently and sufficiently. (Crispin and Gregory 2009)

Running a full manual regression test on a release gives the team confidence that everything is working as expected and any changes introduced did not adversely impact existing behavior. Full manual regressions, however, hamper the team's ability to release quickly or make changes during the testing phase. When testing new functionality, those test cases may be added to the regression suite and over time the tests become unmanageable, timely to update and difficult to trace.

3.5 Quality Efficiency

Testing alone is not sufficient to ensure a quality product. Most individual testing methods, which include manual testing, unit testing and regression testing, have only a 35% effective rate at catching defects. If testing is the sole preventative measure for a large scale project, it will seldom top 80% of defect removal. A combination of quality methods, including pre-test methods such as static analysis for defect detection, can bring defect removal rates up to 95%. This is the difference between good quality and mediocre quality. The later you catch a bug in the process, the more costly it is to fix. It is expensive in both people hours and project time. (Jones, Subramanyam, and Bonsignour 2011)

With so much focus on testing, and the cost of catching a bug late in the process, we are inadvertently creating the dreaded QA bottleneck that slows the momentum of agile development. Even working in an agile environment with daily builds, manual regressions begin just before releasing to a production environment. If a defect is raised at that point, it may be in code written early on, be more complicated to fix, and have a negative impact on the release date and ultimately on the end user.

3.6 Typical Mistakes

3.6.1 Agile is not mini waterfall

Attempting to condense a waterfall workflow into a two-week sprint is not agile; it is a shorter waterfall. An example of this is having requirements ready before the sprint, developing the stories for the majority of the sprint, and delivering a build to QA for testing at the end of sprint. This leaves little time to meet the sprint commitment. With so much quality work packed into so little time, quality and efficiency suffer and necessary testing slips to the next sprint. This can lead to a backlog of features and defects. As a result, the final release is large and unmanageable, ultimately impacting the team's ability to release on time. (Crispin and Gregory 2009)

3.6.2 Lack of (any) documentation

The Agile Manifesto states: “Working software over comprehensive documentation.” Unfortunately this is usually interpreted, “Working software and no documentation.” A lack of documentation makes it difficult to verify functionality and reference changes. On small projects, personal interactions may be enough, but on larger agile projects the tickets are an integral part of interaction. Informative tickets will reduce confusion and time spent filling in missing details.

3.6.3 Quality ownership

Who is responsible for quality? Everybody.

I have worked on numerous projects at a number of companies where members of the team did not know how to use the software they were developing. Everyone should test. Testing provides insight into how the customers and end users see and interact with the finished product.

3.6.4 Loss of traceability or coverage

In an agile environment, comprehensive or exhaustive documentation should not be necessary and often does not exist. But, many organizations still rely on comprehensive and exhaustive test plans in a test case management tool, like Quality Center, to provide regression coverage. Manual testing is already taking too long; maintaining test plans only increases project cost.

Taking a paragraph-length user story and turning it into multiple, large test cases defeat the goal of working in agile to begin with. That time could be spent in other, more valuable areas, like actual product testing and creating test automation. Since automation tests are reusable, focusing on them instead of manual testing will pay for itself in the long run. (Crispin and Gregory 2009)

Since the software world has started the shift to agile, many new tools now exist to help track the new workflows. Unfortunately, to date, there has not been a revolution for creating a traceability matrix or test cases. Nearly all requirement coverage and test case tools are still based on traditional waterfall methods to manage quality.

4. The Approaching Stage

Quality has come a long way over the last 10 years. Shorter development cycles are proven to return higher quality in comparison to longer-running waterfall projects. The previous transitional period, however, has left resource, process, and documentation gaps that prevent teams from delivering high-quality projects within an agile timeline.

4.1 Determine Coverage

In an agile environment, software change occurs on a consistent basis. Compounded with a reduced tester-to-developer ratio, there are more people introducing change and less people validating it. In this scenario, it is virtually impossible for teams to successfully execute all of the changes introduced into an application.

4.1.1 Traceability matrix

To move forward, we need a more robust traceability matrix to determine coverage. It needs to be lightweight, integrated into ticketing systems, and create and track automated and manual test runs. The matrix could align to user stories, incorporating newly defined capabilities and the existing capabilities of the application.

Google practices the ACC model for rapid test planning and requirements tracking.

> **Attributes**: What types of behavior the application has. Examples are Fast or Secure.
>
> **Component**: The different components of the systems.
>
> **Capability**: Intersect of attributes and components, and lists the expected functionality in that intersection. With each capability there is an assigned risk factor that will determine the priority of testing.

With the list of capabilities combining attributes and components, we can tie automated and manual testing to cover each individual capability. We can identify the high-risk, high-value testing areas to prioritize testing and automation efforts. The capability can be validated through different unit testing, automated testing, or through a scenario test case. (Whittaker 2012)

ACC is only one example of a process for identifying capabilities of applications. The ability to tie tests back to an individual capability allows the tester to calculate test coverage. By tracking the risk, testing high risk areas first, we can prioritize and reduce risk in each release. Parsing the application into consumable test focus areas (the capabilities) means we can create tests that validate each capability.

4.1.2 Manual test cases

In a perfect world, all regression tests would be automated. We don't live in a perfect world. That means keeping manual tests cases light reduces the need for constant updating and duplicate testing. If the tests are too detailed, testers will spend more time writing and maintaining the test cases than actually testing the application. Step by step test cases also typically keep testers on a specific path, limiting exploratory testing.

There are tests scenarios in which automation is not ideal, for example, usability testing. Part of usability testing is looking at the page across supported browsers. Having a lightweight test case will result in quicker execution and allow the tester more time for exploratory testing. With the multiple configurations that need to be tested, creating an execution matrix will give traceability and insight to what was tested.

4.1.3 Cross-functional teams

When building a cross-functional team you look for specialized skills. There may be a product owner, a UX designer, front-end developers and back-end developers. Test engineers bring valuable skills and a quality mindset to the table. They will watch builds, run automation, provide test environments, and work closely with each member of the team; always in the mindset of making the best product possible.

Requirements can introduce 20% of all the software defects. (Jones, Subramanyam, and Bonsignour 2011) Making assumptions about how the software works, or not having fully considered all of the possible scenarios, will create unwanted design defects that are usually discovered during testing. Test engineers tend to be more familiar with the product, or type of product, than anyone else on the team. Having them present during design and planning will help expose design defects before they are implemented. This will help the team avoid unnecessary rework efforts.

4.1.4 Quality at all times

Working in agile, continuous integration can mean the difference between having a great product or having no product. A continuous integration server needs to build every time there is code committed into the repository. With each commit of the code it needs to be scanned for language guidelines (linted) and have static analysis performed to detect potential coding issues. Faster running tests, like unit tests, should be executed with every build. Slower tests, like automated regressions, may run once a day during off hours, to be reviewed first thing the next morning.

Since every code change is built and deployed on a regular cadence, test engineers will monitor the builds and automation, providing immediate feedback to developers. Many regressions defects are found less than 24 hours after the offending code is introduced. When a defect is discovered, the team should create a test to verify it and prevent it from happening again. Ideally this will be as close to the actual cause of the defect as possible, such as a unit test of a component or function. (Jones, Subramanyam, and Bonsignour 2011)

Reducing the feedback loop will allow issues to be fixed faster. The status of the build and testing should be made public, and a passing build needs to be the top priority of the team. Having the application in an always-releasable state allows you to release anytime, without waiting for regressions to run, and gives you confidence that the application will work as expected.

4.1.5 Maintainable/reusable automation

Having a robust and easily-maintained system is vital to working in an agile environment. Automation should always pass, unless there is a defect. Time spent digging through logs or manually checking failed tests negates the value of automation. There should be logic in place to address instability that is common with certain types of testing. That might include adding hooks on the web page to signify loading, or running a two-out-of-three pass/fail test.

Far too much cross-browser testing is done manually. Tools like webdriver can run across all browsers, but they only test functionality. This means the web page might render incorrectly and automation will not discover any issues. A page that does not render correctly is a serious issue. The ability to test for this through automation would allow greater coverage in less time. Comparing the layout of browsers can be integrated into the automation with image-compare programs like Sikuli and ImageMagick. There are issues with this approach, including maintaining the browser-rendered images, but it can be useful to validate page layouts quickly.

4.2 Skills in Demand

4.2.1 Open source

If you ever work in a start-up or with cutting edge technology, you will most likely be working in Open Source. The hottest automation technologies (Cucumber, Watir, Selenium, Robot Framework, Sikuli) are all Open Source projects. They may take a little longer to incorporate, but their capabilities usually surpass that of expensive alternatives.

4.2.2 SQL

Companies today are investing in data mining, data gathering, and other metrics to inform and make key business decisions. The need to validate raw data is increasing. For the next generation of QA, the ability to read and create SQL queries is vital.

4.2.3 Object-Oriented (OO) languages

Record and playback automation, and basic scripting automation are not sufficient for complicated, modern applications. Full-fledged programing languages can be used to make automation reusable and maintainable, and should be Object Oriented. Common OO programming languages used for automation are Ruby, Java, and Python. Testers should seek out opportunities to learn and use these languages. There are excellent online resources as well as traditional college courses.

4.2.4 Unit test writing

The closer the tests can live to the actual code the better. Unit tests are able to provide coverage, testing a small portion of the code, and should be implemented with every code change. A tester with unit test writing can provide additional coverage while learning more about how the application works.

4.2.5 Web technologies

It is important to have an understanding of all the pieces required by a web application. Knowing the difference between a JavaSript error and a CSS error means faster troubleshooting, well-written tests, and the ability to provide accurate details for defects.

4.2.6 Virtualization

The QA environment used to include physical hardware supporting all of the configurations needed to test. In today’s QA, with many configurations required to test, the only sensible solution is virtualization. This cuts hardware costs, enables environments to change quickly and allows multiple environments to run side by side. There are many different virtualization solutions, in varying price ranges, with local or cloud accessibility. For popular platforms, Microsoft offers test virtual machines for its browsers. (http://www.modern.ie)

4.2.7 Mobile

Mobile devices are commonplace today. This year, it is expected that Internet connected mobile devices will surpass the number of connected desktops and laptops. (Standage) From native applications to mobile-specific web pages, mobile is an important segment that cannot be ignored. The tool Appium (http://appium.io), based off of webdriver, recently emerged for automated mobile application testing.

4.2.8 Social media

Everyone is on it and everyone wants a piece of it. People working on a given project should be familiar with applicable social media sites, understand the difference between the sites, and have a basic understanding of how the sites work. Many websites now incorporate social logins and social sharing, both of which require testing.

5. The Next Generation

We have witnessed a significant transformation over the last 10 years and there is much more to come in the next 10. The industry is constantly changing, with new methods and tools. Looking forward, there are a few changes I feel confident we will see.

5.1 The testers

The ability to design and build reusable automation in a continuous integration environment allows companies to deliver software faster. The demand for purely black-box, manual testers will continue to decrease, replaced by a new, hybrid tester who can implement automation, has programming know-how, and is cognizant of the end-user experience. With the typical agile team size of five to eight, the hybrid tester will also need critical thinking and leadership capabilities to advocate for the quality of the application.

5.2 The leadership

As companies move towards cross-functional teams, QA will become more embedded in the development team. As that happens, the need for a dedicated QA team will slowly decrease and eventually cease to exist. This will reduce the need for QA managers, placing the onus on individual QA leads to represent and promote testing as part of the development team. Larger organizations, with QA testers spread over multiple development teams, may implement a QA Center of Excellence (COE) to span all of QA, creating efficiencies, training and best practices. The QA COE will be responsible for the overall QA strategy and automation tool sets.

5.3 The tools

Web automation is becoming robust, with the ability to provide excellent coverage. All major browsers now support webdriver making it easy to write cross-browser automation. Verifying web page layout across browsers will eventually be simplified through unification of the rendering engines, better web standards and better comparison tools. As we hone and innovate in this area, I foresee the next phase of web automation as self-testing web pages. Some of the testing could be built in, like link checking, or if an image appear, which would not require any test setup. More complicated tests may become test code, embedded on the web page or in a separate file, which the browser can execute to test and provide results on the spot. Each code change will be instantly tested with potential issues known immediately. This will allow for rapid defect detection and resolution resulting in faster development.

6. Conclusion

Software development is an ever-changing field. The constant introduction of new technologies is the new norm. In an industry that is constantly evolving, the test engineer must evolve with it or fade away. Many start-ups today forgo hiring testers, relying (hopefully) on other methods to ensure quality. Many of these organizations are beginning to realize testing is a unique and valuable skill set. The biggest challenge testers face today is the expectation of high-quality output in an relatively short period of time. In the agile environment, manual testing alone is not a viable option.

Over the past 40+ years software quality methods have changed. Shorter development cycles, methods on discovering defects, and automation have reshaped how testers work. Unfortunately we are still holding on to some legacy methods that aren’t working, or aren’t working well. By doing so it holds back innovations for quality. We are entering a new period for software quality. Some of the old ways and tools used in testing no longer apply. I believe it is important for us to understand our roots, learn from past experiences, and find new ways to move forward. Software teams, including test engineers, need to redefine how they determine quality and create new methods to help them achieve it.

7. References

Savoia, Alberto and James Whittaker. "Test is Dead" Recorded October 26 2011. Google Test Automation Conference. Web, http://www.youtube.com/watch?v=X1jWe5rOu3g.

Royce, Winston. "Managing the Development of Large Software Systems" *Technical Papers of Western Electronic Show and Convention* (WesCon) August 25-28, 1970, Los Angeles, USA. http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf (accessed August 3, 2013).

Gelpering, David, and Bill Hetzel. "The Growth of Software Testing." *Communications of the ACM*. no. June (1988): 687-695. http://clearspecs.com/joomla15/downloads/ClearSpecs16V01_GrowthOfSoftwareTest.pdf (accessed August 3, 2013).

Jones, Capers, and Olivier Bonsignour. *The economics of software quality*. Upper Saddle River, NJ: Addison-Wesley, 2011.

Crispin, Lisa, and Janet Gregory. *Agile testing : a practical guide for testers and agile teams*. Upper Saddle River, NJ: Addison-Wesley, 2009.

Whittaker, James A., Jason Arbon, and Jeff Carollo. *How Google tests software*. Upper Saddle River, NJ: Addison-Wesley, 2012.

Standage, Tom. "Business: Live and unplugged | The Economist." *The Economist - World News, Politics, Economics, Business & Finance*. N.p., n.d. Web. 10 Aug. 2013. <http://www.economist.com/news/21566417-2013-internet-will-become-mostly-mobile-medium-who-will-be-winners-and-losers-live-and>.

"Browser Statistics." *W3Schools Online Web Tutorials*. N.p., n.d. Web. 10 Aug. 2013. <http://www.w3schools.com/browsers/browsers_stats.asp>.

Value Oriented IT Process Design and Definition: Integrating Customer Value into Process Quality

Shivanand Parappa and Mangesh Khunte

shivanand.parappa@cognizant.com , mangesh.khunte@cognizant.com

Abstract

Today's competitive environment and increased customer demands require companies to respond quickly and effectively to market opportunities. Business organizations are leveraging Information Technology (IT) solutions as a means to accomplish the business objectives and to fulfill the growing customer demands. As business always expects significantly higher return and value on investment made for IT, it is critical for IT division to demonstrate the quality and value added deliverables. Hence, it is extremely important to understand the concept of value from a process design and definition perspective while formulating IT processes that create and deliver value to the business and customers.

This paper details a framework named Value Oriented Process Paradigm (VOPP) that provides adoptable guidance for designing value oriented IT processes and integrating customer value with process quality. This proven framework has been developed by leveraging the lean concepts and value creating techniques. This paper focuses on value analysis in line with Dr. Genichi Taguchi's loss function theory and opportunities for creation of value utilizing the lean principles while defining the processes. This paper details a technique named Potential Waste and Impact Analysis (PWIA) that provides guidance towards identifying potential waste during process definition and prioritizing them to put in place controls to reduce or eliminate them. The approach and concepts detailed in this paper would be useful for process consultants involved in process design and definition.

Biography

Shivanand Parappa is a Senior Consultant at Cognizant Technology Solution, currently working with Process and Quality Consultancy team. He has over 16 years of experience in Manufacturing / Software Quality Management and Consultancy. He has extensively worked on driving process improvement projects through Six Sigma and Lean methodologies to deliver tangible results. Shivanand holds a Bachelor's degree in Industrial Engineering and a Master's degree in Production Engineering. He is a certified professional with Six Sigma Black Belt (DMAIC and DFSS), PMP and PRINCE2 Practitioner.

Mangesh Khunte is a Senior Manager – Consulting at Cognizant Technology Solution, currently working with Process and Quality Consultancy team. He has a total professional experience of more than 15 years across process consulting and project delivery areas. He has considerable experience and proven expertise in process consulting, interfacing with clients to get business, perform business analysis. Mangesh holds a Bachelor's degree in Electronics and Telecommunications and a Master's degree in Marketing. He is a PMP certified Project Manager and PMI-ACP$^{SM.}$

1. Introduction

Business organizations are leveraging Information Technology (IT) solutions as a means to accomplish the business objectives and to fulfill the growing customer demands. Business needs are formulated based on the customer requirements that come from customers in terms of their business needs. These needs are fulfilled by providing value add solutions through appropriate business and IT processes. The business and customer requirements are triggers for producing IT solutions that enable business to achieve its objectives and customer satisfaction. Timely delivery of IT solutions assists in enhancing business efficiency, improving quality of services and products and increasing the value provided to the customer.

Process can help an organization's workforce to meet business objectives by helping them to work smarter, not harder, and with improved consistency *[CMMI-DEV v1.3 2010]*. Processes are critical entities in an organization and its operational success is tied with the effective performance of the processes. One of the never ending challenges for an organization is to act on customer requirements and translate them into product or service with agreed quality attributes at low cost. The actions involved in conversion of the requirements to products/services are enabled and supported by well-defined processes coupled with adequate resources. Processes can be grouped and classified into: core; enabling; governance; or support processes. This classification is only to bring the distinctions in the organization process asset instead of indicating its priority or ranking. Each of the organizational processes has equal importance with respect to achieving the objectives and goals of the organization *[Shivanand Parappa 2012]*.

2. Value Definition and Analysis

2.1 Value versus Quality

The perception of value varies based on the customer's needs and changes in their business objectives. Also, we can see in the Kano model (a tool used for categorizing the customer needs into basic, performance and delighters) *[Basem and David 2005]* that the basic and performance needs are more sensitive to the value as compared to the delighter needs. Customers expect more value from the basic and performance needs. Value is always defined from the customer's perspective. The customer value perception can be derived from the requirement attributes like speed & responsiveness, accuracy & quality, flexibility, capacity, compliance, stability & reliability, availability, security & risk, etc. From the Lean concept perspective, value is "a capability provided to a customer at the right time at an appropriate price, as defined in each case by the customer." If we take a closer look at the value attributes, one can visualize that quality is an inherent characteristic within the value attributes. For example, the value perception against the requirement attribute 'capacity' in a service provider company will have a negative impact on quality of service if the service quality character fails in terms of not providing services within the agreed SLA.

On the other hand, the quality is defined as "Degree to which a set of inherent characteristics fulfills requirements" *[ISO-9000:2005]*. In this definition: degree refers to a level to which a product or service satisfies the requirements, inherent characteristics are those features that are a part of the product and are responsible to achieve satisfaction, and requirements refer to the needs of customer, needs of organization & those of other interested parties (e.g. regulatory bodies, suppliers, employees, community & environment). The quality of products or services can be determined by comparing a set of inherent characteristics with a set of requirements. The quality depends on a set of inherent characteristics and a set of requirements and how well the former complies with the latter.

It is apparent that it is the customer who defines the value of a product or service by making an analysis of the performance of the product/service relative to its cost. On the other hand, the quality of a product or service is always in the hands of the organization, governed by the processes that are adopted by organization and it depends upon the ability of an organization to provide a product or service that gives a performance that customer is seeking.

2.2 Understanding the Process Value Creation

In order to produce quality and value added product or service, it is necessary to define the value oriented processes that enable the value added outputs with desired quality. So what is the qualification of the value oriented process? A value oriented process should produce the required outputs, without defects, as efficiently as possible, and at the right time *[Dr. Hugh L. McManus 2005]*. Value oriented processes should enable the organizations to provide value to the internal and external customers and to its stakeholders as well. To ensure this, value must be understood in two different contexts: the value of the process output to the entire organization and the creation of that value during carrying out the individual sub-processes/tasks that make up the process.

In an organization, the processes are organized based on the core operations, support functions and line of businesses. These processes collectively produce the outputs to meet the customer requirements. Customer requirements and expectations are gathered from market survey, research, and customer interactions and converted into products or services with the specified attributes to be provided back to the customers. During the formulation of customer requirements, the customer value/perception is also captured and carried all along the conversion process. Each of the processes that are involved in producing the services or products are responsible for ensuring that the customer value is integrated with the process outputs along with specified quality. The value is created along with quality outputs during the execution of the processes and transformed to the downstream processes for their consumption and to enable them to add value to their outputs and subsequently the final product.

The next obvious question is how to define, visualize, create the process value and integrate with process quality. Let us consider a hypothetical business organization where different processes are executed in different areas and functions to produce and deliver the products or services to the customers as per the specified quality. The following diagram (Figure 1) depicts a model for process value questions at every phase of the product or service realization stages *[Dr. Hugh L. McManus 2005].*

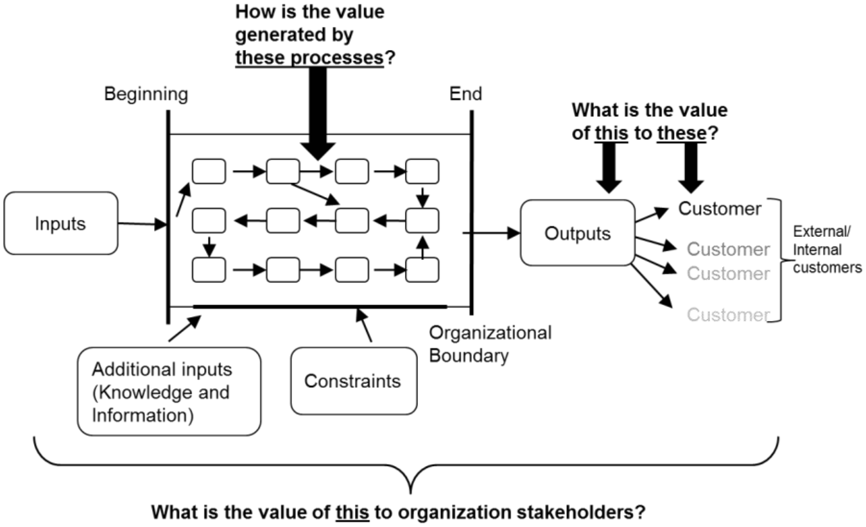

How is the value generated by these processes?
Beginning
End
What is the value of this to these?
Inputs
Outputs
Customer
Customer
Customer
Customer
External/ Internal customers
Organizational Boundary
Additional inputs (Knowledge and Information)
Constraints
What is the value of this to organization stakeholders?

Figure 1. Process Value Question model

From the above diagram, it is evident that the process value has to be questioned and ensured that it is being carried along during process execution, in intermediate process outputs and in the final outputs as well. Holistically, the entire system of processes needs to be designed so that value is created at each process stage and process output quality (for both intermediate and final outputs) is achieved.

2.3 Sources of Process Value Loss

From the Lean perspective, the process value is significantly impacted by the waste generated during individual process execution and this subsequently leads to a greater value loss in the entire enterprise value chain. In order to achieve the maximum value for the process, the waste has to be identified and eliminated from the process in all of its forms during process definition. The seven forms of waste sources in the IT process ecosystem have been tabulated in the following table (Figure 2). These wastes have to be analyzed, addressed and eliminated through appropriate process controls during defining and designing the IT processes.

Type of Waste	Examples in IT
1. Over production Adding extra features to functionalities.	While designing and developing an application, adding more features which are not asked by customer/requester. For example, in search functionality, including extra criteria/filters for search function in addition to what customer asked for. This leads to more test cases and requires more testing effort.
2. Over processing Performing extra processing steps	During application development through SDLC, performing extra processes like multiple reviews, checks, etc. Examples of extra processes include all kinds of requirements, analysis, design, and test documents that no-one will ever read, the endless meetings with more people than are really needed.
3. Defects Defects not caught by reviews/tests	Defects not captured in early stages due to ineffective reviews and testing. The longer a defect is allowed to stay in the system, the more it costs to fix it. Also, defects leads to do a costly recall, repeat a lot of manual testing, revise the documentation, etc.
4. Motion Effort in finding information	Spending more time and effort in getting/ finding required information and clarification. Searching of the documents. It occurs when we need to go somewhere else to get information/clarifications- if the customer or an expert is not at hand when we need answers to questions or if the team is spread out over multiple offices, floors, buildings or even time-zones.
5. Waiting Waiting for resources, information, approvals. Mismatch between the in-flow of tasks and the production capability	Most of the features to be included in an application/ software spend their lives sitting in queues, waiting for the next Change Control Board meeting, living in Requirements documents, waiting in Analysis documents before being implemented, waiting for system integration, waiting for the testers to test them, or waiting in a bug tracking system to be corrected or waiting for the next bi-annual deployment window.
6. Transportation Too many handoffs related to work and information	Handoffs can be identified as assigning people to multiple projects which is a source of waste. Every time software developers switch between tasks, a significant switching time is incurred as they get their thoughts gathered and get into the flow of the new tasks. Anytime you hand a deliverable off to a different party, there is some loss in the transfer of knowledge 1. Developer hands off to another developer. 2. Developers hand off code to testers to test 3. Handing the code over to deployment teams 4. Handing off to customers.
7. Inventory Partially done work	In software development, inventory is anything that you've started and you haven't gotten done. It's "partially done" work. Partially done work is essentially work-in-progress. Until this work is done, you don't know that there are quality issues i.e. you don't know that the customer will be happy. So the idea should be to complete work-in-progress as soon as possible i.e. minimize work-in-progress as much as possible. 1. Code that is completed but not checked in to your version control system 2. Undocumented code 3. Untested code (both unit tests and functional tests)

Figure 2. Seven Forms of Waste in Information Technology

2.4 IT Process Value Analysis

Before moving to process value analysis, let us understand how the customer value can be visualized and mapped with IT processes. Let us consider a hypothetical software service organization where the IT solutions are being provided the customers to meet their business needs and run the business. The customer business needs and software requirements are captured in terms of functional and non-functional requirements by the software solution provider organization. These requirements are processed through various processes and converted into useful software as per the customer specifications. The following Figure 3 depicts the schematic diagram of value attributes that need to be addressed during execution of the processes.

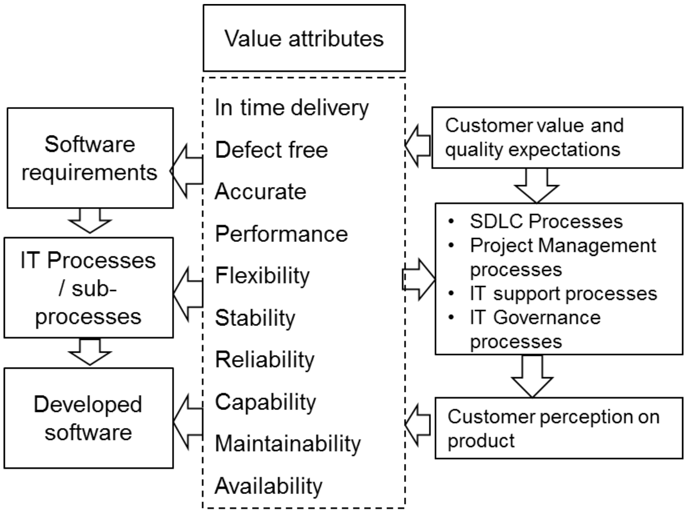


Figure 3. IT Process Value Attributes

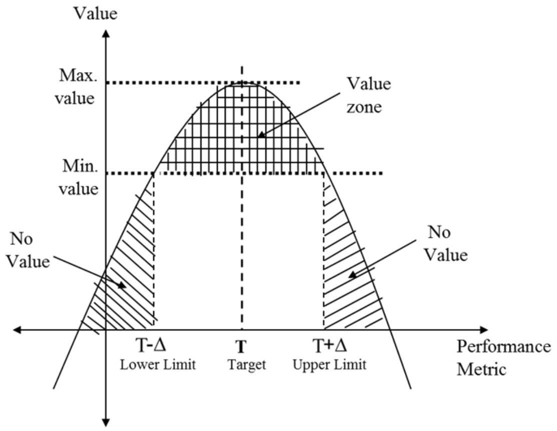
During the process execution, it is necessary to measure the performance of the processes through appropriate performance metrics to ensure quality and value of the process outputs. For example, during execution of testing process, the performance of the test case preparation sub-process must be measured in terms of appropriate metrics like schedule, review defects and productivity.

According to Dr. Genichi Taguchi's quality loss function theory *[James and William, 2008]*, any deviation from a quality characteristic's target level results in a loss; a higher quality characteristic measurement is one that results in minimal variation from the target level. Specifically, if a quality characteristic measurement is the same as the target value, the loss is zero, otherwise there exists value loss that can be measured using a quadratic function. Extension of this theory to process performance characteristics of a process provides an insight on process value analysis. That means a process or sub-process that is responsible for achieving product quality results in process value loss if it fails to perform at its target level of performance metric. Specifically, in order to attain maximum process output value, it has to be executed with zero deviation from target value.

In order to achieve maximum process value, the process should perform at its target level of identified performance metrics. Any deviation of performance metric from its target level indicates the existence of factors impacting the output value. These factors are due to the presence of waste in the process that hampers the process execution and there by impacts the process output value.

Let us consider the scenarios where the process performance targets are different for different processes depending on the process output objectives. The following figures (Figures 4 to 6) depict various scenarios in which the value zones are positioned with respect to the different locations of performance metrics. The 'X' axis represents the performance metric location and the 'Y' axis represents the location of process value against the corresponding performance metric. As the performance metric deviates from the target, the process value start decreasing and reaches to minimum value at the given limits and beyond the limits results into total value loss. This value loss is due to the presence of process waste that impacts the performance and quality of process outputs. When plotted the locus of deviation points in a Cartesian coordinate plane, the resulting graph is a parabola (quadratic function), with maximum value when the performance metric is equal to the target '**T**'.

Scenario 1: Target is the Best



Examples of performance metrics in this scenario are: 1) Resource Load Factor (Percentage)
Target = 80%, Lower Limit= 71%, Upper Limit= 89%
2) Test case preparation productivity (Test cases per person day)
Target = 48, Lower Limit= 37, Upper Limit= 59

In this scenario, the process has both upper (**T**+Δ) and lower (**T** - Δ) limits for performance metric with the target **T**.

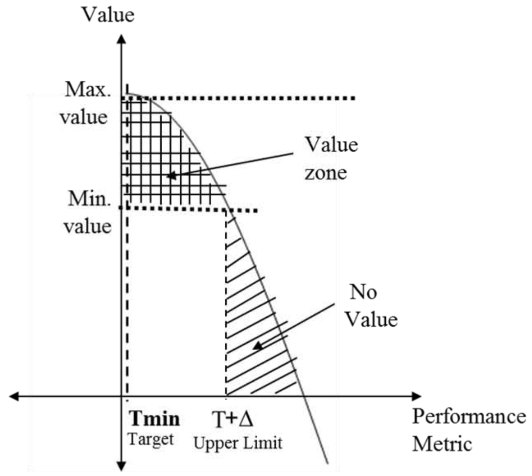
- If the performance metric is at target **T**, the process produces maximum value.
- The value decreases as the performance metric deviates from **T** and moves towards the limits either to **T**+Δ or **T** - Δ.
- Beyond these limits the process doesn't produce any value.

The value zone can be expressed in terms of a theoretical equation as shown below.

$$\text{Value Zone} = F(x) = \int_{T-\Delta}^{T+\Delta} f(t)\,dt = \int_{T-\Delta}^{T} f(t)\,dt + \int_{T}^{T+\Delta} f(t)\,dt$$

Figure 4. Target is the Best

Scenario 2: Lower the Better



Examples of performance metrics in this scenario are: 1) Coding Effort Variation (Percentage)
Tmin = 0%, and Upper Limit= 4.2%
2) Defect Leakage (Percentage)
Tmin = 1.24, Upper Limit= 3.02

In this scenario, the process has only the upper limit (**T**+Δ) for performance metric and the target **Tmin**. If the performance metric is low, process produces better value.

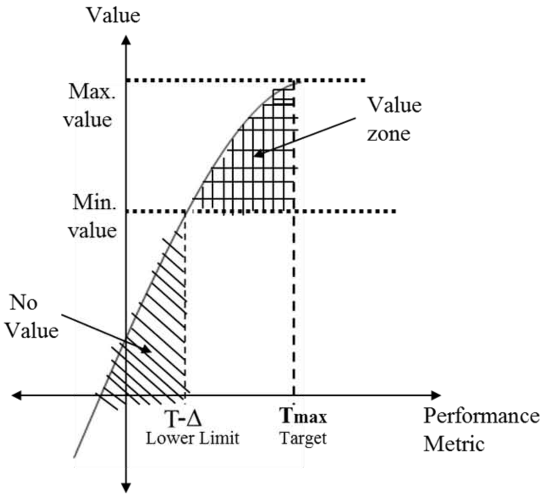
- If the performance metric is at target **Tmin** , the process produces maximum value.
- The value decreases as the performance metric deviates from **Tmin** and moves towards the limit **T**+Δ
- Beyond this limit the process doesn't produce any value.

The value zone can be expressed in terms of a theoretical equation as shown below.

$$\text{Value Zone} = F(x) = \int_{\mathrm{Tmin}}^{T+\Delta} f(t)\,dt$$

Figure 5. Lower the Better

Scenario 3: Higher the Better



Examples of performance metrics in this scenario are: 1) Review efficiency (Percentage)
Tmax = 94%, and Lower Limit= 75%
2) Requirements to Test coverage (Percentage)
Tmax = 90%, Lower Limit= 70%

In this scenario, the process has only the lower limit (**T**- Δ) for performance metric and the target **Tmax**. If the performance metric is high, process produces better value.

- If the performance metric is at target **Tmax** , the process produces maximum value.
- The value decreases as the performance metric deviates from **Tmax** and moves towards the limit **T**- Δ
- Beyond this limit the process doesn't produce any value.

The value zone can be expressed in terms of a theoretical equation as shown below.

$$\text{Value Zone} = F(x) = \int_{T-\Delta}^{Tmax} f(t)\,dt$$

Figure 6. Higher the Better

3. Approach for Value Oriented Process Design

The following diagram depicts an approach that has been used to develop the detailed methodology for designing and defining the value oriented IT processes. This approach highlights the components that need to be considered during process definition.

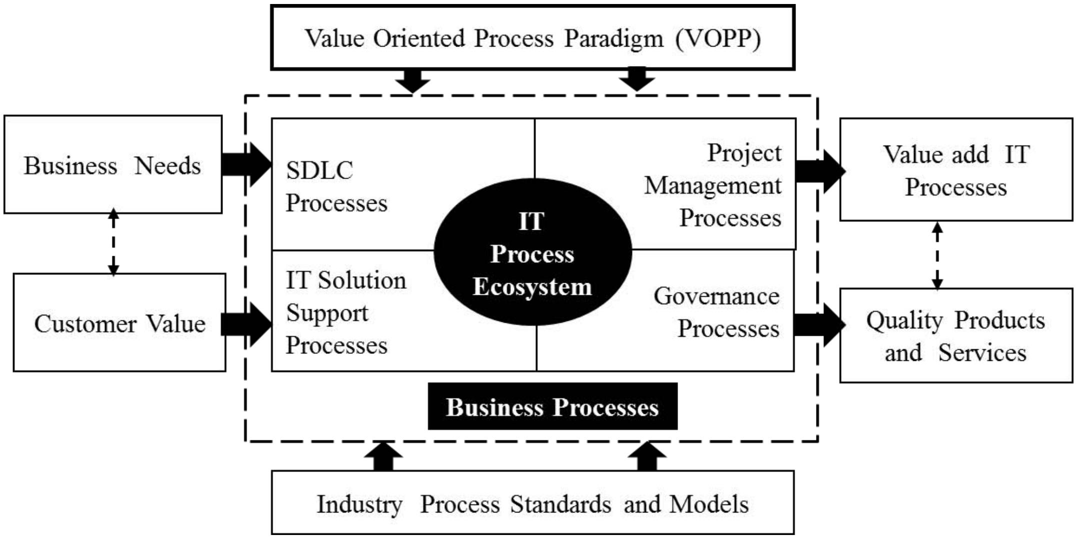

Value Oriented Process Paradigm (VOPP)
Business Needs
Customer Value
SDLC Processes
Project Management Processes
IT Process Ecosystem
IT Solution Support Processes
Governance Processes
Business Processes
Value add IT Processes
Quality Products and Services
Industry Process Standards and Models

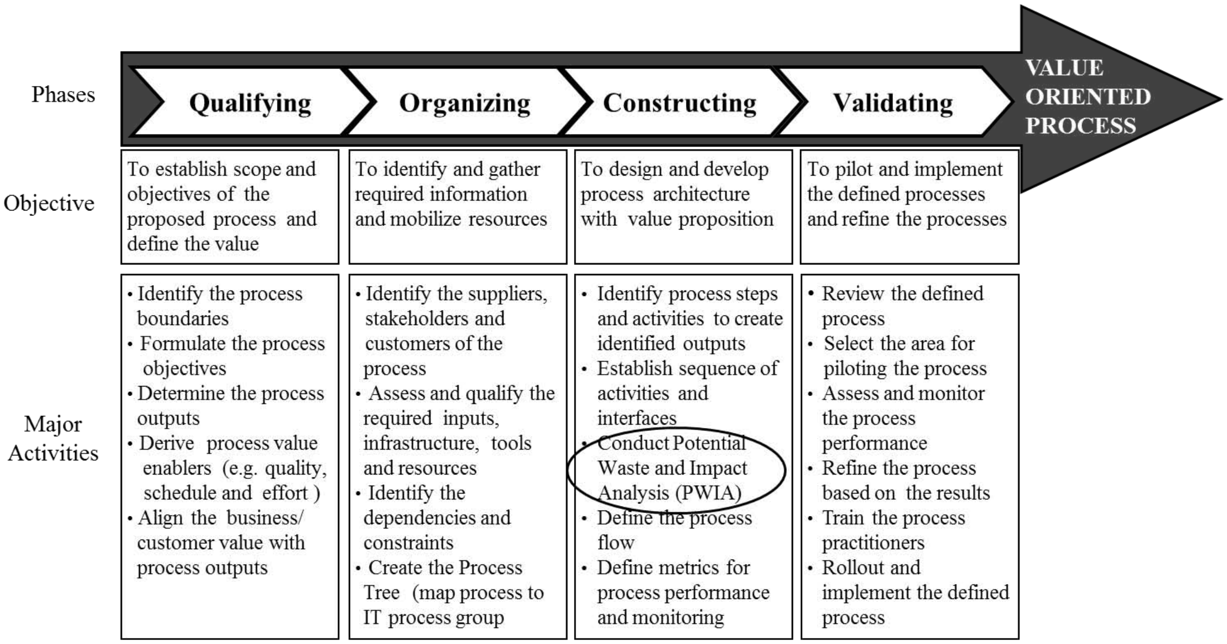
Figure 7. Approach for Value Oriented Process Design

In the Figure 7, business needs and customer value (left side boxes) are critical and important inputs for formulating IT processes that create and deliver value to the business and customers. The IT process ecosystem consists of process groups (indicated in four boxes within a big box with dotted boundaries in the center) that include the processes responsible collectively for producing IT solutions. However, the business processes (the box outside the IT process ecosystem) will have interactions with the IT process groups to ensure the business and customer values are integrated with IT processes. The industry process standards and models (bottom box) like ISO, CMMI and ITIL provide inputs for adopting best practices and process compliance requirements while defining the processes. The Value Oriented Process Paradigm (VOPP) (top box, discussed in detail in following sections) provides a structured methodology including phases and activities to define value oriented processes.

4. Value Oriented Process Paradigm (VOPP)

4.1 Value Based Process Definition Model

In this model (Figure. 8), the traditional process definition approach has been enhanced by addressing value creation concept during defining the processes to produce value add outputs with specified quality. This proven process definition model consists of four phases and the objective of each phase is achieved by performing various activities and tasks in that particular phase.



Phases	Qualifying	Organizing	Constructing	Validating
Objective	To establish scope and objectives of the proposed process and define the value	To identify and gather required information and mobilize resources	To design and develop process architecture with value proposition	To pilot and implement the defined processes and refine the processes
Major Activities	• Identify the process boundaries • Formulate the process objectives • Determine the process outputs • Derive process value enablers (e.g. quality, schedule and effort) • Align the business/ customer value with process outputs	• Identify the suppliers, stakeholders and customers of the process • Assess and qualify the required inputs, infrastructure, tools and resources • Identify the dependencies and constraints • Create the Process Tree (map process to IT process group	• Identify process steps and activities to create identified outputs • Establish sequence of activities and interfaces • Conduct Potential Waste and Impact Analysis (PWIA) • Define the process flow • Define metrics for process performance and monitoring	• Review the defined process • Select the area for piloting the process • Assess and monitor the process performance • Refine the process based on the results • Train the process practitioners • Rollout and implement the defined process

Figure 8. Value Based Process Definition Model

The phases of this methodology detailed out in the following paragraphs:

Qualifying Phase: This is the first phase of the model, which includes the activities to set up preliminary requirements for process definition. The objective of this phase is to establish scope and objectives of the proposed process and define the process value. The process boundaries and objectives are formulated in this phase and corresponding outputs are determined. The process value enablers are derived based on the business/customer value and mapped with the process outputs.

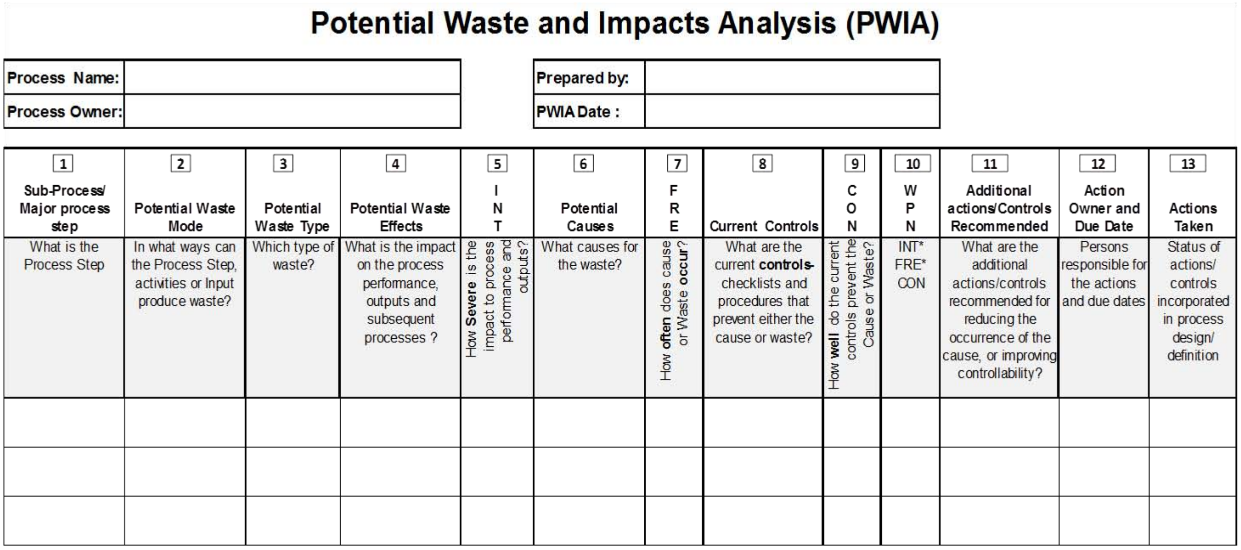
Organizing Phase: This is the second phase of the model with a focus on corroborating the information and inputs that required for process definition. The objective of this phase is to identify and gather required information and mobilize resources. In this phase, the suppliers, stakeholders and customers of the process outputs are identified. The inputs, infrastructure, tools and resources are assessed and qualified for using them during the process definition. The interfacing is established with other processes as required.

Constructing: This is the third phase of the model in which the process definition is carried out by using the information and resources from the previous phases. The objective of this phase is to design and develop process architecture with a value proposition. The tasks, activities and steps are defined, and the sequence and interfaces of the activities are established. The process is critically evaluated from the value creation and waste elimination perspective by a technique called 'Potential Waste and Impact Analysis (PWIA)' highlighted with a circle in the Figure 8. This technique is discussed in detail under subsection 4.2. Appropriate actions are identified to eliminate potential waste in process and suitable controls are incorporated as part of the process definition. The associated process performance metrics are identified and defined to monitor the performance of the process and its outputs.

Validating: This is the fourth and final phase of the model, which includes the activities for piloting and implementing the defined process. The objective of this phase is to pilot and implement the defined processes and to refine the processes as required. The defined processes are piloted in the selected area and the outputs and performance are assessed against the identified value and quality. Based on the results, the processes are refined and fine-tuned before the rollout and implementation across the process groups.

4.2 Potential Waste and Impact Analysis (PWIA) Tool

This tool is developed to identify the potential waste during process definition that affects the process value and thereby impact the quality of process outputs. The objective of this technique is to identify and assess the waste, and to identify and incorporate the controls to eliminate or reduce the impact of waste on process value during process definition stage itself.



Potential Waste and Impacts Analysis (PWIA)

Process Name:		**Prepared by:**	
Process Owner:		**PWIA Date :**	

1 **Sub-Process/ Major process step**	2 **Potential Waste Mode**	3 **Potential Waste Type**	4 **Potential Waste Effects**	5 **I N T**	6 **Potential Causes**	7 **F R E**	8 **Current Controls**	9 **C O N**	10 **W P N**	11 **Additional actions/Controls Recommended**	12 **Action Owner and Due Date**	13 **Actions Taken**
What is the Process Step	In what ways can the Process Step, activities or Input produce waste?	Which type of waste?	What is the impact on the process performance, outputs and subsequent processes ?	How **Severe** is the impact to process performance and outputs?	What causes for the waste?	How **often** does cause or Waste **occur**?	What are the current **controls**- checklists and procedures that prevent either the cause or waste?	How **well** do the current controls prevent the Cause or Waste?	INT* FRE* CON	What are the additional actions/controls recommended for reducing the occurrence of the cause, or improving controllability?	Persons responsible for the actions and due dates	Status of actions/ controls incorporated in process design/ definition

Figure 9. Potential Waste and Impacts Analysis (PWIA)

The usage of the above tool is as described below:

Column 1- Key Process Step: Indicate the name of process activity or step

Column 2- In what ways can the Process Step or Input produce waste? : Mention various possible ways this step or inputs can produce waste

Column 3- Which type of waste? : Mention the type of waste. Select the types of waste from the 7 types of waste (Transportation/Hand-offs, Inventory, Motion, Waiting, Overproduction, Over processing and Defects)

Column 4- Potential Waste Effects: Mention the impacts on the process performance, quality of outputs and subsequent processes

Column 5- Intensity (INT): Assess the intensity on scale 1 – 10 and mention that how severe is the waste impact to process performance and quality of outputs

Column 6- Potential Causes: Mention the potential causes for the identified waste

Column 7- Frequency (FRE): On the scale 1 – 10, indicate the rating on how often does cause or waste may occur

Column 8- Current Controls: Mention what the current controls are e.g. checklists and procedures that prevent either the cause or waste

Column 9- Controllability (CON): On the scale 1 – 10, indicate the rating on how well the planned controls prevent the cause or waste

Column 10- Waste Priority Number (WPN): Calculate the product of the ratings of column 5, column 7 and column 9 and mention the product number (WPN = INT * FRE * CON)

Column 11- Additional actions/Controls Recommended: Provide the additional actions/controls recommended for reducing the occurrence of the cause, or improving controllability for higher WPNs

Column 12- Action Owner and Due Date: Mention the names of persons responsible for the actions and target dates for completion

Column 13- Actions Taken: Provide status of actions/ controls incorporated in process design/ definition

The identified potential wastes are prioritized based on the Waste Priority Numbers by arranging them from highest to lowest. The highest WPNs will be given priority while developing and incorporating the controls to reduce/eliminate the process waste.

4.3 Case Study- Usage of Potential Waste and Impact Analysis (PWIA) Tool

In this section, usage of PWIA tool during the process definition using Value Oriented Process Paradigm (VOPP) framework for an IT division of a large Insurance client at one of its global locations is elaborated

Scope of the engagement: To define and implement the processes of service operation life cycle stage of ITIL framework

Client: One of the leading Insurance Providers.

Process in scope: Incident Management Process, Investigation and Diagnosis Subprocess

Approach: During this process definition engagement, VOPP framework is leveraged for Incident Management process definition. The process consultants have referred to all the phases of the VOPP framework and performed the activities mentioned in each of the phases to produce the process package (relevant artifacts that required completing the process definition). The PWIA tool has been extensively used to establish appropriate controls to reduce process waste and thereby to ensure that the process value and output quality is integrated together before implementation and roll out of the defined process.

Sample Artifacts: Following are the two sample artifacts Process Tree and PWIA worksheet from Organizing and Constructing phases of VOPP framework respectively.

1. Process Tree: The following figure (Figure 10) depicts on how the subprocess 'Investigation and Diagnosis' is rolling up to the IT Process Group 'IT Service Support Management'. The objective of this exercise to understand how and which of the upstream components/processes will be impacted (either positive or negative) by the performance and output quality of the subprocess 'Investigation and Diagnosis'

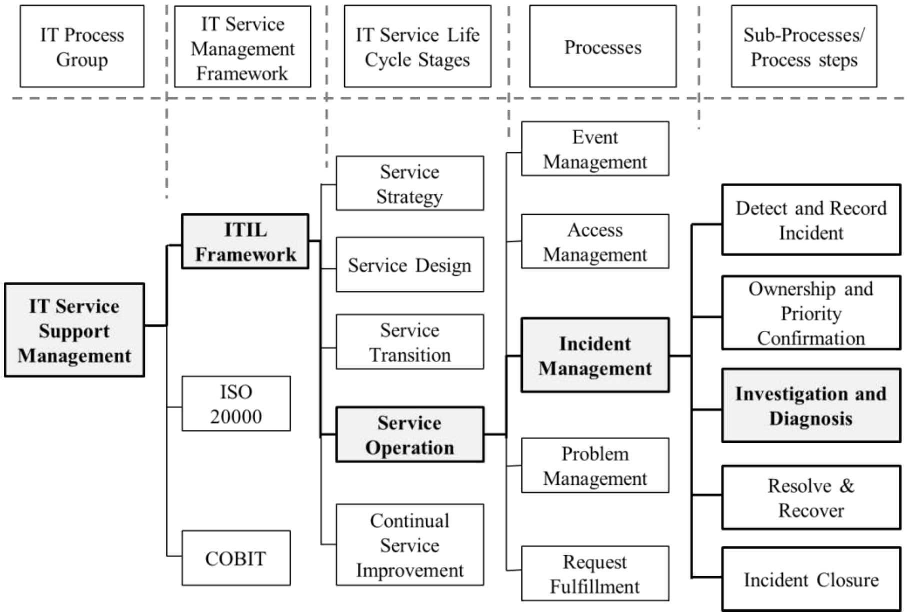
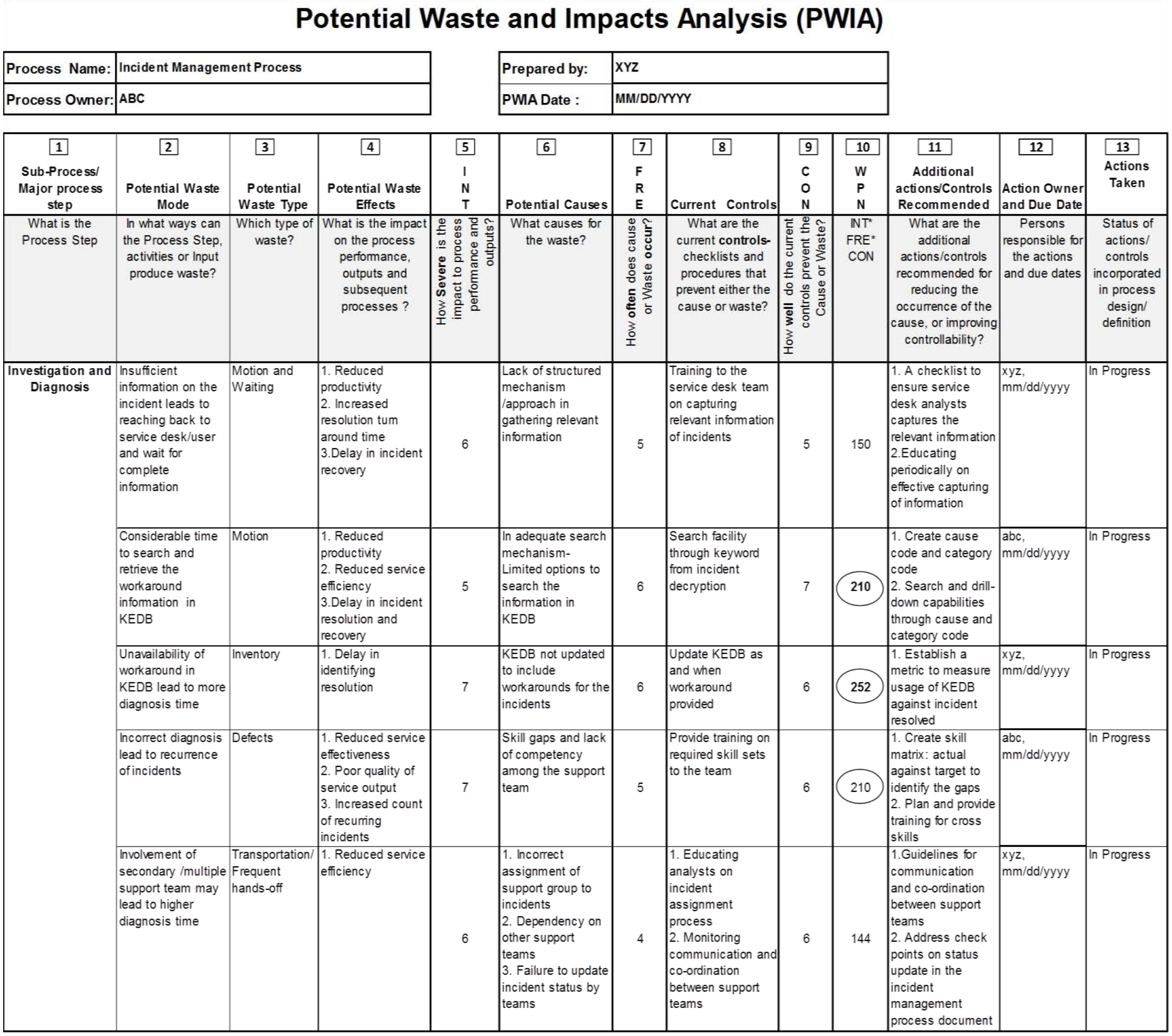


Figure 10. Process Tree

2. PWIA Worksheet: The following figure (Figure 11) depicts the snapshot of the PWIA worksheet created to identify the process waste and create relevant controls to put in place while designing and defining the subprocess 'Investigation and Diagnosis'. Column 10 contains the Waste Priority Number (WPN) for each of the waste mode and the top 3 WPNs are circled for prioritization purpose. Column 11 contains the recommended additional process controls/actions to eliminate/reduce the waste.



Potential Waste and Impacts Analysis (PWIA)

Process Name:	Incident Management Process	Prepared by:	XYZ
Process Owner:	ABC	PWIA Date :	MM/DD/YYYY

1	2	3	4	5	6	7	8	9	10	11	12	13
Sub-Process/ Major process step	Potential Waste Mode	Potential Waste Type	Potential Waste Effects	INT	Potential Causes	FRE	Current Controls	CON	WPN	Additional actions/Controls Recommended	Action Owner and Due Date	Actions Taken
What is the Process Step	In what ways can the Process Step, activities or Input produce waste?	Which type of waste?	What is the impact on the process performance, outputs and subsequent processes ?	How **Severe** is the impact to process performance and outputs?	What causes for the waste?	How **often** does cause or Waste **occur**?	What are the current **controls**- checklists and procedures that prevent either the cause or waste?	How **well** do the current controls prevent the Cause or Waste?	INT* FRE* CON	What are the additional actions/controls recommended for reducing the occurrence of the cause, or improving controllability?	Persons responsible for the actions and due dates	Status of actions/ controls incorporated in process design/ definition
Investigation and Diagnosis	Insufficient information on the incident leads to reaching back to service desk/user and wait for complete information	Motion and Waiting	1. Reduced productivity 2. Increased resolution turn around time 3.Delay in incident recovery	6	Lack of structured mechanism /approach in gathering relevant information	5	Training to the service desk team on capturing relevant information of incidents	5	150	1. A checklist to ensure service desk analysts captures the relevant information 2.Educating periodically on effective capturing of information	xyz, mm/dd/yyyy	In Progress
	Considerable time to search and retrieve the workaround information in KEDB	Motion	1. Reduced productivity 2. Reduced service efficiency 3.Delay in incident resolution and recovery	5	In adequate search mechanism- Limited options to search the information in KEDB	6	Search facility through keyword from incident decryption	7	210	1. Create cause code and category code 2. Search and drill-down capabilities through cause and category code	abc, mm/dd/yyyy	In Progress
	Unavailability of workaround in KEDB lead to more diagnosis time	Inventory	1. Delay in identifying resolution	7	KEDB not updated to include workarounds for the incidents	6	Update KEDB as and when workaround provided	6	252	1. Establish a metric to measure usage of KEDB against incident resolved	xyz, mm/dd/yyyy	In Progress
	Incorrect diagnosis lead to recurrence of incidents	Defects	1. Reduced service effectiveness 2. Poor quality of service output 3. Increased count of recurring incidents	7	Skill gaps and lack of competency among the support team	5	Provide training on required skill sets to the team	6	210	1. Create skill matrix: actual against target to identify the gaps 2. Plan and provide training for cross skills	abc, mm/dd/yyyy	In Progress
	Involvement of secondary /multiple support team may lead to higher diagnosis time	Transportation/ Frequent hands-off	1. Reduced service efficiency	6	1. Incorrect assignment of support group to incidents 2. Dependency on other support teams 3. Failure to update incident status by teams	4	1. Educating analysts on incident assignment process 2. Monitoring communication and co-ordination between support teams	6	144	1.Guidelines for communication and co-ordination between support teams 2. Address check points on status update in the incident management process document	xyz, mm/dd/yyyy	In Progress

Figure 11. PWIA Worksheet

5. Conclusion

In this paper, the process value and quality were discussed extensively by providing insight on process value definition. The IT process value creation was discussed in detail through a process value question model. The sources of waste responsible for process value loss in IT ecosystem were presented and extensively discussed with examples.

An approach for visualizing the customer value and mapping these value attributes with IT processes has been explained to understand value flow across IT process groups. Extension of Dr. Genichi Taguchi’s quality loss function theory to analyze the process value loss has been extensively discussed with different scenarios. These scenarios were elaborated further to demonstrate the pattern of value deviation against the variation of process performance metric from the target level and corresponding process value.

An approach for value oriented process design was presented. This approach helps process practitioners to understand the components that need to be focused on during value integrated process definitions. Also presented was a framework named Value Oriented Process Paradigm that provides adoptable guidance for designing value oriented IT processes and integrating customer value with process quality.

Finally, a technique named Potential Waste and Impacts Analysis (PWIA) that provides guidance towards identifying potential waste during process definition and prioritizing them to put controls to reduce or eliminate waste has been demonstrated. Also, a practical case has been presented to demonstrate the usage of PWIA tool to identify waste and establish relevant process controls.

The authors feel that the importance of process value analysis and the value oriented process definition model used to ensure value addition across the enterprise value chain, as detailed in this paper, can be useful for a process group involved in process definition and process improvement initiatives.

References

Dr. Hugh L. McManus, "*Product Development Value Stream Mapping (PDVSM) Manual 1.0."* Lean Aerospace Initiative September 2005.
http://lean.mit.edu/downloads/cat_view/94-products/575-product-development-value-stream-mapping-manual

International Organization for Standardization, *ISO 9000:2005 Quality management systems -- Fundamentals and vocabulary.* 2005.

CMMI Product team, *CMMI® for Development, Version 1.3*, Software Engineering Institute. 2010.

Basem El-Haik, David M. Roy, *Service Design for Six Sigma—A Road Map for Excellence,* John Wiley & Sons Inc., 2005.

James Robert Evans, William M. Lindsay, *Managing for Quality and Performance Excellence*, Thomson Learning Inc., 2008.

Shivanand Parappa, "Process Dynamics, Variations, and Controls in Software Engineering," 30th Annual Pacific Northwest Software Quality Conference Oct 8 – 10, 2012.
http://www.uploads.pnsqc.org/2012/papers/t-74_Parappa_paper.pdf

Process Fitness – Process Improvement Lessons from Personal Trainers

F. Michael Dedolph

fmdedolph@netscape.net

Abstract

Many people have seen the "before" and "after" pictures of people who have successfully undertaken a rigorous fitness program. What you don't often see is the pictures of those who fail to make the changes--just as companies don't advertise their failed process improvement efforts.

One key to success is to involve a personal trainer (or process improvement coach) early in the transformation effort. Michael has gathered some lessons from some premier personal trainers that can be applied either to personal or organizational improvement efforts.

This paper explores some of the areas where personal training and process improvement coaching are analogous, and then expands on one of these areas – dealing with difficult clients.

Michael will pass on lessons about how a trainer might deal with the know-it-all alpha male, the overly content person, the client who always goes off track, and others. These lessons may save you from a client or team member who (wittingly or unwittingly) sabotages your best efforts.

This presentation is suitable for both gym rats as well as couch potatoes - as long as you have a desire to improve.

Biography

Michael has 30+ years experience in the computer industry, with interests spanning agile methods, architecture, metrics, process improvement, project management, quality management, software risk management, and teambuilding. Currently an independent consultant, Michael started out as a computer officer doing satellite communications in the US Air Force, and has worked for the Software Engineering Institute, Lucent Technologies (Bell Labs), and Computer Sciences Corporation.

Michael is on the PNSQC Board, and has served as the Invited Speaker and Keynote Chair for the 2011 and 2012 conferences

He recently embarked on a Personal Training certification program, and noticed many similarities between what personal trainers and process coaches do.

1. Introduction

One day, standing in front of the mirror, you see someone staring back at you who doesn't quite look like the person you remember staring back at you a few years ago. Possibly the paunch is a bit paunchier, the posture not as upright, and the arms not quite as firm. But there it is, you are who you are.

Lying on the table nearby is a magazine, with a cover picture of someone who looks quite different – firm, toned, tanned, sleek. There it is - they are who they are, and it looks like it is way more fun to be *them* than it is to be *you*.

You shake your head, and say to yourself, "it is time to get back in shape."

The fitness premise is simple. There is **us**, there is **them**, but YOU can be one of THEM, if . . . you get yourself "in shape".

Putting vanity aside, there are some real questions you should ask yourself about your personal fitness level, such as: "am I fit enough to do the things I want to do, and do them well?"

Process fitness is analogous to **personal fitness** in many ways. For process fitness the questions might be "can our team do the things we want to do, and do them well?"

Some of the similarities between personal and process fitness start with these questions. What does it mean to be fit? What does it mean to "do them well"?

Personal fitness might be defined as being good looking, participating in a sport you like, having a healthy cardiovascular system, being able to perform enjoyable every day activities (such as taking a walk in the evening), or in terms of being a world-class competitive athlete.

If you are successful, you may or may not get recognition for your effort. The lady shown here was featured in a major Women's magazine, but recognition was certainly not what sustained and motivated her for the extended time she worked to lose weight and become more fit.

As to **process fitness**, fitness might be defined as being the envy of the other agile teams, being able to add more fun features, maintaining system performance under load, being able to apply maintenance fixes without user interruption, or producing ultra-reliable software for life critical applications. Your fitness goal might be strictly commercial - to sell a million copies of a mobile app, or to get a leg up on your competitor.

A lot of information about HOW personal trainers approach their work can be found online at the Personal Trainer Development Center (PTDC) website, http://www.theptdc.com/. Most of this paper was gleaned from pondering the lessons there, and selecting ones that applied to process improvement as well.

The premise of this paper is that there are many areas where personal training and process improvement are analogous. The similarities provide a rich source of potential lessons learned. The first part of the paper explores some of the high-level similarities and high-level lessons.

2. Lessons from personal training

Seven lessons are listed below; each is expanded in the following sections:

- Lesson 1: People are more interested in the product or outcome than the process
- Lesson 2: The process is a major contributor to the eventual outcome
- Lesson 3: Client needs differ, BUT,
- Lesson 4: For overall fitness, there are basics that are apply to everyone
- Lesson 5: As a coach or trainer, you need to lead by example
- Lesson 6: There are common paradigms for improvement
- Lesson 7: You will need to deal with difficult clients.

2.1 People are more interested in the product or outcome than the process

6-pack abs are 6-pack abs, no matter how you got them. A working app is a working app, no matter how it was developed.

People really don't want to hear about your 4 AM mornings doggedly spent chasing the last bugs before release, the nuances of your development processes and bug tracking software, or the 20,000 crunches you did to develop your 6-pack abs.

> Note: this begs the question – are 6 pack abs what you really want?
>
> Although six-pack abs are attractive to many, it is the deep core muscles that need to be strong, as these are what support you. If you only strengthen the superficial abdominal muscles to the point of significant definition, this is actually over-tightening these muscles. There should be balance between flexibility and strength, and overdoing abs (or any other muscle group) can be counter-productive. But, if being attractive is your primary goal, you may make this choice.
>
> Equivalently, does your app really do something useful that people need, or is it the commercial equivalent of eye-candy? What should it be?

The lack of attention to process and focus on what is immediately visible leads people to make a lot of interesting (and often suboptimal) choices. Fitness, like software, is filled with quick fixes – the "magic muscle pills", the "wonder diet", and the "only ab exercise you need" are equivalent to the elusive software "silver bullet", you know, the tool that talks to the user and writes the software for you, all the while putting giant sums of money into your checking account.

Unfortunately, quick fixes don't work in either arena, and don't lead to either sustained performance improvement or overall fitness.

Another way to improve your fitness is the "do-it-yourself" approach, which actually can work. You can set goals for yourself, research methods for reaching them, and implement a fitness program. There are advantages to this approach. You own it, and it will work if you do your research well and are intrinsically motivated enough to follow through with your program.

With either a quick fix or a sustained do-it-yourself program, at some point you might realize you are falling short of your goals. When the realization hits, you have choices. You might:

- Change (or even abandon) your improvement goals
- Work harder at what you are currently doing
- Implement a new or modified improvement program to change what you are doing
- And/or hire a coach or trainer to help you modify your improvement program

All of these things can help, **or hinder,** your success. For example, changing your goals may make them realistic, but may also just be giving up. Working harder may cause a breakthrough, but also may fit into the definition of insanity - “doing the same thing and expecting different results”. Implementing a new fitness program may lead to success, or to “thrashing” from side to side without any forward progress.

Assuming you go the coach/trainer route, your new coach will undoubtedly work with you to clarify and improve your goals, motivate you to work harder, and suggest ways to change what you are doing that can lead to better results. Here, the role of a process improvement coach and personal trainer are very similar. You focus on the work; the coach focuses on the methods, and makes sure the methods are appropriate for your goals.

Hiring a trainer or coach is the antithesis of the quick fix. Not only is it a commitment to change, it is an acknowledgement that tapping someone else’s expertise is likely to achieve better results than going it alone.

For any successful fitness program, the results you see will come from having a plan and a method, following it, monitoring your progress, and making adjustments when needed. In other words, results come by following and improving a process.

2.2 The process is a major contributor to the eventual outcome

To a large extent, how you go about your quest for fitness determines the outcome. How you go about your work process determines the types of products you can successfully build and the services you can provide.

To improve your either your personal or process fitness, you will need to:

- Know where you want to be (*motivation*)
- Know where you are (*observation*)
- Have plans and methods (*preparation*)
- “Do it” (*perspiration*)
- Monitor progress (*recalibration*)
- Experience success (*celebration*)

As a trainer or coach, you can help with everything, **except** the perspiration part. You can’t lift the weight, swim the lap, or perform the software testing for your client. Your goal as a trainer is to improve your client’s chances of success, not to do the work for them.

> *Imagine you are a personal trainer who was hired by someone’s spouse to “get them into shape”. They show up for Session 1 and they make it clear that their idea of being in shape is being able to walk to the refrigerator for a snack and some beer, and their favorite sport is golf. On TV. Nonetheless, you work hard to develop a program for them. The program you “mutually” agree to*

includes weight training for strength, specific cardio work to facilitate weight loss, and a yoga program for flexibility on their rest days.

You try to keep them motivated by reminding them how happy their spouse will be if they succeed in meeting their weight loss goals, and you work to interest them in other pastimes that are conducive to fitness. In week 12 of the program, the spouse calls to tell you they are getting a divorce. How well do you think this will work out?

In a previous job, I was hired by a Senior Vice President to implement a process improvement program with two goals – passing a compliance appraisal, and improving operational efficiency. Things were moving along swimmingly until she retired. Immediately, all cooperation disappeared, and I realized I was in the same position as my hypothetical personal trainer. I'll leave you to guess how it worked out. (We did pass the compliance appraisal . . .)

You can help your client build and sustain their motivation, but you can't create it from nothing. And, without sustained motivation, sustained improvement won't happen.

2.3 Client's needs differ

Realistically, process improvement or personal training alone is NOT enough. Although it is tempting to recite the mantra "the quality of the process determines the quality of the product", in the real world, high quality workouts and premier personal training will not turn me into the next heavyweight champion. There are several reasons for this unwanted reality intrusion - I am too old, too small, and have some minor health issues that will prevent me from ever reaching this goal.

A more realistic mantra for improvement is the old recruiting slogan "Be all you can be." For me, completing a duathlon or half marathon, doing 100 pushups, or lowering my cholesterol by 20 points are much more realistic goals, and probably better for both my mental and physical health than chasing an impossible dream.

The training needs for an elite, world-class competitive athlete are very different than mine. Even among elite athletes, the training protocols for a marathon runner, power lifter, or boxer will be very different. The marathon runner will need techniques to build speed and endurance; the power lifter needs brute strength and explosive quickness, and the boxer needs a bit of everything.

But, a more typical personal training client might be interested in weight loss, optimizing their skills for the weekend softball league, meeting a specific goal such as completing a marathon for the first time, or improving their performance in an activity they have already mastered.

This is analogous to process improvement in software development. The team may need to make release schedules more consistently, fix a customer's perception of poor web site quality, improve the burn down rate so people can spend more weekends at home, create a high performance system that can handle a million hits during the Super Bowl halftime, or field error-free safety-critical systems.

Part of your job as the process coach is to tailor the process improvement (or workout) plans to the client's needs. You also need to make sure you are the right coach for the client. An Olympic power lifting coach may be intimidating to a small-framed woman that wants to lose weight, and may seem completely irrelevant to a marathon runner that wants to shave 10 seconds per mile off her time. Analogously, a rigorous performance model may be irrelevant to a small, single user app.

A final note on client needs – the goals the client expresses may not be the real goals, the goals that are driven by their core values or fears. You will need to dig a bit to find those.

2.4 For overall fitness, there are basics that are apply to everyone

One thing you will find is that there are fundamentals that apply to almost every client.

For personal fitness, some level of cardio training, strength training, and flexibility (or mobility) training are needed for a balanced program. Everyone will benefit from strength training, although the nature and intensity of that training will differ significantly.

For software development, requirements development, configuration control, release management and testing are among many basic processes that need to be performed well no matter what the software is, although the way those activities are done will differ depending on your goals and methods.

Also, in both personal and process fitness, the need to articulate the improvement goals and plan the process is essential. For success, the client (individual or team) will need to actually do the work. Along the way, you and the client will need to monitor progress and make adjustments based on reality. As a coach, this means you will need to have a system to track your client's progress, and have the credibility to provide feedback.

When the world intrudes, you and your client will need to recalibrate. A bout of pneumonia may put their marathon plans on hold, and if two team members leave you may need to delay process improvement changes to make the date for a scheduled release.

In addition adjusting to external factors (you do have a risk management plan, don't you?), you need to keep in mind that keeping the team motivated is always a primary goal. Any technique you use to motivate people won't work well if you don't set an example.

2.5 As a coach or trainer, you need to lead by example

As examples - don't tell people to do an exercise you can't do (there are very rare exceptions to this for elite athletes). Don't lecture people about watching their diet while snarfing down bacon cheeseburgers and a large order of fries.

As a process coach, don't implement documentation standards that you wouldn't follow your self, and don't prescribe metrics unless you are willing to set a personal standard for using metrics as well. In the jargon of the process improvement trade, "drink your own tea."

2.6 There are common paradigms for improvement

For process improvement in its simplest form, this is exemplified by Shewhart's plan-do-check-act cycle. There are hundreds of variants; here is one that could be applied to almost any improvement program.

- **Motivation.** This is the starting point, the belief that things can be different than they currently are; motivation provides the inspiration for change.

- **Observation.** You will need an accurate assessment of the current status for planning improvements and selecting the right methods.

- **Preparation.** You know why (motivation), you know what (observation), now you need to chart the *how* – planning and methods.

- **Perspiration.** It is really unfortunate, but the plan won't implement itself, unless the plan is to do nothing.

- **Recalibration.** You will need to check your progress, and also will need to incorporate changes due to changing circumstances. Both require ongoing observation and honest re-evaluation.

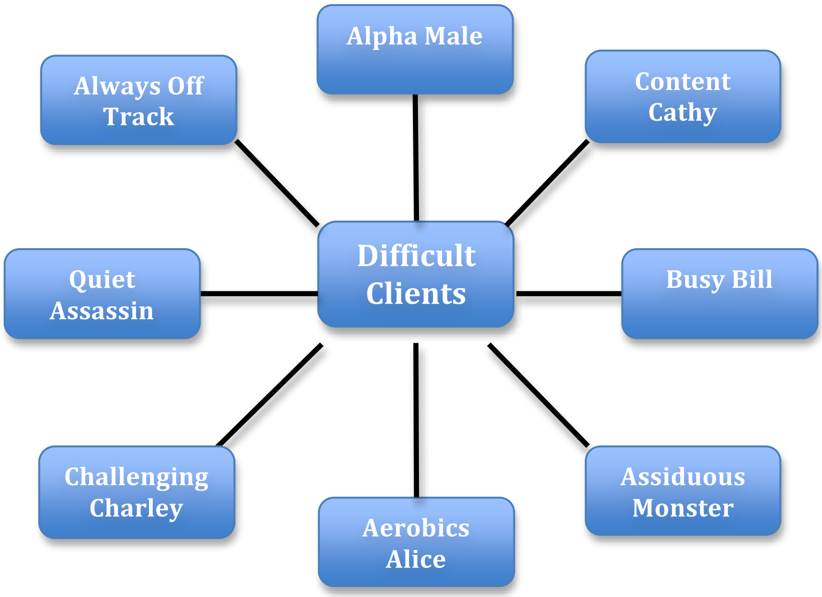
- **Celebration.** This comes with achievement, and can mark the end of an effort, or the beginning of a new improvement cycle.

With all of these lessons in mind, what could go wrong? Well, you might encounter "people problems" along the way, in many forms. Sometimes it will be explicit or unintended resistance to the change effort, in other cases, a difficult client or team member may be the problem. People are the problem, and the solution to the problem is the people.

2.7 You will need to deal with difficult clients

Alas, even though they pay for your services, clients aren't always easy to deal with or totally on board with your program; ditto for the process improvement team members. This holds true for personal trainers as well as process improvement coaches.

I was attracted to advice provided by Jonathan Goodman about difficult clients, and found myself chuckling because I had encountered each of these characters in the different context of process coaching. So, here are some "typical" difficult clients, along with some strategies for dealing with them.



Note to the Reader:

The format of these examples is to briefly describe the personal training client, describe the challenge, summarize Jonathan's way of addressing the challenge, and then relate it to process improvement coaching.

These examples are paraphrased (or quoted directly) from Jonathan Goodman of PTDC, and can be found on the PTDC web site or in his book "Ignite the Fire". Any errors from the paraphrasing are mine.

As you read these examples, think about people in your organization that are involved with process definition, management and improvement. Some of these characters may seem strangely familiar (you may even resemble one of them yourself.)

Ask yourself, "Would these strategies work to motivate people in my environment?" Finally, keep in mind that while we are primarily discussing the client, the description may also apply to one of the team members on the process improvement team.

3. A personal trainer’s view of dealing with difficult clients

3.1 The Alpha Male

From Jonathan Goodman:

He was an A-type. About as much of a know-it-all personality as I've ever come across. He was more interested in telling me about his self-prescribed workout than getting my advice. He wasn't in the gym because he wanted to be there; instead, he was coerced into coming. So when I met him I knew I would have my work cut out for me.

***Challenge for the Alpha Male**.*

There is one overriding challenge: establishing a relationship where you are perceived to be an equal. Without equality, your advice will be ignored.

Techniques for dealing with the Alpha Male

Here's what I did:

1. *Sat down and listened to him. I needed to gather as much information about his own workout as possible to generate my plan. I let him talk and took notes. I made him speak about his pain and how the artificial hip had limited his ability to do activities he loved, like hiking.*

2. *I didn't perform an assessment. He would not have listened to me at this point because I hadn't shown my value. Instead I let him show me his warm up. At the end of the day I prescribed a 10-minute warm up for him. That was all. I asked him to have it done before we met 2 days later.*

3. *Instead of showing him an exercise during the next meeting I asked him to show me the exercise that I chose. So I would say, "show me a pushup", and watch.*

This was what shifted the power relationship. One of two things happened when I asked him to show me the exercise.

- *He had no clue what it was. Mission accomplished. I was now free to instruct.*

- *He knew the exercise and would show me. Here I used what I call the "* ***Sandwich****" technique. It has three parts:*

  a) *I told him something he did well in the exercise.*
  b) *I asked him to improve upon one or two of the most important facets of the exercise. (This part of the sandwich is stuff that really isn’t good.)*
  c) *I finished with one more point he did well.*

The Sandwich technique worked beautifully. By stroking his ego first, he was more willing to listen and take instruction.

Sometimes you need to be a bit creative when dealing with clients. Don't jump into the instructor role right away. By stroking alpha-types egos you can slowly shift the power without hurting their ego and increasing resistance.

3.1.1 Analogy for Process Improvement Coaches

For me personally, it has always been tempting to go toe-to-toe with the Alpha male, particularly if you have executive sponsorship backing you.

However, harkening back to my earlier story, that is exactly what I had done with the Alpha Male on the team I was hired to lead. When my sponsor retired, the Alpha Male went out of his way to sabotage the effort, and he had 30-plus years of political connections inside the organization. Even though I won the initial battles, I lost in the end, and, so did the organization. Big time.

Next time, I'll listen more, and try a Sandwich. It may or may not avoid all confrontations – but it may help to transform the power relationship into a peer-to-peer collaboration.

3.2 The Content Cathy

From Jonathan Goodman:

> *The Content Kathy is just happy to be exercising. This client had a previous injury with loss of function. Because of that previous injury she was ecstatic to be exercising again. A Content Cathy could also be an elderly client who is happy just to stay active and maintaining their function. The Content Kathy doesn't care about progression or pushing heavy weights. They love lots of stretching and 'feeling' the body working. They may also resist quantifying progress goals, because they feel trapped by numeric goals.*
>
> ***Challenges for Content Cathy**:*
>
> *Despite not having lots of hefty goals that you, as their trainer, are accountable to, the Content Kathy usually has a host of concerns. Keeping them healthy is the biggest challenge. A secondary challenge is often that **you** may want all of your clients to push themselves hard and get stronger. The Content Kathy loves working out but doesn't want to work hard.*
>
> ***Techniques for dealing with Content Cathy***
>
> *The Content Kathy needs some direction. They've come to you because you have the knowledge to help them with their problems. Your job is twofold:*
>
> - *Organize their workouts in a safe and efficient manner.*
> - *Provide a subtle push and pleasant conversation.*
>
> *Sneak in the assessments without making it overt, and couch improvements in terms of doing something new or fun, or in terms of injury prevention.*
>
> *The key in dealing with this client type is to be willing to veer from your training model and mold yourself to their needs. Have some fun. You have to accept that you can't get this client the kind of results that you expect from others because they don't want to push themself. Sometimes as a trainer it's difficult to not expect hard work from clients we want to push. You must accept the* [valid] *differences in reasons for exercising and work within them.*

3.2.1 Analogy for process improvement coaches

A team that is close to retirement and doing maintenance releases for a legacy system may not be interested in improvement if it involves a lot of change. Keep in mind, the team members that wanted to push hard into new frontiers left the team years ago. Changes may have been tried in the past, and they led to crashed systems and/or unhappy customers (the equivalent of an injury).

The system is quirky, and their process is quirky too. The process probably has a lot of home grown checks and balances that have evolved over time because of specific aspects of the system environment. But, the process probably isn't very efficient. Roll with it.

You may be able to implement some improvements while planning for the team's transition to the next project, to facilitate handing off the system to another project team, in the guise of risk avoidance (equivalent to injury prevention), or if team members see the improvements as a way to stay employed a few more years.

Analogously to the way Jonathan handled his client, you may need to perform process assessments quietly, to avoid the perception that your are judging them or holding them to unrealistic goals.

3.3 The Busy Bill

From Jonathan Goodman:

> *Time is the issue with the Busy Bill. He runs in on his cell phone and rushes out the minute the workout is done. Sometimes during the workout he'll even stop to take a call.*
>
> *This client understands the value of working out and usually wishes that he could be more consistent. Their busy schedule forces them to cancel regularly without knowing when they can reschedule. They've simply taken on too many things in their lives. It's impossible to program even short term.*
>
> ***Challenges for Busy Bills***
>
> *Getting results for a Busy Bill is difficult. They work out sporadically so any programmed progression is close to impossible. In addition, because their lives are so busy, stress gets in the way of regular workouts. The typical Busy Bill's eating habits are usually sub-par and a weakened immune system due to stress may cause frequent illnesses and missed workouts. From a scheduling point of view, he can be frustrating because of the frequent cancellations.*
>
> ***Techniques for Dealing with Busy Bill***
>
> *I enthusiastically took on a Busy Bill when their previous trainer left. The guy already had great form and always seemed to both work hard and have a great time. He had an athletic background as well.*
>
> *My training with BB started off great. I got him back on track and laid out a long-term plan for him that he was excited about. BB started to monitor his eating, cut down on his alcohol consumption, and diligently completed his cardio. He was losing inches and gaining strength, **but then work started to take over BB's life**.*
>
> *Making time for exercise became close to impossible. When he did make it in, his attention was elsewhere. The stress caused him to increase his alcohol consumption, and the gains he had made over the past 3 months disappeared quickly.*
>
> *I was frustrated, but I wanted him to stay active during a difficult time in his life. So, I decided to switch our focus. I gave him two simple 30-minute workouts and some homework. Every time he was feeling stressed or overwhelmed, he could come into the gym on his own and complete the workouts. **My goal was to make the gym his sanctuary.** In the meantime, I cancelled all of our organized sessions and told him to call me if he wanted to book a session. This pattern lasted for nearly 5 months. During that time, he didn't make any progress, but he did continue working out, which was my primary goal.*

If your Busy Bill's life is always going to be stressful, here are some additional ways to deal with it.

- *Don't overwhelm a Busy Bill.*
- *Do a mini-assessment every time the client walks in. His physical and mental state will be a wild card. Some days he will come in relaxed, and excited for a break in his routine. Other days he may be so stressed he can hardly move. It's important to read your Busy Bill and take as much time as needed.*
- *If you find that his mind is jumbled it's probably a good idea to nix work that requires a lot of clearly focused attention.*
- *Give homework. Twenty minutes of light exercise, even if it's before bed, will help them feel good when they can't get to the gym and give them something to look forward to.*
- *Finally, if Busy Bill refuses to leave his cell phone in the changing room, hold it during the workout. If someone calls, I tell him who it is and ask if he needs to take the call. That way he only answers the important calls, and can let others leave messages.*

3.3.1 Analogy for process improvement coaches

The IT industry is full of Busy Bills, and their counterparts, the Busy Barbs. There are often a few key players that are essential to any sustained improvement effort. The program manager (PM) comes to mind, and the PM job either breeds or attracts Busy Bills.

In the same way Bill knew exercise was important, the PM knows that process improvement is important, but often the process improvement payoff is long range and corporate, such as better company performance on the next program or project, or a competitive advantage in a future bid. But, the PM's focus is immediate and pressing, often with contractual obligations that supersede everything else.

So, process improvement is crowded out by day-to-day demands. In Stephen Covey's terms, the urgent overwhelms the important.

Most of Jonathan's strategies will help. In addition, there are three implicit strategies that Jonathan used but did not articulate fully. They are to:

(1) Simplify, simplify, simplify. This keeps you from overwhelming your Busy Bill; keep them focused on one or two critical things that will move things forward until there is more time.

(2) Eschew timelines for improvement. Especially, don't box yourself in with fixed dates for meeting goals that depend on Busy Bill.

(3) Build a relationship that will allow more focused work when (and if) the crises du jour passes.

Following Jonathan's model, continually assess the PM's mental condition. Do they need to blow off steam? Provide a vent. Try to become the PM's sanctuary. If they are unable to keep regular appointments, study their routine and find ways to be visible, catching them in the hall.

Give them homework by asking for specific short-term help on critical items. For example, "I need you to push the test team a bit to finish evaluating the new tool. I know they are very busy, but we can't implement it in time for the next release without the evaluation."

Good luck with the cell phone thing. If you figure out how to get the PM's phone, let me know. You may be able to get the PM's assistant to screen calls during your meetings. One thing I have seen work during larger meetings is to get the PM to announce that everyone needs to turn their cell phones off. Then they will obligingly set the example.

3.4 The Assiduous Monster

From Jonathan Goodman:

> *The Assiduous Monster is the hardest working person in the gym, day in and day out. This type of client loves to "feel the burn" on every exercise, and he won't stop until he's given everything he has. He leaves the gym exhausted after every workout, and rarely misses a day of training.*
>
> *You may be thinking – "So what's wrong with that?"*
>
> ***Challenges for Assiduous Monsters***
>
> *It's difficult to convince Assiduous Monsters to slow down when needed. But going all out every workout may eventually lead to injury and in order for progression to happen, you have to program lighter days among the tough ones.*
>
> *After years of training, the Assiduous Monster may also have bad form along with significant injuries and imbalances that must be fixed. A thorough assessment will help you determine how to address these issues.*
>
> ***Techniques for Dealing with an Assiduous Monster***
>
> *These clients have usually participated in exercise programs before, but the programs probably weren't well rounded. They may have focused on strength training while ignoring mobility and cardio, for example.*
>
> *Assiduous Monsters may not see the value of a personal trainer, so your first step should be to educate them on the benefits of working with somebody who has the requisite knowledge and passion—namely, you.*
>
> *A full assessment will reveal what the Assiduous Monster needs to work on, so be frank in telling him what needs to be done. By showing him his weak points, you're giving him a challenge, and his personality will have him chomping at the bit to fix his imbalances.*
>
> *You must take charge when working with Assiduous Monsters. Don't let them dictate their workouts; you're the trainer. Make sure they're aware of why and how your plan is different from what they were doing before, and most of all, that they know how it's going to help them.*
>
> *If the Assiduous Monster refuses to change his ways, encourage him to look elsewhere for a trainer. The short-term gain of training the Assiduous Monster is not worth hurting your reputation, or worse yet, a lawsuit if he gets hurt.*

3.4.1 Analogy for process improvement coaches

By analogy, the Process Monster has participated in improvement programs before, and may wonder why everyone else didn't improve. They may have extraordinary skill with a particular tool or method—say 6-sigma or a particular estimation method—but may not see that their approach is unbalanced and doesn't fit all projects.

They probably don't understand why the organization hired a coach – after all, they have the key.

The Process Monster may be the executive who is sold on a tool or method, and wants it all done now. Or, your Monster may be an over-enthusiastic improvement team member whose missionary zeal for their part of the solution turns off key players on the development team.

The Process Monster may expect everyone else to work as hard as they do. After all, any team member assigned to work on process improvement half time is only expected to work half days. As in, there are 24 hours in day, so a half-day is 12 hours.

In both cases, you probably have a person who knows intellectually that process improvement is only one of many things happening in the organization, but has trouble backing off.

In process improvement, like sports, timing is everything. Implementing the new test tool without having the testers and developers ready to use it is usually an exercise in futility. Rolling it out in the middle of a new release will affect the schedule—in a bad way.

As Jonathan noted, you have to take charge. If your Monster can't be tamed, then they need to be taken off the process improvement team. If they are sponsoring the improvement effort, then you may need to move on.

3.5 Aerobics Alice

From Jonathan Goodman:

> *Aerobics Alice doesn't care about form. She wants to look good and wants to feel each workout, but she's not interested in detailed explanations of exercises, workouts, or the physiology behind them. The Aerobics Alice has already been active by taking exercise classes or working out with friends. She'll see you as a tool for quicker results and will hold you accountable if she doesn't get them.*
>
> ***Challenges for Aerobics Alice***
>
> *The Aerobics Alice client forces you to walk a fine line. Focus too much on form and you'll lose her, but focus too little on form and she may get injured. Without proper form, progression is impossible, which makes it hard to improve beyond a certain point. Without improvement, there are no visible results to show, and you will lose the client. This leaves you in a difficult position.*
>
> *The Aerobics Alice often has bad habits to address as well. It's not uncommon to re-teach exercises weekly. The Aerobics Alice may also want to work out with friends between sessions, which can make it hard for her to stick to the routine you establish. Therefore workout adherence between sessions may be low.*
>
> *In one case, my Aerobics Alice told me that she wanted the workouts to be faster paced, and I pretended not to hear. To my surprise, after our initial sessions, she dropped me as a trainer. Some time later she came back for more sessions, and I agreed, but only after I negotiated a longer-term contract.*

[Authors Note: In many cases, he would have lost the client forever. If Alice were a typical development manager charged with implementing a process improvement effort or a new software development tool, this would have been "all she wrote".]

> ***Techniques for Dealing with Aerobics Alice***
>
> *There are two basic approaches to take with a typical Aerobics Alice client. You need to recognize that your client will go back to group exercise classes or working out with friends, and prepare her for it. Add some aspects of group exercise into your workouts and focus on proper form, and you may be able to keep the client on an irregular but continuing basis, such as once a week or even once a*

> *month. For example, invite her to bring her friends to the session (for a nominal charge).*
>
> *Alternatively, you can take a hardline approach. If you think that Aerobics Alice would benefit more from your training than from what she was previously doing, tell her. Just keep in mind that it's her goals that matter, not yours.*
>
> *If you do take this approach, it has to be an all-or-nothing conversation. You cannot bend on your opinion of the most effective way to train. If she refuses to buy in after you've extolled the benefits of your system, move on. But I've found that often clients respect the hardline approach because you have their goals in mind and are holding true to your values as a trainer.*

3.5.1 Analogy for process improvement coaches

The first step is to determine the decision-making mode of the client. An "Aerobics Alice" is likely to be more collaborative and group decision focused – just as she wants to work out in a group class in the gym, she will want everyone to be part of the team in the work place. This is <u>not</u> a bad thing for a process improvement effort, and actually will help things along if channeled correctly. She may be socially focused rather than intrinsically goal driven. For a socially focused person, gaining team consensus is more important than achieving a metrics-based improvement goal.

She may also be motivated by appearances. This may express itself as a form of risk aversion—"don't do anything that would make us look bad". Alternatively, it may express itself as "We need to change this because it will look good to the VP."

Based on her decision making model, you can "sell" necessary changes as ways to involve the group, ways to reduce risks, or ways to showcase the team. If consensus is involved, be prepared to take some extra time.

The second step is to listen. When any client tells you that what you are doing isn't working, you need to find out why. It may be there are critical players that need to be involved, or that the processes you are developing don't fit the group or client's needs. Your client may not see the connection between an improved Configuration Management system (your goal) and reduced customer complaints (her team's goal), so you need to make the connection.

If you can't gently "sell" the improvements you are working on, you will need to change your approach **<u>or</u>** take a hardline – and be ready to walk away.

3.6 The Quiet Assassin

From Jonathan Goodman:

> *This client kicks butt! They'll come into the gym every time and give you 100%. It won't matter how their day went. This is convenient because they'll never speak about their day.*
>
> *In fact, the Quiet Assassin will never speak about anything other than the workouts and, even then, conversation is few and far between. Getting any information about this clients work, family, or social life is like pulling teeth.*
>
> ***Challenges for the Quiet Assassin***
>
> *It's hard to gather any information about the Quiet Assassin. This makes establishing a relationship difficult. Every workout is like a bad first date. You will receive one-word answers to every question that you ask a Quiet Assassin. They won't ever ask anything about you.*

Since you never know where the Quiet Assassin's head is at, programming in long-term progression is difficult. They'll tell you that they're going on vacation the day before they leave, and often have a preset (and un-communicated) notion as to how long they want to train with you. Getting a grasp on the Quiet Assassin's goals can also be tough. You will find that you catch yourself falling into the trap of disorganized sessions. In addition, any scheduled rest will consist of an awkward silence. Any questions are followed by a one-word answer before the silence returns.

Because of the lack of feedback, dealing with the Quiet Assassin forces you to second guess yourself and make [possibly bad] *assumptions about what their goals are. Retaining this client is difficult because you lack the relationship that you're able to build.*

Techniques for dealing with the Quiet Assassin

The best advice that I can give in dealing with arguably the most difficult client type is to stay the course. The Quiet Assassin has issues that they're dealing with and sees the gym as an oasis. It's important for you, as their trainer, to not get offended when they don't share any information about their personal life with you.

Because she worked so hard and was internally motivated, the trap that I felt myself falling into was becoming complacent.

You might feel like you're caught up in a bad relationship. You'll be working hard planning the workout and trying to get closer to the client. The client, on the other hand, won't seem appreciative and it's easy for you to get frustrated. It's not your job to be their psychiatrist. It's your job to give them a great workout and educate them properly. By sticking to your guns and always giving them a great workout you're providing the best service possible for that client.

I promise that if you keep showing the Quiet Assassin respect and resist getting complacent they will pay it back.

3.6.1 Analogy for process improvement coaches

Your client for the process improvement effort may be massively introverted. This may mean they are not engaged, it may simply mean they don't talk much, or are dealing with personal issues that are, frankly, none of your business.

For some introverts, you may need to find communication channels that are more comfortable than face-to-face (some engineers really prefer e-mail.)

You will need get creative and find other ways to check their support for the effort. A simple way to do this is to ask for help, assigning a homework task more involving than "read and approve". For example, ask for a visible sign of sponsorship. "Can you please send out a note stressing how important it is to attend tool training? There is some resistance and a nudge from senior management will help." Then wait. Don't draft the note for them – tell the sponsor it needs to be in their words. This will help you gauge their understanding of the process improvement effort, and their level of involvement.

I had a sponsor for architecture reviews who scheduled a review for every major release of the products under his control. We never got any feedback directly, except once, when we wrote a finding questioning the business case for a product. He told us that was out of scope for us (I disagreed, so I found another way to word the finding.)

I found out later he was trying to protect the team from a layoff, hence did not want anything in writing that would call the overall effort into scrutiny and make it a target for cuts.

Despite our one disagreement, he kept scheduling reviews, and his teams kept correcting the findings. The dynamic of his organization was that the reviews were the acceptable way for teams to bring issues out into the open, and provided him a venue for correcting things that might not get addressed otherwise.

All in all, an absolutely awesome sponsor, and I didn't even know it until afterward.

3.7 Challenging Charley

From Jonathan Goodman:

> *The Challenging Charlie ~~knows everything~~ **thinks** they know everything. They're skeptical of you and your gym and won't commit to anything long-term. This client type may be experienced in exercise or may not be. Often the Challenging Charlie has done research prior to starting an exercise program.*
>
> *It may seem like they're testing you early on. In fact, they are.*
>
> *In many cases, they have been trained badly in the past, and are working from a model of "once burned, twice shy." Charley may come across as being quite a jerk, when in fact he is just jaded.*
>
> ***Challenges for Challenging Charlie***
>
> *Challenging Charlie doesn't leave himself open to a lot of conversation. Your questions will be met with one-word answers. They've often done some research before meeting you and therefore want to test you before they commit to working with you.*
>
> *On one hand I love it when clients take on the task of self-educating. The problem is that their research is usually internet-based and not always accurate. Setting out a long-term plan with Challenging Charley is difficult due to the inaccurate information they have and their lack of commitment. They'll stop you midway and remind you that they haven't signed up for anything yet.*
>
> *Charley may refuse an assessment, on the grounds they haven't committed to anything, so why waste time?*
>
> ***Techniques for Dealing with a Challenging Charley***
>
> *Since the Challenging Charlie's resistance stems from bad experiences, it's important to take your time to educate them on how your approach is different.*
>
> *Your plan of action has to change. Since Charlie refused an official assessment I secretly worked the assessment into the first 3 workouts.*
>
> *After each exercise I would take a minute to tell him what was going on in detail. When I came to know his patterns, I started to answer his questions before he had a chance to ask them. As the workouts went on Charlie gained more faith in me. I made sure to show him that I had a reason for the exercises that he was doing. I like the process of educating a Charley before they ask. Before every workout with this client type I'll take 5 minutes to explain my plan for the day and why I'm doing it. By closing all the doors to objections before they raise them you prepare yourself for success.*
>
> *The Challenging Charlie should be a short-term problem with two possible situations emerging.*
>
> - *You don't explain yourself well. They'll quickly give you the boot as their trainer.*
> - *You explain yourself well and prepare for their questions or, better yet, answer their questions before they ask them. The Challenging Charlie will shift into a long-term client with a great understanding of exercise in addition to having faith in you.*

3.7.1 Analogy for process improvement coaches

A senior process improvement coach told the story of going into an engineer's office to talk about the improvement program they were starting. As he began talking, the engineer quietly reached into the desk drawer to retrieve a pocketknife. He then used the knife to cut a notch into the expensive wooden top of the desk.

The consultant, startled, looked at the desktop and noticed a row of 8 notches. "May I ask why you are cutting notches into the desk?"

"Yes."

Silence. (This is Challenging Charley at his best – you asked if you can ask the question, now you have to ask the question.)

"Why are you cutting notches into your desk?"

"Each notch represents a new whiz bang program that is going to straighten out all the problems here. The programs go away, the problems remain, and this is my record of it."

Better have a plan, and facts, and persistence.

There is a trap here. A continual stream of challenges may be a deep-seated resistance to the effort, stemming from another cause. To succeed, you will need to find and address the root cause of the resistance. Where Jonathan was able to win over his Challenging Charley by anticipating his questions and having a greater level of expertise than the client had, you may not be in that position. The software development field is so complex, no one can master all aspects of it, and Charley may have specialized expertise you don't have. Then you have to co-opt their expertise and make it part of the change plan.

One way to do this is to ask the Challenging Charley what they would do. They may respond with a detailed story about a previous improvement program they were actively involved with. Then ask – "How did that work out?" They will often reply with a diatribe about how it failed because of X, Y and Z. This gives you an opening. The follow up question is, "How can we address those barriers?" If Charley has an answer, make them responsible for the result, incorporate it into the plan, and step aside. If there isn't a good answer forthcoming, then in the best case you can suggest trying something new (with some buy-in), and in the worst case, they will sideline themselves along with their challenges.

For example, during a risk identification session I was challenged by a mid-level manager "Why are we wasting time with this, when we already have standard lists of the problems and risks?"

My response: "That should save us some time! Please bring me the standard list, I will gladly include those risks in the prioritization sessions long with the ones we identify in the interviews." No list was forthcoming, and the manager disappeared, leaving the other managers free to work on the risks and issues they identified.

3.8 Always Off Track

From Jonathan Goodman:

> *The Always Off Track client never focuses on the workout. It's constantly a struggle getting this client to give 100%. They enjoy your company and are happy to chat about whatever pops into their head. On the bright side the Always Off Track client makes time fly by entertaining you with stimulating conversation.*
>
> ***Challenges for Always Off Track***

> *Focus! Your job is to get the client results. Results are what they came for and they won't get them unless they're engaged in their workouts. Also, if they're not focused on the exercise then you will have to repeat yourself constantly. Getting the Always Off Track client to complete their workout in the allotted time can be a challenge, since 45-second breaks inevitably turn into 3 minutes.*
>
> *This client will also get bored with your workouts since they don't take the time to dive in and understand the purpose. They constantly ask you to change it up.*
>
> *Lastly, the Always Off Track client is an avid reader and, since they are now excited about fitness, will want to try every new invention or workout regime. You'll find that you spend a lot of your time at the beginning of sessions explaining whatever new fitness trend is out there* [and why they aren't doing it today].
>
> ***Techniques for Dealing with an Always Off Track Client***
>
> *The Always Off Track client might take a little creativity. Your goal is to figure out a way to increase their focus without losing sight of their goals. Using a stopwatch works well, as it communicates a strict end to the break. "When you hear the beep, you go!" "Positive punishments" such as plyometric pushups for non-compliance may work well.*
>
> *Sometimes all it takes is sitting the client down, revaluating their goals and making sure that they buy-in to the program. The Always Off Track client might talk because they don't know what else they should be doing during the breaks. When I feel that's the case, I make sure to educate them on the importance of staying focused and teach them visualization techniques.*
>
> *I've never had an issue where a client has gotten bored of the workout that I'm putting them on – but, it may take some extra work on your part to ensure this. Boredom* [often] *stems from a lack of understanding on the client's part. If you explain your workout, progressions and vision properly to your client you won't have an issue. They'll be motivated by small consistent progress because they know what to look for.*

Analogy for process improvement coaches

Ah yes, the always off track team or client. Meetings that drag until everyone's bladder gives out because the senior architect keeps talking, the manager who always wants to implement the latest buzzword trend, the free spirit who has wonderful creativity but no follow through, or, better yet, a team with all three aspects in play at the same time.

By analogy to Jonathan's stopwatch and beeper, meeting management is a must. Agendas help. Timers help. Hard stops help. "Parking lots" help. Standing up helps. Agreed upon "penalties" for meeting tangents help. I heard of a manager who replaced the meeting room clock with one that had dollar amounts on the dial to show the mounting cost of the meeting in real time. A trained facilitator to keep it all moving forward is invaluable.

You may be able to distract the senior architect by asking them to come to lunch with you and tell you more about the fascinating approach that worked back in '82, while reminding them that there is a hard stop today. This protects the other team members, and you may learn something as well.

Jumping from trend to trend is tempting for many managers, but trying to do too many things in one improvement cycle is lethal, and in many cases, the efforts may be orthogonal. For example, if you have trouble managing release configurations for four hardware platforms now, adding the additional complexity of supporting mobile devices and a mobile testing lab is not a good idea right now. If security is an issue, moving critical data storage into the cloud is unlikely to help.

Pairing people with unlike temperaments is another technique that can help a lot. For example, if the free spirit can be paired with a slightly anal detail wonk, they will drive each other nuts, but may achieve great things. They will need to be rewarded and praised as a mini-team, and you may need to educate them about how their differing strengths complement each other.

4. Summary

Reasoning by analogy is always suspect, and while it can lead you to insights, it can be misleading as well.

The premise of this paper is that there are many areas where personal training and process improvement are analogous. This occurs naturally because the focus in both cases is on changing the way we do things to achieve better results. The similarities provide a rich source of potential lessons learned, but they must be applied judiciously. Any time you reason by analogy, there are areas where the analogy won't hold, and may even be misleading.

For example, there are differences between individual difficult clients and teams. The personal training analogies may hold with individuals on the team or sponsors of the process improvement effort. However, they totally fail to account for team dynamics that may mitigate or exacerbate individual weaknesses.

A final word **- avoid complacency**; by some measures, the fitness business is failing (and process improvement isn't doing much better). The number of obese Americans is going up, to the point that it has been described as an epidemic. This is occurring in spite of boatloads of evidence linking weight to heart problems, strokes, and diabetes.

According to the University of Scranton Journal of Clinical Psychology, 12/13/2012, 45% of Americans make New Year's resolutions, and of those, the biggest number have to do with personal improvement or weight loss. And, although people are ten times more likely to meet their goal if they express them explicitly as a resolution, only 8% of people achieve their resolution; 92% have partial or no success.

While this is only one survey (beware of cherry-picked data), it showcases a simple fact—improvement is hard. Although a LOT of effort and money is spent on it, based on results, much of it is misspent.

In the process improvement field, data from Crosby (Quality is Free) and others indicate that almost 80% of improvement efforts fail to meet all of their success criteria.

In both cases, while coaching improves the chances of success, there is a dearth of hard data to substantiate this. In fact, we can cherry pick data to contraindicate coaching. For example, in a ten-team football league, there is one team that is the champion, so by one measure, 90% of the coaching efforts can be categorized as failures, or partial successes at best. (Since all of the teams have coaches, there isn't a control group to determine the success rate of un-coached teams.)

Elite athletes have coaches, the "biggest losers" have coaches, and Little League baseball teams have coaches. Successful coaches provide scholarship and domain knowledge, motivational skills, encouraging words when the going is tough, impartial evaluation of progress, and a bit of "tough love" when needed. This combination makes the difference between mediocrity and being the best you can be.

References and Credits

This paper draws on 30+ years in the IT field, most of it spent in product reviews, risk management, quality management, and process improvement.

The bulk of the personal training stories are lifted directly with minor modifications from Personal Trainer Development Center (PTDC) website: http://www.theptdc.com/ .

The difficult client examples are also contained Chapter 9 of “Ignite the Fire: The Secrets to Building a Successful Personal Training Career”, Jonathan Goodman, 2012, ISBN 978-1468168273

I would be remiss if I did not mention my intellectual debt to Jerry Weinberg, Watts Humphrey, and W. Edwards Deming.

Turning Projects into Products (and Back Again)

Adam Light

adam@sotechadvisors.com

Abstract

Agile methods provide a product-centric model for software development. Projects, however, are deeply ingrained in corporate management systems and in the business models of service companies. Having established agile methods at the team or department level, many software development groups find further improvement hampered by conflicts with project-based policies, procedures, and practices.

How can you take full advantage of cross-functional teams and a product-centric implementation approach within a project-based planning process and in the face of policies and procedures that seem designed to prevent scope trade-offs? When software development is only one part of a larger work effort, how can you make a successful case for optimizing globally rather than locally? And with so many project, program, and product management roles involved, how can you maintain role clarity and a measure of stability even as you work to evolve those roles?

This paper presents a simple framework for successfully integrating project-based and product-centric approaches within a single, continuously improving workflow at the enterprise-scale. It describes an approach that explicitly defines the boundary between project and product-centric processes and maintains delineation as changes and improvements occur. If your organization depends on projects, you are experienced with Agile methods, and you have already tried the scrum of scrums, this paper may help you determine the next step to take.

Biography

Adam Light is Management Consultant and Principal at SoTech Advisors where he helps organizations apply lean and agile methods to unlock greater value from software development. Adam works with enterprise clients to adopt and scale agile methods, design and operate agile work systems, plan and deliver critical projects, and build the knowledge foundation necessary to sustain continuous improvement.

Adam has nearly twenty years of experience planning and executing complex software initiatives across a range of applications and industries. Prior to founding SoTech Advisors, Adam was Director of Planning and Program Management at TransUnion. Adam graduated with honors from Dartmouth College, holds a PhD. in Geography from the University of Oregon, and is a certified Project Management Professional.

1. Introduction

The Project Management Institute defines a project as "a temporary group activity designed to produce a unique product, service or result." (Project Management Institute, 2013) This definition emphasizes the unique aspect of projects and explicitly contrasts projects with routine work or operations. But many project elements may be viewed as routine work by the specialists who often provide services to deliver a project. In the development of software applications, for example, sizing and provisioning hardware may be accomplished by dedicated staff or outsourced to a provider of "cloud" computing services.

The adoption of Agile methods by software developers is causing many to re-think what they consider routine work. To be sure, knowledge work such as building software remains highly variable and still requires creativity, but stable cross-functional teams may be able to accommodate variation more readily, effectively, and reliably than projects can assemble a new group of people into a team capable of delivering a high-quality product. Many organizations and managers report shifting from a project-based approach that assembles groups of individuals to meet fixed scope and schedules to a product-centric (West and Wildeman, 2009) approach that focuses on maintaining stable self-organizing teams who deliver features and functions on a continuous basis.

But while product-centric development techniques are on the rise, projects remain everywhere on the landscape. Project-related concepts are often deeply embedded in organizational systems for selecting, funding, and monitoring work. A large enterprise, for example, may select and fund IT investments in the same regular annual budgeting cycle used to fund investments in facilities or factory equipment. And in relationships between companies, projects are the means by which service businesses usually transact with their customers. Contracts for outsourced software development, for example are often structured along project lines.

So why can’t we simply combine project-based with product-centric approaches? Working together, shouldn't project managers and product developers be able to start out on a sure footing, deliver value rapidly, and stop when the time is right to move on to something else? Product and project-based approaches can co-exist within the same workflow, but the two approaches represent distinctly different strategies with fundamentally different assumptions and work best when the boundaries between them are clearly delineated.

This paper explains why "hybrid" approaches to delivering software that mix methods unintentionally often perform poorly, and presents an approach for successfully integrating project-based with product-centric business processes. After outlining the challenge inherent in combining the two approaches, I introduce the metaphor of air traffic control to explain the recommended approach and outline a framework for managing the flow of development and delivery initiatives through pipeline stages. I then present six steps to implementing a project delivery pipeline with a product-centric core.

2. The Challenge

At the center of the challenge lies a familiar paradox. Whether you are conceiving a ground-breaking new product or starting a book, beginning something new is often difficult, yet once you get going there is never enough time and you always wish you had gotten rolling sooner. Even if you have undertaken many similar initiatives before, the inertia of beginning frequently feels insurmountable. In what Donald Reinertsen (Smith and Reinertsen,1998) has famously called the “fuzzy front end” there are so many degrees of freedom that both the problem and the solution seem unbounded. A range of stakeholders and stakeholder groups with varying interests and perspectives often multiply the difficulties of finding a starting point. Product-centric and project-based methods approach this challenge differently.

Projects attempt to solve the problem of getting started by forging agreement on scope, schedule, and budget - the so-called "iron-triangle" of project management. If we can just agree on something (anything) about what we want, when we need it, or how much it should cost, then – the argument goes – we will gain some kind of toe-hold and the basis for moving forward. Constraining design options may become a goal in the name of progress. Product-centric approaches, by contrast, acknowledge that starting is very hard and try to avoid the problem altogether by stopping as infrequently as possible, striving to operate a "design factory," through which product ideas flow.

Combining product-centric and project-based methods proves difficult because, in theory and in practice, the two approaches make different assumptions when accomplishing the same activities. Often those assumptions are not clearly stated or even consciously acknowledged by those who hold them. Friction emerges when people try to work together without clearly stating their opposing assumptions. Under such circumstances, unprincipled compromise does not necessarily make processes work better. Duplication of effort often results. In the worst case, incompatible approaches may result in different segments of the development workstream actively working against one another as factions form and one group strives toward the ideal of flexible scope while another strives toward the ideal of fixed scope.

The ability to increase value by varying scope lies at the heart of product-centric development methods. Well-crafted software built using modern languages and techniques remains malleable long into the development process. And the development process is as much a matter of learning and knowledge creation as a matter of construction. By fixing scope too early, organizations relinquish their main lever for value creation, while assuming that scope is going to change helps foster discipline on the part of developers by creating an incentive to emphasize flexibility and re-use over speculative feature development.

Variable scope forms the foundation of delivering business value with Agile methods. When flexible software is released frequently, along with mechanisms for rapid feedback to the people who are in control of the product scope so they can make trade-offs, the premise is that the best results will emerge. The evolution of technology makes this increasingly possible. In a world of connected devices, and with the decreasing overhead of releasing, software release cycles have shrunk and users have gradually learned to tolerate, expect, and even welcome regular software updates.

Between organizations that agree to share the work of software creation, however, it is not always possible or desirable to contract for a well-defined product from the outset. The value of supplier expertise often arises from the ability to frame problems and define the product, creating a chicken and egg situation. Counterparties frequently need to commit to working together before the product becomes well-defined so they craft agreements. Whether explicitly or implicitly, the "iron triangle" of project management is at the center of these agreements, often embedded in legal contracts, and this binds both sides to a project-based relationship. Scope that has been fixed early in the development process may prevent effective trade-offs that would later increase value.

In organizations of all sizes, established management systems assume the existence of projects. Budgeting processes often allocate funding by projects. Management controls release funds according to project milestones, staffing processes assign people to projects, and performance incentive plans refer to project outcomes. Companies that choose to capitalize software development base their accounting on guidelines that emphasize activities as much as the actual product of those activities. In many organizations, the existence of projects will be a fact for some time to come.

Formalized project management concepts run through many contracts and much has been written on the topic of creating Agile contracts (Arbogast, Larman, and Vodde, 2012). This article does not address Agile contracts directly, but if the proposed approach for reconciling project-based and product-centric processes within a single workflow can lead to more unified understanding of the development process, then contracts based on that understanding should better serve all parties.

While product-centric methods work better for some portions of the system delivery workstream, project-based processes may make sense for the other parts. And it may prove easier to embrace product-centric methods in stages. Organizations that can effectively integrate project and product-centric methods are able to start where they are and continuously improve from that point. The boundary between product and project-based workflows can then be managed based on changing conditions and as a function of continuous improvement.

3. Air Traffic Control for Projects

For most people, the phrase “air traffic control” brings to mind an image of the people with binoculars who staff the tall towers standing above every major airport. In reality, however, the modern air traffic control system is more complicated than most travelers understand (Freudenrich, 2001)

As a plane moves from the gate at its airport of origin to the gate at its destination, it passes through multiple domains of control whose overall structure resembles the “zone” defense used by basketball teams. Figure 1 shows the path of a flight through these multiple zones. Each zone is suited to a different purpose and the total air traffic control system employs a well-defined set of protocols to knit the multiple zones together. Among other things, these protocols include the use of well-defined travel corridors and communication standards for handing off responsibility from one control center to the next. Key information about each plane and about the airspace has been standardized; controllers use the standardized information to coordinate air traffic under changing conditions.

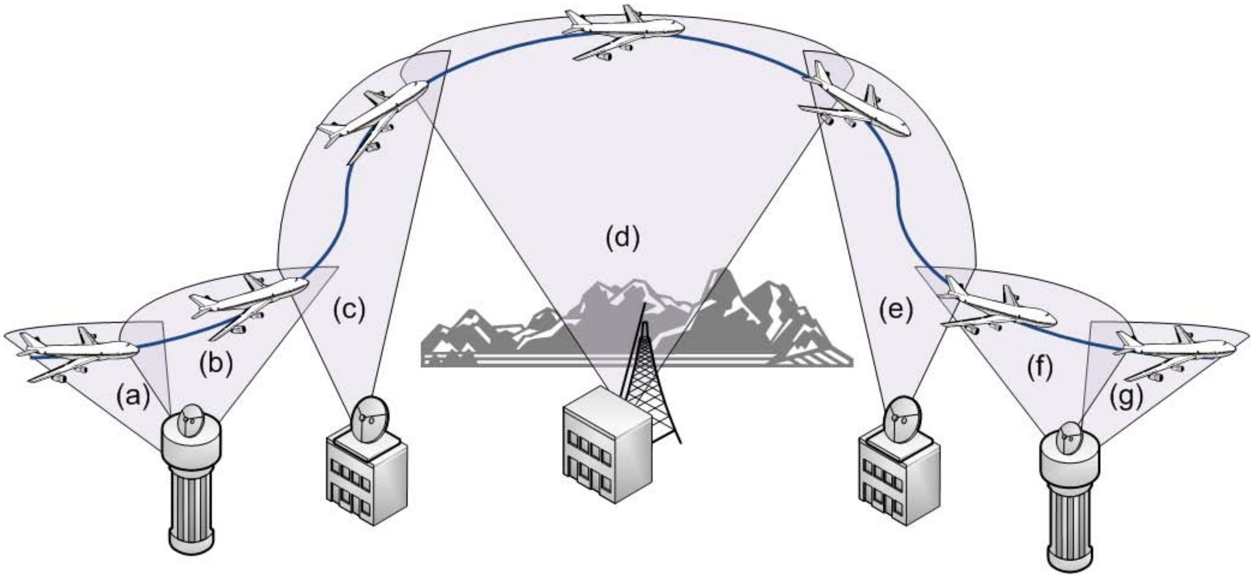


Figure 1. Flight path of an aircraft through air traffic control divisions.

The ground controller (a) is responsible primarily for the safe and orderly operation of traffic on the ground: getting planes from the gate to the runway as efficiently as possible. Once the ground controller completes his task, he hands off to the local controller (b) who then hands off to the departure controller (c) who hands off in turn to the center controller (d). Landing reverses the sequence (e, f, g).

The air traffic control system serves the safety and information needs of individual flights, but its more important role is to prevent collisions between flights and to minimize congestion. Maintaining safe and efficient airspace requires much more than providing a collection of uneventful flights. The complexities involved and the sheer size of the task (at peak times there may be 5,000 airliners in flight over the United States alone) make it impractical to simply dedicate a single controller to each flight over its entire journey. Such an approach would be both inefficient and inadequate for the task since it would simply magnify the challenges of coordinating so many flights. Its costs would be prohibitive and the system could probably not ensure safety or success across all the flights.

Within most large enterprises, and certainly between companies who contract with one another for services, sponsors and stakeholders tend to view themselves as customers in a service-provider relationship. They may escalate complaints to senior management when forced to make scope trade-offs or accept delays. But attending to each individual initiative or project cannot necessarily meet the needs of all the initiatives and all their customers. Modern enterprises are systems: large, differentiated, and complex. Individual projects frequently interact and conflict with one another, requiring trade-off decisions, and most projects must operate within constraints often shared with other projects.

Successfully delivering multiple concurrent and overlapping initiatives, then, requires a more sophisticated solution than simply assigning one (or more) project managers to each sponsored initiative. Regardless of how stakeholders view their separate projects, managing a collection of projects solely by dedicating individual project managers to serving individual projects and their sponsors over the entire lifetime of the project is no more adequate than dedicating individual air traffic controllers to serving individual flights. Such an approach is both inefficient and ineffective.

When a plane in flight experiences mechanical difficulties or encounters air turbulence, its pilots are capable of responding or adjusting course without direction from outside the plane. Like the flight crew of an airliner, self-organizing work teams, such as those defined by the popular Scrum framework (Schwaber, 2004), possess the necessary skills, experience, and resources to operate independently – both under routine circumstances and in the face of unpredicted events. The role of a project manager where work is organized in a product-centric model is to function like the center controller who monitors planes as they cruise across the country, the project manager can tune into information that the team doesn't regularly keep an eye on. The value added by the project manager in a product-centric model comes from maintaining a bird's-eye perspective and in monitoring the environment within which the team operates.

While a flight is on the ground, and during take-off and landing, by contrast, the ground controller, local controller, and departure controller monitor much more intensively and instruct the flight crew directly to keep the flight safely within established travel corridors. Like planes, projects are much more expensive in flight than at rest and they are much more likely to require guidance when taking off and landing. Project-based processes usually make the greatest sense in the beginning and ending stages of software delivery initiatives.

And flight crews are clearly not equipped to plan profitable routes, lease equipment, organize crew member schedules, manage fluctuating ticket prices, secure landing slots, handle luggage, or indeed to conduct any of thousands of other value-producing activities necessary for a flight to meet the needs of its passengers, crew, and airline investors. While these many disparate functions have been standardized in the operations of an airline, development work is highly variable and project managers usually find opportunity to add value by organizing and aligning elements surrounding the core development work.

In the past, demand for project management was highest during implementation, but with the advance of Agile methods this is no longer the case. To maintain the flow of development work, project managers should function more like air traffic controllers than pilots, deploying their resources in zones across the development cycle according to the needs of the organization and recognizing that continuous improvements in the flow of operations may cause their focus to shift over time.

4. Success Framework

While traditional approaches to project management gave the responsibility for assembling and aligning teams around the required work to the project manager, Agile methods form and maintain stable teams, devolving responsibility for alignment to teams and team members. What remains is the task of bringing the right work to the team at the right time.

The "ideal" Agile organization aligns stable cross-functional teams around specific products or product components. Where the work for any given team is characterized by a high degree of continuity over time, the team can simply “pull” the next unit of work, and the organization functions like a factory with Agile teams as its work cells. In circumstances where self-organizing teams flourish, project management may play no role or may play a supporting role by providing specialized services to the team.

But where work is not characterized by a high degree of continuity, making appropriate work available to a specific Agile team at the right time becomes critical to success. Without a steady flow of available work, the Agile team cannot function smoothly and project management can play a central role in maintaining the flow of work.

The framework described below contains three basic elements: (1) the Agile “core”, (2) a pipeline for delivering work through this core, and (3) the ideal of continuous flow. With these three components in place, a stable, continuously improving system can emerge.

The Agile core

Giving responsibility to the development team typically works best when that team is stable and cross functional, and operates within an environment where it can control or influence things that enable it to carry out its responsibilities. Such conditions enable the team to experiment in order to find out what practices work most effectively.

Everything that follows in this article assumes the previous or concurrent implementation of Scrum or another approach to Agile work management. Organizations with any scale and longevity should also implement protocols for staffing, forming, and managing teams. And Scrum can only succeed to a limited degree without the tooling and knowledge to implement modern technical practices like continuous integration, test-driven development, and test automation. Empowered self-organizing teams form the backbone of Agile software organizations and coupling them with the right technical practices is the reliable way to improve productivity.

Making mistakes and taking accountability for not repeating them is an essential part of learning in cross-functional teams, and teams where a project manager regularly intervenes to prevent or curtail mistakes are less likely to exhibit effective self-organizing behavior. There is much more to forming and maintaining self-organizing teams than making day-to-day project management a team responsibility, but doing so is a prerequisite.

In the recommended framework, cross-functional teams make up the core of the pipeline through which work flows. Organizations that rely on projects but also want to achieve benefit from Agile methods should plan and prepare to explicitly re-deploy project management resources away from the Agile “core” as it forms.

Pipeline management

Managing a pipeline means tracking and actively controlling individual work items through a series of workflow stages in order to optimize flow and maximize the combined value of all work items delivered.

Pipeline management is often seen in sales functions where a typical business goal might be to grow the number of strategic accounts while maintaining or increasing revenue, or to maximize the total amount of sales in a particular time period. Pipeline management enables the sales organization to optimize globally rather than locally, and centralizing information in a sales force automation tool, such as Salesforce.com, helps managers maintain continuity when employee turnover occurs.

Making all the work visible in one place is a commonly cited benefit of pipeline management. Tracking and reporting on the pipeline may reveal patterns and help identify opportunities to simplify workflow. With a global view, managers (and other team members) can assess and respond to constraints in the work system rather than focusing efforts on expediting individual work items.

Pipeline management is also commonly applied to new product development which is modeled as a funnel whose purpose is to progressively develop ideas into a portfolio through active (rather than passive) selection processes, but the extent and effectiveness of pipeline management often degrades as work flows from portfolio selection into implementation.

In response to the complexity of development work, many organizations form functional “silos.” In a product company, this may mean product management, engineering, and “go-to market” teams.” Within the engineering function, especially inside internal IT organizations, the silos may divide into functional units for quality assurance, business analysis, user experience design, operations and so on, and developers may also specialize in particular technical competencies such as integration or business intelligence.

Figure 2 shows an example of pipeline stages defined for an IT organization where project-based and product-centric processes coexist. Following project startup and once work has been transitioned to the Agile team and Product Owner, project management plays a supporting role. The PMO (project, program, or portfolio management organization) and business decision makers, actively manage the flow of work by tracking work in progress (WIP) and queue length in each pipeline stage to identify emerging bottlenecks and prevent overload in later stages. Once development completes, project and program managers re-engage in a leadership role to perform release management and launch activities.

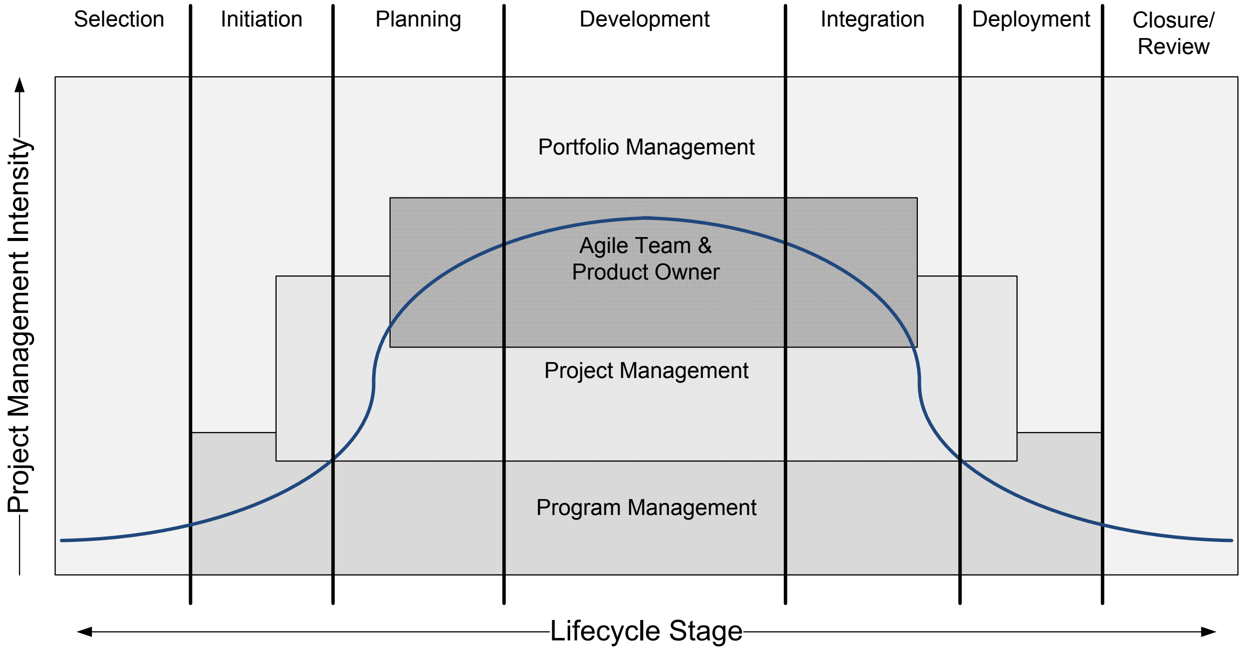


Figure 2 – Pipeline stages in an organization that integrates project-based and product-centric processes

Managing an individual project or initiative across functional silos is a familiar challenge to anyone that has ever worked in a development organization of any size, and project managers specialize in meeting this challenge. Related projects are often grouped into programs and PMO organizations work to “roll-up” projects and programs in order to provide visibility but as described above using the metaphor of air traffic control, focusing measurement and accountability at the level of individual initiatives does not optimize for the whole.

Managing a delivery pipeline implies assessing priorities and the performance of the overall portfolio at every stage and following policies and practices designed to maximize overall portfolio goals. Effective pipeline management can increase pipeline velocity – the rate at which individual work items move through the pipeline, and also improve throughput – the value or quantity of volume of items that move through the pipeline.

Pipeline stages, as illustrated above, may call to mind concepts made popular in software development under the heading of Kanban (Anderson, 2010). Kanban describes a specific approach to lean value stream improvement. The first step in Kanban is to make work visible and visualize workflow, which is exactly what the project pipeline does. The primary distinction is one of scale. While Kanban has most often been applied beginning with individual feature requests or even tasks, pipeline management looks at overall work in progress in the overall project portfolio and could therefore be seen as an initial step in the direction of Kanban at the enterprise scale.

The ideal of continuous flow

The term "lean" entered the vocabulary of business with the 1990 publication of *The Machine that Changed the World* (Womack, Jones, and Roos, 2007). Since the publication of "Machine," many authors have devoted themselves to studying the inner workings of the Toyota motor company. Among these is Mike Rother. In his 2010 book *Toyota Kata: Managing People for Improvement, Adaptiveness and Superior Results,* Rother describes the critical role of a guiding *process-based* ideal for continuous improvement at Toyota. This vision, which he describes as Toyota's "true north" for production can be summarized as "synchronized one-by-one (1x1) flow from A to Z at the lowest possible cost." (Rother, 2009, p. 45)

Value chain analysis and related lean techniques can help everyone see the big picture, but ongoing continuous improvement comes less from re-configuring the value chain and more from improvements at the detailed process level. Often, as easier improvement ideas are implemented, progress grows more difficult. Without a guiding vision like one piece flow, process-level improvements may oscillate between extremes, and improvements in different parts of the value chain may not be aligned.

The third element of our success framework, then, is the ideal of continuous flow that maximizes the total value of the portfolio delivered. Visionary statements can always be refined; the right way to state your initial vision initially is in a way that resonates within your organization.

5. Six steps for putting it into practice

Given an Agile core of cross-functional teams capable of operating in a product-centric manner, how should you implement a system for managing projects through this core? This section describes six steps for creating a work system that combines product-centric and project-based processes within a single workflow.

This article recommends first implementing a framework that clearly defines pipeline stages and the boundaries between project-based and product-centric processes, and then implementing protocols for making decisions about work flowing through the pipeline. Only after process boundaries are established and management aligned and educated should detailed roles and responsibilities be worked out for Project Managers, Agile teams and other actors in the different processes.

5.1 Build a coalition and communicate the vision for a pipeline model

The first step is educating decision makers on the concept of a project pipeline with an Agile core and, in the process, building support for subsequent steps. This step is extremely important, and should generally be the first step of any visionary change. This step corresponds to steps one to four in John Kotter's classic 8-step model (Kotter 2012.)

When implemented in concert with other Agile adoption or scaling efforts, the pipeline model is one component of the overall change. In situations where earlier Agile adoption efforts have not addressed governance and project management effectively and holistically, this change may stand on its own and coalition building may require overcoming disillusionment or misunderstanding about prior work.

5.2 Implement a master decision record

After obtaining buy-in and educating decision makers around the pipeline concept, the second step is establishing an effective framework for decision making.

Begin keeping a single record of important decisions about work and the organizational system that performs work. The structure of the decision record will evolve over time, but the most fundamental elements include the decision, the element of work or work system about which the decision was made (project, team, etc.), the decision makers who were present, and the information (presentation, report, etc.) that formed the basis for the decision.

Establishing a basic record of decision making is important for several reasons. First, it creates transparency that makes inconsistencies visible. The decision record provides a foundation for communicating effectively about decision status and outcome. Contradictory decisions also become immediately evident.

Secondly, ad hoc decision making represents a significant source of variability in many organizations. If two managers or teams independently make different decisions based on the same information, it may or may not be important to identify and reconcile the differences at the time decisions are made, or to standardize any particular decision type, but eventually, the organization will want to standardize certain types of decisions as guidelines, policies, or norms in order to ensure certain results and this will be impossible without understanding the variable processes that produced them. The decision record serves as a key basis for continuous improvement.

Last, and perhaps most importantly, culture change begins with behavior change. Few behaviors are as central to organizational culture as the way decisions are made and communicated. The simple act of making decisions explicit is a neutral action if done properly and in the context of making the pipeline visible. But asking who made a decision and what it was begins the process of rationalizing decision making.

5.3 Define explicit processes around making and communicating decisions

To avoid dilution of responsibility and a silo mentality that prevents flow, a single group or committee of decision makers should oversee the pipeline from one end to the other.

As with the decisions themselves, you cannot systematically improve the decision making process unless you make it explicit. While decisions that affect development may be made by different parts of the organization, some kind of single forum should be devised to review all decisions. A good starting point is the Operations Review format popularized in the Agile community by David Anderson (2003). Implementation can vary greatly but, consistent with the pipeline model approach to project management, agendas should focus on discrete segments of the pipeline. Referring to Figure 2, this might include separate segments for portfolio selection, initiation, planning, and development.

Complex modern organizations contain overlapping functions and pursue multiple aims simultaneously, requiring shared priorities and collective action. This typically means some kind of decision making committee. As with any team, there will be turnover, but you should keep decision-makers together throughout the pipeline

If structure and complexity did not already mandate some level of coordinated activity, providing consistent management to cross-functional teams requires management itself to become cross-functional. Whereas Scrum creates a container for self-organization and prescribes networks across which relationships form readily and reliably, the open-ended structure of committees contributes to their notoriously slow and uneven formation.

Why focus on something as seemingly unglamorous as committee protocols? Just like any other team, decision-makers need to learn to work as a team. Assuming realistically that some of the most important decisions in the organization will be made by committees of managers, and that those committees will take longer than teams to become proficient at decision making, it makes sense to (1) begin down the path to proficiency as soon as possible, and (2) aid the process by whatever means available.

Expressing processes as fractals is a common Agile concept. From an Agile standpoint, a committee is a team of decision makers who follow a more or less formal process to break down epics (projects) into releases and deliver them incrementally. Their work is accomplished through the direction they give to other teams and the decisions they make as a group.

5.4 Tie decision-making cadence to delivery cadence

Guided by a lean vision of one piece flow, value should be “pulled” by the customer and upstream processes should be pulled by downstream processes. The teams that form the pipeline are the work cells that produce value, so the cadence of decision making should be dictated by the iteration cycle of the teams.

If cadence is aligned, you can calculate decision cycle time from two important perspectives. First, if a team identifies a decision that needs to be made, what is the average or maximum time before the decision-makers will be able to convene and make that decision. Second, if decision-makers make a decision, how long before they are given feedback on that decision?

An important point to make here is the relationship between planning horizons and decision cycle time. Avoiding disruptions in flow requires planning at least as far ahead as the maximum decision cycle length and explicit acknowledgement that there is more waste from inaction than from action. That is to say, lack of preparedness for a decision should not prevent work in the short term.

5.5 Define project management roles explicitly

Projects are managed with varying degrees of formality in different organizations, and many authors use the term “project” in a relatively imprecise way to mean something like “the enterprise we’re all engaged in to achieve a specific aim.” Imprecise communication frequently obscures conflict between those who aspire to stabilize processes by improving project management and those who wish to institute and improve continuous processes. Once the pipeline is in place and decision making has been rationalized, project management roles can be defined precisely relative to specific pipeline stages.

Many times people start from the beginning of the business process to define roles, but it is better to proceed from areas of certainty to areas of uncertainty in the end-to-end flow. Also, it is better to start with the to-be process rather than the as-is process. Combining these two things implies that you should start with the product-centric part of the workflow and then define the handoffs into and out of it, agreeing to revisit the roles at a specified time in the future.

The most critical hand-off is from initial mode where scope may be fixed and schedule and budget vary to the condition where scope is varied to maximize value.

5.6 Pursue retrospective improvement toward the ideal

Just as an effective team is more than the sum of its parts, an effective organization is more than a collection of teams. Cross-functional teams lack defined positions and so they must form by working together and improving iteratively. When program and project managers are interacting with teams and decision makers to make good decisions, then all must be involved in effective retrospectives.

If the ideal is explicitly stated as product-centric flow, then at each stage you will try to chip away at the necessary wastes of project management; perhaps moving upstream to forge different relationships with sponsors in the client organization, or developing new policies for capitalization.

6. Making Room for Projects While Scaling Agile Delivery

Initially, Agile methods developed and evolved at the level of the development team. Building on the potential of dynamic, object-oriented programming languages, Agile methods demonstrated enormous potential to improve productivity and rapidly gained favor following the publication of the Agile manifesto in 2001.

A large part of the mechanism by which Agile methods grew rapidly in popularity was the marketing of Scrum through certification. Scrum became an accessible starting point for understanding Agile methods. Community support grew around Scrum as a sort of open standard. Scrum had many similarities to other Agile methods that were prevalent at the time of the Agile manifesto; for a variety of reasons it was simply Scrum that won out.

Since before the rise of Scrum, Agile practitioners around the world have been exploring what works for scaling Agile methods to the organization level. Fitting Scrum into the larger organizational context is a problem of great complexity. Scrum provides a blueprint for Agile work management at the team level; but deliberately avoids providing a similar answer to the question of what happens outside the team.

Given the practical and commercial success of Scrum, and the fact that there is no equivalent for Scrum at the enterprise level, it is perhaps inevitable that somebody would try to fill the gap. Two attempts to package and market frameworks for Agile methods at scale have recently gained attention: Scott Ambler's Disciplined Agile Delivery (DAD) and the Scaled Agile Framework (SAF) championed by Dean Leffingwell. While a thorough discussion of DAD, SAF, and related concepts lies beyond the scope of this paper, the rest of this section briefly compares the two and then explains how the recommendations above relate to these two topical reference points on scaling Agile methods.

There is a great deal of overlap between DAD and SAF in terms of the concepts and principles, and a great deal of intellectual weight combined behind them. This is good news for the field of software development because it indicates that the state of the practice is advancing rapidly. But with so many experienced, well-qualified thought leaders contending for mind-share and consulting business, there is bound to be debate and disagreement in any field

One long-standing controversy revolves around the certification business model. As mentioned, Scrum training and coaching have been extremely successful from a marketing standpoint. This helped spread effective practices and has enlarged the overall Agile pie greatly, but some parties have grabbed larger slices of the resulting pie than others, which inevitably excited some envy. Also, there is a legitimate point that the industry might be better served by a more diverse dessert table -- particularly as the center of gravity evolves beyond team-level Scrum. Proponents of the Scaled Agile Framework are, from a marketing standpoint, following in the footsteps of Scrum by using a training and certification model while the proponents of Disciplined Agile Delivery reject certification, branding their approach as "disciplined" while claiming to reject branded methods.

SAF focuses more explicitly on setting up a factory for development work. The Agile release train, which features prominently in SAF as the principal means of scaling development is roughly equivalent to what this article has described as the "Agile core" of the value chain. DAD acknowledges the release train as a valuable means for coordinating the work of multiple teams, but includes a variety of other building blocks and focuses on "tailoring" an approach based on multiple variables.

DAD and SAF both recommend product-centric approaches (as do Agile methods generally) and neither includes a specific role for project management. Both frameworks describe ways for accomplishing the traditional functions of project management through different roles. And their authors recommend that current project managers transition into new roles or evolve their roles. In the case of SAF, the authors recommend that project managers move to a release manager role. DAD recommends that project management is a part-time role on most teams filled by one or more existing team members.

Taking SAF as the more concrete reference point, the approach described in this article can be seen as a specific way to implement effective processes for interfacing the product-centric Agile release train in software development with project-based work elsewhere in the organization.

7. Conclusion

In practice, project-based approaches tend to fix or control scope closely, while Agile methods are product-centric and attempt to vary scope in order to maximize value. The work of Agile adoption is in many ways the work of recognizing and coming to terms with this fundamental difference. This article’s premise has been that, from a development standpoint, product-centric processes are optimal, but that practical reasons exist for project-based processes.

The nature and degree of project management support required outside software development will depend on the organization, but within the product-centric Agile core of the value stream, project management should typically operate more as a form of air traffic controller that serves the team and the overall work system. The hazard of a "blended" or "hybrid" approach is that it will be based on the lowest common denominator and that the benefits of each approach will be diluted. Instead we need an approach that maintains clear boundaries so that everybody can move continuously toward an ideal.

Since the two approaches make fundamentally different assumptions, maintaining them both within a smoothly operating workflow requires reconciling the differences explicitly. Where inconsistencies remain unaddressed, systemic misunderstanding may remain underground in the organization, erupting periodically in the guise of conflict over specific projects.

References

Anderson, David J. 2010. *Kanban: Successful Evolutionary Change for Your Technology Business.* Sequim, WA. Blue Hole Press.

Anderson, David J. 2003. *Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results.* Upper Saddle River, NJ. Prentice Hall.

Arbogast, Tom, Craig Larman, and Bas Vodde. 2012. “Agile Contracts Primer,” http://www.agilecontracts.org/agile_contracts_primer.pdf (Accessed July 27, 2013)

Freudenrich, Craig. 2001. “How Air Traffic Control Works,” HowStuffWorks, http://science.howstuffworks.com/transport/flight/modern/air-traffic-control.htm (accessed July 27, 2013)

Kotter, John P. 2012. *Leading Change.* Cambridge, MA. Harvard Business Review Press.

Project Management Institute. 2013 “What is Project Management?” http://www.pmi.org/About-Us/About-Us-What-is-Project-Management.aspx (Accessed July 27, 2013)

Rother, Mike. 2010. *Toyota Kata: Managing People for Improvement, Adaptiveness, and Superior Results*. New York. McGraw Hill.

Schwaber, Ken. 2004. *Agile Project Management with Scrum*. Redmond, WA. Microsoft Press.

Smith, Preston G. and Reinertsen, Donald G. 1998. *Developing Products in Half the Time*, 2nd Edition, New York , John Wiley and Sons.

West, Dave and Roy C. Wildeman. 2009. “Product-Centric Development Is a Hot New Trend: A Distinctive Approach To Software Development That Delivers Exceptional Value,” Forrester Research, December 23, 2009.

Womack, James P., Daniel T. Jones, Daniel Roos. 2007. *The Machine that Changed the World.* New York, Free Press.

The Tail that Wags the Dogma

Rhea Stadick

rhea.d.stadick@intel.com

Abstract

It is a common pattern for quality advocates or advocates of any kind really (Agile, Waterfall, etc.) to start off banging a drum with almost religious zealotry. “This is the way to achieve quality!” “Follow me or be cast aside!” I have certainly done this and many of the change agents and forward thinkers I respect in the industry have struggled with this as well in their careers. However, riding a train of dogmatic viewpoints and practices has, in many cases, ruined the original efforts of change agents and more tragically, created bad patterns of development. This paper and presentation will explore how we can clarify our intentions and work towards creating an environment where the people involved are empowered to think and experiment but still march in the same general direction. As product development becomes more and more complex, different approaches are needed to get to the right outcome and we can no longer afford to apply rigid methods that worked for a simpler environment. This paper will help the audience think critically about where they are being too dogmatic in their quality efforts while introducing some new ways of thinking around complex adaptive systems and complex product development.

Biography

Rhea Stadick is an Organizational Coach at Intel, Corp. She has spent the last eight years in software quality and development of engineering teams. Today she helps organizations across her company develop cultures and competencies to create thriving work environments that support excellence in product development. She received her B.S. in Computer Science from Oregon State University and M.B.A. from Willamette University. For the past several years she has organized the Rose City Software Process Improvement Network (SPIN) in the Portland-metro area that gathers professionals in the area to learn and network.

1. Introduction

Part of my early career began in a quality advocacy role. In my organization I was immersed in a mindset that "we needed to control quality and that if we only used our expertise we could devise great processes that would lead to the high quality goals we sought." Unfortunately, the people using the processes got in the way ("why do they keep doing this incorrectly?") and reality hit ("why is schedule pressure always trumping quality investment?"), among other exasperating issues. I began to see that despite everyone's best intentions, the process and practices we put in place just weren't working even though they were well accepted as best practice. Something else was happening in the environment that we weren't acknowledging.

This paper describes my continuing journey of growing my understanding of systems to create an environment that allows people and business to thrive. While I don't prescribe a recipe or a new profound way of working, it may help you think about how you are approaching your work and change agency today. Perhaps it's great, perhaps you may take another approach next time. The important thing is to continue to learn and not take your current state as the only way you'll ever be or approach your job. We all could do better to not just use recipes but to really learn how to cook in any context.

2. View from the Ivory Tower or "You're Not Trying Hard Enough"

The "beat the drum" management technique of telling everyone they just need to work harder should be relegated to one of those de-motivational posters. You don't create high quality products through brute force, so why do we think putting in more metrics, criteria, and process will help? Ironically, despite the overwhelming power of reality, we as humans have a tendency to rationalize away that reality to fit the story we've created in our minds. I've done this with quality processes, blindly attributing lack of adherence to the process to lack of understanding or discipline. There have been countless case studies, thrown at MBA students, demonstrating companies that, despite all evidence to the contrary, chugged along with the same approach and mentality in product development right up and over the cliff of oblivion. Remember DEC?

Historically, quality methods have taken the form of prescribed processes to enforce control of a system to produce the desired result. While leaving some room for adaptation, it was understood that there was a defined way to produce high quality products. Further, the process to this day is often king. Unfortunately, this is often too much of an ivory tower view where from far away, people look like rational parts of the system. However, people and organizations rarely, if ever, operate in a rational way. Our approach has to accommodate "imperfections" (aka, people being human) and not try to force a rigorous and unbending process on them. Otherwise, we risk taking the humanity out of a very human system.

I see this exemplified by many of us orthodoxly holding to the pseudo-waterfall, stage gate life cycle of the company (driving process based on it, collecting metrics supporting it) despite evidence that no one is really following it as it was intended. One of the stages didn't go right but we still pretend that we can continue to use the same process even though it is based on the assumption that we cleanly exit one stage in order to enter the next stage? Where's the emergency lever here?

Another case in point is collaboration. We desperately want teams to communicate more across our products but our organizational structure has been created based on the current life cycle of the company that, by nature, creates silos. When we take certain portions of the process for granted and consider them unchangeable, it's very hard to see another way of working outside of just hammering on ourselves to get better at the process.

At this point, blame becomes a favorite tool. "If only marketing had gathered the requirements correctly we wouldn't be in this mess." Fast forward several months of working nonstop to "do whatever it takes" to get the product out the door. We hold a retrospective to talk about the fact that if we had only followed the process we would have been fine. Subsequently, we get a little bit better at following the process but still run into many of the same issues as before.

This happens at all levels of organizations and to the best of us. I'd advise right now if there's anything you're trying to change and you have a little voice telling you, "I wish they'd just listen and actually try this," and that voice has been saying this for a few months then QUESTION THAT VOICE. Perhaps there's something going on that doesn't fit that story line and you need to quiet that blaming/justifying voice if only to allow some room for reflection. One way to get this reflection is to get support. Ask someone outside of your domain to play devil's advocate for you and just listen to the questions they raise about how you're approaching your work. Don't be afraid to use your customers as a source for good feedback.

2.1 Herding Cats

"Nothing pleases people more than to go on thinking what they have always thought, and at the same time imagine that they are thinking something new and daring: it combines the advantage of security and the delight of adventure."

– T.S. Eliot

But what else could be going on? At one point in my journey, I discovered methods to enable the quality change that I wanted to make. I found Agile (queue the inspiring music) and saw that we had to look not only at the mechanisms within the life cycle where we worked but at the life cycle itself, the people (surprise!), and the principles we followed. The system level approach was revolutionary in my mind and corrected the many problems I had found in my past approaches to driving product quality. One of the key things that I had been missing was that not understanding the whole system meant I was optimizing for only a few parts of the system. However, these parts were affected and changed based on other operators in the system, of which I had no visibility and frankly wasn't paying any attention to. There is no way to really way to get the sustained outcome we want unless we start to understand the system in which we're working.

Unfortunately, I still approached it with the same black and white stick – "follow this Scrum process or else!" "Agile is the only way!" I spent more time then I care to admit brute forcing this method on our team with a religious zealotry. I provided examples of teams and data to support why Agile was better. While this made me feel better, it didn't go a long way toward helping change the situation or people's perception. In retrospect, I was using the same dogmatic approach but substituting "quality" for "Agile." I also focused much more on something that was easy (the Scrum framework) than the harder cultural and mindset change that Agile endorses. This is not to say that we didn't see enormous improvements in quality, team health, execution, and engineering by making work transparent and building in mechanisms to reflect and take action as a team. But by favoring the process as a mechanism for change versus focus on the ultimate goal of being an Agile organization, it just got us to be really good at one thing. Meanwhile, we weren't looking around at what other ways we could approach our work in order to do even better.

Over the course of a couple years, through hard work and dedication, we were able to get good at getting things done in the sprint. However, we saw some anomalies such as parts of the code being hard to change, validation getting squeezed at the end of the sprint, and the team not becoming the dynamic, happy group of professionals that Agile promised. In many ways we had plateaued in our improvement efforts and weren't seeing a better way to work other than to try harder. Luckily, we hired an engineer with a background in Extreme Programming (XP) that lead and inspired the team to use Agile technical practices. The engineering team took a big step forward and moved to a much more Agile state in only a few months. Where I had been coaching the team on a particular methodology, our new hire brought in new practices and was able to take the team to the next level. Having someone with a new perspective come in to coach the team was invaluable.

Given how much the team evolved in such a short period of time, I realized from this experience that I was just using one tool and acting as if it would be the solution to all problems. Yes, Scrum was my hammer and everything in our development was a nail ("if only we just did Scrum correctly it would be fine!"). I was fixing my mindset in a way that didn't challenge trying new approaches to the detriment of increased improvement. I had put so much emphasis on the process, I didn't step back and focus on the reason we were doing this process and what our ultimate goals were. Looking back at that experience, I would tell myself and my team to always look at the why, look outwards, experiment, and crave continuous improvement.

If this sounds familiar, then know that you're definitely not alone. It's very easy to adopt new tools versus new behaviors and approaches. Every day now, I challenge myself to see if I'm really approaching my work differently and if I'm trying something new or if I'm carrying along my old mindset baggage. The quote at the beginning of this section helps keep me humble! However, it's best to ask those on the outside. Ask your co-workers or your clients. My team actually goes out to other corporations and speaks to people in a similar line of work as ours. This helps us get fresh ideas and challenge our current approaches. We also bring outside change agents in to take a look at what we're doing and give us an outsider's perspective.

3. Peeling the Onion or "Stories We Tell Ourselves When Things aren't Working"

Our engineering teams were doing exceptionally well but when one part of the process broke down, such as the Product Owner not fulfilling their role, we found ourselves in fairly serious circumstances. Many had a sinking suspicion that no one was really challenging what was valuable or not on the project. In fact, how did we really know the product was even worth doing other than our upper management saying that it was valuable? The reality was that we had lack of organizational support or understanding for this role and subsequently did not put people into this position that would be successful. Instead of accepting the reality that no product owner support was coming, we dug our heels in the dirt. We kept our focus on our 2-week scrums and heavily rejected new work coming into the sprint because we assumed the Product Owner was not prepared. Had we created a monster? We had fixed our mindset that the process was king and that people just needed to try harder.

Using Christopher Avery's Responsibility Process^{TM} as a guide, we were definitely in the blame and justify space. What we weren't doing was taking responsibility. The story we were fixed in was that we were the victims and, if other people did their job right, everything would be fine. We had stopped looking at why we were doing the things we were doing and acting from the present reality versus an ideal one we sought. Agile tells us that we should always be focused on creating business value. Thus, we should have stopped and asked ourselves, "if we don't really have a focused product owner, aren't we just guessing about whether we're creating value?"

Unfortunately, we went along with “the program” and didn’t challenge what was happening. I wish I could say that this story ended well but after the team completed the entire year and a half project and right before launch, the business group pulled the plug and decided not to ship it externally.

Earlier in the project, Scrum was doing its job as a tool by telling us that something was wrong and sending up warning signs. Unfortunately, instead of listening, we hunkered down and focused on the development, blaming the rest of the system for the problems instead of challenging the organization to really make a choice about whether this was a valuable product or not. To make this kind of challenge would not have been easy; challenging the status quo is never a walk in the park. But, we have to ask ourselves: isn’t questioning the validity of the product better than wasting months of our time on work that doesn’t matter?

This extends to anything we’re doing whether it’s launching a new initiative, working on a new set of quality criteria, or developing our architecture. It’s rarely easy to make dramatic changes in a large organization, especially when it comes to people, jobs, and rewards. In retrospect, I can look back and challenge the larger system: Were we rewarded more for making and meeting schedules or providing the most value for the company? Were people’s jobs dependent on certain products in a product year and therefore had little incentive to cancel the projects even if they were wrong? Did we have a culture that enabled us to give up a project and switch to focusing on innovation or to help out another team if it was more valuable? The answers wouldn’t have been pretty but they may have helped us start addressing the really gnarly problems in the system instead of blaming a single part of it. (In this case the product owner).

One of the things to ask ourselves when we keep focusing on one problem and it just isn’t getting better is, “what story are we telling ourselves?” What is the storyline that keeps running through our head? “If only someone would do this,” “I have to do this or else I won’t be rewarded,” “This is just the way it is.” That may be one story, but there’s always other stories to tell and we have a huge ability to change our storyline if we take a new perspective.

4. Stop Asking Permission: Empowering the Troops

To tell a more positive story, for many years I was part of a ragtag virtual team that supported each other in our madness to work in a way that made more sense to us. This virtual team had stopped really asking permission to make changes about five years ago. At that time, we saw that there were many silos of people who were doing crazy new things like Agile. It was important, we thought, to get that community talking to each other. So four of us decided to create our own conference. “Can we do that?” we asked ourselves. We supposed we could try. No one stopped us and 200 people showed up! Then we just kept doing it every year and more people showed up. Then the adoption of new practices and new culture at the company started growing exponentially. Fast forward to a year ago when we asked a senior vice president to show up to our conference and she agreed! Someone from the audience asked, “Why are there thousands of Agile adopters at the company but no one supports us?”

I’m not sure who that brave soul was, but thanks to him standing up and asking such a great question, we capitalized on the momentum and proposed a new team, again asking, “Can we do that?” Amazingly, our executives let us create our own team! We were nervous to push some of these assumed boundaries but there was no harm done and great things came out of it. We began paying it forward by helping teams to get past their own assumed boundaries and challenge the impediments. It turns out, a lot of what we do today is give people permission to challenge the things that are holding them back from being great teams.

If you've said, "this is just the way it is," ask yourself, "who said so?" If you've said, "we could never do that," ask yourself, "why not? What would it hurt to take one step forward to start changing it?" Go do something about it, even if it's a small step! You have my full permission to go try something for what it's worth. I still struggle today with my assumption of the existence of certain barriers even though I have yet to discover that any that were real. The thing is, you won't know that there's a boundary until you try to cross over it. Sometimes what's holding us back the most from making big impacts are our own preconceived ideas about what is allowed or not allowed in our companies.

5. Welcome to the Real World

"No plan survives contact with the enemy."

-Helmuth Karl Bernhard Graf von Moltke

Empowered to help groups to the best of our abilities, my team has started to explore new worlds. Specifically, we've begun to delve into the nature of our own world and the systems in which we work. Earlier this year we became engaged with learning about the Cynefin framework and the theories developed by Dave Snowden and his company Cognitive Edge. I couldn't write this paper and not introduce this framework as it has given me a completely new understanding of why Agile has worked well and what aspects of our environment it doesn't address. I won't go into great detail but I encourage you to start learning about the Cynefin framework and the nature of complex adaptive systems.

As an introduction, the Cynefin framework clearly shows the different system domains that make up our world. On one side we have ordered domains, Simple and Complicated, where there is cause and effect relationships that can be determined. In the Simple domain, we can use best practice, in the Complicated we start to use good practice and consult experts or do analysis to get the answers we seek. This is great, but it turns out there are unordered domains as well. These domains include Complex and Chaotic. Here, there is not a cause and effect relationship that can be determined. In the Complex space we can use new management techniques and tools to navigate but in the Chaotic space, we just have to act and see if anything breaks us out of the Chaos.

Simply, the Cynefin helps us to understand the nature of the system we're working in so that we can act appropriately. No domain is better than any other, it's just the way things are. However, if we assume we are in a simple system and act that way when we're really in a complex system, things can go direly wrong. Think of people gaming the draconian rules of their organization or blindly following the process without asking why we're doing the process in the first place.

Whoa! This explains so many things but demonstrates that we have so much more to learn. As I discover more about complexity thinking, I'm continually excited about the prospect that we can really effect the big hairy problems of our business that we are not addressing well today. Through this understanding, I've learned that it is most important for us to face reality. As the quote at the beginning of this section demonstrates, in both war and in development, we need to be fully aware of the situation and react accordingly. Beating a dogmatic drum is not going to make the cats go any faster in the direction you want them to move.

Since my understanding of complex adaptive systems is still a work in progress, I will defer it to a later conference. However, there are a few things I've learned based on Cynefin but also based on discussions with other coaches that I was able to apply immediately and have helped me to change my perspective in a number of ways.

5.1 Learn, Unlearn, Relearn

The first thing I learned was that I was being entirely too dogmatic, and it was to the detriment of the company. For example, our Agile pendulum had swung so far at times that managers looked like enemies and waterfall was the destroyer of all things. This was not only wrong but it pushed people away and there began to be a sense in the community of an “us vs. them” mentality. Now that I have an understanding of the different domains, I realize that there is a place for waterfall in the complicated/simple space. Agile and Lean work well in some areas of the complex/complicated spaces. And there are many more techniques we can apply depending on the system we’re working with.

I also learned about the power of human cognition and why enabling self-organizing in teams had created such large results in our organization. I realize now that we haven’t gone far enough in enabling self-organizing outside of single teams and that it’s not so scary to let go of direct control.

The third important thing I learnt was that we must stop assuming that there is a right answer to the complex problems we’re dealing with. This seems a little concerning at first, but it changes the way the problem is approached. We have to start experimenting more in a disciplined way and learn to accept that our experiments may not always produce the result we want. This isn’t bad thing, and is actually valuable. What’s important is that we are able to learn, try again, and see where it takes us.

6. Distributed Cognition

"The role of the teacher is to create the conditions for invention rather than provide ready-made knowledge."

-Seymour Papert

Managing in a complex environment is much different than managing in a simple or complicated space. While our goal is to continuously move from complex to complicated and vice versa, just wishing or pretending we can use ordered methods (hard rules, hierarchical command, etc.) will not magically transform our complex system into a complicated one. Circling back to the opening journey I went through, I realize that I was and have always been working with complex systems. Additionally, those systems became more complex as I began working with larger, interrelated groups and transformations. Had I understood the nature of systems and the methods used in complex systems, I would have saved myself a lot of heartache. It’s not simple, but at least I wouldn’t have been trying to fit a square peg into a round hole.

One of my favorite discoveries from the great work of Dave Snowden and his team is the concept of distributed cognition. It turns out that humans can figure out a lot of things if you just let them. Novel concept, I’m sure, but in practice we often completely forget this. Even as a huge champion of self-organization and empowerment, I still had many pre-conceived notions on what it meant to really let the power of human cognition thrive. Perhaps it was my ego or my lack of understanding. Or perhaps it was performance management systems that viewed only direct intervention as a rewardable behavior. Whatever it was, I had a really hard time letting go of the idea that I couldn’t be the expert on change and transformation. In fact, there is no one expert or set of experts in a complex environment! Those that think there is a right answer when you’re dealing with hundreds of people or changing the culture of an organization can think again!

In this kind of complex system, leaning on distributed cognition and the ability of diverse viewpoints to scan and better understand the system is your best bet. Now that I can accept this, I see over and over again that we are trying to put too many controls on complex systems leading them to a fragile and potentially catastrophic state. A great example of this (thanks to Cognitive Edge for bringing it to my attention), is the "Magic Roundabout" in Swindon, England (*Figure A. Magic Roundabout*). The civil engineers who developed the roundabout knew that they would need to enable movement of traffic despite unpredictable situations. The roundabout today flows traffic from a football stadium, sits right next to a fire station, and combines those facilities with five other major roads. Not only did the engineers need to enable flows of extreme variable traffic but also allow locals to get to where they needed to go quickly, protect those unfamiliar with the nature of such a roundabout from causing traffic jams and accidents, and overall ensure a very low incidence of traffic accidents.

Now, had we been using the old scientific management approach, we'd gather our traffic engineering specialists, look at the most advanced traffic light systems with proven algorithms for traffic management, and come up with a system that controls all movements of cars. After years of planning, we'd put in the fixed traffic control system all at once and stand back and appreciate our work as the population encounters it for the first time. Of course, then you would have caused traffic build up for miles as it wouldn't be possible for such a system to anticipate and account for the high variability in traffic patterns and your government funding would have likely run out so there would be little possibility of improvements anytime soon. Additionally, there would likely be a higher incident of accidents because people are paying more attention to the rules of the stop lights than the cars around them. In fact, the Magic Roundabout solution has the added benefit of causing people to have to pay more attention to cars around them (because they aren't relying on automated controls) and really use the power of human cognition and problem solving. One look at the final roundabout and you know it's not something you want to go into half-asleep or talking on your cell phone. Further, the engineers didn't just create the perfect solution from the beginning. Over a few rounds of experiments, they played with various temporary barriers and looked at the effects on the system. Once they had the system at acceptable thresholds, they locked in the barriers.

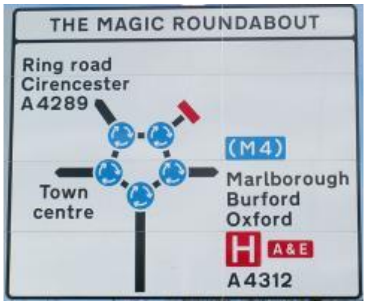


Figure A. Magic Roundabout

Applying this concept to development forced me to think, how many control mechanisms are we putting in place that actually take away from critical thinking and do not harness the power of distributed cognition? I've seen over and over teams that have a well-intentioned manager who simply allocates work to people versus putting the responsibility on the team to think through and practice developing robust designs. What happens is what one of my Agile coaching friends at John Deere refers to as "learned helplessness." The team is robbed of actually learning the skills they need to stand on their own and worse, the solution design is subject to the scrutiny of only the person that developed it (the manager). Going back to the Cynefin framework, this sounds like we've assumed that we're in the complicated space when in fact our solutions and the environment lie in the complex. One person can't just come in with the "right" answer as there are many answers and we need to subject the solution to a diverse set of viewpoints.

To avoid getting ourselves into another dogmatic trap and assuming everything is complex, there are complicated environments where it is perfectly good to apply techniques that rely on expert analysis. What is important is to understand what system you're in and act appropriately given that context.

Identifying and experimenting with boundaries in the system and ensuring minimal controls while enabling the outcomes you want is a great way to do this. Then, sense what's going on with the system. What things are happening that we want to amplify? What things do we want to dampen? Continuously inspect your results and adapt from there. This notion of decentralized control with governance through boundaries is carried out more and more in the new development methods. What's critical is that we realize that the system is changing and these changes may make great improvements in our organization. But, as the organization evolves, our boundaries need to adjust to accommodate that growth.

Again, I wish someone would have told me this years ago. I regret coaching teams to just work within Scrum, as this did nothing more than enable fixed mindsets. Now we go into teams and tell them upfront that Scrum is just a method on the path to Agility and, really it is just the training wheels. They will learn and fix a lot with Scrum but eventually will pick up many new techniques as they move towards continuous flow. Ultimately, it is their responsibility to figure out why they're doing the things they're doing and to improve. They have the power! In this way, it is everyone's job to continuously improve – not just get good at Scrum and plateau after that.

7. Now Experiment!

Alright, you say, we see complexity around us but now the approach has to be one of continuous experimentation? That's a lot to ask! One major impediment we've found to developing a mindset that works well in the complex space is the fear of failure. Often companies build this fear of taking risks into the system (intentionally or not) for many reasons but, regardless of the original motivation, it exists. Knowing that we often work in a complex environment, not experimenting is actually negligent. We can no longer pretend that everything is ordered and that, if we just plan really well, we can get to the outcome we want. This idea of identifying an ideal target and then filling in the gaps simply doesn't work in a complex system. We may have a general idea of direction, but we don't need to know what the final outcome will look like as it's unpredictable. To appropriately manage in this system we need to experiment not just at the development level but at the product portfolio level and, from a change agent perspective, at our own initiative level.

This leads me to my latest mission which is not to make everyone good at Scrum or Agile or Lean but to grow a strong "Agile" mindset at our company. I'm using the term Agile as I don't have a better one (please feel free to suggest!). Ultimately, the point is that we must create an environment where everyone in the system thinks critically about the reasons why they are doing the things they're doing and, if they see that they are working in a complex domain, they experiment appropriately. Using Linda Rising's definition of Agile Mindset, based on the work of Carol Dweck, people are continuously learning and improving and accept failure as a natural part of life.

I have teams coming to me all the time simply asking for Scrum training with no understanding of why they're doing it except that they saw others be successful by using the method. I sit down with these teams and figure out what their business reasons are for wanting to change. Often, the solution may have absolutely nothing to do with the need for Scrum but rather something completely different such as helping the leadership team to define better products for their markets. I encourage everyone to ask why! By doing this, we can better avoid going through the motions of a process (whatever it may be) without understanding the intention behind all the activities we use to get our work done.   The lack of understanding of why we're operating in a certain way is a big problem I see in many teams. We have to understand what we're trying to accomplish in order to have hope of improving the way we work. Further, we need to give people permission to fail and start shifting out of a fixed mindset.

Thanks to Linda's work, I now realize that beating that drum and criticizing lack of adherence to the process was producing exactly the opposite of what I hoped to achieve, which is an Agile mindset.   Now I encourage people to just start trying something new (practice) in order to change behavior and then start really experimenting. I also make it very clear whenever I'm coaching teams that this isn't necessarily easy. It will take work and some things may not work out, but it's ok because we're all learning this together. As change agents, we have to make a safe environment for people to fail and learn because, from my experience, a lot of companies typically punish this behavior. Our companies won't be successful if we don't begin to cultivate an Agile mindset. The very nature of complexity means that we don't know the right answer, so we will have to live with failures. Pretending that this is not the nature of our system is a pretty bad course that we should flee as quickly as possible.

An important thing to note is that Snowden proposes the use of Safe-to-Fail experiments. These are small experiments that won't tank your entire product or group if they fail. The purpose of these probes or experiments is to approach issues from different angles that will enable more visibility into emergent possibilities in your complex system. Start thinking of some things that you can try today to start addressing that big problem you keep running into! Within 15 minutes, I'm sure your group can come up with a couple. And don't shy away from dissent – get some people outside of your team to really hammer on the experiment and adjust it - but quickly!

Another technique that the Agile community has used for years is fast feedback loops. How can you continuously get information back that helps you know that your experiment is producing good or bad results in your system? We use pair programming, test driven development, unit level tests, system testing, customer testing, etc. as early and often as possible. As we get more into the complex space, we will have to use new techniques. But this is great! We have to know that we'll continuously be learning and adapting to the nature of the world in which we work.

8. Crisis of Faith

Don't get me wrong, after learning more about complex adaptive systems, I suddenly experienced panic. What does this mean – Agile is wrong? Have we gone too far in thinking Lean and Agile were the silver bullets? In the earlier stories of this paper, I had my identity wrapped up in being a Software Quality Engineer. Now I found myself wrapped up in the identity of an Agilist. This makes change incredibly hard as we start to question our very purpose. From years of struggling with this, I can offer only this advice: Relax!

I appreciate David Hussman's approach and mindset on this. As he says, "if it's working, use it." Agile isn't wrong, it's just applicable in different spaces particularly in transitioning complex systems to complicated. Waterfall isn't wrong, it may work very well in complicated/simple spaces. We may employ other techniques earlier in the life cycle and to enable resiliency in our company but that's perfectly fine too. For example, how does Agile help us to build our project portfolios and enable knowledge management across our company? It doesn't explicitly provide us the tools to do this! We have to use other mechanisms but that doesn't mean that Agile methods aren't fantastic in our product execution.

I'm saddened to hear of the religious wars going on in the Agile and Lean community recently. While debate is good, this type of attack on certain methodologies that are different than what we're used to isn't doing the community any favors. That's why we're seeing a lot more companies turn toward each other to learn how to take their organizations to the next level while many of the former thought leaders in the industry are arguing over old methods. There is no perfect method here. Some things work in some situations and fail in others. There are many different techniques that we should be applying based on the nature of different types of systems.

9. Conclusion

"The illiterate of the twenty-first century will not be those who cannot read and write, but those who cannot learn, unlearn, and relearn."

- Alvin Toffler

My purpose in writing this paper is to help save you from spending years of taking a dogmatic approach only to find that you aren't affecting the change you really want. I also want to let you know that if you realize you've been doing this, it's ok! You haven't failed life, you just learned from this experiment! Now take what you've learned and try something different; see what happens! The biggest risk we run is getting stuck in a fixed mindset. It turns out that if we're working in a complex domain, we won't have the answer. This is a very freeing concept if you think about it. One great step to take is to begin to understand the nature of systems and techniques for working in the complex domain (as this is one of the more unfamiliar domains). You can also cultivate an Agile Mindset and help those around you to use an Agile Mindset as they are approaching their day to day work. Good luck and remember to ask why, experiment, and learn!

References

Avery, Christopher. “Responsibility Process.” http://www.christopheravery.com/responsibility-process# (accessed August 11, 2013).

Cognitive Edge. http://cognitive-edge.com/

Snowden, David J. and Mary E. Boone. 2007. “A Leader’s Framework for Decision Making.” Harvard Business Review (November).

Rising, Linda. “The Agile Mindset – What’s Next?” 2013. https://submissions.agilealliance.org/system/sessions/attachments/000/000/221/original/PNSQC_Mindset_-_60_min.pdf. (presented at the Agile 2013 Conference).

“The Magic Roundabout.” http://www.cbrd.co.uk/indepth/magicroundabout/ (accessed August 11, 2013).

“Safe-to-Fail Probes.” Cognitive Edge. http://cognitive-edge.com/library/methods/safe-to-fail-probes/. (accessed August 21, 2013).

Test Automation: A Project Management Perspective

Amith Pulla

amith.pulla@intel.com

Abstract

For most QA leads or managers, it's always difficult to get the project manager (PM) to invest in test automation. The term project manager refers to the person who manages the project funding and expenses. Sometimes program manager or project sponsor owns the function of managing the funding and resources. Project management as a function exists in both traditional and agile environments, as project cost management is not in scope for most agile development practices including Scrum. It's important to learn how project managers and stakeholders view test automation and what the impediments in investing in test automation are. Test automation costs are mainly associated with resources, training and tools. For example, a senior program manager once said, "Test automation is great thing to have, but not a great thing to invest in". This paper looks into how project managers view test automation and try to measure its value or ROI (Return on Investment). This paper also offers guidance to QA and Testing teams on how to show the value of test automation in the terms that project managers can easily understand.

Project managers view the project from the classic triple constraint model that mainly includes Scope, Cost (Resources + dollars), Schedule, and Quality management. Project managers don't see a direct correlation between these four project variables with investing in test automation and are usually reluctant to invest in test automation. Investing in test automation has a measureable value on at least 3 out of 4 project attributes and individual metrics can be created for each attribute to show that overall ROI can be in positive territory over time. With the right metrics PMs can be more confident in investing in test automation.

To make a case for investing in the test automation, the QA lead needs to translate investments and ongoing costs for test automation into positive impact on project attributes: Cost, Schedule, Quality and Risk. Once the decision is made to invest in test automation, the QA lead needs to provide an individual metric for each of the four projects' attributes that shows tangible and measurable value to project managers in the terms they most care about.

This paper tries to offer methods, examples and metrics to QA and test leads to show the value of test automation in both traditional and agile environments, with a focus to get the project and program managers to invest and trust in automation efforts.

Biography

Amith Pulla is a QA Manager at Intel Corp, currently working at the Intel site in Hillsboro, Oregon. Over the last decade, Amith has been involved in software testing strategies and processes for applications mainly focused on sales and marketing. Amith has worked extensively on projects involving multiple platforms and complex architectures. Amith also worked on developing test methodologies and techniques to meet the business needs and timelines. As part of his QA lead role, Amith focused on improving and refining QA processes and standards for efficient software testing in agile environment. Amith also worked as a project manager and ScrumMaster in short assignments.

Amith has an M.S. in CIS from the New Jersey Institute of Technology; Amith got his CSTE and PMP certifications in 2006. Amith got his CSM certification in 2012.

1. Introduction

Over the years, with the emergence of test automation tools and growing complexity of software projects, many experts agree that test automation does bring value in terms of better quality and faster delivery, but only small percentage of software development projects actually use test automation today. Even in agile projects, not all projects use test automation as much as the team wants to. The need for test automation and the level of test coverage varies for each application and product, but most QA leads agree that using test automation can cut testing time significantly leading to faster delivery and higher confidence in quality.

Most QA leads and test managers agree that they should use some level of test automation in their project, but lack of funding and priority are some of the biggest obstacles. Implementing test automation can be expensive, there are costs related to additional resources with automation skills, automation tools, test machines, servers and training. In addition, there are ongoing costs of maintaining the automation framework and scripts as long as the application and product is used.

The QA Leads need to work with project managers and stakeholders to make a case for investing in test automation efforts. They need to show the value of test automation to project in the terms that are easily understandable. In the book Experiences of Automation, the authors say "Make your benefits visible to managers. You need to continually sell the benefits of automation." (Dorothy Graham & Mark Fewster, 2012) In this paper we'll discuss how test automation can bring value to the project and some of the ways QA or Test leads can demonstrate the value using metrics and examples.

2. Project Management Constraints

Most project managers, manage the project using the triple constraint model. Triple constraint is the balance of the project's scope, schedule (time) and cost. The traditional triple constraint model now evolved into five variables of project management. Let's look at how test automation impacts each of the project's variables.

2.1 Scope: No impact, we all agree that regardless of amount of testing or test automation of test cases has no impact to product scope or business value delivered. The product scope, features or capabilities are defined by the business or product owners, test automation may impact quality but not scope. Test automation is a task(s) on the project, but not a story or feature that will be delivered to users. Test automation is tool used by testing teams to improve quality and to complete testing faster; it's not visible to end users.

2.2 Cost: Investing in test automation can have an initial spike in project costs. Costs go up with investments in automation resources, training, automation tools and setting up a suitable framework to maintain automation. The initial investment and ongoing resource costs need to be offset over a period of time. The QA lead needs to come up with a cost per each manual regression cycle and automated regression cycle (factoring in cost/investments in test automation), depending on number of regression cycles the project needs, the ROI should move to positive territory at some point. We can calculate the inflection point so that the PM can see the ROI depending the duration of overall project including support phase.

2.3 Schedule: We can clearly show positive impact to this attribute. Regression test cycles will shrink from weeks to days or hours enabling the teams to go to production faster. Again, the QA lead needs to calculate estimated time for manual vs. automated regression cycles per fixed test coverage. The project's release schedule should change as the test automation is built out, clearly showing the value to PM.

2.4 Quality: Testing or test automation won't by itself change the product/application and improve the quality, test automation is a tool to execute functional or system test faster, test-fix-test cycles can be run faster, allowing the project team to find and fix defects faster, leading to improved quality of the application or product given the same schedule. Defects found per each regression cycle can also be a good indicator of the efficiency of test automation. The test automation can enable teams to have multiple regression cycles given the same project schedule, this allows the team to improve overall quality and increase the users' confidence. Test automation can be used with Continuous Integration (CI) framework to run build verification tests after each code check-in.

2.5 Risk: Test automation increases the speed and efficiency of testing, with good test automation coverage, teams can execute more test cases compared to manual testing in the same time. More test coverage means better understanding of application behaviors and defects. By increased test coverage, teams can manage risk effectively.

3. Test Automation and Quality

3.1 Regression Testing

Regression testing, the type of testing that is performed to make sure that previously working functionality is not altered or impacted, is usually done by re-executing existing functional test cases or test suites. The goal of regression testing is to uncover software defects or regressions in existing functional areas of the application or product after new changes are introduced. The new code changes can be related to adding new functionality or features, enhancements, patches or configuration changes.

A typical regression testing cycle will have all or a subset of existing test cases that need to be run depending on the application or platform, the existing test case numbers may range from hundreds to thousands and running them manually can take days or weeks to complete. Sometimes due to the scheduled constraints, teams may take a risk-based approach to regression testing, executing only a portion of test cases, but if the application is large with hundreds of functional, integration and system test cases, running a subset of test cases can take days or weeks.

3.2 Test Automation

Regression testing, running the same test cases or test suites manually in multiple cycles can be laborious and time consuming, depending the on desired quality, this may significantly impact the project schedule. Automating these test cases brings down the time taken to run the regression testing cycles. Manual testing that could have taken weeks or days can be now run in hours.

Once the tests cases or test suites have been automated by the right automation tool and framework, they can be run on-demand and repeatedly faster than manual testing. Running regression testing using test automation is a cost effective method especially if the test cases don't change often and frequent regression cycles are necessary. Frequent maintenance activities like upgrades or patches over the lifetime of the application can cause working features to regress; this means a regression cycles need to be run for any change that goes into the environment.

In the article Cost Benefits Analysis of Test Automation, Douglas Hoffman says that "Test automation is not always necessary, appropriate, or cost effective. In cases where we are making decisions based upon an expected return on investment, analysis can direct us to where test automation can benefit us" (Douglas Hoffman, 1999).

3.3 Test-Fix-Test Cycles

For any software application, quality is achieved not just by better testing or better test coverage, but by going through series of Test-Fix-Test cycles until defect counts are reduced to below thresholds. Test-Fix-Test Cycle is basically the testing team running a set of functional tests on the application and logging defects, then the development team fixes the defects and deploys the code changes to the test environment, the testing team re-runs the same test cases and verifies the fixes, but may find other defects. Testing in itself doesn't change or improve the quality of the product or application, the goal of testing is to find defects or deviation from expected behavior so that the development teams can fix the defects or issues reported. Depending on the number of defects found per regression cycle, the project team needs to run several regression cycles to ensure the defect numbers are below acceptable levels.

4. Metrics and Measures

Metrics help the project managers, project team members and stakeholders visualize the value of test automation. ROI on test automation can be calculated in many ways, but we will discuss few examples like test coverage, time gain and defects found. Good test automation metrics can quickly be related to key project attributes like cost, quality, risk and schedule. Below we will outline good test automation metrics with some examples that project teams can use.

4.1 Test Automation Coverage

Every application or product will have test cases in the regression test suite that need to be run every time there is a change. As teams are actively developing new functionality and features, the regression test suite grows.

Test automation coverage metrics tells the team how many test cases are automated vs. total number of test cases that can be automatable. Some teams may call this metric as Max Automation Coverage. Better test coverage means finding defects faster, allowing the team to fix them sooner leading to improved quality and reduced risk.

Other metric that is popular is Automation % = # of Automated Test Cases / Total Test Cases [manual + automated]. This metric tells the team how many of the test cases are automated in the total test suite or test bed.

4.1.1. Example Graph

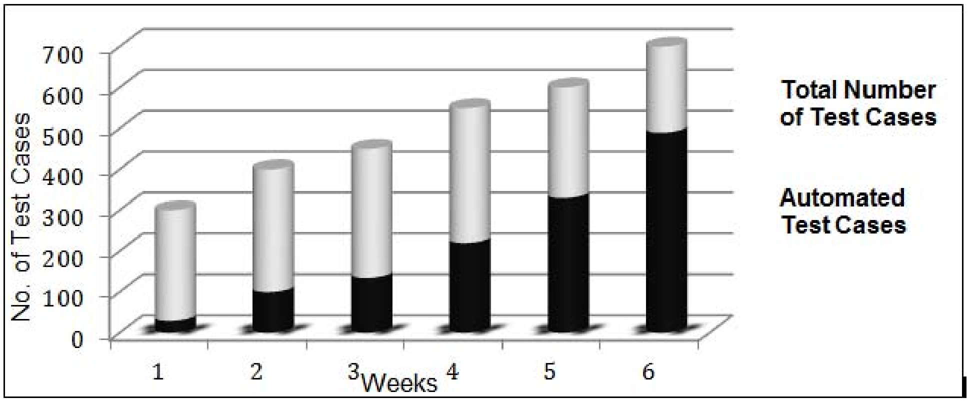
Test automation coverage is typically expressed in percentage (%); it is the ratio of number of test cases automated to total number of automatable test cases.

$$TAC = \frac{\text{No. of Automated Test Cases}}{\text{No. of Test Cases that can be automated}} \; x\ 100$$

For instance, if the project team has 1200 test cases that can be automatable and only 500 test cases are automated and ready to run on demand:

$$TAC = \frac{500}{1{,}200} \; x\ 100$$

$$TAC = 41.6\%$$



4.2 Time Gain per Regressing Cycle

Regression cycles need to be run frequently and probably daily if team has Continuous Integration. Running regressing cycles by executing test cases manually can take days or weeks depending on the number of test cases in the regression suite. Test automation can significantly cut that time. With automation, regression test cycles can be run faster, allowing the teams to run them multiple times in a shorter time reducing the overall project schedule and cost.

The time gain and effort reduced for each regression cycle can be converted into cost saving considering that the project needs fewer testers as number of test cases to execute manually declines, but the team needs to consider the initial cost of automation (effort and tools required to automate the test cases). The initial cost of automating a set of test cases is always high, but with each regression cycle there is a certain cost saving, after a certain number of regression cycles the cost saving for each regression cycle will add up to the initial cost of automation. After this point, the ROI on test automation efforts moves into positive territory. The project team needs to find that inflection point based on the number of regression cycles, sometimes the initial cost of automation may be offset after 10 regression cycles or it may take 50 regression cycles to get to that point.

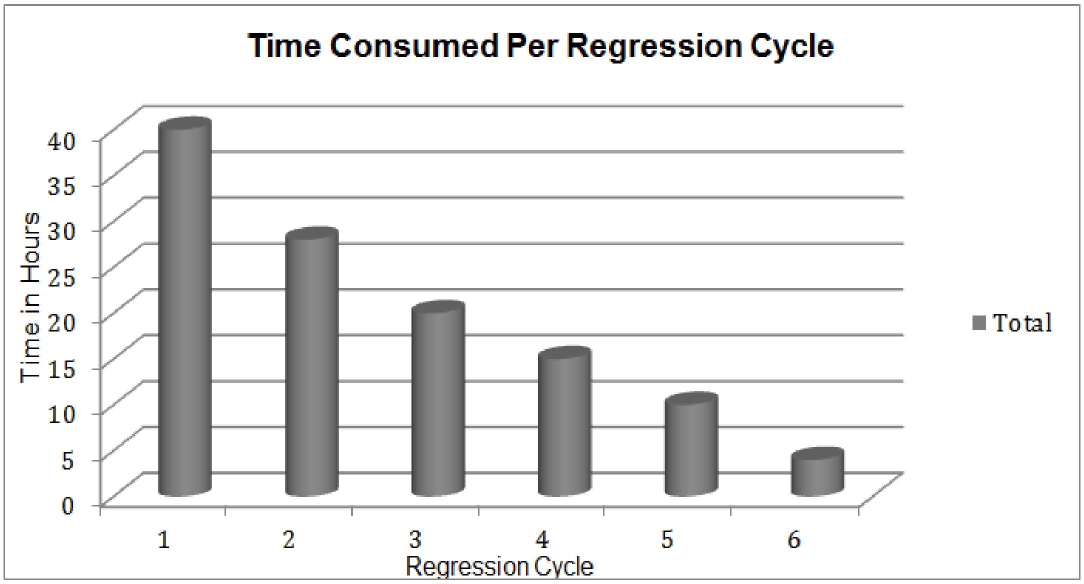
The project team needs to look at the overall life of the application or product; a typical life of a software application can fall between 2 to 6 years. Depending on the number of regression cycles needed throughout the life of the application or product, the team can make a decision on if investing in test automation will have positive or negative ROI.

4.2.1. Example and Graph

Time gain per regression cycle is comparing the time it takes to run the regression cycle manually vs. running it using automation tool. For instance, if it takes the testing team 40 hours (5 working days) to execute 200 test cases manually and same 200 test cases can be executed by the automation tool in 4 hours, the time gain is 8 (hours) x 5 (days) – 4 (hours) = 36 hours

This 36 hours savings can be converted into cost savings my multiplying it by number of testers on the team and hourly rate. But again we need to factor in the initial cost of test automation.

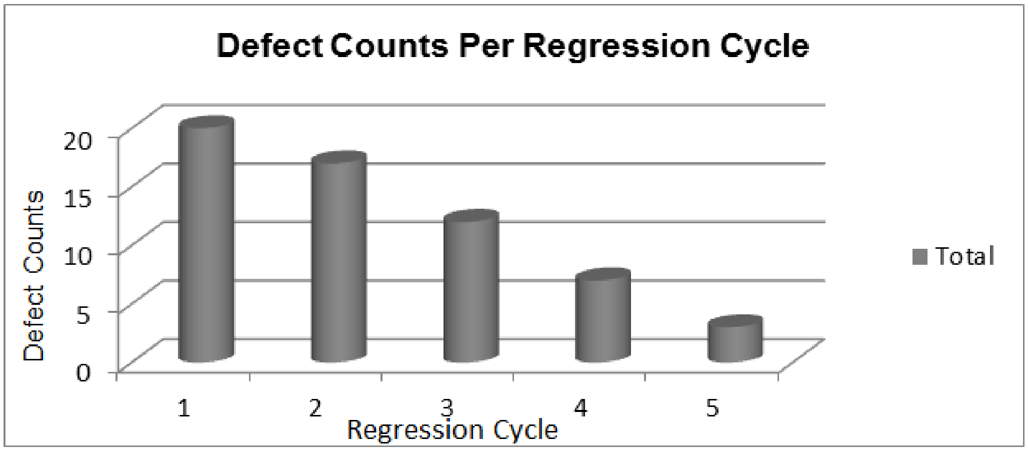
For example, if it takes $50,000 to automate 100 test cases and there is a cost saving of $3,000 for each regression cycle with the use of automated test cases, it will take 17 regression cycles to offset the initial cost, after the 17th regression cycle the ROI on test automation is positive.



4.3 Defects Found per Regression Cycle

With every regression cycle, the number of defects found can tell the team about the quality of the application and effectiveness of the test automation. More defects found indicate poor quality and less means better quality. However, if the test automation is finding more defects, it says your test automation is working and effective. Defects found per regression cycle are a good indicator of quality and as they trend downwards over a series of regression cycles, it builds the confidence of the stakeholders.

4.3.1. Example Graph



4.4 Cost Savings (ROI) and Challenges

Cost-benefit analysis can be applied to test automation; implementing test automation for a project can add additional costs, mainly for resources with automation expertise, training, tools and hardware. Depending on the number of regression cycles needed or projected for the life time of the application, the ROI on test automation can move into positive territory.

As discussed in section 4.2, ROI on test automation can be simply calculated by the below formula, but not many quality experts agree that this is the right approach.

ROI = (cost of manual testing– cost of automation) / cost of automation

Teams need to consider the ongoing costs for automation, mainly related to maintenance. Every application goes through changes throughout its life, all these changes impact test automation scripts. The automation scripts need to be tweaked or sometimes re-scripted depending on the change to the application. Defect tracking costs may also vary depending on the manual effort required as number of defects found can fluctuate for each regression cycle.

According to Michael Kelly, the real return on investment for automation is based on different types of automation done by the team and the value it adds to the overall testing effort (Michael Kelly, 2004). Automation does offer tangible and intangible benefits. Tangible benefits may be related to hours saved and time gained, intangible benefits can include faster feedback from users, finding defects sooner in the development cycle.

5. Test Automation with Continuous Integration (CI)

Continuous integration (CI) is a software development practice of merging all developer workspaces into a common codebase once or several times a day. Continuous integration originates from extreme programming (XP). Though CI was originally intended to be used with automated unit tests written through the test-driven development, later running automated functional test cases has become a standard.

Most agile teams today have some level of continuous integration (CI) built in. Continuous integration is not just actions of source control integration and automated builds, but as a complete process including integrating code to baseline, automated compilation and build, running unit tests and automated regression tests (if available) and most importantly if the tests fail, all the activities that follow to make the build clean and stable again.

6. Automation Tools

Test automation is basically using a software product or tool that is outside of the application under test to execute tests and report failures. Test automation tools can automate repetitive tasks or steps that are originally documented by a test analyst. The automation tools can offer additional benefits by expanding the assortment of testing that is difficult or time-consuming to execute manually.

Choosing the right test automation tool is a key decision that project team needs to make, the decision can be influenced by cost of the product, the technology used in developing the application under test, skillset needed to create and maintain automation, scalability, body of knowledge, compatibility etc.

6.1 Open Source vs. Vendor Tools

Open Source tools are freely available and can provide lot of flexibility, but they need some upfront work to create a framework. If the team has programming and development skills, open source tools may be ideal. On the other hand, vendor tools usually come with a framework, are user friendly and designed to hit the ground running. Vendor tools or vendors also provide support and training if needed.

There are many open source UI based functional test automation tools available in the market today like Selenium, Watir. On the other hand QTP, Ranorex, TestComplete, Test Studio (Telerik), eggplant, Certify are popular vendor based tools that used by many teams.

6.2 ATDD Frameworks

ATDD (Acceptance Test Driven Development) is an agile engineering practice that is built on test automation framework. ATDD is a complete paradigm shift from other agile software development practices where developers build the application code based on user stories and acceptance tests; an ATDD framework like Cucumber, FitNesse is used to integrate automated tests to application under test. ATDD frameworks have a Wiki mechanism to document test cases in plain language and the tests are typically defined by users or product owners. The tests are run by the product owners on a day-to-day basis and it creates a constant feedback loop between development team and business.

ATDD integrates developers, testers, product owners and users into development effort, a kind of forced collaboration as the product is being developed. ATDD brings the concept to quality is everyone’s responsibility into practice.

6.3 Cost Benefit Analysis on Tools:

Cost benefit analysis is the important aspect of test automation tool selection. There are many factors that influence the decision; here are few things that teams need to consider.

1. Initial investment, licensing and ongoing costs for the automation tool
2. Flexibility and scalability of the automation tool
3. Initial framework setup necessary for the automation tool
4. Skillset and training required to use the tool
5. Compatibility with application under test

7. Test Automation in Agile

Test automation is becoming a foundational practice in agile development and it should be the way agile teams need to build software. Agile teams see the value of test automation in terms of Velocity and Quality. The value of Test Automation is clearly visible within the CI model; it's obvious that teams can deliver better quality and business value faster. Test automation in itself cannot deliver much value if not used daily within the CI model. Test automation has limited value if it's run every few weeks and just before the release. The true value of test automation is only realized when team can run it every day as part of CI, it allows the teams to find defects faster when there is enough time fix them.

7.1 Test Automation as an Engineering Practice

For most agile teams, test automation with CI is a standard engineering practice, a job zero (must do) to keep up with the velocity, release cadence and quality expectations. Test automation scripts need to be run daily in an integrated environment to find and fix defects more frequently. If this doesn't happen, teams are not using automation to its full potential to deliver best ROI. The true ROI of test automation is realized when Scrum teams can use automation every day in the development sprints, find and fix defects faster during the sprints.

In agile, test automation certainly increases quality, improves time-to-market, but not necessarily decrease costs because of the iterative nature of changes to applications. Many projects use test automation to increase velocity, quality though there is significant increase to the cost. Test automation enables teams to release more frequently as testing can be completed faster, this is more important for some projects than cost savings. If it's an agile project with a working CI model, test automation should be fully integrated with the software development process,

8. Continuous Deployment with Test Automation

In the past couple of years, Continuous Deployment has revolutionized software delivery. Continuous Deployment takes the CI and test automation to new level of agility and delivery. With Continuous Deployment agile teams can transform its software delivery and can deliver features and fixes as they become ready without having to wait for set release date.

8.1 Automated Deployments and Infrastructure Automation

Infrastructure automation or automated deployments have been emerging in the past couple of years and are seen by some as standard engineering practices for agile teams. Infrastructure automation is a key aspect of DevOps, where development and operations team work together. Test automation and Continuous Integration (CI) have been around for some time now but if the agile teams cannot push working features, patches to production (to users) quickly and efficiently without pulling in all the development and operations teams, then they may not be as agile as they think. There are many tools, some open-source that teams can use to reduce the time consumed in doing a production release by automating deployment tasks. Automated deployment tools can deploy code to production in minutes, saving time for development teams and reducing production downtimes.

8.2 Industry Examples

LinkedIn uses Continuous Deployment to transform its software delivery, they significantly improved the frequency and speed of their production deployments, and with this they are able to deliver features and fixes as they become ready. Other companies seeing value of Continuous Deployment are Facebook and Etsy.

LinkedIn, the largest online professional networking site, adopted Continuous Deployment as the software development methodology. Under this model, a developer writes code for a new feature and checks it into the main line of software that is shared between all developers, a line known as "trunk" within the software version control system. The newly-added code or feature is subjected to large number of automated tests that will identify any defects. Once the tests are completed and no issues are found it is merged into trunk and classified as ready for deployment, this will allow product managers to select and push new features into production as needed.

Etsy, an online marketplace for handmade goods, vintage items and craft supplies, has embraced Continuous Deployment practices in last couple of years. Etsy's development team develops small change sets and deploys them frequently (Ross Snyder, 2013). This process allowed Etsy to keep up with user demands and stay profitable. Etsy uses engineer driven QA, test automation and solid unit testing meet the quality standards, all these practices were integral to their development process.

Facebook, the world's largest and most successful social media site has adopted Continuous Deployment to safely update the site with hundreds of changes including bug fixes, new features, and product improvements (Facebook Tech Talk, 2011). Facebook has a test centric engineering culture and uses a test automation framework build with Selenium and Watir. The automated tests run with every change and revision, this helps Facebook engineers to find failures and fix them faster.

9. Conclusion

Project managers or stakeholders sometimes may not see the value of test automation and benefits it offers. This could be an impediment to investing in and prioritizing test automation efforts. The project team or the test lead can influence the decision to invest in automation by showing the value in terms of positive impact to key project management attributes.

If project team or the test lead can show the value of test automation in terms of positive effect on project's schedule, cost and quality, the aspects that project managers and stakeholders most care about, it will be much easier to make a case for test automation.

References

Douglas Hoffman. 1999. Cost Benefits Analysis of Test Automation

Dorothy Graham and Mark Fewster. 2012. Experiences of Automation. Pearson.

Michael Kelly. 2004. The ROI of Test Automation

Rayn Tate. 2013. The Software Revolution Behind LinkedIn's Gushing Profits
http://www.wired.com/business/2013/04/linkedin-software-revolution/

Ross Snyder. 2013. Continuous Deployment at Etsy: A Tale of Two Approaches
http://www.slideshare.net/beamrider9/continuous-deployment-at-etsy-a-tale-of-two-approaches

Chuck Rossi. 2011. Pushing Millions of Lines of Code. Facebook Tech Talk.
https://www.facebook.com/video/video.php?v=10100259101684977

Nine Low-Tech Tips for Project Management

Moss Drake

drakem@dmcdental.com

Abstract

Project management can be tough, especially when technology gets in the way of the goal: to develop software that provides value to the users. Creating a complex software system becomes even more difficult when your processes rely on other complex systems.

Experience shows that project management is made easier by reducing complexity whenever possible. Considering this, perhaps it is better to choose the simpler, low-tech solution when one is available. Additionally, some low-tech practices can lead to higher visibility, more democratic processes, and simpler solutions.

This paper is an experience report on nine simple, low-tech solutions used for project management at our company that helped improve our process, reduced complexity and kept the project focused on the goals rather than the technology. These tips include using story cards, prioritizing the backlog, sizing stories, using information radiators for project status, collaborating using drawing and simple screen mockups, and increasing face-to-face interaction.

Biography

Moss Drake is a software developer and project leader at Dentist Management Corporation in Portland, Oregon. He has been developing software for over twenty-five years, focusing on solutions for the health care and insurance industries.

Moss has a B.S. in Computer Science and a B.A. in English Literature from Oregon State University.

1. Introduction

"Somewhere today, a project is failing." (DeMarco 1987) These are the first words in chapter one of Tom DeMarco's book *Peopleware*. His thesis is that the major hurdles encountered in software development projects are not so much technological as they are sociological. For years, people have been developing and redeveloping the same or similar projects, and yet one somewhere today is in trouble. The problem is that high-tech people, enamored with the technology, look to code themselves out of a problem rather than interact with the team. DeMarco points out that because we go about software development in teams and projects and other tightly knit working groups, we are mostly in the human communication business. In 1987, this may have been a cry in the wilderness, but by the time of the Agile Manifesto (Beck and al. 2001) in 2001, it was commonly accepted. As the manifesto proclaims, the focus should be on "individuals and interactions over processes and tools." (Beck and al. 2001)

Peopleware proposes changing physical workspaces and mental attitudes to promote communication, but communication can also be improved by simpler project management processes. Project management can be tough, especially when technology gets in the way of the goal of developing software that provides value to the users. Creating a complex software system becomes even more difficult when processes rely on other complex systems. Project management is easier if you can reduce complexity whenever possible. Considering this, it is better to choose a simpler, low-tech solution when one is available. Additionally, some low-tech practices can lead to higher visibility, more democratic processes, and simpler solutions.

2. Background

Dentists Management Corporation (DMC) is a subsidiary company of Moda Health. DMC develops clinical and business management systems for dental offices. Our development team is small, about six or seven people. Projects usually last six months to a year, although multiple endeavors are in play at any time. In addition to customer-facing projects, DMC must also adapt to regulatory changes such as HIPAA and the Affordable Care Act.

The path that led to our current project management process was not a quick one. Having explored multiple project management methods over the years, including modified waterfall methods and a couple failed attempts at Agile, we now use Scrum for most of our non-regulatory projects. We found that some of the techniques learned during our initial Scrum training continue to shine through as excellent practices, especially because they are so simple. Other aspects, however, were more opaque at the time. It took repetition, over multiple sprints and projects, before we realized the full value of the activity.

3. Low-Tech Tips

In his *Ignite* talk, Adam Light asserts, "Scrum moves organization from the individual level to the team level." (Light 2013) For managing projects, this means:

- Helping people take their ideas into a shared work space in a structured and efficient way.
- Getting the right people working on the right things at the right time.
- Building on the shared ideas to create better solutions

In short, it means collaboration: moving the project management process from the project leader to the team. The Scrum Master becomes a team member who has the role of facilitating the progress of the project rather than the project leader.

The Scrum Master provides the tools and environment to make the project easier, and allows the team to self-manage. Self-managing, however, is not a goal but a habit, and like any habit it takes time to become innate. The nine tips discussed in the following sections provide a set of good habits that lead to self-managing teams. They all have the common goal of using low-tech methods to make ideas visible and shared, enhancing collaboration.

3.1. Take a Field Trip

At the beginning of every project, we develop a vision document. The goal of this document is to provide a vision of how the product will work once the project is complete. The contents of this document are words and diagrams, pages and pages of writing in an attempt to bring imagination to life. The vision document is supposed to be the inspiration for the project, but often it is a flawed vision only approximating the actual results.

There are two good reasons why a vision document cannot possibly hope to communicate everything. The first is because of the fuzzy front end (New product development n.d.). A vision document is meant to inspire, but the project could be killed at any point so only a limited amount of work is spent defining the edges of the system. Second, while the vision document may talk about personas, it often does not describe actual system users in the way that a novel might bring a character like Harry Potter to life.

One way to fully realize the environment and characters that will populate your software solution is to take a field trip to sites where the software will be used. Visiting a site brings concrete images to the abstract ideas summarized in the vision document. While it may not be feasible or affordable to bring the entire team on site, even video and phone calls give an enhanced understanding of the environment. The goal is provide a deep impression that will help with development later in the project.

The most obvious observations to make are the workflows. The business analyst is best at this process. For the technical staff, the environment may play a factor in developing a solution. For example, dental operatories must be hygienic, so keyboards and mice are wrapped in disposable plastic covers. Since this makes touch screens unfeasible, we had to investigate voice and motion recognition solutions. In other offices, we discovered that staff had such cramped working spaces that they did not have desk space for their paper notes, so they ended up holding them in one hand and typing with the other, which raised a usability issue.

The field trip is a good time to put faces to personas. Previously you may have thought of the receptionist as “Receptionist 1,” but now you know her as Betty, who is extremely good at multi-tasking until it comes to her software. We would not have imagined the reception desk was such a hub of activity until we saw Betty interrupted nearly fifteen times in a fifteen-minute interval. Observing this not only impressed on the programmers and testers the importance of avoiding any software failures, but also provided data for system response times. Betty would be extremely unhappy to take time to call for support if something went wrong with the software.

There are other ways to observe the system: shadowing through screen-sharing sessions, conference calls, setting up role-playing scenarios at work, but none of these bring the situation to life as well as a field trip.

3.2. Make the Story Cards Work Overtime

Numerous articles on Scrum discuss story cards. They usually explain how the cards contain a user story, an informal statement about the value of a feature in the system to someone. A common form of the statement is:

> **As a <role>, I want <goal/desire> so that <benefit>.** (Wikipedia entry User story n.d.)

The articles explain that the cards are merely an artifact to begin the conversation about the full requirements of the feature. Some teams transcribe the stories into spreadsheets, documents or project management systems.

At this point, some people may toss out the cards, but wait! Even if the story is documented electronically, the physical card has value for the project. Since the principal goal of story cards in Agile development is to raise visibility and focus for each feature, keeping story cards posted in a public place is a good way to do this. Think of the card as an icon for the full feature.

The card can also be a shortcut for the additional details associated with each story. For example, on each card we write a unique tracking ID, the story size, and the initial priority. If the story has a particular champion, the PO writes that person's name on the back of the card.

As the project progresses, each story card will become a familiar artifact, bringing joy or dread, memories and ideas. The story card becomes memorable token that stands for the full feature. The position of the card can also carry meaning, like a flag. Turn the card sideways to remove it temporarily from the action. Turn it upside down to indicate that the feature is in distress. If conversations about the story become contentious, use the story card as a talking stick, allowing each person to speak their point of view as long as they hold the card.

Even when a story is tracked electronically it is useful to revert to low-tech by printing the summaries on labels and putting them on sticky notes. For multiple, parallel projects use different colored cards for each one.

A word of warning about using story cards: watch out for the nighttime janitor or strong air conditioning. If there is a mysterious gap in the backlog, you might want to look around on the floor. This emphasizes the need to keep an electronic copy of the stories in a spreadsheet or requirement management system.

3.3. Use an Insertion Sort for Prioritization

To be successful, Scrum requires a prioritized backlog of stories. Teams focus their efforts during the sprint on the items with the highest priority. Often, explanations of Scrum imply that the Product Owner (PO) already has an optimally prioritized the backlog (Cohn, Learn About the Scrum Product Backlog n.d.). That sounds like an easy request, until you realize the process required to achieve prioritization. Supposedly, the PO sets the priority, but that assumes a fully informed PO, which may be asking a lot of one person. Each stakeholder involved with the project has his or her own set of priorities, and backroom lobbying for features can cause conflicts. It is better to have a transparent prioritization process.

There are many ways to prioritize the backlog, but some tend to exclude participation more than others do. For example, when using a spreadsheet, database or document to organize the priority, only one person at a time can make modifications, so it is not an inclusive process. In planning meetings, no matter how large the display, the entire backlog cannot be seen at once, so attendees end up seasick watching the screen bounce around while one person sets the priorities. In other cases, when someone has assembled a printed document with the complete backlog, the prioritization process becomes a bubble sort on paper, stepping through the list, comparing items and swapping them if they are in the wrong order; not the most efficient method. In all of these cases, providing a pre-processed list allows some priorities to slip by without any discussion.

A simpler solution is a physical process that involves an insertion sort. In this method, the PO convenes a prioritization meeting and anyone who has a stake in the project can attend. The meeting space must have enough space to display all the story cards on a wall or window. The process steps are:

1. The PO places a random story on the wall, and then pulls another from the backlog.
2. The PO asks whether the current story has higher or lower priority than the one on the wall.
3. People in the room may offer their advice, but the PO has the final authority for placing the story card.
4. If the current story has a higher priority than the one on the wall then the PO places it to the left (meaning higher priority), otherwise it is placed to the right (lower priority). All stories must have a unique priority, so they should all be at the same level horizontally (see Figure 1).
5. Leave space between the cards so it is easier to organize them if a subsequent story is prioritized between two others.
6. One by one, the PO continues with the remaining stories, reading them aloud, getting advice, and then placing them into the correct priority.
7. When all stories have been prioritized, write the initial priority on the front of each story card for tracking purposes because priorities change over time. It is also useful to track the priority in a spreadsheet or requirement management system.

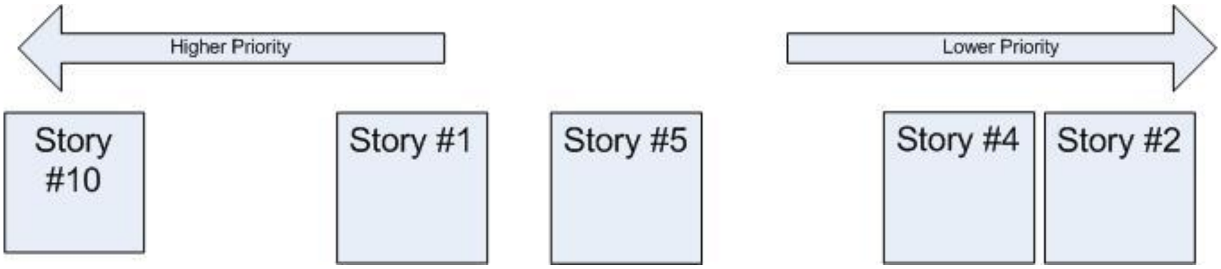


Figure 1 Uniquely Prioritized Stories

Using this process, our team has been able to prioritize up to 50 stories in a two-hour meeting. It is not recommended to try for more than this since people get decision fatigue and attention spans decline. Depending on the size of the team and of the stories, it may not be necessary to prioritize beyond the top 50 items. Since the process is an insertion sort, it is easy to split the process into multiple sessions.

The benefits of this approach are:

- Each story has its own moment under the spotlight—there is no a chance for it to be lost.
- It is a more democratic process, where the stakeholders can lobby the PO on behalf of their preferred stories and the PO becomes more fully informed.
- The unique prioritization avoids the situation where multiple stories end up with the same priority. It forces people to acknowledge the constraints of time and resources early in the project. It also avoids vague priorities where some stories are "high," some are "medium" and some might be "medium-high."
- The space allows everyone to participate, and the spatial orientation helps people better understand the scope of the project.

There are some potential variations to this process. When two project teams are working in tandem, it is acceptable to create two physical lines. This may expose dependencies. The prioritization process can also help define the minimal viable release points. The PO can move along the horizontal axis to identify the earliest acceptable release point. We maintained the prioritization as we moved the stories to the information radiator (see section 3.5). When a story is added to the backlog, it will have the lowest priority until the next sprint planning meeting when there will be an opportunity to re-prioritize.

3.4. Keep the Sizes Relative

In Scrum, the product backlog shows the amount of estimated effort required to deliver each story. While there are other methods of estimating, such as Mike Cohn's Planning Poker (Cohn, Planning Poker: Agile Estimating n.d.), T-Shirt sizing is a simpler way to get a baseline idea of effort. The human brain is better at estimating relative sizes than absolute sizes. Relative sizing is simpler since less information is required. Team members need only to recognize which stories are larger than others. The goal is to give the stories T-Shirt sizes (XL, L, M, S, and XS) relative to each other. Then, after several iterations of tracking the stories completed in a sprint, the team will have a better idea of the schedule for the remaining stories.

A problem, however, is introduced when some team members try to correlate relative sizes with absolute amounts, such as "S = one day", while others may use a different scale. Workdays are a subjective unit of measure and depending on other commitments, ideal days will vary. Using a simple mechanical method avoids this problem.

At DMC, only the Scrum team participates in the sizing process. Team members arrive at a sizing session with an idea of the stories and with any information that might inform their estimates.

The sizing process is similar to the one described above for prioritizing stories, except the Scrum Master leads the session. After an arbitrary story card is placed on the wall, the Scrum Master pulls the next card, reads it aloud, and asks if people think this story is larger, smaller, or about the same size as the one the wall. The vertical axis represents the size of the stories with higher on the wall indicating more effort; lower on the wall is less effort. People in the meeting say "higher" or "lower" and the Scrum Master follows their suggestions. If the team cannot agree on the size, then it is open for a five-minute discussion, and the Scrum Master decides the final position.

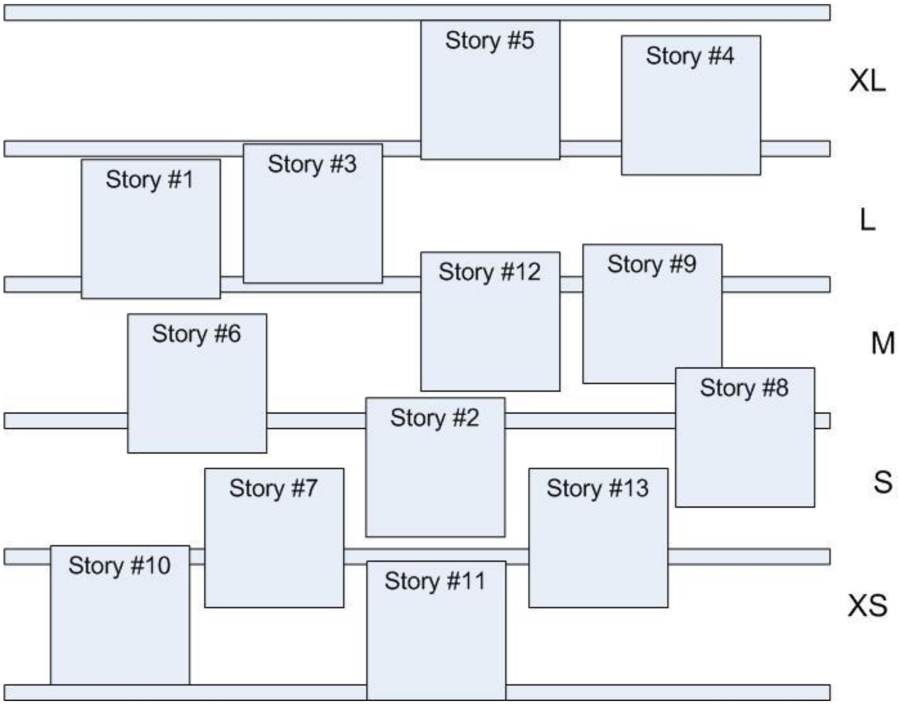


Figure 2 Relatively Sized Stories

After all the stories have been placed on the wall relative to each other, divide the wall into five equal vertical sections. Mark the sections XL, L, M, S and XS, respectively. Drawing the five sections is a physical method for categorizing the stories by size (See Figure 2).

Sizing the stories this way creates a visual map of the project in terms of effort. Reviewing it may highlight red flags hidden in the project. The Scrum Master should watch for L and XL stories that have lower priorities; in the last third of the project, for example. Large and XL stories toward the end of a project represent riskier work. Software projects have enough uncertainty that teams will want to avoid planning large stories at the end of a project. The PO may want to think about raising their priority, dropping them from the project, or trying to break them into smaller features.

Since the stories sizes are relative, the majority of the stories should fall into the small, medium and large categories. If there is a larger ratio of XL stories then the team should re-examine the stories that fall between L and XL. Perhaps the team has overlooked some details of the stories, or perhaps they are epics that should be split into multiple, smaller stories.

Note: relative sizing only works for stories considered during the same session, but the simplicity of the process makes it especially accessible for teams and team members who are new to Scrum.

3.5. Broadcast with Information Radiators

First coined by Alistair Cockburn, an information radiator is the “generic term for any of a number of handwritten, drawn, printed or electronic displays which a team places in a highly visible location, so that all team members as well as passers-by can see the latest information at a glance” (13Ag). Information radiators provide succinct visual messages for anyone who is in the area; similar to how a construction zone sign proclaims “187 days without an accident” to both workers and those who drive by. The goal is to convey current project status to people who are not involved on a day-to-day basis and to provide enough information to avoid interrupting the team with questions. Posting this information also implies that the team acknowledges the status of the project and is willing to share it with anyone.

Information radiators are one of the best low-tech devices for broadcasting the status of a project. Since they are visibly posted, it is a passive process. No one has to seek out the status or refresh a web page. As Cockburn mentions “online files and web pages generally do not make good information radiators, because an information radiator needs to be visible without significant effort on the part of the viewer.” (Cockburn 2004)

At DMC, the information radiator shows:

- The current sprint’s stories and related tasks
- A Kanban-style board with work under progress
- The current build number and timestamp
- The most recent release number and timestamp
- All the stories from the product backlog

Other items that might be useful to post are:

- A graphical display of the number of tests planned versus passing
- Any planned resolutions discussed during the most recent sprint review
- The status of any of the team members if they will be on vacation or otherwise unavailable

At a glance, anyone can tell how many more items are in the backlog for the current project and which stories are under development during a sprint.

When we first started using an information radiator the workspace presented some challenges. It lacked large areas of windows or walls, and the facilities staff was reluctant to hang whiteboards due to the newness of recent renovations. Also, the development area was in a quieter spot, so not many people passed by.

Fortunately, we overcame those problems. Instead of a whiteboard, we used the cubicle walls for posting the backlog. Since it was difficult to get sticky notes to adhere to fuzzy cubicle walls, putting a long piece of strapping tape on the cubicles made a non-porous surface that worked better. The development area lacked space to display everything together, so we compromised by posting the backlog in one area, and the sprint status items in a slightly different area, but still visible from one spot. Moreover, passing traffic increased when sales materials were stored in an area just past development. Now the sales and training staff pass by the information radiator on a daily basis. Recently I saw a sales person looking at the cards, understanding what we are working on.

The biggest benefit of the information radiator is for the team itself. It creates a sense of completion as the backlog whittles down, and the sight of a once-daunting project dwindling down to a small tail of outstanding stories is a sure morale booster.

3.6. Draw Out Ideas

Reviewing architecture and design documents can be tedious and boring since the process is passive: sitting in meetings trying to evaluate someone else’s design work. Participants easily check out, often keeping their eyes and hands occupied by doodling in their notebooks while the meeting uses their ears and brain.

Drawing, however, is an excellent approach to engage the brain in multiple ways, making a more creative and memorable meeting for everyone. A recent Wall Street Journal article began "Employees at a range of businesses are being encouraged by their companies to doodle their ideas and draw diagrams to explain complicated concepts to colleagues." (Silverman n.d.) Drawing can be used to promote engagement, visual thinking, and enhance note taking.

I used to prepare diagrams or schema using Visio and hand them out in meetings only to realize that people were not fully engaged. I had gone through the journey of creating the visual, but for everyone else it was just a static image. Looking at a picture can be a fleeting event and the less energy one puts into it, the less memorable it will be.

In design meetings now, however, I arrive with an image in my head, and make sure I have a space to draw it out. The process is more engaging and we share the journey as the drawing is created. Describing the design and drawing it out simultaneously combines two methods of learning, sight and sound, helping to reinforce the ideas. This technique has a number of benefits.

- Telling stories while drawing helps people who learn using different styles. Drawing is a process and a result, your mind carves out the lines as you examine the subject.
- The images will transcend language: drawing out ideas works even when colleagues do not share a common first language or have the same level of technical understanding.
- Drawing is participatory: The brainstorm is directed and scoped, but democratic. Others may draw on the board or paper as well. It helps foster involvement in the critical thinking process, whereas a completed and polished image may give the impression that it is beyond criticism.
- Drawing brings out creativity. As the cartoonist Lynda Barry says about her graphic memoir/how-to book *What It Is*, "drawing or expressing your image is a therapeutic biological change that occurs in your body and makes you say 'ah!'" (Barry 2008).
- Studies have shown that drawing enhances retention. In *Moonwalking with Einstein,* Jonathan Foer devotes a chapter to discussing mind maps and how they correlate with memory palaces (Foer 2011). A memory palace is a visual mnemonic device used to help organize and recollect bits of information. As the team works with the image, they inadvertently create spatial memory palaces of the system in their minds. Later, when writing code or testing, the image can help clarify details and provide a context within the larger system.

To take it to the next level and get people involved, give them a pen and ask them to include their ideas in the drawing. If people object, saying they cannot draw, suggest instead that they "make marks on paper."

In addition to using drawing in meetings, drawing helps with other parts of the design process. Kevin Cheng, author of *See What I Mean* (Cheng, See What I Mean: How to Use Comics to Communicate Ideas 2012) promotes the idea of drawing comics for business reasons. Creating storyboards and scenarios as comics not only engages people in the process, but also saves time and effort. The sequential comic frames are simply another way to represent the steps in the user stories. Drawings can also be used to visualize the user experience, which is the main idea behind his web comic "OK/Cancel." (Cheng, OK/Cancel n.d.)

Save the drawings at least until the end of the project. Even when working on a whiteboard you can still take a photo of the board and the memory of drawing it will stay with people.

3.7. Simplify Design with Paper Prototypes

Requirements people, designers, and developers think on different levels. When they meet to plan what needs to be done they focus on different aspects of the specifications rather than on the problem that needs to be solved.

One way to avoid this is to embrace paper prototypes. When the team goes to work on a screen design, the fastest way to get everyone's participation is to use paper, markers, scissors and tape or glue stick. In a just a few minutes, one team member can sketch out an initial screen and then let others add to the work. Using black markers enforces simple screen design and removes colors and fonts from the equation. If people make mistakes or suggest changes, tape and scissors make it easy to rearrange the screen. A photocopier can be used to duplicate parts if necessary.

Working interactively on the page turns it into a shared design problem. Everyone in the team can participate in naming items, and the business analyst can remind the team of the real-world terms. Once the prototype takes shape, testers can apply their tests to the paper: How many characters? Is this required? Where do you go when you click on this?

Paper prototyping has been used for years, and it continues to be useful in this age of mobile devices. Whether you are designing a screen for a tablet, a smart phone, or a computer, the basic rules are the same. Using a paper model makes initial design easier. Print out an oversize template of an iPhone, photocopy it, and then use that to draw in the screens with markers. To replicate the user experience, you can stack up the pages, hold them in your hand like a phone, and then flip through the story.

3.8. Increase Communication by Pairing

Many Scrum proponents endorse creating cross-functional teams, but they do not always describe in detail the team mechanics. The high-level description is usually something like "Scrum relies on a self-organizing, cross-functional team. The Scrum team is self-organizing in that there is no overall team leader who decides which person will do which task or how a problem will be solved." (Cohn, What is Scrum Methodology? n.d.) As a result, team members may end up creating their own personal silos, with developers writing code, testers checking it, the business analyst reviewing, all without collaboration. Consequently, the hand-off points become discrete, and the communication between team members is limited.

Similar to cross-functionality, many Agile evangelists also point to pair programming as a way to create better code with fewer bugs than when programmers work alone. But, why limit pairing to only developers? By pairing developers with testers or testers with the technical writer, or the business analyst with the developer, you can achieve benefits similar to pair programming for the entire team.

For example, we occasionally have the developer write the rough draft of the user documentation. This process helps the developer better understand the user workflows and often uncovers gaps in functionality. Moreover, as the technical writer sees early versions of the system, he can look at it from the point of view of documenting user workflow and suggest changes to the UI to improve usability.

In the same way, asking a developer to sit with the tester while creating the initial test plan increases the informational bandwidth between them. The developer can point out areas that may need more scrutiny or were touched in development but not obviously connected to main functions of the code.

And, the tester may notice areas where the developer may not have fully understood the implementation of a feature. As they create the test plan, the developer may recognize that the tester is not working with the same mindset and steps that he assumed while writing the code.

These pairings are a great way to shorten the loop. Throughout the development process, the goal of the Scrum framework is to increase participation, ideas, and the general quality of the product through shortened feedback loops. This means each minor iteration of the process should have feedback, and meeting face to face is the best way to accomplish this.

Managers and Scrum Masters can do the following to make pairing sessions successful:

- Make sure that the team has time to pair and suggest including pairings as tasks attached to stories. This raises the visibility of the pairings and allocates the time.
- Ensure that both sides get encouragement. Developers like to show off their code, but can be protective of it. If the interaction is confrontational, it will not work and will not become a habit. The Scrum Master should plan to attend the first couple pairings and encourage a positive outcome.
- Keep the sessions concise. When working on complex problems, pair programming and other types of pairing is useful, but difficult. It is two different minds working together on the same problem and sometimes people pull in different directions. Set aside time for the pairing, but avoid forcing the issue. Let the team self-organize.

The result of pair programming, testing, writing, and other sessions is that members of the cross-functional team will get a better understanding of the specialties of the other members in their group and seeing other people’s perspectives helps everyone create better solutions.

3.9. Keep a Project Diary

All of the previous tips have provided ways to increase project visibility, participation, and collaboration. Those ideas were designed for a disparate audience: the team, managers, other employees, customers, and executives. This last idea, however, is to help you collaborate with one specific person: your future self.

Keeping a log of activities is not a new idea. Projects were tracked when Stone Age people drew pictures of a hunt on cave walls. Most tracking consists of structured metrics used to show managers and executives that the project is on target for the impending future release. Keeping a diary, however, is a way to track where the project has *been*.

A daily diary, like a personal diary, is meant for your eyes, and should be written for you as the audience. It can contain many bits of information that might not incorporate well into the project process. For example, if you have a complicated email exchange, do not leave it in your email client. Spend time extracting the exchange into paragraphs and annotate with pertinent details. Paste everything into the diary, and date it. Later, when someone asks, “What were we thinking at the time?” the diary will act as a personal reference to jog your memory.

Another use of a diary is to note exceptional efforts by team members. If someone on the team has a shining moment, make a note of it in your project diary. For supervisory staff who write employee performance reviews or provide feedback to Human Resources for employee assessments, these examples in the diary will prevent a lot of head scratching later and will benefit everyone. Most importantly, the project diary is a way to provide a history for you. After many projects, you may wonder what you have accomplished, and how you got there. Since the diary entries were personal for you, you can include ideas, guesses, feelings and emotions among the timeline and events.

During retrospectives, the diary can help you prepare your notes. Years later, when you recognize a situation, the project diary provides a reference, of not only how the situation was handled, but also your thoughts at the time.

4. Conclusion

The process of developing software is essentially one of taking ideas and making them tangible. Since it is usually accomplished with a team of people assigned to create a complex software solution, it is necessary for the team members to communicate their ideas and opinions to others. High-tech teams often navigate toward high-tech solutions to accomplish this goal, but those solutions come at a cost and can complicate the project management process in unexpected ways.

Even Agile projects are complex. Although Scrum reduces some of the complexity, developing modern software solutions is fraught with risks and unforeseen dependencies. Simplifying the project management process frees the team to focus on creating the solution, rather than occupying themselves with administrative tasks. Additionally, involving the team more in the project management process achieves the Agile goals of collaboration, short feedback loops, and user-centric development.

While all of the tips presented in this paper have been used in conjunction with Scrum, they are not limited to an Agile project framework.

Many of these tips may not be new to you, but I hope that you see them in a new light. The low-tech goals of collaboration, increased communication, and democratic processes align with the tenets of the Agile Manifesto to promote face-to-face communication, people over process, and collaboration over silos.

References

Agile Alliance. http://guide.agilealliance.org/guide/radiator.html (accessed 07 09, 2013).

Barry, Lynda. *What It Is.* Drawn & Quarterly, 2008.

Beck, Kent, and et al. *Agile Manifesto Principles.* 2001. http://agilemanifesto.org/principles.html (accessed 07 07, 2013).

Cheng, Kevin. *OK/Cancel.* http://okcancel.com/ (accessed 07 07, 2013).

—. *See What I Mean: How to Use Comics to Communicate Ideas.* Rosenfeld Media, 2012.

Cockburn, Alistair. *Crystal Clear: A Human-Powered Methodology for Small Teams.* Addison-Wesley Professional, 2004.

Cohn, Mike. *Learn About the Scrum Product Backlog.* Mountain Goat Software. http://www.mountaingoatsoftware.com/scrum/product-backlog (accessed 07 07, 2013).

—. *Planning Poker: Agile Estimating.* Mountain Goat Software. http://www.mountaingoatsoftware.com/topics/planning-poker (accessed 07 07, 2013).

—. *What is Scrum Methodology?* Mountain Goat Software. http://www.mountaingoatsoftware.com/topics/scrum (accessed 07 07, 2013).

DeMarco, Tom and Lister, Timothy. *Peopleware: Productive Projects and Teams.* New York: Dorset House Publishing Co., 1987.

Foer, Joshua. *Moonwalking with Einstein: The Art and Science of Remembering Everything.* New York: The Penguin Press, 2011.

Light, Adam. "Why Scrum Works (and How to Sustain it)." *Ignite.* 05 16, 2013. http://igniteshow.com/videos/ignite-tao-v4-may-162013-adam-light (accessed 07 07, 2013).

New product development. Wikipedia. http://en.wikipedia.org/wiki/Fuzzy_front_end#Fuzzy_Front_End (accessed 07 07, 2013).

Silverman, Rachel Emma. "Doodling for Dollars." *Wall Street Journal.* http://online.wsj.com/article/SB10001424052702303978104577362402264009714.html (accessed 07 07, 2013).

"Wikipedia entry User story." http://en.wikipedia.org/wiki/User_story (accessed 07 07, 2013).

Preparing the Next Testers: An Undergraduate Course in Quality Assurance

Peter A. Tucker

ptucker@whitworth.edu

Abstract

For most undergraduate institutions, a degree in computer science has a heavy emphasis on programming skills. Students learn programming fundamentals, data structures, and software development skills. Along the way, they also learn data management, operating system concepts, and computer architecture. Rarely, however, does a degree in computer science include the skills, techniques, and thought processes involved in software quality assurance. Indeed, anecdotal evidence suggests that quality assurance is a career choice considered by some academics as a “lesser” craft. We believe that a course in quality assurance is an important elective in a computer science education; that it will open opportunities for students and make them more attractive to employers looking for QA engineers. Further, we believe a course in quality assurance will help those students who take positions in software development or management better understand their roles in assuring quality in software.

We began offering a course in quality assurance in 2006. Our QA course has gone through four iterations. The most recent iteration focused on the five views of quality outlined by Garvin and then applied to software quality assurance by Kitchenham and Pfleeger. Garvin’s five views are: Manufacturing, Production, User, Value, and Transcendental. By focusing on each of these five views, students are able to take a more structured approach to thinking about quality assurance, and are better able to understand how quality assurance fits in with the difference aspects of software engineering.

This paper outlines our most recent iteration of the QA course. We discuss the different views and how they were presented to students, as well as the various activities students participated in during the semester. We discuss what went well and what kinds of things can be improved. It is our hope that a good discussion can take place between those in academia and industry practitioners to ensure that the exposure students get to QA skills and techniques is appropriate, and does indeed prepare graduates with degrees in computer science for various careers in software engineering.

Biography

Peter A. Tucker has served for the past ten years as a professor of computer science at Whitworth University. His regular course load includes quality assurance, database management, software engineering, and mobile application development. His research interests include quality assurance and data stream management. In addition, Dr. Tucker is a consultant at NextIT in Spokane, WA, where he is exploring QA automation techniques and implementing tests specific to some of their core technology. Dr. Tucker began his career at Microsoft, spending five years as a software design engineer in test, and then three more years as a software design engineer. Dr. Tucker received the Ph.D. degree from Oregon Health & Science University in 2005.

1 Introduction

College students in computer science typically interview for one of three kinds of internship or full-time positions: developer, quality assurance, or program management. Most CS students will take many classes that focus on programming techniques, so they often have some programming skill. They often will have done small-scale software projects as well, so they will have at least a simple understanding of software design and program management. However, students rarely have any experience in software testing, making them ill-prepared for interviews for positions in quality assurance. Those students that do make it through the interview for a QA position are not often prepared for what that job will entail, and have to learn even the most basic QA skills on the job. A course in quality assurance could help toward that end, preparing more students toward a career in quality assurance. Indeed, there exists a recognized need for a course in quality assurance as at least an elective for undergraduate students in computer science:

- Many companies look for testers with strong technical skills in software design and/or development.
- The Association for Computing Machinery (ACM) Computer Science Curricula (2008) mentions that QA was one topic that received attention in discussions with industry.

At Whitworth, we have offered a course in QA four times over the past eight years, designed to be accessible to computer science students in their second year of study. The motivation for the course came from a Whitworth alum who had been in quality assurance at Microsoft for 20 years. She requested that the course be offered to increase the number of employee candidates for QA positions. In their experience, few QA candidates have strong technical skills and strong testing aptitude. Talking with employees at others companies, the need for more strong candidates in QA seems to exist across industry. A course such as this, housed in a computer science department, would expose more students to the technical skills and the quality assurance skills required for a position as a software quality assurance engineer. Partially as a result of offering this class, Whitworth has produced a number of graduates who are or have held positions in quality assurance, including employees at Microsoft.

In the course's most recent iteration, offered in fall 2012, we modelled the course topics using Garvin's (1984) views on quality (and built upon by Kitchenham and Pfleeger [1996]): Manufacturing, Product, User, Value, and Transcendental. Our goal was to expose students to different points of view for approaching software quality. Students were asked to develop a test plan and implement tests, and to write bug reports as they find bugs.

The goal of this paper is to present how Garvin's views on quality played out in our QA course, and to start a dialog on what kinds of things industry would like to see in such a course. We are working toward a textbook on quality assurance, suitable for sophomores in computer science, so more universities can also offer a course in software quality assurance.

2 Overall Course Structure and Practice

Our QA course is designed to be very hands-on. As new concepts are introduced, students are asked to put those concepts into play. For example, as students learned about boundary test cases, we would present them with a specific function or interface and have them define the boundaries they saw. Over the course of the semester, students were to test a real software product of their own choosing (usually downloaded from a site such as sourceforge.com. Again, as new concepts were considered in class, students were to apply those concepts to the software they were testing.

2.1 Textbooks

The main textbook used in this course is Jorgensen's Software Testing: A Craftsman's Approach (2001). This text takes a practical approach to software quality assurance. It focuses largely on functionality testing, with minimal discussion on other "-ilities" defined in ISO 9126 (reliability, usability, efficiency, maintainability, and portability). As such, it was heavily used early in the semester as we learned fundamental QA techniques, and less as we explored QA from other angles, guided by Garvin's views. In addition, we recommended students use Kaner's Lessons Learned in Software Testing (2002). This text has many short lessons, and we ended nearly every class session with one or two lessons. Students really looked forward to this part of class; the lessons are short and practical, and are often applicable in any career direction in computer science.

2.2 Final Project

Students were also expected to form teams of 2-3 students for a project that would last most of the semester. They were to find and test a real software product. There were only two requirements: first, the software was not considered a release-quality version, and second, that it wasn't software that they had written. Students were encouraged to go to SourceForge.com and find software that was in alpha or beta stage. The motivation for this requirement is that students would be more likely to find bugs in software that was not ready to release. It was also important that the students have access to the developers' source code, so that they could perform code reviews, static code analysis, and code coverage. Details on these code-level tasks are given in Sections 4.1 and 4.2.

In the first iteration of this course, I found the software product that we tested as a whole class. I took this approach for two reasons: First, so that I could ensure that the application was testable, and second, so I could simulate a real world testing organization by assigning different parts of the software to different groups. The downside to this approach, though, was that I picked a software product that most of the class hated. Rather than trying to find software that would be more accepted, I have since allowed students find their own project. When students find something they are interested in testing, they demo the projects to me, so that I can determine testability. This approach has worked better since students are more motivated to work with a product that they actually like.

3 Initial Class Session Details – Fundamental Techniques

To start the semester, we worked on fundamental techniques and practices in quality assurance, so that students would have basic skills. We covered boundary testing, random testing and fuzz testing, equivalence class testing, and decision table testing. All of these topics relied heavily on material from Jorgensen. We then discussed test case creation, version control systems, and issue tracking. I wanted students to understand these basic tools so that we could apply them throughout the semester as we moved into each of Garvin's views.

Homework for these initial sessions asked students to apply these concepts in simple, contrived situations, largely based on exercises from Jorgensen. Students were given simple functions or interfaces, and asked to: come up with boundary tests, apply random value testing and fuzz testing, define equivalence classes, and define decision tables.

3.1 Quality Assurance Techniques

We spent the first three weeks of the semester covering a number of fundamental quality assurance techniques. As a running example, we used the classic `IsTriangle` function (Myers 2004) as a running example, with the added stipulation that the triangle sides had to be in the range [0,200]. The `IsTriangle` function takes three integer parameters representing the lengths of the three sides of a triangle. Based on those side lengths, the function should return the appropriate type of triangle: `Scalene`, `Isosceles`, `Equilateral`, or `NotATriangle`.

We started with boundary testing. That is, use test values that operate at the limits of the input. Following Jorgensen, we built up different levels of boundary tests. Students started with testing at or near the boundaries for a single parameter using legal values (e.g. `IsTriangle(0,10,20)`, `IsTriangle(10,20,199)`, `IsTriangle(10,20,200)`, and so on), then added tests to include illegal values for a single parameter at the boundaries (e.g. `IsTriangle(-1,10,20)`, `IsTriangle(10,20,201)`, and so on). We then added tests where all boundary cases were hit together (e.g. `IsTriangle(0,0,0)`), and finally special case boundary tests. The `IsTriangle` function illustrated this well for students, and they were able to pick up on this concept well. Particularly, `IsTriangle` was a very good illustrative example for the special case boundary tests. We wanted to hit boundaries for each kind of triangle. For example, we could test boundaries for isosceles triangles with `(5,5,8)`, `(5,6,10)`, and `(5,5,10)`.

We then moved into random value testing, by using randomly-generated values as testing values. Students understood quickly how to apply random value testing. We included discussion on the issue of test reproducibility when using random value generation as well. This discussion helped motivate the need for seeding a randomizing function, so that the same sequence of "random" numbers could be generated multiple times. Again, the `IsTriangle` function served us well here. Students saw the problem of blindly come up with three random values for a triangle, and instead were able to see how to generate combinations of random values that defined specific kinds of triangles. Finally, students were shown fuzz testing (Miller 1990) as an application of random value testing. A fuzzer takes a string input, and modifies it slightly, and randomly. For example, given the string "hello world", a fuzzer function might return any of the following strings: "hell▬ wo∟ld", "helo world", "hello wûrld", or "hhello?wArld". Students were shown a simple fuzzer function, and were expected to use it in their own testing.

Our next topic was equivalence classes, where groups of tests are classified as redundant, and we motivated this topic by discussing the number of tests required for `IsTriangle`. For example, the test cases `(5,5,5)`, `(10,10,10)`, and `(100,100,100)` are of the same class (equilateral triangles), and so executing all three would likely be redundant. Students were able to see that many of the test cases belonged to the same class. Students also could see the applicability of boundary testing random value testing here: to test boundaries within an equivalence class, and be able to come up with random values within a class, as useful testing strategies. Again using the case of equilateral triangles, `(0,0,0)` and `(200,200,200)` are boundary tests in the class, and then we could pick some random value `x` in `[1,199]` and test `IsTriangle(x,x,x)`.

3.2 Quality Assurance Tools and Practices

With some QA techniques in mind, students learned about test case development, using whichever programming language they were comfortable in. Students were given some general tips, including: test cases should one test one thing, and be self-contained; test cases should clean up after themselves; and QA engineers should look for opportunities to share code through libraries.

Students were exposed to some existing testing tools and frameworks. Most in-class software development at Whitworth uses Microsoft's tools, so students were shown the Unit Testing Framework in Visual Studio. Students learned about the `Assert` class, and how to check results against expected values through `Assert.Fail()`, `Assert.AreEqual()`, and `Assert.IsTrue()`. Students also saw how to develop specific test cases for library functions (we used vector and List classes as examples), and how to execute them in Visual Studio. From there, we worked on the UI automation tools also in Visual Studio. Students were able to record scripts through the tools and generate the code used for those scripts, then again execute them.

Finally, we discussed bug reports and issue databases. We used Bugzilla as an illustration. We discussed a bug's lifecycle (from bugzilla: http://www.bugzilla.org/docs/2.18/html/lifecycle.html), and the different parts of a bug report. Students were given various tips for writing bug reports, including discussion on the differences between priority and severity, the importance of being specific in reproduction steps, and how critical it is to make the title short and descriptive.

As students worked on testing their software product, they were expected to use these tools, or similar tools as appropriate. Test scripts were to be checked into a version control system so I could keep up with their work, and bugs were to be posted to some issue tracking system.

4 Class Sessions Applying Garvin's Views

With the fundamental techniques and tools behind us, we moved into Garvin's views on quality and how they applied to software testing. Thinking through all five views gives students more angles to think about as they consider the quality of the software they are testing. Students were expected to apply what they were learning directly to the software project they were testing. Homework, therefore, consisted of some contrived scenarios for testing, but also some more open-ended work as they considered how best to test their product.

4.1 Manufacturing View

According to Garvin, the manufacturing view focuses on the supply side of software development, particularly with an eye on engineering and manufacturing practices. Software should be developed using accepted practices and foundational techniques. In considering manufacturing practices, we focused on ISO/IEC 25010:2011, and the Capability Maturity Model. We also discussed other QA tools, including code coverage tools, source profiling tools, and test automation strategies. One difficulty we encountered in this portion of the class was in finding good, free tools for each project. Since we left the assignment open, and allowed students to find software to test that interested them, they all needed different tools. For example, each project was in a different language (C#, C++, and Python), so each team needed to find their own code coverage tool. This search was easier for some languages (e.g. Python) than others (e.g. C++). However, just the act of searching was a good lesson. Next time we offer the course, we will do a better job of providing students links to various tools that are available.

Focusing on ISO/IEC 25010:2011 gave us the motivation for developing test plans, as part of its General Requirements. Thus, we were able to get students thinking about test plans for their software early in the process. Initially we kept the test plan document fairly simple. We built it up throughout the semester as we learned new concepts. See Figure 1 for the initial test plan contents.

- Title
    - Software, including version and release numbers
    - Document Revision History
    - Table of Content
- Purpose of the document
    - Intended audience
    - Software overview
- Naming Conventions
- Test Outline
    - Testing approaches used by feature, functionality, process, …, as applicable
    - Boundary value analysis, equivalence classes, decision tables
- Test automation
- QA Tools to be used

Figure 1 – Initial Test Plan Contents

Students then were introduced to practices and tools for code and test case reviews. To think about code reviews, we used concepts from Spinellis' Code Reading, and considered specific code snippets to review. We considered specific "red flags" QA might look for in code reviews, including:

- Using conditionals to determine boundary cases
- Reviewing loops to try to cause forever loops, and also to look for test cases that make sure the loop executes zero times, one time, and many times
- Reviewing pointer and resource usages
- Reviewing storage, particularly temporary storage
- Sterilizing user input

With these ideas in mind, students were presented with simple code snippets, and asked to look for instances of those red flags.

Finally, we also considered static analysis tools, and how they can be used in QA. We considered the results from tools including Visual Studio's Code Analyze, Lint, and Resharper. As we worked through these tools, students were asked to consider how the results could be used to help in quality assurance. To motivate this discussion, we considered the results reported by Ruberto (2012), and how the quality of that mature product improved through the use of existing static analysis tools.

4.2 Product View

In the product view, quality is measurable, and determined by specific attributes of the product. Are there features of this product that make it unique from other, similar products? Is the software more efficient than existing products, in terms of speed or memory? Are there security features that make it a higher quality product? The product view focuses on measurable attributes, and so we can talk here about common software metrics including bug counts and bug rates, code coverage results, and mean time to failure.

We started this section with the practice of exploratory testing. Certainly, this discussion could have fallen with the user view just fine. In fact, we will probably move it to the user view next time. We put it here as a way to discover unique features of the software. Students were given tips and direction, then asked to perform exploratory testing on a particular dialog in GIMP, an open source image processing tool (http://www.gimp.org). At first, students were lost. They didn't know where to start. However, once they started down a path, they did a good job of keeping curious, of writing things down, and of learning new features of the product.

To discuss code coverage, we first took the approach of describing code as a directed graph, based on introductory discussion in Ammann and Offutt (2008). Students were given a small section of source code and asked to define a graph for that code. They were then asked to cover the graph with test cases. This visual approach really helped students understand what we mean by some of the different levels of code coverage. We then got into code coverage tools, and students were shown some examples. Finally, we considered the discussion from Roseberry (2013) regarding the myth that high percentage of code coverage means high quality. Students quickly picked up on why high coverage doesn't necessarily mean high quality software, but it was a good discussion that helped students think more carefully about what code coverage results really mean.

Finally, we used this area to introduce security testing, starting with the concept of taintedness. We took ideas from Whittaker on testing for security, and tried out many of his attacks. Obviously, students found this section fun. Security testing could easily be an entire semester of material; what we did here was simply intended to be an introduction.

Student homework during this section of the course consisted of applying the material to testing their own project. Students were to find tools for performing code coverage on their software, looking for places where they'd need to add more tests. In Section 5, we will see that this part of the assignment caused quite a lot of frustration. Student groups were to spend 30 minutes in exploratory testing, documenting all they did and the issues they ran into. They also were to try out Whittaker's security testing, and again reporting any issues. Finally, they were to update their test plan to include more discussion on how exploratory testing and security testing would be employed on their project.

4.3 User View

In the user view, quality "lies in the eye of the beholder." That is, if the product is of high quality, the user will tell you so. (Or, more likely, the user will tell you when the product is low quality.) For our class, it allowed us to focus on the "-ilities" outside of functionality: reliability, usability, maintainability, and portability, and ways to address testing in those areas. Students were introduced to stress testing, targeting heavy network activity, huge data files, high levels of concurrent activity, and activities that are long running. We also considered performance testing as both a way of testing reliability (especially with some standard benchmarks) and usability. Here students were asked to look at the PerfMon tool in Windows, to be able to watch CPU and memory activity during testing. For usability and portability, we discussed the notion of "eating your own dogfood" as well as usability labs. Additionally, and related to usability, we discussed user experience data and usability tools from Page et al. (2008).

As before, student homework was to discuss the different -ilities, and how they were to be used in testing their own project. Would they incorporate some kind of stress testing? If so, what would those tests look like? Would they use monitoring tools such as PerfMon to track performance, or develop other performance testing tools? And finally, would they set up dogfood sessions, or "bug bashes" to test the product's usability. This discussion was also added to their test plans.

4.4 Value View

The value view allows for cost to be considered in the discussion of quality. Users may be willing to sacrifice a level of quality if it means paying less for the software, as long as the software continues to meet the needs of the user. In class, we discussed managing the tension between quality, time, scope, and resources. In terms of the tension of quality and time, we discussed bug triages, and the decision of which bugs to ship with. Regarding quality and scope, we focused on decisions related to which features to keep or drop in order to ship on time. For quality and resources, we talked about hiring good people, the costs of configuration labs, and interactions with clients. All put the value view into a very practical light for students. We also got into some discussion from Jones and Bonsignor (2011), and the cost of releasing low-quality software. Students were very surprised at the costs of low-quality software after the product is release.

At this point in the semester, students were finishing up their final work on testing their software. There wasn't any specific homework assignment here, other than to continue testing and prepare for their final presentation.

4.5 Transcendental View

Finally, according to the transcendental view the quality of software is obvious to those who use it. Quality is not measured. From Pirsig, "But even though quality cannot be defined, *you know what Quality is*." Certainly, this is the hardest view to teach, as it requires experience as much as anything else. We worked through discussion here from Emery (2012) on "testing quality in". We focused on the need for quality to be a culture within the development organization, including management, designers, developers, and testers. Everyone on the project needs to expect quality in their work. Further, we discussed practices that can help within each group in the organization, including hiring the right people, employing good practices (e.g. test-driven development), and respecting the work of others.

5 Course Evaluation

Over the years, the quality assurance course has been well-received. The course is not required for our majors, but we always get a good number of students in the class. Students realize that having some exposure to quality assurance may help them in searching for jobs, whether in a QA role or as a developer or designer. At the very least, students get job interviews in part because they had a course in quality assurance on their resume. I motivate the subject to all computer science students by telling them that QA skills can be applied in every aspect of software development.

In many ways, the course has been a success. Our first goal was to try to encourage more students in computer science to consider quality assurance as part of a career path, and that has worked. Students who have taken this course have moved on to take internships and full-time jobs in QA, at companies including Microsoft, Adobe, NetApp, and NextIT. Some of those students have remained in QA roles, while others have moved into other roles in software development. Even those students who took development or design positions right after college have reported that taking the quality assurance course helped them succeed in their careers.

More generally, students report that taking the QA course has opened their eyes to the different careers in software development. When they enter the major, most students believe that development is the only career path. This course has shown them that there are other interesting directions they can take.

Students offer two main critiques of the class. First, that finding tools to help them test their project is not easy. Students pick their own software project to test, then they are asked to find tools for static analysis, code coverage, memory profiling, and so on. These tools are easily available for some projects (e.g. we were able to get code coverage in Python easily), but other projects are more difficult (e.g. we weren't able to get good code coverage results for some of our C++ projects). Students would like more help in finding these tools.

- Introduction
  - Title
  - Software, including version and release numbers
  - Document Revision History
  - Table of Content
  - Objective of Testing Effort
- Software product overview
  - Relevant related document list, such as requirements, design documents, other test plans, etc.
  - Relevant standards or legal requirements
- Overall software project organization and personnel/contact-info/responsibilities
  - Test organization and personnel/contact-info/responsibilities
  - Assumptions and dependencies
  - Testing priorities and focus
  - Scope and limitations of testing
- Test outline - a decomposition of the test approach by test type, feature, functionality, process, system, module, etc. as applicable
  - Outline of data input equivalence classes, boundary value analysis, decision tables
  - Test environment - hardware, operating systems, other required software, data configurations, interfaces to other systems
  - Code reviews and test reviews, including a schedule of what was done, and when
  - Code coverage – discuss tools used, and code coverage goals
  - Exploratory testing, including a schedule of what was done, and when
  - Security testing techniques and tools
- Non-functional Testing
  - Reliability
  - Usability
  - Maintainability
  - Portability
- Test automation and tools - justification and overview
  - Test tools to be used, including versions, patches, etc.
  - Test script/test code maintenance processes and version control
  - Problem tracking and resolution - tools and processes

Figure 2 - Final Test Plan Contents

The second critique is in the text books we've used. Most texts for QA seem to be written for students in at least their fourth year of undergraduate education, and assume students have a good foundation of software engineering. I want this course to be open to students who have completed their second semester of computer science, so that they can use these skills in their projects starting in their second year of their education.

6 Conclusion

As mentioned earlier, this course has become a regular offering at our school, and has been quite successful. Students appreciate learning the different skills and processes involved in quality assurance, and are glad to see new career paths for their skills.

One important goal we have heard from professionals in quality assurance is that students learn about writing test plans. As mentioned, students created an initial test plan (Figure 1), and then many of the homework assignments had students update their plan as we discussed new material. The final test plan was more complete, though certainly there could have been more added. The final test plan contents are shown in Figure 2.

In the next iteration, we will keep the focus on Garvin. It allows students to think about the whole task of software development. As mentioned before, we will move the exploratory testing piece to the user view discussion, and look for better tools that students can use. As we continue with this work, material will be compiled into a textbook.

Also, as mentioned earlier, we need to do a better job of helping students find tools they can use. For example, free code coverage tools are easier for students to find for projects written in Python than for projects in C++. More work needs to be done before the semester begins to find these tools so that students can use them.

It would also be worthwhile to devote more time to the “-ilities” of quality assurance. This section seemed a bit more ad hoc, and the requirement of adding appropriate testing for the “-ilities” was too vague. For example, we might devote a class session as a bug bash session, awarding prizes to students who find the most bugs. Another addition might be to spend more time with some of the efficiency measuring tools, including perfmon, as well as giving students time in class to use them.

7 Acknowledgements

The author would like to thank Microsoft Corporation, and particularly Cherie Ekholm, for their support for our QA course, as well as Whitworth University for its support as the course has developed. Finally, the author would like to thank the reviewers for their helpful comments on early drafts of the paper.

References

ACM/IEEE-CS Joint Interim Review Task Force. 2008. Computer Science Curriculum 2008: An Interim Revision of CS 2001, Report from the Interim Review Task Force. http://www.acm.org/education/curricula/ComputerScience2008.pdf

Ammann, Paul and Jeff Offutt, 2008. Introduction to Software Testing, Cambridge University Press.

Emery, Dale, 2012. Testing Quality In. *Pacific NW Software Quality Conference*, Portland, OR.

Garvin, David A., 1984. What Does “Product Quality” Really Mean? *Sloan Management Review*, (Fall):25–35.

ISO/IEC 25010:2011. Systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – system and software quality models.

ISO/IEC 9126-1:2001. Software Engineering – Product Quality – Part 1: Quality Model

Jones, Capers and Oliver Bonsignor, 2011. The Economics of Software Quality. Addison-Wesley.

Jorgensen, Paul, 2001. Software Testing: A Craftman's Approach, CRC Press, Inc., Boca Raton, FL.

Kaner, C, James Bach, and Bret Pettichord, 2002. Lessons Learned in Software Testing, Wiley.

Kitchenham, Barbara and Shari Lawrence Pfleeger, 1996. Software Quality: The Elusive Target. *IEEE Software*, 13(1):12–21.

Miller, Barton P., Lars Fredriksen, and Bryan So, 1990. "An Empirical Study of the Reliability of UNIX Utilities", Communications of the ACM 33, 12.

Myers, Glenford J., Tom Badgett, Todd M. Thomas, and Corey Sandler, 2004. The Art of Software Testing, Wiley.

Page, Alan, Ken Johnston, and BJ Rollison, 2008. How We Test Software At Microsoft. Microsoft Press.

Pirsig, Robert M., 1974. Zen and the Art of Motorcycle Maintenance. Bantam.

Roseberry, Wayne, 2012. Code Coverage Isn’t Quality. It Isn’t Even Coverage. *Pacific NW Software Quality Conference*, Portland, OR.

Ruberto, John, 2012. Introducing Static Analysis in a Mature Codebase. *Pacific NW Software Quality Conference*, Portland, OR.

Spinellis, Diomidis, 2003. Code Reading: The Open Source Perspective. Addison-Wesley.

Whittaker, James A., 2003. How to Break Software Security. Addison-Wesley.

Quality Begins with Design

Jeyasekar Marimuthu

Jeyasekar_marimuthu@mcafee.com

Abstract

In most software development organizations, QA has been a passive member in the design phase. QA's role has been assumed to begin only when the implementation phase kicks off. Had QA not missed capturing some of the valuable details during design phase, wouldn't their role be much effective than now?

This paper is about involving QA from the design phase. The amount of gain that organizations will benefit by having a QA architect is explained in this paper. There are some key advantages if QA penetrates deeper in design. Some of the noteworthy data points are abstracted in this section.

QA's involvement in the design phase will help the QA team understanding weightage of each user story/ theme/feature-group in terms of complexity, inter dependency, number of functions involved, distributed coverage across various layers of the application and the required number of unit test cases.

QA involvement in the design phase will also help detecting issues in advanced stages of the project and the same will help estimating QA costs with better accuracy.

In design oriented QA planning, test plans will be created in accordance with code flow. Test cases will be executed in a systematic approach. Test cases will be optimized as required by the design. A lot of guess work during QA planning will be avoided. Test ownership can be effectively assigned when the fundamental design is known in advance. Sometimes having a single QA owner for common- code (designed to cover multiple stories across different feature groups) can help avoiding multiple owners.

The efficiency of automation significantly increases when QA knows the design well. Effective automated cases will increase the code coverage. Also QA will gain visibility into the non-functional requirements like fault tolerance, extensibility, reliability, maintainability and availability.

How can QA begin their contributions to design? Let us see in the sections below.

Biography

Jeyasekar Marimuthu is a senior technical lead with 13 years of development experience, currently working with McAfee India Center at Bangalore. Over the past 7 years, he has been on a technical leadership role involved in software development. Out of 10+ product releases, his collaboration with QA has given him wisdom in implementing Quality in much advanced level.

Jeyasekar has worked with a number of security management products for enterprise in McAfee. His development contribution on Middle ware, database technologies over a decade has helped seeding some good thoughts at work. Jeyasekar has a Master of Computer Applications degree from Madurai Kamaraj University of Madurai, India.

1. Introduction

In the past, the primary role of quality engineers was to gate-keep releases and to certify products against a set of quality tests. The products that followed a waterfall model took a hit with a huge number of defects towards the end of release cycles. Since a large amount of use cases were addressed in the end, the load on quality staff used to be high and the same has resulted in creating a huge backlog of unaddressed defects.

Also, it has been realized that defects that were pushed to later project cycles incurred a greater cost than the ones that were caught earlier. As per corporate records, the defects caught in early cycles have reduced customer call volume by 20% to 30%. This metric has been proven in my last two projects that I delivered in my organization.

When companies shifted their development model from waterfall into agile, the defects were caught and fixed within the iteration cycle. The agile model helped the engineering teams to correct flaws and issues within the time window of iterations. This approach has distributed the QA workload more evenly across the project.

Though the situation had improved for QA in terms of distributed load, the following issues continue to bother. In Agile, QA deals with functional use cases by means of stories and tasks. Each story gives them an idea on what is being delivered, not how it is delivered. Implementation details are unknown. The focus is only on results. For example, if the story states "user should be able to add the values on click of ADD button". QA traditionally tests whether the addition is implemented correctly. QA never had an insight how the addition is done. Had he or she known the "how" details, QA would have steered their testing more effectively.

Let us see the pain points that projects encounter by not involving QA in the design phase.

2. Pain points

SDET (Software Design Engineer in Test) staff work closely with developers, test APIs and automate white box tests. They have a fair idea on what code has been implemented and what functional areas are to be unit tested and automated. Though having a SDET improves the situation, the process is still not fully effective.

1. Their focus is mostly on the code that is being implemented. The pain point is that if the implementation is reworked a number of times by developers, their white box test cases also need to be reworked.
2. The focus is around unit tests. There are no tests to validate the design. For example, if a developer has written the following function,

   public int sum(int a ,int b)

The designated SDET writes unit tests for the above mentioned function with a bunch of input combination. What if the developer has overlooked that this function could perform summing of more than two numbers?

3. When there is a change in the fundamental architecture of the system, the whole bunch of automated and manual test cases are rewritten. The QA plan misses an important element of "QA architecture" document. In the presence of that document, the amount of changes in rewriting test cases could have been as minimal as possible, because a good design will always prevent any large scale changes.
4. Tests are tightly coupled with implementations.

5. QA tests are not given weightage based on the complexity of the task. The high complex tasks should be preempted while testing so that the project’s risk is reduced.
6. Nonfunctional aspects like scalability, performance, and concurrency are given less focus while writing a test case. For example, if a SDET engineer is writing tests for the following function, he/she often overlooks having boundary conditions and concurrency checks tested.

   public int sum(int[] intArray)

   Expected boundary conditions: Populate the array with max number of integer values.

   Expected concurrency condition: Create as many number threads and invoke the function with random input samples and assess the behavior.

Though having a SDET improves the situation further, most of the white box cases written by SDET are around the implemented code. They only help developers to add additional unit tests or discover automatable functional tests.

Still, there is huge potential for advancing the quality process by introducing a Quality Architect or SDET right from the design phase.

Shouldn’t the approach of a SDET be more design oriented and systematic rather than just being a unit test engineer?

3. Need for Quality Architect in the design phase

During the design phase, high level and low level designs are created against a set of requirements. In agile, stories are picked from a backlog and designs are done at modular levels. This makes easier for QA staff to understand the design and to derive QA specific designs. The design phase also allows architects to come up with detailed system diagrams that capture the following:

1. Logical breakdown of functionalities against functional and nonfunctional requirements.
2. Separation of concerns - An established way of reducing complexity.
3. Interface designs for internal and external facing APIs. Internal facing APIs are meant for SUD (System Under Development) whereas the external facing APIs are meant for external systems that want to integrate with SUD.
4. Defining nonfunctional system attributes such as Fault tolerance, specific set of tradeoffs, extensibility, reliability, maintainability and availability.

In order to integrate quality with the design phase, the QA architect should prepare test plans for each of the item mentioned above.

Organizations seldom have a designated owner for creating quality designs for Systems Under Development. In the absence of such role, it is important for organizations to designate a Quality architect or a senior SDET to take control of quality designs. The designated owner will work closely with product architects and other design owners and will create a quality design document for the System Under Development. The owner of the quality design document is expected to have wealthy knowledge about the system both functionally and technically.

4. Responsibilities of a Quality Architect

The proposed approach is to have a quality architect with the following responsibilities:

1. **Quality document creation:** The Quality design document needs to be created by deriving it from architecture documents and development- design documents. The number of QA architecture/design documents should be equal to the number of development documents. There should be a one to one map between the number of Development and QA design documents.

2. **Focus:** The focus of quality design documents should be on possible automation areas, required Build Verification Test (BVT) cases, Integration tests, Performance tests and User Interface tests, at a high level. Basically the QA design document will capture all sorts of test plans with granular details. The test plan will include coverage on system testing, black box testing, white box testing and BVT. When the document is complete, QA will have a fair idea about the length and breadth of tests the project will need to have.

3. **System testing:** The quality design document will give a fair idea about how tests are spread across multiple sub-systems. This knowledge is helpful to perform system testing effectively.

4. **Education:** The Quality architect needs to educate the team on how does development designs translate into QA focus areas in terms of Complexity, Inter dependency, Number of lines of code, Distributed coverage across front end (User Interface), Middle tier (Java, .NET etc.) and backend (database).

5. **Organized test plans:** The QA architect will help the QA team to execute their test plans in an organized way and in a timely manner.

6. **Synchronized design:** The QA architect should ensure that the QA plan is well in sync with development designs.

7. **Capturing actions items:** The QA architect should map all non-functional requirements to QA centric action items.

8. **Guidelines to Test engineers:** The QA architect should give pointers to QA engineers who will estimate individual user stories. The pointers can be on complexity, number of function points, number of interfaces etc.

5. System Design Document versus Quality Design Document

The system design document defines the architecture, component, modules, interfaces and data for a system to satisfy specific requirements. The proposed QA design document will be an extended version of the system design document with QA perspective added to each section. Some of the key highlights of the quality design document are below.

1. **Call out dependencies:** All use cases have a dependency section if applicable. Those dependency sections will denote how the module is to be approached.

2. **Define complexities:** The complexity of each component in the document is assigned with weightage number and the stories are categorized according to their assigned weightage. The other way to categorize the stories is to categorize them with complexity levels such as low, medium and high. Based on the weightage or complexity levels, QA planning needs to be done accordingly. If the weightage is already assigned by the development team, then QA can follow the same weightage numbers. It is always recommended that QA revalidates the complexities before rolling out their test plans.

3. **Test optimization:** Test cases (both manual and automation) are optimized to a greater extent. When there is reusable code that is serving multiple modules, automating one of them would pass all the tests. There is no need to write multiple automation tests for each user story. Optimization of test resources results in significant cost savings.

4. **Technical knowledge:** QA gains insight into deeper system design by working against low level design. Since each user story design connects with a distributed set of components (Front end, Middle tier and Backend), test ownership gets assigned breadthwise. The quality design document captures the details that are not captured by the system design document. For example, if a system design document that describes the functionality of use case “Registration”, it will capture how the request is initiated, how the business component processes the request and how the response is obtained. The quality design document will have the same details with additional test information. The additional points captured include the quality impact in terms of possible UI tests, business component tests, response tests, non-functional tests, automation opportunities, integrations tests etc.

5. **Cost savings:** When SDETs/ test engineers log defects around design, the benefit that the defect brings is much higher than the ones caught in later cycles. One uncaught design defect may have the potential of introducing multiple functional defects across the system during later cycles of the project.

   For example, if a developer has designed the following function to sum two numbers,

   public int sum (int a,int b)

   QA can file a defect asking “how can this function support more than two numbers”.

6. **Well-defined boundaries:** System boundaries are well-defined. Since QA's design document covers all parts of the system, QA knows what tests make the system 100% covered. System tests are derived from integration-interface designs. Hidden parameters like “network throughput”, “connectivity”, ”system load” are very well known in advance.

7. **Collaboration:** Tight collaboration between development and QA: Both development and QA units become interlocked as both know the system design equally well. QA can also push for critical functions to get prioritized so that there are fewer surprises at later cycles.

8. **Automation Plan:** The quality design document gives a clear picture, in the design phase, of what can (and can’t) be automated. A number of redundant automation cases can be eliminated when QA knows the design well. In time-crunched projects, QA can automate only the necessary or important cases rather trying to automate the whole number of test cases.

6. QA design diagram: Illustrated from a use case

QA design diagrams are part of the quality design document. Development teams may use different tools and techniques to prepare their design. It is the QA architect’s responsibility to understand each representation (like block diagram, use case diagram, sequential diagram, and class diagram) and to come up with a QA equivalent design.

For example, if developer uses a class diagram, QA should understand the following to extract QA designs.

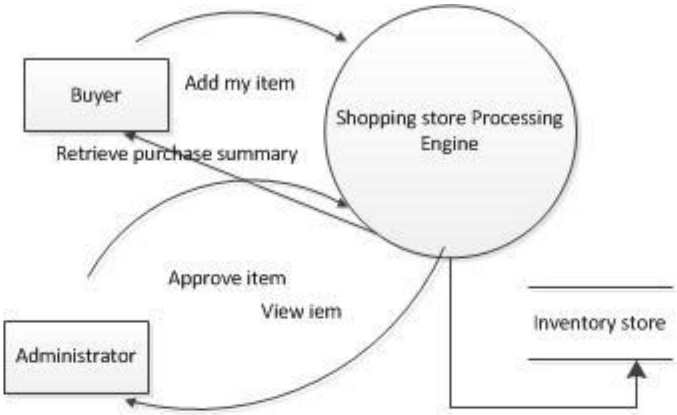
1. Number of public members
2. Number of public APIs
3. External dependencies for the class
4. Relationship with other classes (one to many, many to one, many to many)

5. Possible number of automatable APIs
6. Possible number of critical APIs that has many to one relationship

As an example, let us consider a shopping cart application.

At a high level, the activity diagram looks as below.

1. Buyer adds his item.
2. Buyer retrieves his purchase summary.
3. Administrator views the item ordered and approves.
4. The shopping processing engine or business component services the above 3 activities by persisting the user(s) data into inventory store.



Development design

1. Buyer object sends the shopping information to shopping cart.
2. The shopping store business component identifies the buyer by his session id and adds buyer's shopping data to inventory store.
3. Admin object requests for shopping info added by buyers.
4. The shopping store business component validates the admin's id and retrieves purchase requests from inventory store.
5. The shopping store Business component constructs purchase object and sends it back to admin.
6. Admin object views the info in the front end (view JSP) and clicks on approve
7. The approve request is submitted to shopping store business component.
8. The shopping store Business component commits the order.
9. End.

QA design mapped from the development design

1. Make sure unit tests are written for buyer object in order to check whether the object contains valid values like session id and shopping parameters. The unit tests can cover with positive values, negative values, lower boundary value, upper boundary value, empty value etc.
2. Write a test API to check whether the business component commits the data into data store successfully.
3. Check the maximum number of requests that the business component can handle (nonfunctional Performance test).
4. Check the correctness of admin id and its corresponding request object.
5. Validate the validation logic by exercising with a number of unit test values.

6. Write unit tests for checking the completeness of the purchase object.
7. Ensure all fields in the UI are validated.
8. Write a test API to check whether data submission was successful.
9. Check relevant tables in the inventory store and make sure the commit operation happened End.

Deriving non functional tests from QA design

1. Interoperability tests covered between Buyer object, Business component and admin object
2. System availability, durability and maintainability test cases.
3. Load test, performance test coverage.
4. System test boundaries: API test, serviceability of the component in standalone and distributed environments.

7. Conclusion

It has been discussed what is lacking in the traditional QA process. The pain points were analyzed in detail. The merits of QA involvement in the design phase were discussed in detail.

In order to overcome the existing challenges, it is important that QA should address the items that have been lacking in the design phase. Organizations should consider having a QA architect to handle the design aspects of the quality process.

Some of the key learnings from this paper are:

1. QA should approach each story based on its criticality, complexity, and interdependencies.
2. QA should prepare an architecture document capturing all important design decisions that can influence their testing.
3. The designated QA architect should guide his team to derive tests from designs.
4. Involving QA in the design phase results in well controlled QA plans with significant cost reduction. It also allows creating extensible QA designs.
5. QA gains adequate system knowledge when they become co-owners of designs.

8. References

1. Wikipedia, “Systems Design”, https://en.wikipedia.org/wiki/Systems_design (accessed August 01, 2013)

High Availability Testing – Stories from Enterprise and Consumer Services

Srinivas Samprathi
snsrinivas@rocketmail.com

Abstract

Quality of a cloud service is more than having a defect-free service which meets the requirements. If the product or service has downtime or is slow in recovering from faults, it will directly impact the product or service adoption and customer satisfaction.

High availability is often an implied need; but customers expect it to be there and its absence will impact the business. With global users and mobile devices using cloud services, testing for high availability, also known as fault tolerance, becomes much more important.

One useful measurement to gauge service availability is customer feedback and the customer perceived downtime. If the user feels the cloud service is erroneous or has frequent downtime, it does not matter what process or tools are used for high availability, the customer is always right.

This document explains high availability testing with fault model techniques that can be used to improve the availability of a cloud service (enterprise or consumer) and to plan for the right thing, and get the right things for the right user at the right time – every time.

Biography

Srinivas Samprathi is a Senior Test Manager at Microsoft, working in the Windows Application Experience team.

He is passionate about inventing engineering processes to help manage and deliver high quality software and services. His passion is engineering excellence and he enjoys coaching engineers, managers and leaders to build high potential teams.

For the last 15 years, Srinivas has managed and mentored engineers in the software industry. He frequently drives projects on software engineering, project management, talent development, and quality assurance testing.

At Microsoft he was actively involved in developing innovative engineering methodologies and has led the implementation of fault modeling and high availability testing for the Microsoft Office 365 Platform. He has been awarded the Leadership Award, Microsoft Gold Star Awards, Quality of Service award, etc. He has managed the quality for products like Windows, Microsoft Sharedview, Microsoft BPOS, Microsoft Office 365, Microsoft Windows 8 Metro apps, and Amazon Kindle Content Services.

1 Introduction

Conventionally, software testing has aimed at verifying functionality but the testing paradigm has changed for software services. Developing a full-featured and functioning software service is necessary; however service real time availability is equally important. Service downtime will increase customers' negative perception towards service quality and reliability. Services can be consumer services such as Kindle Services or corporate services such as Office365 messaging and sharepoint. A service engineering team's goal is to understand the common faults associated with user scenarios. A service is used by novices and experts. Therefore, the service needs to provide high availability for all the customers.

High availability testing is essential to ensure service reliability and increased fault tolerance. Fault tolerance is defined as an ability of a system to respond gracefully to an unexpected hardware or software failure of some of its components. High availability testing can be used to prevent outright failures of the online cloud service and to ensure continuous operation, therefore increasing fault tolerance and reliability. High availability testing does not necessarily focus on preventing faults, but ensures designing for tolerance, recoverability and reduced occurrence of high severity faults. This testing measures business impact from faults and helps in planning for reducing impact.

A theoretical understanding of how the service or system in test behaves under possible faults is a prerequisite to high availability testing. This paper starts with a discussion of stories from both the enterprise and consumer services. With such a reference model, the paper delves into providing a systematic method to model faults. The fault modeling section introduces key concepts of faults, categories of faults, and the component diagram along with the fault modeling itself. Once the system has a fault model, the next step is to test for fault tolerance. Finally, the paper summarizes the results seen with the Microsoft Office365 platform by following fault modeling and testing methodologies outlined in this paper. The results section highlights the savings achieved from reduced operational effort costs.

2 Discussion – Stories from Enterprise and Consumer Services

This section calls out some patterns of availability issues experienced in service teams. These patterns imply the need for a paradigm shift in designing and testing software services for high availability. Below mentioned stories call out the need for a systematic design and testing approach for achieving high availability. There is no silver bullet to the challenge of high availability; one can build a cloud service which meets the availability expectations of users by planning it for fault tolerance, managing it with the right set of monitoring, recoverability metrics and leveraging fault model techniques.

2.1 Fault Tolerance with High Volume and Deep Dependency Levels

Amazon Kindle Service endpoints receive millions of calls in a day which in turn can lead to several million outgoing calls (average ratio of 1:12) to underlying services (for a peak service call rate of over 1000 requests per second). Faults are guaranteed with these many variables in the system. Even if every dependency service has an uptime of say 99.9%, if a service is dependent on 36 other services, each with 99.9% uptime, then the service will have less than 99.9% uptime due to the chain of dependent services' downtime. When a single dependent service fails at high volume, it can rapidly lead to abnormally increased request queues to or from dependent services (Christensen Ben, Netflix TechBlog). Similar high availability issues can be addressed via systematic analysis of faults. The next section mentions, an instance of how much monitoring is good.

2.2 Is More Monitoring Good for High Availability?

A service received an alert indicating a failure in one instance. Within the next 12 minutes, there were 12 alerts which looked similar but affecting all instances of that service in all datacenters. Since such monitors run separately against each datacenter, 4 alerts were triggered for those failures, for a total of

16 failure alerts so far. The first step was to root cause the issue and understand the customer impact. It was sufficient to look at one of the failures, as all alerts were for the same failure. Redundant alerts wasted valuable investigation time that could have been spent on failure mitigation. The cause of this issue was erroneous entry of service configuration values in the production environment.

There are two advantages to reducing alert duplication. Firstly, it increases the signal to the noise ratio of alerts. Secondly, it prevents ignoring alerts with the assumption that alerts are duplicates. The only drawback of reducing alert duplication is missing genuine alerts during the reduction process. The next section highlights with an example, the pros and cons of production environment developer access.

2.3 How Much Developer Access to Production Environment is Fail-Safe?

On 2012 Christmas Eve, Amazon Web services experienced an outage at its Northern Virginia data center. Amazon spokesmen blamed the outage "on a developer who accidentally deleted some key data. Amazon said the disruption affected its Elastic Load Balancing Service, which distributes incoming data from applications to be handled by different computing hardware." Normally a developer has one-time access to run a process (Babcock, 2012).

The developer in question had a more persistent level of access, which lately Amazon revoked to make each case subject to administrative approval, controlled by a test team signoff and change management process. The advantage of this approach is the reduced business impact from a controlled review of changes made to production environments. The drawback of the change management process is the additional delay in implementing changes in production environments.

2.4 How to Recognize and Avoid Single Point of Failures?

A web service, *WS1,* for enterprise customers sends data from enterprise systems to cloud subscription. This web service *WS1* depends on a 3rd party Identity service such as Windows Live for authenticating enterprise users. There is one call in the Windows Live web service that the developer knows about. That call depends on the Windows Live auditing database which is prone to single point of failure. Another WS2 call also provides same functionality without the single point of failure. WS1 call was used as the developer knew about it. When the third party Windows Live Service auditing database was brought down for maintenance, the WS1 call resulted in high impact failures. Therefore, there is a need to understand the impact to availability due to each dependent service, even though the dependent service is a third party service. In this case, change the design to invoke the service call WS2 that is resilient.

Another example of single point of failure: The servers have 2 Top of Rack (*ToR*) switches. If one switch fails, then the other switch takes over. This is a good N+1 redundancy plan but is it sufficient?

If a rack was supported by only one switch (which had a hardware fault) and the redundant switch was in maintenance, then this can lead to single point of failure. Therefore, all the hosts serving an externally facing web service endpoint were arranged on that single rack. It is a high impact single point of failure to have all the hosts that serve an externally facing web service endpoint to be placed on a single rack.

Questions to ask: Are one of these switches regularly taken down for maintenance? If yes, how long on average? Do we have to know where all hosts serving the web service *WS1* implementation are arranged in datacenter? – Yes, absolutely to the extent of knowing there is no single point of failure.

2.5 How Important is Special Case Testing?

Testing the behavior of service on February 29 (leap year day) or on December 31st (the last day of the year) is one example of date-time special case test. On February 29, an Azure protection process inadvertently spread the Leap day bug to more servers and eventually, 1,000-unit clusters. The impact was downtime of 8-10 hours for Azure data center users in Dublin, Ireland, Chicago, and San Antonio. As Feb. 28 rolled over to Leap Day at midnight in Dublin (and 4 p.m. Pacific time February 28), a security certificate date-setting mechanism (in an agent assigned to a customer workload by Azure) tried to set an expiration date 365 days later on February 29, 2013, a date that does not exist. The lack of a valid expiration date caused the security certificate to fail to be recognized by a server's Host Agent, and that in turn caused the virtual machine to stall in its initialization phase. This story indicates the need for system-wide fault safe special case testing. In this case, it was important to test how the entire system behaves on special date-time values (Babcock, 2013).

3 Fault Modeling

Fault Modeling is a service engineering model to capture the behavior of the system against faults. In order to understand fault modeling, it is necessary to agree on fault definition and the goals of fault modeling. Faults can be a single non-desired state of a component, entire system, service or group of components (Hardware or Software). Faults can be represented by transitions or states in addition to being represented as a single state. Such a representation is possible irrespective of the type of faults or nature of faults (permanent, transient or intermittent).

The goals of fault modeling are to:

- Identify fault domains (machine boundary, process boundary, datacenter boundary) of each component that is part of service implementation.
- Identify the single and double point of failures in the entire system.
- Derive a comprehensive list of possible faults, business priority and recovery from faults.
- Prepare a recoverability and redundancy model for deployment and maintenance activities.

3.1 Team member roles in Fault Modeling and Testing

Table 1 describes the typical roles and responsibilities played by team members in high availability testing and fault modeling:

Table 1: Roles and Responsibilities

Typical roles	**Responsibilities**
Component testers	Work with component developer to create a fault model for component(s). Define and document high availability test cases. Test component's fault tolerance and follow through to fix issues. Test faults with dependent components. Participate in the system level test pass. Automate fault tolerance test cases to enable ongoing faster execution.
Component feature teams	Contribute to component or role fault modeling. Review the fault tolerance component test plan(s).
Fault tolerance Subject Matter Experts (SMEs)	Defines and documents technical templates for fault modeling and test plans. Generally, members from each discipline (Testing, Developer, and Program Management) are preferred. Train rest of the team about designing and testing for fault tolerance. Handles Scheduling and Status updates. Works with other fault tolerance subject matter experts (SMEs) in the company, and outside the company to learn principles and best practices. Aggressively follows through downtime caused by faults in the service. Completes Root Cause Analysis (RCA) on an ongoing basis.
Operations/hosting team(s); All datacenter infrastructure teams	Collaborates with Component and System level fault tolerance testers to Understand what kind of faults (high customer SLA impact) can occur in the service or system. Participates in executing high availability test cases. Execute recovery steps (where recovery is manual); File bug reports if recovery steps are unclear or recovery fails. During test passes, works with the engineering team component tester (s), developer(s) to execute, and clarify recovery steps when recovery fails. During the execution of high availability test cases, works with component

	tester (s) and developer(s) to recover from the component or system level faults in a time-bound manner.
Monitoring SMEs	Investigate and propose solutions needed for fixing holes where faults were not monitored and alerted. Ensure all non-automated recovery faults have documented Trouble Shooting Guides (TSGs) (These guides will be verified by fault tolerance SMEs; executed and signed off by Operations team SMEs during pre-production or live production testing).
Higher management	Approves and sponsors fault tolerance activities. Provides guidance to improve and implement design changes.

3.2 How is The Faults Modeled for Each Component (Software Component, Infrastructure or Hardware) of a Service?

For each fault category, identify and model:

- Fault definition: Define the fault (described in the above section).
- Customer Attributes:
  - Severity of customer impact can be high or low financial impact.
  - Scope of impact can be all users or one user.
- Historical data or measured attributes:
  - Probability of fault occurrence: Is the probability High or Low?
  - Time to Recover.
  - Time to Repair.
- Monitoring:
  - Does monitoring exist for a specific fault?
  - If monitor exists, how soon is impact notified by the monitor?
- Recovery:
  - Document Manual or Automated recovery expected.
  - Document Standard Operating Procedures (SOP) or Trouble Shooting Guides (TSG) for manual recovery.

To achieve a fault model for a service component, begin with the service component diagram as shown in Figure 1. This component diagram depicts a single active service instance. This service has high write load and is designed to be in the same datacenter as the active storage units. This service interacts with four layers which are the storage service, business logic workflows, simple mail transfer service (SMTP), and remote power shell. Windows live provisioning for user data happens through business logic workflows. The service authenticates clients through windows live tokens. The reads, writes, and searches are done directly through storage service API. The service sends emails for a batch of data using the SMTP server located in the datacenter. All datacenters have the exact same configuration. GSLB handles Global Server Load Balancing.

A component diagram is a diagrammatic representation of the component and it’s dependencies as it will operate inside the service offering’s datacenters. The important thing to capture with component diagrams is dependencies, especially out of process dependencies, so that when something is not working as expected, the escalation on-call engineer could look at a component diagram and say, “well, the service needs to talk to Windows Live in order to work.” Enumerate dependencies to one level. Include datacenter boundaries and load balancing components in the component diagram. The diagrams identify the easy and most common interfaces that go faulty and the dependency faults. This makes a good first set of faults – what happens when dependencies fail, or perform poorly?

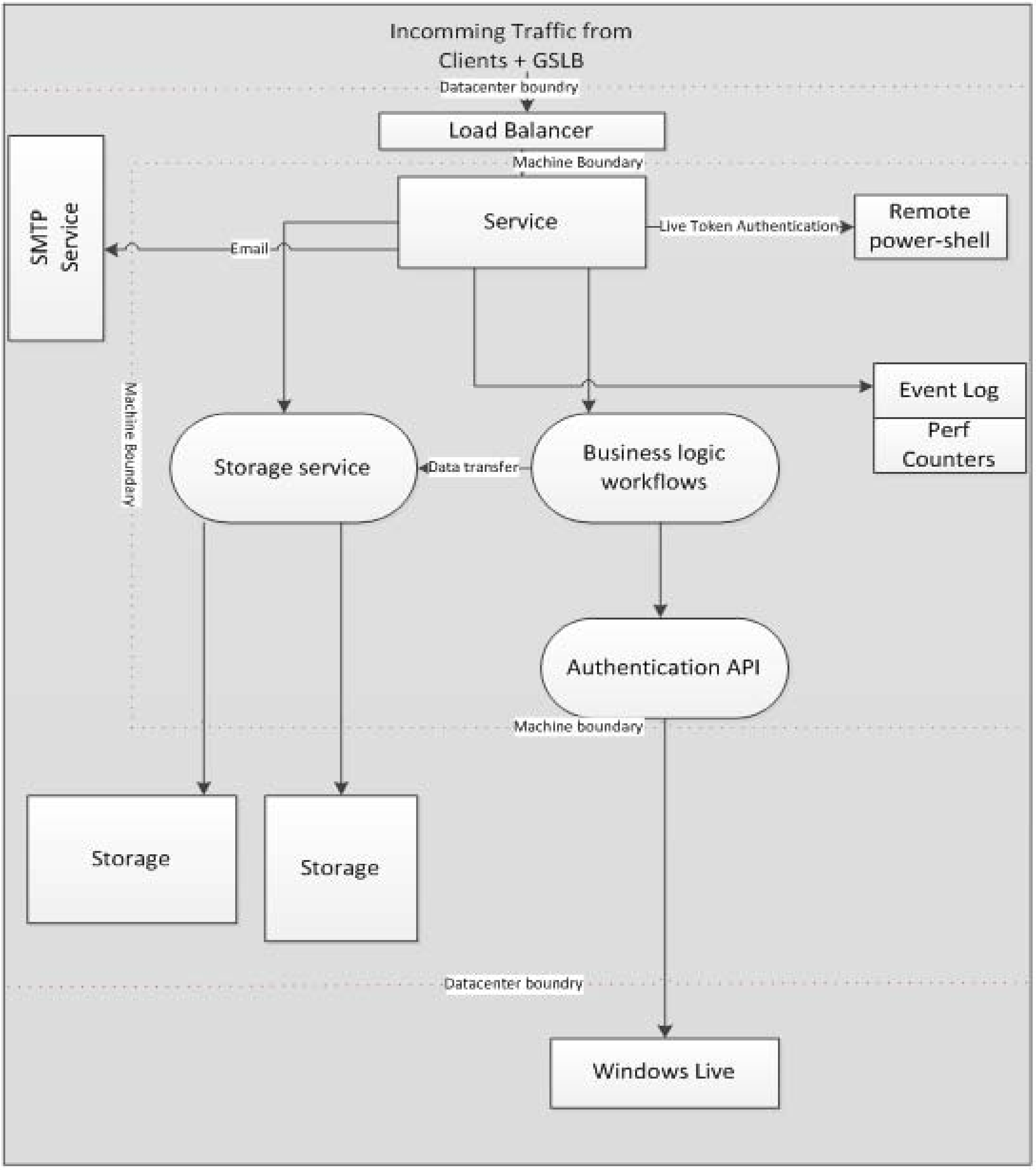


Figure 1: Example component diagram

3.3 Fault Categories with Examples

3.3.1 Within Machine Boundary

This category includes faults that can occur within a service component's physical or virtual machine boundary. This is not an exhaustive list but an illustration of fault types in this category.

Web service application pool crashes are common for service endpoints hosted in IIS (Internet Information Services) application pools. It is important to detect application-pool crashes configure application-pools to be self-recoverable.

Resource starvation such as CPU hog, memory starvation, and I/O hog are a common set of faults that can occur within a machine boundary. Key aspects of these faults are to keep them machine boundary specific and not cascade to all instances of a service. Also, to design for monitoring and recovery of service instances from resource starvation faults.

Certificate faults are another set of faults that can take reliable service operations by surprise. These faults can vary from certificate servers being single failure points or forgetting to renew key authentication certificates, e.g. Virtual IP certificates. It is crucial to have an infrastructure in place that monitors certificate expiration and acts upon them.

Excessive non-circular service logging (log files not cleared automatically) is a common fault seen in bigger service systems spanning hundreds of services. Ensure service logs are circular and regularly archived to a query able fault redundant database.

A single physical or virtual machine representing a single failure point for the service is an obvious fault that can easily cascade into dependent service failures as well. Such faults can be avoided by building in redundancy of virtual or physical machines.

3.3.2 Across Machine Boundary

This category includes faults that occur across the machine boundary of the same component or different components. This is by no means an exhaustive list but an illustration of fault types in this category. Dependent service high latency or unavailability is a high frequency fault in today's service offerings such as Kindle Services or Enterprise Office365. Only one instance of a service behind a Load balancer or Virtual endpoint is expected from services. But there are fault situations where this can happen. One such situation is when there are 'n' instances of a service representing a virtual endpoint. However, the certificates for all but one host of the service expired on the same day causing a single point of failure. Such issues can be mitigated by detecting dependent service failures and mitigating.

3.3.3 Infrastructure and Site-Level Faults

This category includes faults that happen in a datacenter network or in network infrastructure across datacenters. This is not an exhaustive list but an illustration of fault types in this category. High network latency in datacenter is a high impact and a low frequency fault. With Global Server Load Balancing (GSLB), internet traffic can be distributed among different data-centers located at different locations of the planet. This technology is highly efficient in avoiding the impact from local datacenter downtimes. GSLB failures or mis-configurations can cause downtime for end users and can be a high frequency fault if configurations are modified often. Cross-datacenter replication unavailability or high replication latency is often a high business impact fault and needs to be properly monitored.

A network infrastructure single point of failure such as a single primary Domain Name Service (primary DNS) with no updated warm secondary Domain Name Service (secondary DNS) can be an unrecoverable high business impact fault. These failures can be mitigated by monitoring updates to secondary instances, such that the secondary instances can replace the primary instances at any time.

3.3.4 Operational Faults

This category includes faults that happen in operational and hardware infrastructure in datacenters. This is not an exhaustive list but an illustration of fault types in this category. (Barroso Luiz André and Hölzle Urs). Bad rack allocation such as physically allocating all service instance machines on the same rack will make that service behavior prone to rack failures. Network rack switch failures with no redundant switch can cause entire racks to be down for days while repair and maintenance on switches is scheduled. Load balancer failures are expected to be low frequency but load balancer traffic management scripts (iRules) can change often and cause unhandled faults. Ensure testing of hardware configuration changes in a staging environment prior to production deployments.

Lack of datacenter redundancy indicates a cross datacenter operational fault and needs to be fixed to avoid physical datacenter or geographical disaster failures. Datacenter expansions or expansions of new services within a datacenter can also be faulty. These faults can be reduced by adding monitoring and testing performance pressure from expansion on the rest of the existing system. Faults in monitoring infrastructure itself can be critical. Service monitoring must catch downtime issues before external customers report issues. It is critical to ensure monitoring infrastructure itself is stable.

3.4 Real-Time Monitoring of Customer Impact

In addition to fault modeling, it is necessary to model customer scenarios and automate execution of these scenarios as real-time monitors in production environments. This helps to identify immediately what customer scenarios are affected by fault(s) occurring in the system rather than waiting for customers to report issues. These automated monitoring modules can be referred to as Customer Scenario Synthetic Transaction Monitoring.

4 How to Test for High Availability?

Testing for high availability starts with the system fault model as a reference. From the fault model, it is essential to design and execute high availability test cases.

Steps in designing high availability test cases include, but are not restricted to:

- Converting faults derived from fault modeling to actionable test cases.
- Categorizing test cases into:
    - Component level fault test cases which can be within machine boundary or across machine boundary.
    - System level impacting fault test cases.
    - Third party service or cross datacenter fault test cases.
    - Infrastructure and operational level fault test cases.
- Prioritize fault test cases according to customer perceived impact.

Some of the execution steps for a typical high availability test case are:

- Inject fault. Fault injection is the test technique to introduce faults to execute and test error handling and other uncommon code paths in a component.
- Verify the monitoring system raised expected alert(s) to indicate the fault has occurred.
- Verify that no noisy (unexpected or too many) alerts are raised.
- Verify ‘Customer Scenario Synthetic Transaction Monitoring’ behavior shows expected failure. See section 3.4.
- Verify expected component or system behavior during or after fault has been induced
    - Test cases to execute during or after the fault injection.
    - Component or Role specific impact.
    - System level impact.
- Note time to recover (manual or automated recovery).
- If manual recovery: Execute documented manual recovery (Standard Recovery Procedure) steps.
- If Standard Recovery Procedure does not exist yet, then track a task item to create one.
- Execute test cases after recovery (automated or manual) to verify normal expected functionality
- Verify ‘Customer Scenario Synthetic Transaction Monitoring’ behavior shows expected success.

Documenting and executing a high availability test case in the above manner also facilitates modular test automation of each test case and easy manual execution.

4.1 Testing Environments and Testing Frequency

Table 2 explains which testing environment should be used for testing each fault category. This table also suggests the typical testing frequency for each fault category. For example, infrastructure fault cases must be tested in production as pre-production can be different or low scaled compared to actual production environment. However, the simplest case of within machine boundary faults can be simulated and verified in single test machines.

Table 2: Fault testing environments

Fault type	Testing environment	Testing frequency
Within machine boundary faults	Test in a single test machine environment (commonly referred to as one-box testing).	Automate tests and execute high impact cases when a service component is modified.
Across machine boundary fault cases	Test in pre-production and production environments. Focus on testing high impact scenarios.	Schedule monthly or quarterly, based on the business need and change frequency.
Infrastructure fault test cases	Test in Production environments. Focus on testing high impact scenarios.	Schedule monthly or quarterly, based on the business need and change frequency.
Operational fault test cases	Test in production environments. Focus testing the high impact scenarios.	Combine testing with planned operational maintenance or deployment or expansions.
Third party service or cross datacenter fault test cases	Test in pre-production and production environments. Focus testing the high impact scenarios.	Schedule after major updates are deployed to individual data centers. Schedule during and after expansion of data centers.

5 Why Test for High Availability in Production?

Because pre-production environments, which are scaled down versions of production environments, often lag behind on infrastructure upgrades and maintenance compared to production. This causes pre-production environments to be different from production. Since faults (software, firmware and hardware) can be environment specific, testing in pre-production environments alone is not sufficient. It is also important to test regularly in production as production environments have regular deployments, expansions for scaling, new datacenters added, and more database hardware added.

Some examples of issues found by testing in production environments are:

- Secondary Active Directory Integrated DNS (INS) server not in sync with Primary Active Directory Integrated DNS (INS) server; therefore Primary INS is a single point of failure.
- Some key role hosts in production may not have monitoring infrastructure client installed therefore these hosts or their service failures were not being monitored.
- Replication servers (Active Directory bridgehead servers) are misconfigured in production and there was no monitoring in production to verify replication status.

6 Results and Conclusions

Adopting fault modeling and testing method in Microsoft Office365 platform resulted in an estimated savings of 600 work days per year. This savings was achieved in alert escalation, the manual operational cost to resolve issues, and cost of communicating impact to enterprise customers. The savings comparison is attained by comparing the first version of Office365 called ‘Microsoft Online Services’ (MOS) (which did not use a fault modeling and testing approach) with the Office365 platform (O365).

Estimated savings was determined using all of the below data:

- The number of fault recovery and fault identification issues found pre-release.
- Historical data from versions of Office365:
    a) frequency of severity 1 faults as illustrated in Figure 2, Chart 1
    b) time to resolve customer issues as illustrated in Figure 2, Chart3.
- The cost of fault modeling, high availability testing, and redesign for handling recovery.
- Team’s increased understanding of the single and double point of failures and designing for resiliency as illustrated by Figure 2, Chart 2.

- Realization that software or hardware faults are not necessarily prevented, but proper design, fault modeling and testing can ensure resiliency.
- Paradigm shift and the progress with:
    - Testing availability of cloud services.
    - Integrating high availability testing with monitoring testing.
    - Including different teams in availability and monitoring testing.
    - Testing regularly for availability in different environments, including production.

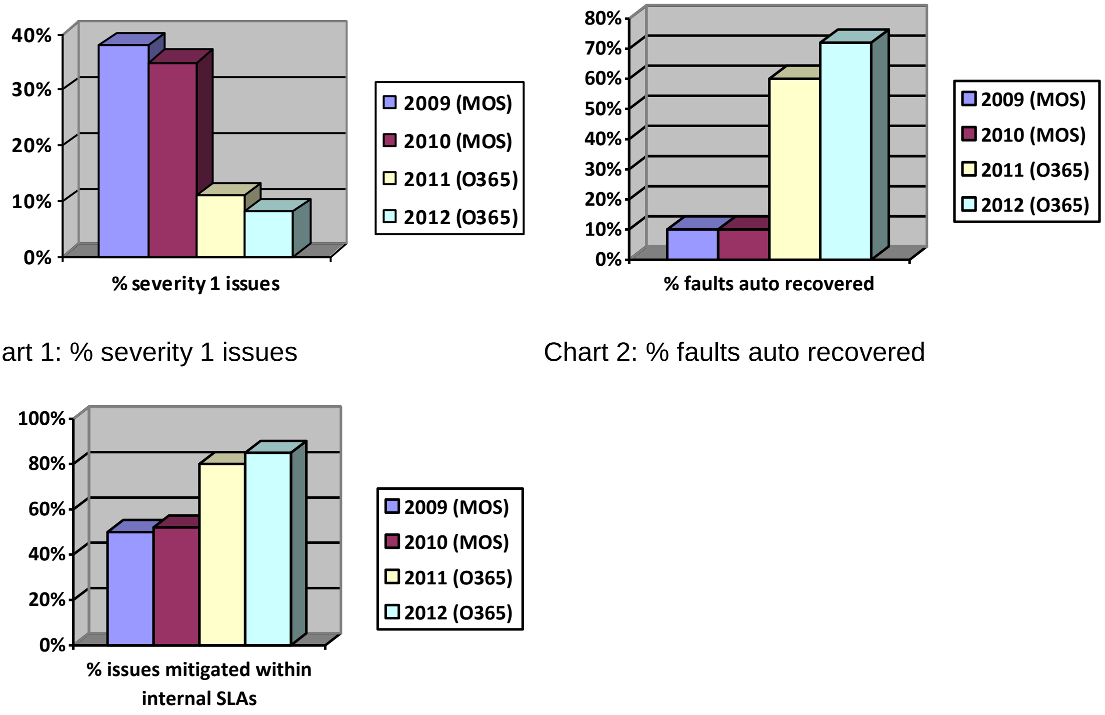


Chart 1: % severity 1 issues

Chart 2: % faults auto recovered

Chart 3: % issues mitigated within SLAs

Figure 2. Historical data from versions of Office365

From above mentioned results and charts, it can be derived that adopting fault model and testing methodology contributed to reducing high severity alerts by at least 65%, which is a statistically significant result. It can also be seen from the charts that the focus on auto recovery and testing manual recovery steps helped reduce mitigation times in 50% of issues compared to when fault model and testing methodology was not utilized.

References

- Christensen Ben, Netflix TechBlog, available http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html
- Babcock Charles, “Amazon's Dec. 24th Outage: A Closer Look”, Information Week, January 2013, available http://www.informationweek.com/cloud-computing/infrastructure/amazons-dec-24th-outage-a-closer-look/240145533
- Babcock Charles, “Azure Outage Caused By False Failure Alarms: Microsoft”, Information Week, March 2012, available http://www.informationweek.com/cloud-computing/infrastructure/azure-outage-caused-by-false-failure-ala/232602382
- Barroso, Luiz André and Hölzle, Urs, “The Datacenter as a Computer”. Examples of operational faults, frequency are available from: http://blog.s135.com/book/google/dc.pdf Chapter 4.

Are You In It For the Long Haul?

Brian Rogers

brian.rogers@microsoft.com

Abstract

Unit tests, regression suites, and end-to-end scenarios are invaluable in establishing a baseline measurement of software quality. They are also wholly insufficient for determining whether a product is truly ready to ship. To aid in this decision, we must regularly collect data on how well a system stands up to longer periods of use (and abuse). Long-haul testing is a methodology uniquely suited for this task.

This paper demystifies long-haul testing and shows how this valuable but often poorly understood technique can pay dividends if appropriately defined and implemented. Learn the principles and practices for building effective long-haul tests to simulate production usage patterns, leverage controlled chaos, and achieve high coverage while maintaining reasonable cost.

With ever-increasing user expectations around high availability and reliability, it is now more important than ever to adopt and embrace long-haul testing for your project. So are you satisfied with short-term quality indicators, or are you in it for the long haul?

Biography

Brian Rogers has been a software tester for over a decade. In his current role as a principal software development engineer in test in the Windows Azure group at Microsoft, Brian designs and develops tools and automated tests for large-scale distributed systems. Brian has a B.S. in Computer Engineering from the University of Washington.

1. Introduction

The software industry is now enlightened enough that it is no longer a question of *if* you test but rather *what* and *how* you test. Developers are test-infected[1] and unit testing is on the rise[2]. Test organizations at companies like Microsoft[3] and Google[4] boast a wealth of automated functional and integration suites which run against every build and protect against regressions. Such testing practices are generally treated like oxygen – essential for survival – and few would argue against their necessity in establishing a baseline software quality measurement.

However, today's software runs on devices like tablets and smartphones that may go for weeks without a restart. Cloud services and web applications operate in always-on mode and despite the unreliable nature of the underlying infrastructure, user expectations around availability and reliability are high[5]. Using simple test cases to exercise the software for mere seconds or minutes at a time is clearly not enough to determine if a product is ready to ship. A comprehensive software testing effort must evaluate longer periods of use with adequate coverage of faults and error paths.

When faced with this challenge, many test organizations turn to the relatively well-known practice of **stress testing**. While useful, stress by definition is meant to cover narrow, extreme situations. As it turns out, the lesser-known but related methodology of **long-haul testing** is a better choice for evaluating a broader set of typical application workloads.

2. Defining "long-haul"

Stress, performance, long-haul, and a variety of related non-functional test types can all be grouped under the blanket term **load testing**. The exact meaning of load testing differs slightly depending on the source but I use a working definition of "observing and evaluating a system at varying usage levels." Given the amount of confusion and conflation that exists with these terms, it is instructive to clearly delineate stress and long-haul testing to compare and contrast the two practices.

2.1 Stress testing

Stress testing is a specialization of load testing and is defined by IEEE as "testing conducted to evaluate a system or component at or beyond the limits of its specified requirements."[6] As such, failure of some sort is essentially the expected behavior in a stress test. Various subcategories of stress testing exist such as soak testing and spike testing; the main difference among all these test types is the way load is applied to the system under test.

In stress testing, the goal is to determine if the system degrades gracefully after hitting a breaking point and perhaps most importantly, if it *recovers* after restoring functionality or reducing load. For example, a stress test may intentionally overwhelm a server with so much user traffic that the system stops responding in a timely manner. After reducing the traffic, the system should bounce back and serve requests just as fast as before the stressful condition was introduced.

A good stress test is repeatable and sufficiently constrained in coverage such that it can reliably reach a known load target while exercising a small set of components. These qualities allow the test to establish a baseline, thereby providing data on unexpected changes in the defined operational limits of the software under test.

2.2 Long-haul testing

Long-haul testing seems to have no industry-standardized definition and only a handful of online references even mention it. (Arguably, the best overview is a write-up of how the Windows® CE team used long-haul testing[7], wherein the author notably distinguishes the approach from stress testing.) I describe it as “observing the behavior of a system when given varying workloads over a long duration.” A useful long-haul test runs for a minimum of several hours while subjecting the system to a variety of faults and actions (the “workloads”), typically concurrently. The workloads should bear some resemblance to realistic use cases, although they may be weighted to include more faults than typically seen in production to improve error path coverage.

Unlike stress, long-haul testing does *not* attempt to exceed known system limits; it has more in common with accelerated-life testing, where time and scale are compressed to allow more activity in a given interval than what is otherwise possible in a real-life, real-time scenario. A long-haul test passes if it observes no fatal errors (e.g. total system failure, data corruption) and if the system remains operational in the presence of any isolated or incidental faults, subject to previously determined service level agreements (SLAs).

The degree of concurrency, wider scope, and longer durations typical of long-haul tests make them at best semi-predictable (i.e. not *completely* random). Although it sounds less than ideal, this is in fact a benefit; given enough time to run, long-haul tests should uncover a broad range of unanticipated operational issues. As a consequence – and in sharp contrast to more narrowly focused stress tests – long-haul testing does *not* guarantee detection of specific regressions. (In some cases where more repeatability is desired, it is possible to use predetermined random number seeds or more deterministic workloads but these should be used with care to retain the overall benefits of long-haul.)

3. Why do we need long-haul testing?

By their nature, unit, functional, and other similar tests are **short-term quality indicators**. They provide quick feedback on whether a single component or small set of features works as intended when exercised in a reasonably controlled fashion – important and necessary, to be sure, but not *sufficient*. The fact is that real usage patterns for an application rarely involve a single walk through one or two features individually, but rather a diverse set of multi-step actions executed occasionally over hours or even days. For multi-user applications the interactions are even more complex, especially when taking into account the ways in which actions by distinct users (even when separated by long time intervals) can have combined or conflicting effects.

Classic integration testing techniques can help to gain *coverage* of these more complex scenarios but planning, writing, and executing individual test cases to exercise all such interactions is normally very difficult and cost-prohibitive. Add to this the need to cover the usual error paths – for cloud services, consider the annoying but unavoidable cases of network interruptions and backend server outages, to name a few – and the difficulty further increases. In fact, it is even more challenging since we must do all of this *repeatedly* for extended periods of time to move beyond the short-term and assess the longer-term operational behavior of the system.

Instead, we need a testing methodology that relies less on costly pre-planned workflows and embraces **controlled chaos**, that exercises the software under test with days’ or weeks’ worth of user activity but compresses time and scale to **minimize execution cost**, and that **stays within operational limits** in the process. In other words, **we need long-haul testing**.

3.1 The bug payoff

To further motivate long-haul as a uniquely valuable test technique, I present some actual product bugs I have witnessed as the result of long-haul testing:

- **The slow leak.** Objects allocated from certain operations were never destroyed resulting in slowly but steadily increasing memory usage. The issue was only noticeable if operations were performed repeatedly over a period of hours.

- **State poisoning.** A node transitioned into an undefined/unexpected state momentarily due to a software bug. Data in the system was replicated and so, if timed just right, the undefined state was saved to disk and transmitted to another node. When a node was restarted, it recovered its saved state from disk; however, the unexpected state caused a crash, preventing the node from ever starting again. This race condition required just the right mix of state changes and induced faults to observe – unlikely in a shorter, predictable test sequence, but hit relatively often in more randomized long-haul tests.

- **Too much information.** Tracing from certain components was overly verbose, leading to extremely large diagnostic trace files if the system ran continuously for an extended duration. The problem was not apparent until a customer workload test ran; the trace files were flooded with millions of repetitive trace lines which interfered with failure investigations.

Even with a full range of other tests (unit, functional, integration) already in place, these defects and many others like them can languish undetected.

3.2 A note about applicability

What about systems which do not have multi-user state, which have limited concurrency, or which do not need to operate continuously for long durations? As it turns out, even in these cases there is still some value in applying long-haul principles to improve coverage and reduce testing costs. Applying semi-random, lightly concurrent, time-compressed workloads and corresponding faults to the system under test can be a relatively low-cost way to ensure broad coverage of a variety of positive and negative code paths.

4. Can vs. should: goals and non-goals

When discussing the goals of long-haul testing, the mantra "just because you *can* doesn't mean you *should*" is apt. Long-haul tests are invaluable but they are far from the only tests you will ever need.

4.1 Goal: reduce the cost of testing

Unlike comparable functional or integration tests, long-haul tests are not highly orchestrated or strictly prearranged, a definite win in terms of the cost of test planning. The longer running time combined with semi-random actions and faults give long-haul tests an edge in terms of cost vs. benefit – overall, you need fewer tests to cover a wider variety of product code paths.

4.2 Non-goal: supplant functional testing

Long-haul tests are powerful but not omnipotent. Functional testing is still the best way to get guaranteed regression coverage or to evaluate specific product use cases. Twisting a long-haul test to be 100% deterministic degrades quickly into a "worst of both worlds" situation; such a test would not only run for a long time but it would cover only a fraction of possible code paths.

4.3 Goal: uncover race conditions and invalid states

Race conditions occur when shared data in a program is modified by multiple threads without proper synchronization[8]. These and other concurrency issues are notoriously hard to debug as well as hard to find through traditional testing techniques[9]. While no test will uncover race conditions in a reliable and repeatable way, long-haul testing is most likely to catch broad concurrency issues throughout a system. The basic recipe for a long-haul test is "randomness + parallelism + time" – exactly what is needed to reproduce a typical race condition.

An invalid program state occurs when an invariant is violated, often codified by an “assert” in the source code[10]. Long-haul tests, given their breadth, tend to cover a large number of program execution paths while also building up a fair amount of previous state given their longer running time. They are therefore quite good at unwittingly running into situations which “should never happen” according to the product design or specification. When combined with a “fail fast” methodology (i.e. the program terminates as soon as an invalid state is detected[11]), a long-haul test is an inexpensive way to find invalid state defects; simply run the test and generate one bug report every time a new “assertion failed” crash fault occurs.

4.4 Non-goal: exhaustively validate all behavior

Automated testing of any kind must use oracles to make a pass/fail decision[12]. For a directed functional or integration test, the oracle is usually just a simple check against a predetermined expectation, e.g. “when function F is called, result R should be returned.” For a long-haul test, things are not as simple. With a relatively chaotic mix of actions and faults, certain failures are expected and thus should be tolerated at least some of the time. It is impractical to devise a true oracle to cover all possibilities that the test may encounter. Most long-haul tests instead use a much simpler heuristic oracle[13] which explicitly rules out exhaustive behavioral verification of the system under test.

4.5 Goal: provide valuable feedback for ship-readiness

A well-designed long-haul test should make an important statement about product quality. The end result is one aspect, which is a testament to the reliability of the product (“the long-haul test passed after subjecting the product to 24 hours of continuous user activity and faults”). In addition, the test can provide operational statistics which help validate or challenge assumptions about system capabilities (“the product fully recovered from all network faults but the observed downtime was longer than two minutes in 50% of occurrences”).

4.6 Non-goal: provide quick feedback

A “quick long-haul test” is an oxymoron. Reducing the running time of the test will have the likely result of providing less coverage since less state will be built up and fewer actions will be executed. Many other types of tests exist which can provide quick feedback if required.

4.7 Goal: leverage controlled chaos

Chaos is a virtue in long-haul testing when it is *controlled* – instead of “*any*thing goes,” think “*many* things go.” This is the key that allows a long-haul test to have broad enough coverage but with some reasonable expectations around what can be considered correct or incorrect behavior. Randomness should be leveraged judiciously and avoided when its use would add extreme or unnecessary complications.

As an example, consider a test that needs to generate files which must be later validated for data integrity. Generating completely random files could significantly complicate the data validation to ensure the files are uncorrupted. Instead, the test could produce a file using a specific pseudorandom number generator and a randomly generated seed; the files created by this test would be random for all intents and purposes but could always be checked for complete consistency given the original seed.

4.8 Non-goal: “monkey” testing

It is trivial to write a “monkey” test, i.e. one that executes random actions at random times for random durations; consequently, it is nearly impossible to say anything of value about the results of such a test. It is hard to even understand what the test will do until it is run and then what actual use cases its activity maps to. These are problems for both the tester who must analyze and interpret the results and the project stakeholder who must decide on the merit and relative priority of a proposed defect.

5. One category, many flavors

There is not just a single type of long-haul test but rather a large category of tests that come in all shapes and sizes. The specific “flavors” of long-haul, as I refer to them, will differ depending on the needs of the project. The following are some examples of the types of long-haul tests that I have used on previous projects. This is not an exhaustive or definitive list but rather a starting point for practical consideration.

5.1 Low-level

A low-level long-haul test typically runs a stripped down version of the product in a “one-box” test environment, where a single machine hosts all necessary components. These tests are often written close to the code (like unit tests) to take advantage of internal components and data structures. For this reason, low-level long-haul tests are useful when there is a need to manipulate or observe internal state, e.g. to do more detailed validation or to inject low-level faults.

The more local nature of these tests makes them an attractive option in terms of ease of execution and debugging, especially when compared to a fully distributed test of the production system. However, to achieve this locality, many components may be simulated or absent thus removing them from the purview of the test.

5.2 Feature/subsystem

A feature or subsystem long-haul test focuses on functionality relevant to a specific area of the product. This focus necessarily constrains the surface area of the test and keeps it from becoming overly complicated or hard to investigate while still maintaining broad coverage of the area in question. This ensures a favorable cost/benefit ratio but scopes out integration coverage across multiple areas.

A feedback loop can exist between feature long-haul tests and functional tests in the same area. Functional tests can drive requirements upward to long-haul tests around coverage enhancements. Long-haul tests can push data downward about issues that warrant regression coverage or risk areas that require additional functional validation.

5.3 Customer workload

A customer workload long-haul test is a translation of a customer’s actual use cases into a test scenario, almost like a long-running acceptance test. Occasionally, real artifacts such as data files from the customer are used in the test workload. This is not always possible or desirable so these artifacts are often simulated or approximated. I have had success here by meeting with key contacts from the customer team (usually developers and testers) and asking detailed questions about their environment and usage patterns; I would then translate these into test workloads using my own test infrastructure so that I would retain control over the design and implementation of these artifacts. The drawback is that these simulated artifacts would require some maintenance over the lifetime of the project as scenarios change; however, the cost is usually lower than trying to fit real customer code and data into controlled test environments.

Customer workload long-haul tests are far more specific than most other long-haul tests and may only cover a few particular patterns of behavior. However, these tests tend to span many product subsystems, making them higher level relatives to integration tests. They are best employed in cases where a customer uses the product in an otherwise uncommon configuration which has less overall representation in other test areas (e.g. the use of a unique or specialized network topology).

5.4 Full-system

A full-system long-haul test operates at the highest level, integrating multiple areas together and exercising cross-component workloads and system-level faults. These tests are usually the most difficult to develop and analyze given the number of possible sources of errors and defects. However, for the same reason they are also likely to uncover issues that no other test would be able to find.

A feedback loop can also exist here between full-system and feature/subsystem long-haul tests. Defects found at this level that exist within a single area may be pushed down to the appropriate feature long-haul test, while known missing integration coverage can be pushed upward to the full-system long-haul test.

6. Automation design considerations

The eventual success or failure of automated long-haul testing will be due in large part to the design decisions made early in the test lifecycle. Based on my past experience, I present the following considerations for designing effective long-haul tests.

6.1 Define the basic test architecture and topology

Before the test implementation phase, it is wise to decide on what a long-haul test should look like. There are many important decision points as far as the details go, but most tests boil down to the following basic description: a long-haul test **drives concurrent** and **continuous** workloads made up of **actions** which exercise product functionality and **faults** which force **recoverable** errors, subject to **periodic validations** to decide whether the system is operating correctly.

6.1.1 Actions

Depending on the context, an action can be a function, a class or interface implementation, or even a separate executable. The design for actions in turn pushes requirements for the test driver, which in the simplest case can be a loop with a random number generator or in a more complex system, a distributed work scheduler. Where the actions run is dictated by the topology or physical layout of the test which also drives decisions on the usage of parallel threads, separate processes, and remote machines.

6.1.2 Concurrency

The level of activity must be bounded to avoid exceeding operational limits but should be high enough to keep the system busy. It is simplest to provide a predetermined “safe” upper bound to limit the amount of work (e.g. “ensure a maximum of 100 concurrent requests”). It is also possible to use an adaptive design which discovers limits dynamically but this is somewhat difficult in practice to implement, and even more so in a fully distributed topology.

6.1.3 Faults

Fault injection is a massive topic in itself and a thorough treatment would require pages of exposition. Instead, I will summarize the key points as they apply to long-haul tests.

Higher level tests should focus mainly on external faults, i.e. those coming from *outside* the system. Process crashes, network outages, and disk errors are typically easier to inject at a high level and better represent the faults that occur in production. Lower level tests which are closer to the code can opt for a more intrusive internal fault injection strategy, e.g. causing internal functions to throw exceptions or forcing memory allocators to fail.

Ensure that any given fault has a defined recovery action, whether automatically taken by the system or explicitly invoked by the test. It does not do much good to have a long-haul test that intentionally forces the system into a terminal error state and then simply stops. (Of course, if the system is designed to safely recover from the fault but does not, this is an obvious product defect.)

6.1.4 Validations

It is important to clearly specify the set of validations which are applicable to a given long-haul test. These ultimately determine the expected results and the success criteria for the test. Identify the data needed to validate a given set of actions and decide how to collect it. Note that some validations may be able to operate against partial state or incomplete knowledge, while others require the system to “pause” while they gather and check the complete state. Understand the implications of the validation strategy you choose and ensure the product supports it – if not, this could be a sign of a product testability or design issue requiring further discussion within the team.

6.2 Separate actions from validations

Tightly coupled actions and validations are rarely a good idea even in functional tests as they limit reuse and lead to poor test design. This approach simply does not work for long-haul tests where the same actions can be performed at different times under different conditions with different expected results.

Instead, think of actions as data producers and validations as data consumers. An action may produce multiple data items, e.g. request timestamps, the time taken to respond, the error code or codes, and the user state associated with the requests. Distinct validations can then be designed which consume all the relevant data items for the checks being performed with no need to care about which action (or actions) produced them.

6.3 Parameterize test inputs

It is a good idea in general to allow configurability of test behavior without having to change the code. Long-haul tests given their larger scope and longer durations tend to require even more tuning over their lifetime than other more targeted tests. Parameterize early and often, and not just for data values – consider allowing selection of different types of test actions and validations if appropriate.

6.4 Use profiles to guide decisions

Consider a software product which maintains online backups of user data. The full test matrix for such a system could be huge – there are essentially infinite possibilities for how to choose the file sizes, the file contents, the rate of uploads and downloads, and so on. An effective way to turn this into a finite test matrix is to consider user profiles. Use production data if you have it or market research and projections to decide what users are actually doing with your software. Perhaps you can determine that there are three basic profiles that most users can be grouped within, each associated with some general usage patterns (e.g. a “casual user” uploads about 100 files per day, 10 MB per hour, etc.).

6.5 Define partitions for disparate test actors

Validation of a long-haul test is often difficult, but the task can be simplified by partitioning the various test actors and workloads. Again, using the online backup example, imagine that users can create shared folders which get synchronized back and forth between multiple machines. Throwing multiple concurrent workloads at the system to indiscriminately add and remove files from a shared folder will quickly become hard to manage and verify. Instead, you can use the shared folders as partitions to group different user workloads. One shared folder could be associated with a set of users that read and write only distinct files with an expected result that no conflicts should be observed. Another shared folder could be associated with users that read and write the exact same group of files with the expected result that conflicts are occasionally observed. This partitioning allows more detailed validations and coverage of still semi-random but better targeted use cases.

6.6 Coordinate invasive faults

In some cases, relatively isolated faults if injected in quick succession or at inopportune moments will cause undue pressure on the system. For example, imagine a fault that power cycles a single backend server. If the system is designed to take up to one minute to recover from this fault yet the test injects these faults once *per second*, the system will quickly lose all backend servers. These kinds of faults should be properly scheduled and coordinated to avoid turning a long-haul test into a stress test.

6.7 Optimize for diagnosability

Long-haul testing will never guarantee reproducibility; instead, strive for **diagnosability**. Every long-haul test should create a complete and sufficiently detailed log of all actions performed to aid in root cause analysis of any discovered defect. (It goes without saying that this is not a test-only responsibility – the product itself must also have enough instrumentation and logging of its own actions.) Be aware that it is definitely possible to run into log size issues given the long duration of the tests; to avoid running out of disk space or creating files which are too hard to manage, consider using segmented or circular log files for both test and product components.

7. What success looks like: project planning and exit criteria

The decision to adopt long-haul testing is only the first step. To actually make this test effort a success, the team as a whole must come together and commit to seeing it through to the end of the project. The best long-haul initiatives use a **common vocabulary**, clearly define the **scope and target** of the tests, align the work to a **realistic schedule**, and **iterate** to reach a desired end goal.

7.1 Use a common vocabulary

Make sure that your organization understands and uses the same terms to describe the long-haul test effort. In particular, be ready to compare and contrast any similar load testing efforts that your team may already be doing. Although it sounds obvious, a common vocabulary is essential and simple misunderstandings can easily derail an otherwise solid plan.

7.2 Agree on the scope and target

Will you use customer workload tests, feature tests, or a mix of long-haul test flavors? Which behaviors will you focus on and what validations will you apply? Are any areas of the product specifically excluded or out of scope for the test effort? Perhaps most critically, *how long will these tests be run* and *what are the expected results*? A successful long-haul initiative requires clear answers for all these questions and likely many more.

Strive for organizational agreement as much as possible when deciding on the answers. While there are some decisions more naturally left to the discretion of the test organization (e.g. test planning, design and architecture), others must involve partner engineering disciplines and project stakeholders (e.g. expected results, required success criteria).

7.3 Build a realistic schedule

Although closely related to the scope and target of the test effort, the schedule is worth discussing separately. Long-haul tests take time to design and execute and the schedule must reflect this explicitly. If the team agrees that, say, the Beta 2 product release must undergo 24 hours of long-haul testing, the team must also be aware that the Beta 2 release may be pushed out by several days due to last-minute issues found by the tests. Failure to account for these “test resets” can introduce unnecessary risks and uncertainty into the product cycle.

7.4 Iterations: crawl, walk, run

Long-haul tests themselves are iterative, always inching closer to an end goal through a series of smaller steps. The long-haul initiative should be no different. Consider defining a set of “crawl, walk, run” goals to slowly build up the capabilities, breadth, and rigor of the tests. It is important to focus on small, realistic objectives, especially if the organization has never done long-haul testing before. The full progression to the final goal may take days, weeks, or months; the key is to ensure the timelines and associated goals are well-defined, well-known, and agreed upon within the organization.

For example, the very first long-haul tests may set a target duration of only four hours, validate only that the product does not crash, and cover only one major feature area. After allowing sufficient time for the product and tests to stabilize, the target can increase steadily toward a final goal of, say, 24 to 48 hours of continuous operation with a full set of validations covering multiple features and subsystems.

7.5 Hold the bar

Long-haul testing can be quite challenging at times. Tests may find hard-to-diagnose bugs at inopportune moments. Resist the temptation to relax the exit criteria or ignore issues. If the team has set the bar and decided on specific long-haul goals, do all that you can to **hold the bar**. Make reasonable exceptions if necessary but always involve the project stakeholders and be transparent about it. Sometimes a late-breaking issue is truly an outlier and should not block the release; always assess the risk and make an informed decision as a team.

Similarly, do not subject the team to a moving target. Avoid making major or frequent changes to the scope or validations of the long-haul tests without first engaging with those who will be impacted (e.g. developers, release managers). Long-haul testing can pay dividends for almost any team but it will have the best impact if the effort evolves in an intentional, incremental way throughout the project lifecycle.

References

[1] Beck, Kent and Erich Gamma. 2002. “Test Infected: Programmers Love Writing Tests.” SourceForge. http://junit.sourceforge.net/doc/testinfected/testing.htm (accessed June 1, 2013).

[2] Analysis.Net Research. 2012. “7th Annual State of Agile Development Survey.” VersionOne. http://www.versionone.com/pdf/7th-Annual-State-of-Agile-Development-Survey.pdf (accessed June 1, 2013).

[3] Page, Alan and others. 2008. *How We Test Software at Microsoft*. Microsoft Press.

[4] Whittaker, James A. and others. 2012. *How Google Tests Software*. Addison-Wesley Professional.

[5] Osterman Research, Inc. 2007. “Planning for Improved Email Availability.” Osterman Research, Inc. http://www.ostermanresearch.com/whitepapers/or_nev0707.pdf (accessed June 2, 2013).

[6] U.S. Food and Drug Administration. 2009. “Glossary of Computer Systems Software Development.” U.S. Food and Drug Administration. http://www.fda.gov/ICECI/Inspections/InspectionGuides/ucm074875.htm (accessed June 2, 2013).

[7] Cherskov, Sergio. 2003. “Windows CE .NET Long-Haul Testing Scenarios and Results.” Microsoft. http://msdn.microsoft.com/en-us/library/ms836785.aspx (accessed June 2, 2013).

[8] Microsoft Support. 2012. “Description of race conditions and deadlocks.” Microsoft. http://support.microsoft.com/kb/317723 (accessed June 8, 2013).

[9] Beatty, Sean M. 2003. “Where testing fails.” Embedded.com. http://www.embedded.com/design/embedded/4024600/Where-testing-fails (accessed June 8, 2013).

[10] Mahmoud, Qusay H. “Using Assertions in Java Technology.” Oracle. http://www.oracle.com/us/technologies/java/assertions-139853.html (accessed June 8, 2013).

[11] Shore, Jim. 2004. “Fail Fast.” Martin Fowler. http://www.martinfowler.com/ieeeSoftware/failFast.pdf (accessed June 5, 2013).

[12] Kaner, Cem. 2004. “Examples of Test Oracles.” Center for Software Testing Education & Research. http://www.testingeducation.org/k04/OracleExamples.htm (accessed June 8, 2013).

[13] Hoffman, Douglas. 1998. “A Taxonomy for Test Oracles.” Software Quality Methods, LLC. http://www.softwarequalitymethods.com/Papers/OracleTax.pdf (accessed June 8, 2013).

Effective Peer Reviews: Role in Quality

Anil Chakravarthy (Anil_Chakravarthy@mcafee.com)

Sudeep Das (Sudeep_Das@mcafee.com)

Nasiruddin S (nasiruddin_sirajuddin@mcafee.com)

Abstract

The utility of reviews, specifically of artifacts related to software development cannot be overemphasized. However, adopting a consistent and fruitful review process is difficult. There are many challenges to streamlining a review process to be widely accepted and followed in a software development team.

In this paper, we look at the learning gleaned from trying to adopt a consistent, constructive and effective review process across multiple teams. We look at the pitfalls, travails, shortcomings and mistakes while adopting a review process across various stages of the development process. We also highlight things that actually help a review process to be a “baked in” part of development process rather than being a “bolted on”. This paper is primarily based on the authors’ first hand experiences on introducing, sustaining and benefitting from peer reviews in software development.

The paper presents a holistic view of review rather than focusing on just code reviews. Review of other artifacts like designs, plans, and schedules shall also be discussed. It will be useful to managers and development team members in leadership roles. The paper provides insight from practical and firsthand experience about one of the important aspects of a software development process.

Biography

Anil Chakravarthy is a Senior Technical Lead at McAfee with more than seven years of software development experience. Highly passionate about quality, he strives to look for ways and methods to improve software reliability and usability. As an inventor of key technology solutions, his interests include security management, content distribution and updating.

Sudeep Das is a Software Architect at McAfee, with more than ten years of experience designing and implementing software. He started with a three person team and over time has seen it to grow to more than twenty, while improving upon development and testing processes incrementally over the years. His areas of interest span security management, content updating mechanisms, data protection and virtualization.

Nasir is a Principal Engineer at McAfee, with more than eight years of software development experience. He is highly passionate about working on emerging and innovative technologies.

Introduction

Peer review is one of the most effective ways of ensuring quality. Not just in software, but also in research and academic publications. Software that doesn’t use reviews has defects discovered at later stages of development and potentially post shipping. It has also been established that late detected defects are expensive to fix. Yet, software engineering teams struggle to adopt a consistent and effective peer review process. For teams that do adopt a review process, at times, they fail to find enough early defects to justify the investment in a review process. Given the vast and diverse literature on peer reviews, software engineering teams often struggle to adopt a review process that works best for them.

In this paper, we start with presenting the goals of a peer review. We discuss effectiveness of peer reviews. Finally, we discuss factors that contribute to effectiveness of reviews and factors that hamper the effectiveness of peer reviews

Our aim is to share the lessons we learned while trying to adopt a peer review process in multiple teams in our organization. Over the years, we have tried different review processes and methods and refined our processes. We continued the things that worked, and eliminated parts that didn't work. This paper is the outcome of our experience and learning on peer reviews, and is based entirely on our experience

We believe that leaders of software engineering teams would find this paper most useful. However, we also believe that our experiences will be beneficial to other software engineering roles as well.

Structure

While many software engineering teams equate peer review to code review, the fact is that peer reviews apply to all artifacts of software engineering process. Artifacts like requirements, specifications, estimates, design, test plans and documents are all candidates for review. A peer review for each of these artifacts can help with early detection and removal of defects.

Not restricting peer reviews to code review brings up a few fundamental questions. What should be reviewed? By Who? Why? When? How?

We start with trying to answer the “What”. What software engineering artifacts should be peer reviewed? Having identified the review targets, we then try to answer the “Why” of that particular review target. It establishes the goal of the peer review for that particular artifact. Finally, we look at the ”Who, “When” and “How” of the peer review for each of the items

The What:

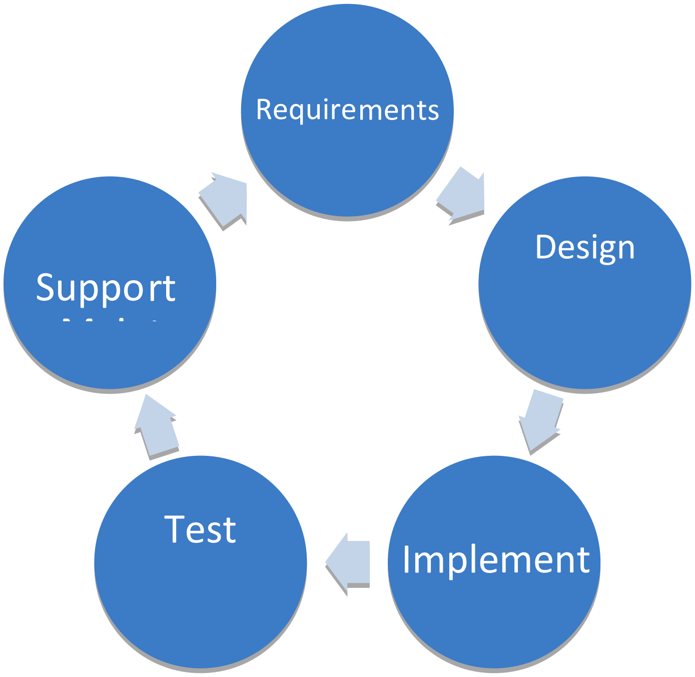
Software engineering encompasses multiple stages and functions, namely, Requirements, Design, Implementation, Testing and Support. While the stages may overlap in time, the primary output of each stage is distinct. It is the output of each of these stages that should be peer reviewed

So, the artifacts that are subjects of peer reviews are

1. Concept Commit
2. Requirements
3. High level feature design and architecture
4. Low level feature design
5. Feature Implementation
6. Review of static analysis findings
7. Regular code review
8. Test plans
9. Test cases

The Why:

Each of the artifacts generated above have a different purpose, and the peer review goals of each of them are different. The key goals of reviewing each of the artifacts are as below.



Concept review goals:

Leaders of a software engineering group formulate a business concept based on a multitude of factors such as market direction, business strategy, and technology advances. While there are a multitude of factors to be considered, the primary purposes of a concept review are to ensure that

1. The concept aligns with the larger strategic imperatives of the business
2. The concept does not adversely impact other successful offerings
3. The concept fills an existing gap in the market
4. The concept is sized for a reasonable time to market

Requirements review goals:

Once business leaders commit to a concept, it is broken down by the concept champion into deliverable features in a way that fulfills the vision behind the concept. The features or epics are then further broken down into user stories to achieve a high degree of granularity. It is at this stage that a thorough review with engineering leadership is warranted to realize the following important goals

1. Ensure technical feasibility of the features
2. Ensure engineering team capacity
3. Prevent feature creep
4. Prevent scope creep
5. Ballpark sizing and alignment with time to market needs
6. Enhance clarity and minimize ambiguity of features
7. The engineering cost of developing the solution.

High level design review goals:

A high level design serves as a straw man, and provides a technical view of the solution being developed. This phase needs in depth review, as the technical choices made during this phase have a very significant bearing on downstream quality. The goals of this review are

1. Ensure that the technology choice can meet scale and performance needs
2. Ensure that the design accounts for security aspects, like authorization, authorization, audit, regulatory compliance
3. Ensure that the design can be implemented on all desired platforms
4. Ensure that economics of technology choices are acceptable. Some technology choices, while easier to work with, force the end user to license building block technologies.
5. Discover if the end user is more familiar with one technology over another that may be used in the solution
6. Ensure that the design covers all use cases agreed to in the requirements review
7. Ensure modularity and extensibility of components

In our experience, we have found that design processes are iterative in nature and multiple revisions are a norm. A design that is subjected to multiple iterations is not symptomatic of a deficient design, but is rather indicative of a fuller and better understanding of the requirements

Low level design review goals:

A low level design is highly implementation centric and is the blueprint of the software being developed. Low-level designs are a magnified view of high-level designs and have a lot in common with the high level design. The key goals of a design review are

1. Ensure alignment with high level design
2. Ensure appropriate choice of algorithms
3. Ensure modularity
4. Ensure extensibility
5. Ensure conformance to established and standard design patterns
6. Improve the accuracy of estimates

Implementation review goals:

Implementation - better known as code. Literature abounds with stated, perceived and purported goals of code review. Goals of code review vary widely and depend on the specific domain and application of the code being developed. Code for a banking application will go through significantly more significant review criteria than, for example, a desktop game application.

However, there are goals that are common across the board…namely

1. Readable code
2. Maintainable code
3. Documented and well-commented code
4. Adherence to design
5. Logical correctness
6. Error handling

Test plan review goals:

A test plan details the high level steps that need to be taken to ensure low defect software.

Engineering team members rely on the test plan to determine if the software has achieved its stated quality goal.

The goals of reviewing a test plan are to ensure that

1. Quality goals are reasonable and achievable
2. All relevant quality indicators are spelled out.
3. All testing phases accounted for… unit test plan, integration test plan, acceptance test plan
4. All testing domains accounted for, for example, security testing, performance testing
5. All features and use cases are covered
6. Automation has been accounted for

Test case review goals:

Test cases lay down the steps that verify that a feature satisfies its stated requirements and quality goals. It is important for test case reviewers to ensure that

1. Test cases are accurate
2. Test cases are exhaustive
3. Test cases are complete
4. Test cases are comprehensive

The How:

The generally prevalent peer review methodology is a formal in-person review where the owner of the work presents his/her work, and multiple reviewers engage in finding shortcomings in the work. The shortcomings are then debated, argued, and some kind of resolution to each shortcoming is expected. Reviewers across multiple functions are involved, ostensibly, to keep all stakeholders aware of important decisions. This approach, in our experience, is found to be severely limiting, and is highly ineffective for the following reasons:

1. Reviewers may have opposing views on issues, and an in-person review with 5 participants turns to a technical duel of 2 seasoned professionals that continues for hours.
2. In-person reviews are found to be intimidating by junior team members, particularly in the presence of reporting managers.
3. Managers are judgmental about individual performance based on code review feedback.
4. For distributed teams, synchronizing time for multiple team members over multiple time zones is a challenge.

Our recommendations for peer review are as follows:

Pre Review :

1. Train the reviewers. Reviewers should be trained on what to review, and be sensitive to the goals and purpose of the review
2. Designate an arbitrator for each phase and artifact. Someone in the team should have the final word in face of disagreements.
3. Utilize automated tools where possible. For example, for code reviews, readability, and maintainability and to some extent, error handling may be enforced and detected using static code analysis tools.
4. Avoid reviewer fatigue. Content to be reviewed should be small. If there is a lot of matter to be reviewed then it should be broken down and each part should be reviewed separately.
5. Avoid multi-level reviews. Some teams have an initial peer review that is then followed up by a “superior review” of sorts, where a “senior” and “expert” person reviews the already reviewed content.

In review:

1. Avoid in-person reviews for code, test cases and design. It is useful to use collaborative review tools such as code collaborator, that lets presenters annotate and upload their work for review
2. It is good to have cross-functional reviewers, primarily with the purpose of identifying gaps in the subsequent development phases. For example, in a code review, developers may find the code suitable for production, but the code may not be testing and automation friendly. Review by testing team members would highlight this shortcoming.

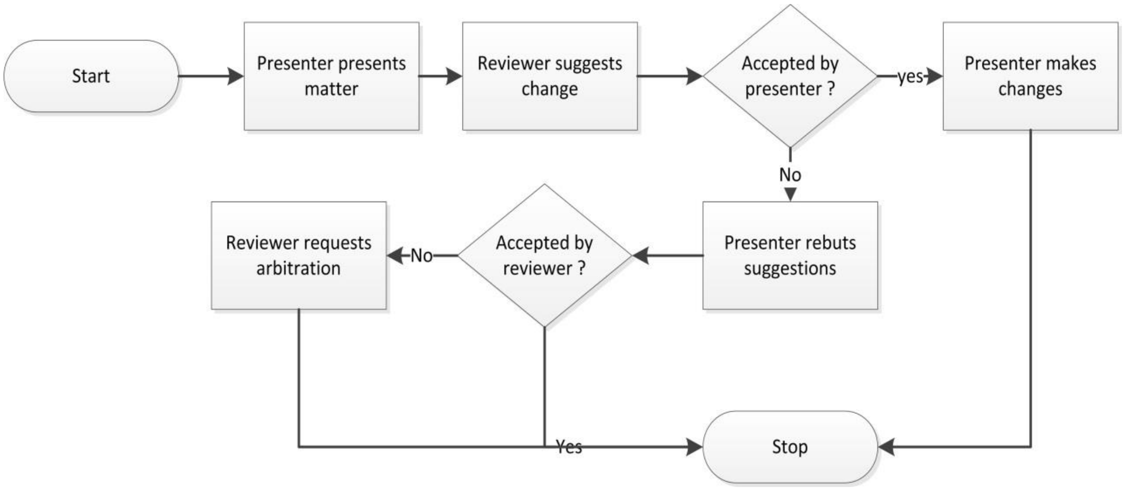
3. Two is the optimal number of reviewers. The review benefits hit a point of diminishing returns with more than two reviewers. We have experienced that there is significant duplication in the issues discovered by additional reviewers.

4. Observers should only observe. If there are stakeholders who need to be kept informed, their role is that of observer who can only watch but cannot comment.

Post review:

1. Timely closure: Having review pending, and going ahead to the next step is a mistake which that leads to rework.

2. Avoid a rally. Unresolved disagreements should be arbitrated rather than being dragged on.



The Who:

The effectiveness of peer reviews has a strong dependency on the number and expertise of the reviewers. Some teams follow a pre-defined matrix of reviewers; some teams gate reviewers on experience and expertise. The key aspects to be considered while deciding on reviewers are

1. In general, reviewers should belong to the same functional group that produces the artifact, and observers should belong to immediately succeeding functional group that utilizes the artifact.

2. The concept review should have participation from product management, possibly early adoption users, and engineering leaders.

3. High level design reviews should have participation from architects and senior developers.

4. Low level design review should include developers as reviewers and senior testers as observers.

5. Test plan review should include senior developers and senior testers.

6. Test case review should include testers and developers.

7. Arbitrators should be senior and respected people in the same functional group that generates the artifact.

Teams generally have mixed composition of experience and expertise, and it's important to ensure that skills are organically developed within the team. The concept of observer provides a unique opportunity for achieving this, where a team member would learn to be more effective watching other team members perform the role

It is found to be detrimental when reviewers from a different functional group are actively involved in reviewing artifacts of a different functional group. A disproportionate of time is spent explaining and clarifying commonplace and obvious facts; context is not obvious to the reviewers. More reviewers do not lead to better reviews.

The When:

Review content can generally be plan-related content or action-related. Code, test cases, and designs may be classified as action-related content, whereas concept, test plans, high level architecture are plan-related content.

For action-relation content, the notion of "Review early, review often" is found to work the best. This is because review presenters, at times, create a large volume of work to make sure the work is complete before they present it for review. This deluges reviewers with too much content to be reviewed in a short time. It is useful to constitute review of the days' work at the end of the day, to be reviewed first thing the next day.

For plan-related content, where completeness is a key aspect of review, it is more useful to have high bandwidth, engaged, potentially in person reviews rather than off line and passive review. Reviews should be completed before moving on the next task. Pending and unfinished reviews are a major source of rework and effort duplication. For example, if a high-level architecture review is unfinished and the next phase of development is already in progress, a change to architecture later would result in rework and effort duplication.

Concept, high-level design, low-level design, test plans and test cases should be reviewed once complete. Feedback should be incorporated and the artifacts reviewed before moving on the next phase. Subsequently, any modifications to these artifacts should also be similarly reviewed.

Conclusion

This paper presents our experiences on the "Who", the "How" and the "When" aspects of different review contents that we have found effective for our teams over a significant period of time while using agile methodologies. Reviews are one of the key contributors to software quality. Reviews need to be effective to useful. Reviews can be made highly effective by following a few simple and easy to adopt guidelines. Different review artifacts may utilize different review strategies, like passive but arbitrated reviews and in person but open debate reviews. Adopting reviews initially present challenges because of its inherent nature of being a fault finding exercise, and teams should be eased into it over time. From a management perspective, it works better if reviews are a pulled process, as compared to being a pushed process. Teams need to be reasonable and guarded in their expectations from a review, as reviews do not guarantee zero defect software. Reviews alone however, do not ensure software quality. Complementary processes like continuous integration and automated verification tests are important and have their place in the software development process. Metrics are important.

How to Review Technical Documentation

Moss Drake

drakem@dmcdental.com

Abstract

In business, people are often asked to become informal editors for specifications, vision documents, user documentation, and technical articles. They are asked to review a document, but are not often given guidelines on how to go about the process.

Written documents are supposed to be about communication, but communication is a fuzzy concept. There are many things to consider: whether the document is complete, whether the correct information has been communicated, and even small details, such as whether the title agrees with the content, may be questioned. Some authors have wonderful ideas, but run into problems sizing and structuring their efforts such as trying to fit a book into an article. How can an inexperienced editor determine when a document is written in a way that makes the most sense? It would be an improvement if there were some way to quantify the review process.

This paper explains a three-step editing process that checks the content and structure of the paper for comprehension, accuracy and completeness, and it also provides tips on style and copy editing. The process can be used for editing software specifications, user documentation, technical articles or blog entries. Benefits include showing the author whether the correct information was communicated to the audience, being able to tell whether the information is balanced and complete, and whether the scope of the content is appropriate to the document.

Biography

Moss Drake is a software developer and project leader at Dentist Management Corporation in Portland, Oregon. He has been developing software for over twenty-five years, focusing on solutions for the healthcare and insurance industries.

Moss has a B.S. in Computer Science and a B.A. in English Literature from Oregon State University.

1 Introduction

How many times in the workplace are people handed documents and asked to review them? Often the person asking fails to provide a context for the advice, whether it is about the content, tone of voice, or spelling and punctuation. In many cases, the reviewer is a peer who is asked to read for content, but even people who are domain experts may not understand whether the writing style is appropriate for the audience, or whether the document is written in a way that is complete. In other cases, the reviewer may have editing skills but does not fully understand the topic and so cannot properly assess whether key points are addressed. Sometimes the author simply wants to know whether the reader "gets it."

Written documents are supposed to be about communication, but communication is a fuzzy concept. Many things may be considered: whether the document is complete, whether the correct information has been communicated, and even small details such as whether the title agrees with the content. Some authors have wonderful ideas, but run into problems sizing and structuring their efforts such as trying to fit a book into an article. How can an inexperienced editor determine when a document is written in a way that makes the most sense? It would be an improvement if there were some way to quantify the review process.

For the past five years, I have reviewed and edited papers for the Pacific Northwest Software Quality Conference. During this time, I have formulated a three-step editing process that tests the content and structure of the paper for comprehension, accuracy and completeness, as well as providing feedback on style and copy editing. This process, which involves creating an outline based on the paper and looking for telltale characteristics in the outline, can be used for editing software specifications, user documentation, technical articles or blog entries.

This paper explains how to apply this method to technical documents. Its benefits include showing the author whether the correct information was communicated to the audience, being able to tell whether the information is balanced and complete, and determining whether the scope of the content is appropriate to the document. Along the way, it also discusses tips for style and proofreading.

2 The Role of an Editor

"That's not writing, that's typing." – Truman Capote

Writing is usually thought of as a solitary endeavor where one person sits at a keyboard composing and expressing his or her thoughts. However, since the point of writing is to share those thoughts in a way that speaks to a reader, before a document is finished it usually requires much feedback and revision to ensure those thoughts are communicated. Without the help of an editor and others who provide advice on the work in progress the process is, as Capote said, merely typing. In some cases, the editor may also be the author, but it always takes two persons to close the loop. Barry Lopez says, "The writer works on the inside and the critic works on the outside." The editor's role is to be a thoughtful critic for the writer. To help him develop his internal thoughts into something that can be shared in the most effective way.

People, however, are not born with these skills. Book publishers and newspapers have trained editors. In other industries, companies may have technical writers on staff that can help edit copy, but more often people who are not trained as editors are asked to read specifications, advertising copy, and user documentation with a critical eye. Ad hoc editors are sure to provide mixed results, and without a method, the process is usually inefficient.

To become more efficient, it is helpful to identify the steps an editor goes through with a document. There are three levels of editing: content, style and copy editing. To avoid rework, each phase should be completed before proceeding to the next. Content editing ensures that the article is complete, comprehensible, and follows a logical flow. Next, review the style of the paper. Does it meet style guidelines? Does the author use a writing style appropriate to the audience? Is it concise and clear, and most importantly does it have the appropriate voice for the audience? The final step involves copy editing the paper.

3 Ground Rules

The editor and author should establish ground rules for the editing process to keep it constructive and professional. For example, the job of the editor is to evaluate the writing, not the writer. Comments, negative or positive, should apply to the words on the page and not to the character of the author. This leads to the second rule: be specific. If a sentence is confusing, the editor should make a note of it and ask for clarification. Both constructive and positive feedback should be applied to specific areas of the paper. Blanket statements saying the article is “Great!” may help the author’s ego, but it fails to provide constructive feedback on the writing. If the writing is great, an editor should be able to highlight specific passages as superior examples, and then compare them with areas that need work. An editor should help guide the writing process, but avoid inserting himself into the writing. Putting words into the author’s mouth and then insisting that version is better is not constructive. The author’s responsibility is to write the paper, while the editor acts as the sounding board. Most importantly, the editor provides the external critic with whom the writer can have a dialog to discover what works.

Before beginning any critique, the editor should read the paper twice: once quickly, to get a basic understanding of the topic and a second time more slowly to comprehend the details. At this point, the editor should record any initial thoughts and set them aside. They may be useful later to remember impressions of the article as a new reader, which is important since most of the audience will fall into this category.

4 Editing for Content

“Don’t get it right, just get it written.” ~James Thurber

The first phase of editing should focus on content. During the initial drafts the author has been struggling with scope, direction, and just getting the ideas on the page. As Thurber says, it is probably not perfect, but at least it is written. In this stage, the editor assesses which parts of the document are “right” and which need work. This process tries to answer the following questions:

- Can I understand the paper?
- Will it make sense to the audience?
- Does it flow logically?
- Does it seem complete?
- Do all sections apply to the main topic?
- Is there enough supporting evidence or real-world examples?

Writing is about communication. The word comes from the Latin "communis," meaning to share. An article should be a shared understanding between the author and the reader. The simplest way to test the communication of ideas is for the editor to summarize his understanding of the paper, and then reflect this back to the author. For short pieces, writing a synopsis of the article may be enough, but creating an outline has additional benefits. Outlining the paper reverse-engineers the content from the point of view of the reader. It allows the author to understand whether key points are successfully communicated. It also exposes to the editor whether the paper is complete, focuses on the right topic, and has the correct flow.

4.1 Outline the Paper

Writing an outline based the content of an existing paper is much like writing one from scratch. Major sections are the top level of the outline, subsections are under that, and under those is a summary sentence for each paragraph. This is known as a "sentence outline."

1. First Major Section ← Top Level
    1.1. A subsection ←Sub-heading
        1.1.1. First paragraph summarized
        1.1.2. Second paragraph summarized
        1.1.3. Third paragraph summarized
    1.2. Another subsection
        1.2.1. First paragraph summarized
        1.2.2. Second paragraph summarized
        1.2.3. Third paragraph summarized
2. Second Major Section ← Top level
    2.1. Another subsection

Like a story, the body of a paper will have a beginning, middle, and end. The beginning often presents itself as a problem, a question, or a challenge. The middle explores the problem, providing background and supporting evidence. The ending resolves with a solution, a conclusion, or a revelation and will often include next steps.

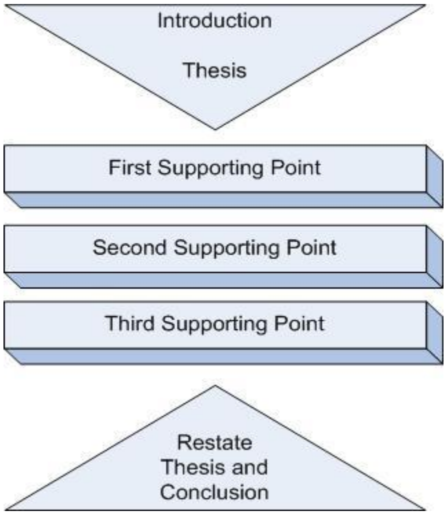


Figure 1 General structure of a document

When writing a summary of a paragraph, it helps to understand the intended underlying structure. A paragraph usually starts with the topic sentence, contains two or more supporting sentences, and ends with a concluding sentence. The topic sentence is a general statement that sets the scene, while supporting sentences explore the topic with more detail. The concluding sentence is optional, but in a longer paragraph it usually summarizes the topic into a literal conclusion.

Generally, an editor should be able to write a one-sentence summary of each paragraph. In a well-structured paragraph this summary should be evident, while less organized paragraphs may have the topic buried in the body of the text. If the general idea of the paragraph is not immediately evident then this is a red flag for the editor.

To ensure comprehension, the editor needs to synthesize the topic into new words. Since this is a test to ensure a communication of ideas, it is not enough to copy and paste the topic sentence from the paper into the outline.

The completed outline is a useful feedback tool in two ways. It is an outline of the major points contained in the paper, communicating back to the author what the editor understands of the article. The editor can also use it to check the content of the paper.

Four rules for writing outlines (Tardiff & Brizee) can be used to test the content of the article:

4.1.1 Parallelism

Parallelism means that outline headings at the same level should preserve parallel structure. If the first heading begins with a verb, the second heading should also begin with a verb. This rule may not be necessarily true for the summaries of the paragraphs since the editor rather than the author wrote them, but the major sections and subsections should adhere to this guideline. Here is an example:

1. Plan the project
2. Team members ← This is not parallel; suggest “Enlist Team Members”
3. Implement the project
4. Release the product

4.1.2 Coordination

With coordination, topics at the same level should have the same significance. Section 1 should have the same significance as Section 2, and subsection 1a would have the same significance as subsection 1b or 2a. For example:

1. Design the program icon ← This is not coordinated; suggest “Develop UI Plan”
2. Develop Back-to-school marketing plan
3. Develop Winter Holiday marketing plan

4.1.3 Subordination

Subordination indicates that information in major headings is more general, while subheadings move toward the specific. For example:

1. Identify beta sites
   1.1. Choose only local sites
   1.2. Favor customers that have previously been beta testers
2. Identify initial release sites
   2.1. Choose customers in Oregon
   2.2. Roll out marketing plan to all customers ← This should not be subordinate to 2.

4.1.4 Division

Division specifies that there is never a single heading or subheading. Each head is divided into two or more parts. For example:

1. Interview current customers for feedback
   1.1. Talk with Dr. Smith ← This breaks the rule of division; add another subsection
2. Interview customers who use the competition
3. Review information with marketing

Given these guidelines, it is possible to check the outline for areas that break the rules. Assumptions can be made based on which rule is broken.

4.2 Check for Scope Creep

Scope creep occurs when the content of a paper drifts away from the main topic. To avoid this, all branches of the outline should have at least two children. Headings and subheadings should be fairly balanced. A section with only one child should either be promoted to its own section, or perhaps the author meant to include more information on the topic, but it was either omitted or not clearly expressed.

4.3 Check for Focus

If one heading has two children, and another has twenty, perhaps the topic is skewed. The editor could suggest that the more heavily weighted topic should become the focus. Similarly, depending on the scope of the document, an outline where each section has ten or more subsections, may be overly ambitious for its size.

Sometimes there will be a sidebar, or a subsection. In these situations, the outline will have another indent. An indent more than four levels deep could indicate too much detail in a particular area, leading to an unbalanced article. Either surface the details in their own section, or reduce the detail altogether. If the information is vital to make a point, consider using an infographic to present complex information succinctly and clearly.

Topics with more than three subsections should be reviewed for duplications or redundant information. Good technical papers require a strong argument, but excessive examples can lose the audience by boring the reader. The outline provides an opportunity to scan the content and see which sections to combine or omit.

4.4 Check for Missing Sections

Reviewing an article or a specification for content can be difficult and tedious, but trying to think what might be missing is a daunting task. How can you spot what is not there? Using the outline makes this simpler.

Each major section should have an introductory paragraph and a concluding paragraph, while each subsection should have paragraphs summarizing background and citing supporting evidence or examples. With an outline it is easy to spot check these areas for completeness.

A strong argument usually has three supporting points. As mentioned in the section on scope creep, a parent with only one or two children could mean that a section is missing. If the section has only two children, the author should reexamine the argument, or come up with a third supporting example.

4.5 Check for Comprehension

The outline also provides multiple ways to test for comprehension. While writing the outline the editor may discover paragraphs where the topic sentence was not clearly expressed, or paragraphs that could not be summarized into a single sentence. In both cases the editor should raise these issues with the author and suggest starting the paragraph with a more obvious topic sentence and if there are multiple topics, breaking it into several paragraphs.

When reviewing the topics of the headings and subheadings, outline levels should be parallel. If they are not then perhaps the organization needs work.

Subordination says each subpoint must be related to, or a part of, the point it follows. This rule helps test whether the paper has a logical flow. If, on reading the outline, the editor finds a point that does not relate to its neighbors in terms of flow, he should make a note of this. Perhaps it should be moved to another section, or adjusted within the context of the current section.

After checking for each of these conditions, the first phase of editing for content should be complete. The editor is ready to provide the feedback to the author. He can deliver the notes face–to-face or in a written document, but he should remember the ground rules. Notes should be concise and specific and evaluate the writing, not the writer. Prepare the author for further steps. The editor can provide the author with the outline and ask if it contains the key points that were supposed to be covered in the article. Should the author decide to make changes to the content, the author and editor may want to repeat this phase until both find the result satisfactory.

5 Editing for Style

"The difficulty…is not to write, but to write what you mean, not to affect your reader, but to affect him precisely as you wish." ~ Robert Louis Stevenson

Style is the manner and form that the author uses to communicate. It is a mix of agreed upon conventions for written English, and the author's personal voice. The goal of editing for style is to maintain a consistent tone throughout the document while providing enough variation to keep the reader engaged until the conclusion. It should also address the needs of the reader, filling in background when necessary. Style affects not what is written, but how it is read. This is important because, as Stevenson mentions in his essay, it is the goal of the author not only to lay the information on the page, but also to effect a change in the reader.

The grandfather of all English style manuals for writing is *Elements of Style* by William Strunk and E.B. White (Strunk & White, 2000). This succinct reference covers most of the points to check when editing writing for style. Since Strunk & White have already written a book on the subject, I refer you to their work for more detail, but I would like to review some of the more common concerns.

5.1 Consider Voice and Audience

The two most important style decisions an author can make are:

- How to frame the article for the audience
- What is the proper tone and voice

For technical writing, the audience may seem obvious: people who read the article must be the right audience because they are interested in the content. Unfortunately, this is not always true. Even within an organization, technical readers and business analysts may read the same document, but with differing perspectives. The editor should read the article from both points of view and decide which assumptions may be implicit, and which should be explained. If the title of the article implies a highly technical topic (such as X.509 certificates), the editor must gauge the level of understanding that the reader brings to the table, and decide whether the technical aspects are unique to a department, an organization, or an industry standard.

The author's voice also has an effect on style. A chatty, casual voice may seem more friendly and accessible, but it could undermine the writer's authority on the subject. On the other hand, language that is verbose and legalistic may present an impenetrable hedge that turns away a potential audience. Generally, technical articles avoid using "you" and "I" to talk about the reader and the author. A good compromise is for the author to write using his or her own voice, and the editor can help change the context to a third person if needed.

Even after questioning assumptions about the audience, it is a good idea to define acronyms and to trim jargon from the article. Write out acronyms the first time they are used, with the initials in parentheses. For example, "Test Driven Development (TDD) is a software development process that relies on the repetition of a very short development cycle."   Avoid jargon, or explain it the first time it appears For example, a reference to "Pokémon exception handling" should explain that it means "for when you just gotta catch 'em all, i.e.: all exceptions must be caught."

5.2 Keep the Writing Clean and Consistent

Several other rules improve the quality and clarity of writing, and help provide a consistent style to any article.

- Maintain a consistent verb tense. As Strunk and White say, "Shifting from one tense to the other gives the appearance of uncertainty and irresolution."
- Use the active voice to make writing more dynamic. Passive voice means adding some form of the helping verb "to be" to an otherwise strong verb that did not need the help.
- Put statements in positive form. Rather than writing, "The user should not be able to enter letters or punctuation," it is more clearly stated as "Restrict user entry to numbers."
- Omit needless words

Beware of "junk" words. Seek out and delete filler words like "simply," "very," and other adverbs that unnecessarily modify actions. These words show indecision on the author's part, and removing them adds authority to the work. Encourage the author to use a variety of words. Look for words used more than twice in a paragraph, and suggest alternatives. Web site tools such as WordCounter.com can help quantify how many times an author uses specific words. Use a thesaurus to look for suitable replacements of overused words. *Elements of Style* has a section on commonly misused words and phrases to use as a checklist when editing for style. As Mark Twain says:

> "Substitute 'damn' every time you're inclined to write 'very'; your editor will delete it and the writing will be just as it should be." ~ Mark Twain

5.3 Keep It Interesting

Part of engaging the reader is to provide a variety of sentence structures. In English, authors have pushed the limits of sentence structure. In his novel *Absalom, Absalom!* William Faulkner crafted a sentence 1,288 words long. *The Rotter's Club* is a novel composed of a single sentence of nearly 14,000 words. Hopefully most technical articles will be more restrained, but it is still possible to be interesting with only a few different types of sentences. The *Concise Guide to Technical and Academic Writing* suggests technical writers only need concern themselves with eight types of sentence structures (Bowman, 2012), shown here using Subject-Verb-Object (S-V-O) notation:

1. Simple sentence (S-V-O)
2. Simple sentence with a simple introductory description (D + S-V-O)
3. Compound simple sentence (S-V-O + S-V-O)
4. Compound simple sentence with a simple introductory description (D + S-V-O + S-V-O)
5. Simple sentences with compound predicates (S-V-O + V-O)
6. Simple sentences with compound objects (S-V-O + O)
7. Simple sentence with descriptive phrase for the subject or verb (S+D -V-O; S- V+D -O)
8. Sentence with ending descriptive phrase or clause (Sentence + D)

The editor also needs to be alert for sentence fragments—those that are missing either a subject or a verb. Every complete sentence must have at least one subject, one verb, and expresses a complete thought. Often the grammar checker in a word processor catches sentence fragments, but not always. Occasionally, it is acceptable to use a sentence fragment for an effect, but inadvertent fragments should be avoided. While reading the sentences for structure, it is also a good practice to make sure the verb conjugation agrees with the subject. The goal is to create sentences that are easy to read and clearly convey the message in an engaging way.

5.4 Check for Correct Punctuation

Checking punctuation can be a twisty maze. One website lists sixteen different rules for writing numbers versus numerals. Experienced editors may eat this stuff for lunch, but ad hoc editors will chew up a lot of time looking for reference material. In general, the best solution is to choose a standard reference and use it. If the article is for publication, ask the publisher for a style sheet. Otherwise, two acceptable references are *The Chicago Manual of Style* (Staff, 2010) and *MLA Style Manual and Guide to Scholarly Publishing*. (Modern Language Association, 2009)

Of all the punctuation problems, the apostrophe is the most misused. So many English writers trip themselves up with apostrophes, adding them to words unnecessarily. To keep it simple, remember that the apostrophe has three uses:

- To show possession, such as "This is Mark's house."
- To show contractions, such as "It's a house" for "It is a house"
- To indicate certain plurals of lowercase letters, such as "Mind your p's and q's."

Lynn Truss, author of *Eats, Shoots and Leaves: The Zero Tolerance Approach to Punctuation*, (Truss, 2004) lays bare the distinction:

The word "it's" (with apostrophe) stands for "it is" or "it has." If the word does not stand for "it is" or "it has," then what you require is "its."

Editing for style is a matter of blending the author's voice with the rules for good writing. The process should allow for some flexibility, but the decision to stray from proper grammar should always be a deliberate one with firm reasoning behind it. Strunk says style is a guideline, but he also reminds us that one should know the rules before breaking them, and rules should only be broken when there is a definite benefit.

"The best writers sometimes disregard the rules of rhetoric. When they do so, however, the reader will usually find in the sentence some compensating merit, attained at the cost of the violation. Unless he is certain of doing as well, he will probably do best to follow the rules." (Strunk & White, 2000) Every writer and editor should heed this advice.

5.5 Use a Checklist

Editing requires close attention to detail. Even an efficient editor must make multiple passes through a document. In some cases, the editor will review a document page by page, checking for specific problems such as verifying the footers in all sections, or ensuring that all bulleted items end with a period. A simple way to handle these often-repetitive tasks is to create a checklist.

Checklists can improve the effectiveness of the editor, and reduce the stress of having to remember every single detail of editing. Lists help the editor remember which items need reviewing and provide a baseline for editing the document. Because styles may change from company to company and from one medium to another, it is common to customize checklists for each type of document.

Appendix 1 shows an excerpt from a checklist used by Tiger Heron LLC technical writers when editing printed technical documentation for Dentists Management Corporation (Profeta).

6 Copy Editing

Editing the final copy is paradoxically the simplest and most difficult step. During this phase, the editor will proofread for any errors missed earlier, but often to do that requires the ability to see the article in a new light.

At this point, many people will run a spell check in the word processor and assume the results are correct. Spell checkers work for finding misspelled words and highlighting oddities in grammar, but there are problems with relying solely on them. First, the spell checker may not be entirely correct. Use the advice to find areas where spelling or the construction of a sentence needs further attention. Copying and pasting the entire contents of the article into a second word processor is a way to get a second opinion. Second, spell checkers cannot understand the context of a misspelled word. For example, this sentence escaped spell checking to make it into print: "The programmer was waste deep in system administration." While "waste" may give the sentence an ironic twist, the actual phrase should have been "…waist deep…" Running a spell check is a good first pass, but editors should use other methods to double-check for mistakes.

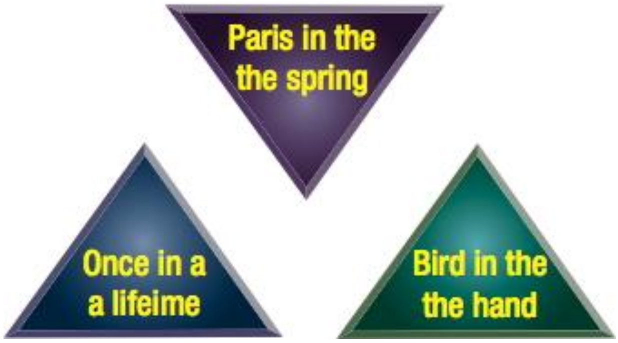


Figure 2 Copy editing requires a different mindset

As mentioned, proofreading requires the editor review the paper from a different perspective. The phrases in Figure 2 may look correct, but you will discover the errors when you read them aloud. A copy editor may be so involved with the content of a document that she is not able to see the small mistakes unless she changes to a new mode of thinking. Here are four ways to get a new perspective:

- Print out the article. If all editing has previously been done on screen, printing out the article changes the mode enough to get your brain to see it differently.
- Read the article aloud. This changes the editing process to engage your ears, as well as your eyes and brain. This method also works for finding awkward phrasing.
- Read the article backwards. Remove the context and content from the process and look at only the grammar.
- Ask a person who has not yet read the article to proofread it.

The printed document is also the best way to find odd formatting problems. Use a red pen or a highlighter to make notes of missing words, odd margins, and alignment problems. This is also the time to make sure all illustrations are readable and properly labeled.

The copy editing pass provides an opportunity to check the format of punctuation. Ensure fonts are used consistently. Check that quotation marks are used correctly. When formatting a sentence with quotation marks, the period generally falls inside the quotes in American English.

Copy editing is time-consuming and tedious, so allow enough time to work through the document without becoming fatigued. Inattention is the enemy of the copy editor, and if you find yourself thinking other thoughts while reading, then you are losing the battle.

Some may ask whether it is worth the results to be concerned with these minor problems. The answer is that the minor problems distract from the content. Similar to hearing an out-of-pitch note during a symphony concert, a typographical error can disrupt the reader’s concentration. People may judge the content by the form, so it is better to proofread as carefully as possible. The goal is to make a good impression in both form and content.

7 Conclusion

Communicating ideas in writing is difficult, but the help of a good editor can make the process more rewarding and effective. The problem is in finding a good editor: not many people have the training or the experience to provide quantitative and qualitative feedback to authors in a constructive way.

This paper has shown that editing is a process that can be broken down into three parts, editing for content, editing for style, and copy editing. Each phase is designed to provide concrete feedback to help the author reach the next level in the process.

In this Internet age, we have many ways to express ourselves and convey information, but it is most commonly as written words. Realizing this makes the service that a good editor can provide even more important.

Bibliography

Bowman, D. (2012). *Concise Guide to Technical and Academic Writing.* Amazon Digital Services, Inc. .

Modern Language Association. (2009). *MLA Handbook for Writers of Research Papers, 7th Edition .* Modern Language Association of America.

Profeta, S. (n.d.). DAISY Master Checklist. Portland, Oregon, USA: Tiger Heron LLC.

Staff, U. o. (2010). *The Chicago Manual of Style, 16th Edition .* Chicago: University Of Chicago Press.

Strunk, W., & White, E. (2000). *The Elements of Style, Fourth Edition.* Penguin Press: Macmillan.

Tardiff, E., & Brizee, A. (n.d.). *Four Main Components for Effective Outlines.* Retrieved 07 31, 2013, from Purdue Online Writing Lab: http://owl.english.purdue.edu/owl/resource/544/1/

Truss, L. (2004). *Eats, Shoots and Leaves: The Zero Tolerance Approach to Punctuation.* New York: Gotham Books.

Appendix 1 – Tiger Heron LLC Checklist

Master Checklist

Document Name____

Done	Branding
	Check that screen shots that contain version numbers have the current version number 4.1.

Done	Pagination
	Check for correct pagination at the beginning and end of each chapter.
	Spot-check three cross-references per chapter, especially to locations in other chapters. If this is a Word document, then check ALL cross-references.
	Check the first and last page number references for each chapter in the main table of contents.
	Verify that chapters begin on right (odd-numbered) pages.
	Scan placement of page breaks throughout the book. Look for page breaks that leave widows or orphans, and for lists or tables that are separated from their lead-in sentences. Other widows and orphans to watch for: ▪ Notes separated from the previous paragraph ▪ Tables that flow to the next page with less than two rows (not counting the header row) ▪ Procedure starting statements separated from procedure steps ▪ A single procedure step (a minimum of 2 procedure steps should be together) ▪ A single line of a paragraph (want a minimum of two lines)
	Verify that no pages or sections are missing.

Done	Chapter titles, headers, and footers
	Check that the wording of chapter titles matches these locations: ▪ In table of contents ▪ In right headers ▪ In left headers ▪ On the first page of the chapter
	Verify footer text is correct in all sections.
	Verify that header and footer lines (if present) line up.
	Check that dot leaders are used consistently in the table of contents.

Done	Procedures
	Check that procedures are numbered in sequence. Word is very bad about this.
	Check that all procedures begin with a “To” statement. MS Word only: Check that all procedures begin with a diamond and a “To” statement.
	Check that all procedure steps end in a period.
	Check that procedure steps are stated the same way, such as “click” rather than “click on,” or ‘select” not “click.”

Done	Notes and Cross References
	Check that the words “Note” “Tip” and “Warning” end in a period.
	Check that the word “Important” ends in an exclamation point (!).
	Check that the words “Note”, “Tip”, “Important” and “Warning” are in Arial Narrow Bold font.
	In cross-references, check that the text drawn from the cross-referenced heading has quotes around it. For example: For more information, see “Deleting a patient” on page 23.

Done	Graphics
	Check graphics for proper placement (from the left margin), and enough space between them and the text that precedes and follows them.
	Check that callouts point to the correct item and are positioned well.
	Check that callout lines are smooth.
	Check that callouts are uniformly indented from the left side of the page.
	Check that the final sentence in the callout text does not end in a period.
	Check that icons within a line of text are lined up well and are of uniform size.

Done	Miscellaneous checks
	Check that the copyright is 2013 wherever it appears.
	Scan for glaring errors and typos.
	Scan for hyphenated words. There should be very few, used only for multiple-word phrases (like this one), and not for splitting long words.
	Scan for consistent use of styles.
	Check the title page for text and correct use of styles.

	Check that web URLS are correct.
	Correct grammar with subject/verb agreement.
	Make sure acronyms are explained with the first usage in a topic.
	Check for proper capitalization.

A Risk-Based Testing (RBT) Approach for the Masses

Bruce Kovalsky, Capgemini

bruce@kovalsky.com

Abstract

Wikipedia defines Risk-Based Testing (RBT) as “a type of software testing that prioritizes the tests of features and functions based on the risk of their failure - a function of their importance and likelihood or impact of failure”. For purposes of this paper, risk related to testing can be identified as the probability that an undetected software defect from a test case will have a negative impact on the user of a system.

Testing teams have a finite amount of time (usually time-boxed) to run their tests. When crafting a strategy of how to address testing risk, organizations should test for the biggest risks first and more often, whether it is a business or technical risk. In other words, what is the minimum testing effort that one should invest in order to maximize risk reduction? It is often difficult to make informed decisions on which tests in a large set of tests planned for a test pass should be focused on -- or should not be focused on.

This paper will describe an easy-to-use process to gather risk-based information about your tests and how to use straightforward metrics in Excel to help quantitatively identify which specific tests should be run earlier and more often, and which tests should not be.

Biography

Bruce Kovalsky, PMP, *has been a Quality Assurance/Test Manager and Automation expert at various Seattle-area companies since 1990, and has been a Testing Consultant since 2002, currently with Capgemini. After receiving a Bachelor’s degree in Computer Science from the University of California at Berkeley, he spent eight years developing software for the Aerospace industry, and then began focusing his career on Quality Assurance and Testing. He has presented papers at several quality conferences over the past 15 years, including the Quality Assurance Institute Conference (1998), Rational User Conference (2000, 2005), and PNSQC (1998, 2011).*

1 Why Risk-Based Testing (RBT) is important

Wikipedia defines Risk-Based Testing (RBT) as “a type of software testing that prioritizes the tests of features and functions based on the risk of their failure - a function of their importance and likelihood or impact of failure”. For purposes of this paper, risk related to testing can be identified as the probability that an undetected software defect from running a test case or test set will have a negative impact on the user of a system.

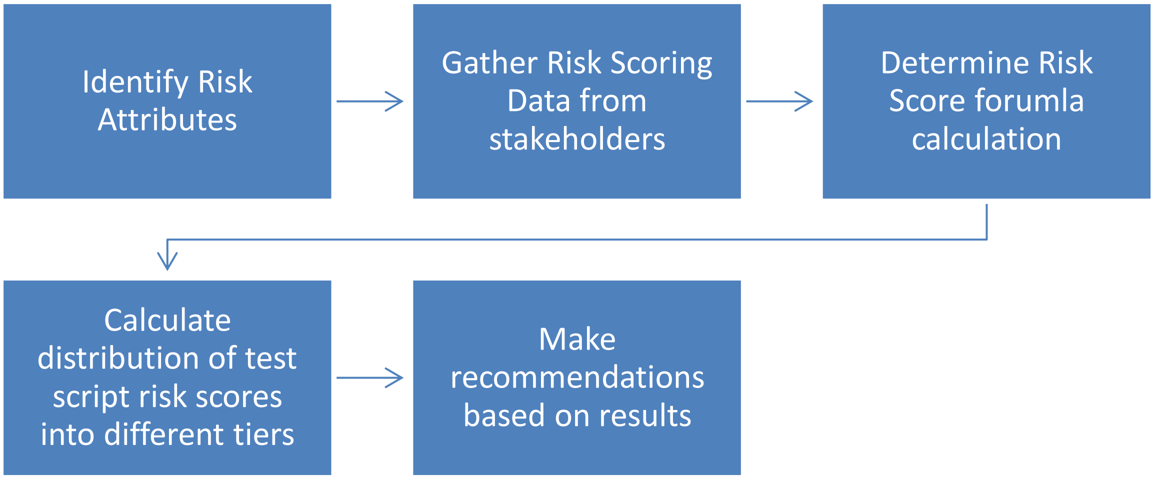
Testing teams have a finite amount of time (usually time-boxed) to run their tests. When crafting a strategy of how to address testing risk, organizations should test for the largest risks first and more often, whether it is a business, technical or other type of risk that is important to their organization.

This question is often asked of testing organizations: What is the minimum amount of testing effort that one should invest in order to maximize risk reduction? In order make informed decisions on which tests planned for a test pass should be focused on, you need an organized approach that will result in allowing your team to making these decisions, which is covered in the following sections.

2 Follow an organized approach to risk based testing

In order to properly assess risk of your testing, following this organized step-by-step approach will allow you to easily make risk-based decisions about your testing:

Figure 1: Risk Based Testing approach steps


Identify Risk Attributes
Gather Risk Scoring Data from stakeholders
Determine Risk Score forumla calculation
Calculate distribution of test script risk scores into different tiers
Make recommendations based on results

2.1 Identify Risk Attributes

The first step is to identify the risk-based scoring attributes that will collectively be used to create a “Risk Score”. The Risk Score will be a sum of individual attribute scores, weighted appropriately as the organization desires; the calculation options for the risk score formula options will be covered in section 2.3.

Examples of types of risk attributes can be across a variety of dimensions:

2.1.1 Business

- Impact or criticality of a subsystem, function or feature failing (business cost of failure)
- High usage or frequency of a subsystem, function or feature

2.1.2 Technical

- Technical impact of failure of a subsystem, function or feature
- Technical probability of failure of a subsystem, function or feature
- Geographic distribution of development team

2.1.3 External Factors

- Sponsor or executive preferences / biases
- Regulatory requirements

2.1.4 E-Business / Failure-Mode Related

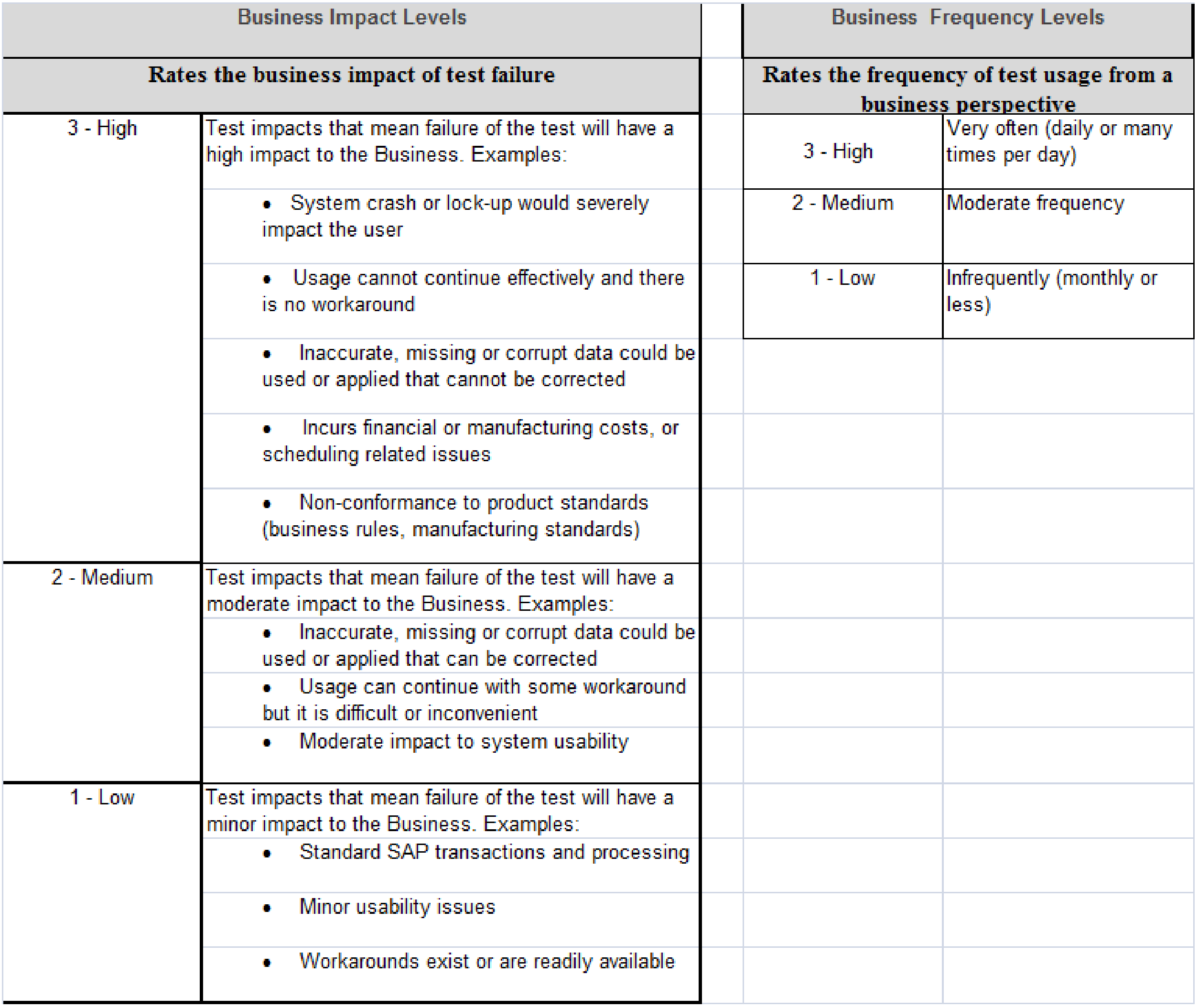
- Static content defects
- Web page integration defects
- Functional behavior-related failure
- Service (Availability and Performance) related failure
- Usability and Accessibility-related failure
- Security vulnerability
- Large Scale Integration failure

Typically, you should pick a mixture of attributes that will make up the Risk Score spread across multiple dimensions that are important to the company, so that each test demonstrates risk impacts across a variety of business, technical or other fronts.

For scoring purposes, a simple 1 to 3 scoring scale should be used for each attribute, where 1 = Low, 2 = Medium, 3 = High, (similar to how testers rate defect severity). You could expand this to a 1 to 4 or 1 to 5 scale, but since the risk formulas will be numerically based on the values and weighting you use for attribute scores, it's important to be consistent on how you apply the scoring values.

Once these attributes are chosen, you should define the criteria that each stakeholder group will use to rate each test attribute so they have a guide for how to score each test (see Figure 2).

Figure 2: Sample Attribute Ratings and Criteria



Business Impact Levels	
Rates the business impact of test failure	
3 - High	Test impacts that mean failure of the test will have a high impact to the Business. Examples:
	• System crash or lock-up would severely impact the user
	• Usage cannot continue effectively and there is no workaround
	• Inaccurate, missing or corrupt data could be used or applied that cannot be corrected
	• Incurs financial or manufacturing costs, or scheduling related issues
	• Non-conformance to product standards (business rules, manufacturing standards)
2 - Medium	Test impacts that mean failure of the test will have a moderate impact to the Business. Examples:
	• Inaccurate, missing or corrupt data could be used or applied that can be corrected
	• Usage can continue with some workaround but it is difficult or inconvenient
	• Moderate impact to system usability
1 - Low	Test impacts that mean failure of the test will have a minor impact to the Business. Examples:
	• Standard SAP transactions and processing
	• Minor usability issues
	• Workarounds exist or are readily available

Business Frequency Levels	
Rates the frequency of test usage from a business perspective	
3 - High	Very often (daily or many times per day)
2 - Medium	Moderate frequency
1 - Low	Infrequently (monthly or less)

As figure 2 shows, you should give criteria guidelines to those who will be scoring each attribute on what each number 1 to 3 means for the specific attribute. This will provide a guide for how each stakeholder will give ratings for the attributes that they will be scoring.

Any number of attributes could be used for RBT scoring purposes, since a formula will be used that weights each of the attributes chosen to arrive at a risk score. Since variety across multiple risk categories is desired, I recommend choosing a total of four attributes, across Business, Technical or other dimensions. For purposes of this paper, I chose the following four attributes to calculate the Risk Score, two in each group:

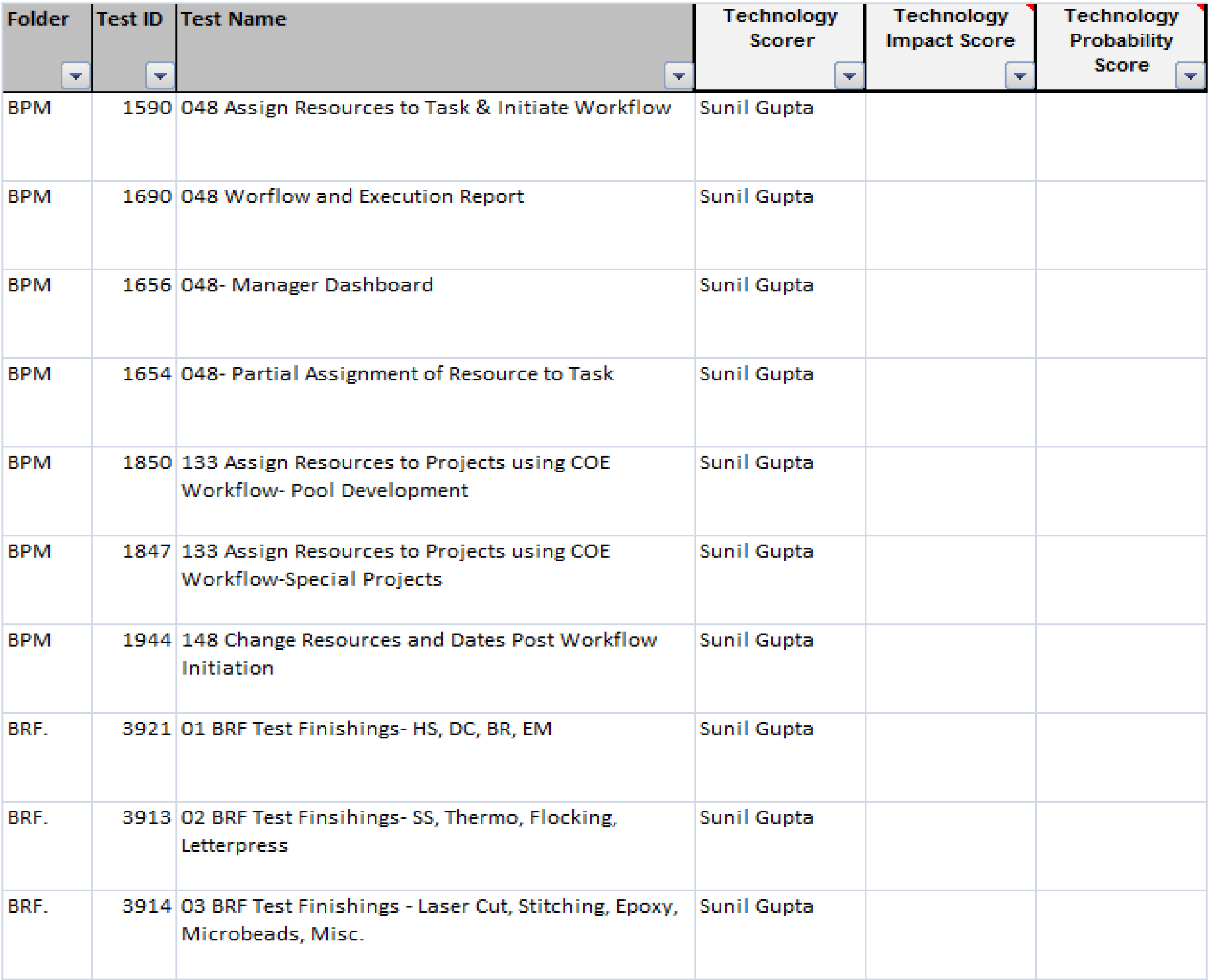
Business	**Technical**
Business Impact	**Technical Complexity**
Business Frequency	**Technical Impact**

2.2 Gather Risk Scoring Data from stakeholders

The next step is to ask the appropriate stakeholders to score each attribute you defined for all tests in your test set according to the scoring system and criteria you have established.

A simple method for this is to extract all of you test names out of your test case management tool (e.g. HP Quality Center/ALM or IBM Rational TestManager), into Excel, and then add the attributes that you choose to include for scoring. Send the spreadsheet with blank values for them to fill out independently (see Figure 3a).

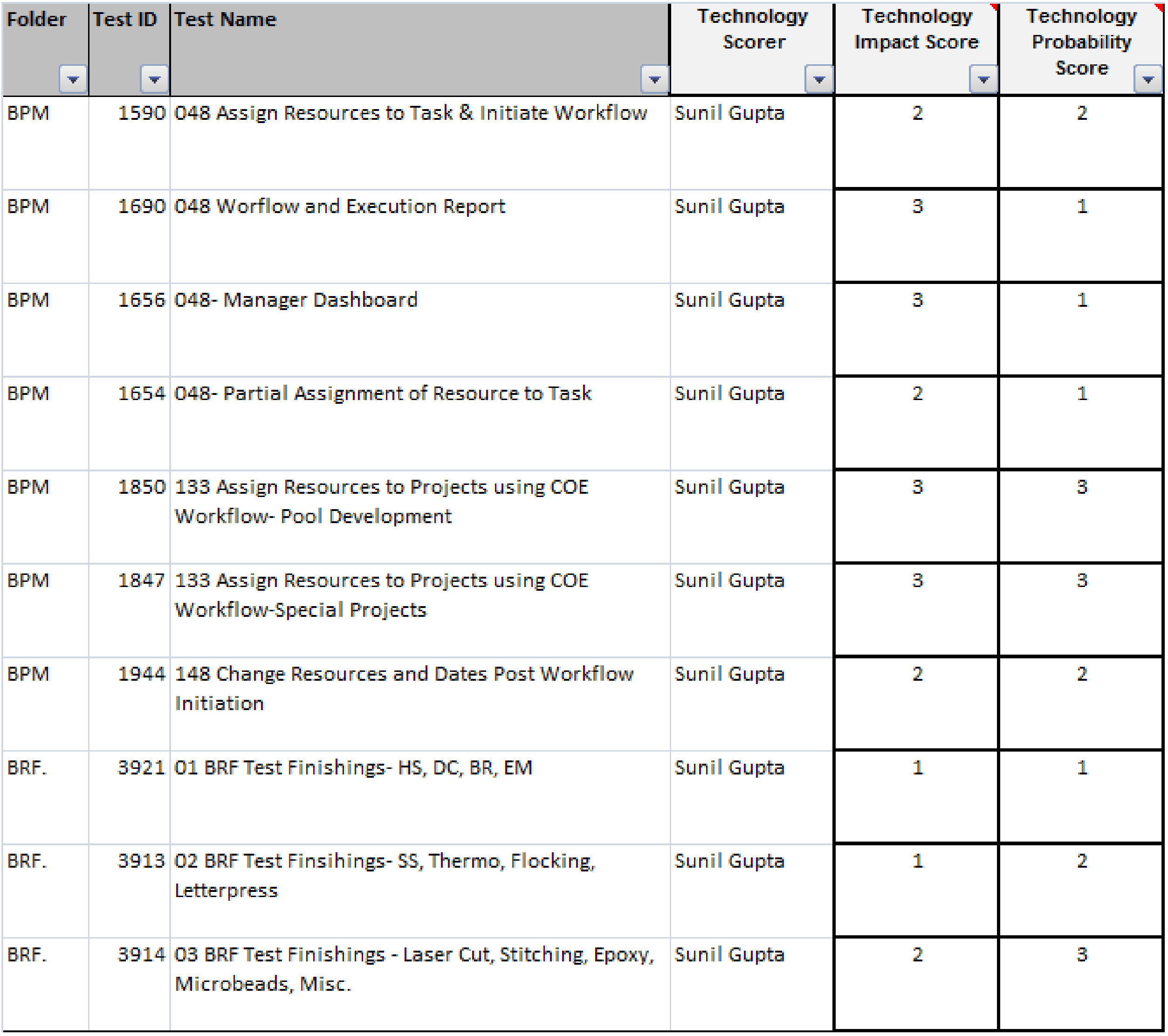
Figure 3a: Sample Attribute Scoring Data - Before



Folder	Test ID	Test Name	Technology Scorer	Technology Impact Score	Technology Probability Score
BPM	1590	048 Assign Resources to Task & Initiate Workflow	Sunil Gupta		
BPM	1690	048 Worflow and Execution Report	Sunil Gupta		
BPM	1656	048- Manager Dashboard	Sunil Gupta		
BPM	1654	048- Partial Assignment of Resource to Task	Sunil Gupta		
BPM	1850	133 Assign Resources to Projects using COE Workflow- Pool Development	Sunil Gupta		
BPM	1847	133 Assign Resources to Projects using COE Workflow-Special Projects	Sunil Gupta		
BPM	1944	148 Change Resources and Dates Post Workflow Initiation	Sunil Gupta		
BRF.	3921	01 BRF Test Finishings- HS, DC, BR, EM	Sunil Gupta		
BRF.	3913	02 BRF Test Finsihings- SS, Thermo, Flocking, Letterpress	Sunil Gupta		
BRF.	3914	03 BRF Test Finishings - Laser Cut, Stitching, Epoxy, Microbeads, Misc.	Sunil Gupta		

See Figure 3b for a sample of a spreadsheet that was filled in for the two technology scores by the developer that was asked, after receiving the spreadsheet in Figure 3a.

Figure 3b: Sample Attribute Scoring Data - After



Folder	Test ID	Test Name	Technology Scorer	Technology Impact Score	Technology Probability Score
BPM	1590	048 Assign Resources to Task & Initiate Workflow	Sunil Gupta	2	2
BPM	1690	048 Worflow and Execution Report	Sunil Gupta	3	1
BPM	1656	048- Manager Dashboard	Sunil Gupta	3	1
BPM	1654	048- Partial Assignment of Resource to Task	Sunil Gupta	2	1
BPM	1850	133 Assign Resources to Projects using COE Workflow- Pool Development	Sunil Gupta	3	3
BPM	1847	133 Assign Resources to Projects using COE Workflow-Special Projects	Sunil Gupta	3	3
BPM	1944	148 Change Resources and Dates Post Workflow Initiation	Sunil Gupta	2	2
BRF.	3921	01 BRF Test Finishings- HS, DC, BR, EM	Sunil Gupta	1	1
BRF.	3913	02 BRF Test Finsihings- SS, Thermo, Flocking, Letterpress	Sunil Gupta	1	2
BRF.	3914	03 BRF Test Finishings - Laser Cut, Stitching, Epoxy, Microbeads, Misc.	Sunil Gupta	2	3

If you have many different stakeholders that are responsible or SME's for different functional areas, a good practice is to separate each set of tests according to area of expertise in scoring when sending them the tests. For example, there could be different developers that score test cases based on the functional areas that they are developing or fixing defects for, in which case you would send spreadsheets to each developer only with test rows only that they are responsible for filling in.

As each group of risk attribute scorers enters their scores for the attributes you have defined and sends them back to you, roll them all up into a single Excel spreadsheet that collects all risk scores for all tests and all attributes.

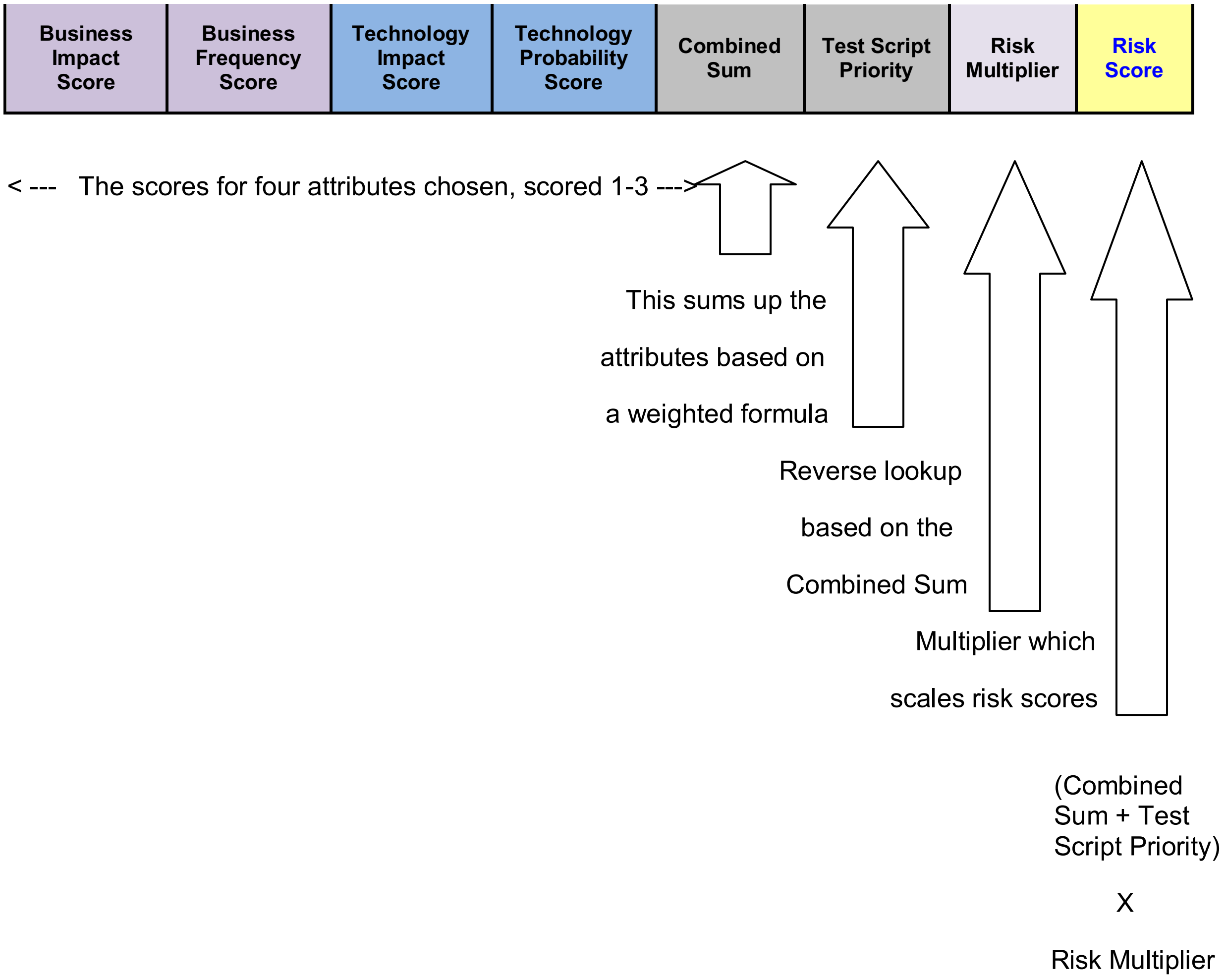
2.3 Determine Risk Score formula calculation

At this point, you have gathered raw attribute scoring data for each attribute that you have defined initially, for all test cases. Next, determine the appropriate formula to be used to calculate your risk score.

The ideal technique is for the total collection of Risk Scores for each test to be distributed over a range of values, so that they can be fit into High, Medium and Low tiers that will allow you to make decisions on which tests are High tier vs. Low Tier.

The Risk Score is based on the following variables shown in figure 4 below:

Figure 4: Makeup of the Risk Score


Business Impact Score
Business Frequency Score
Technology Impact Score
Technology Probability Score
Combined Sum
Test Script Priority
Risk Multiplier
Risk Score
< --- The scores for four attributes chosen, scored 1-3 --->
This sums up the attributes based on a weighted formula
Reverse lookup based on the Combined Sum
Multiplier which scales risk scores
(Combined Sum + Test Script Priority)
X
Risk Multiplier

Assuming you have four attributes chosen at the bottom of section 2.1, some different approaches to calculate the Combined Sum formula (which directly drives the Risk Score formula) are:

2.3.1 Evenly weight the attributes

For this method, you simply weight each of the attributes the same, and sum up the values. So the Combined Sum formula would be:

= (Business Impact + Business Frequency)/2 + (Technology Impact + Technology Probability)/2

2.3.2 Weight one or more attributes higher than others

In this case, you would value one or more attributes some percentage higher than others so that scores for that attribute would have more weight that the others when the final Risk Score is calculated. For example, let's say the business team feels that the Business Impact score should have 50% more weight than the Business Frequency score, and keep the Technology scores the same weight. In this case, the Combined Sum formula would be:

= ((1.5*Business Impact) + Business Frequency)/2 + (Technology Impact + Technology Probability)/2

Note in the above case, by weighting Business Impact by 1.5 times the Business Frequency Score, the other mathematical impact to the Risk Score is that it gives both of the Business Scores a 25% boost over the Technology Scores.

2.3.3 Make one attribute override all other attributes

In this case, you would value one attribute over all others. For example, if someone scores Business Impact Score as a High (= 3), then the overall Risk Score should be the highest possible score and tier, otherwise, they would all be weighted evenly as in the first case above. For this case, the Risk Score formula would be:

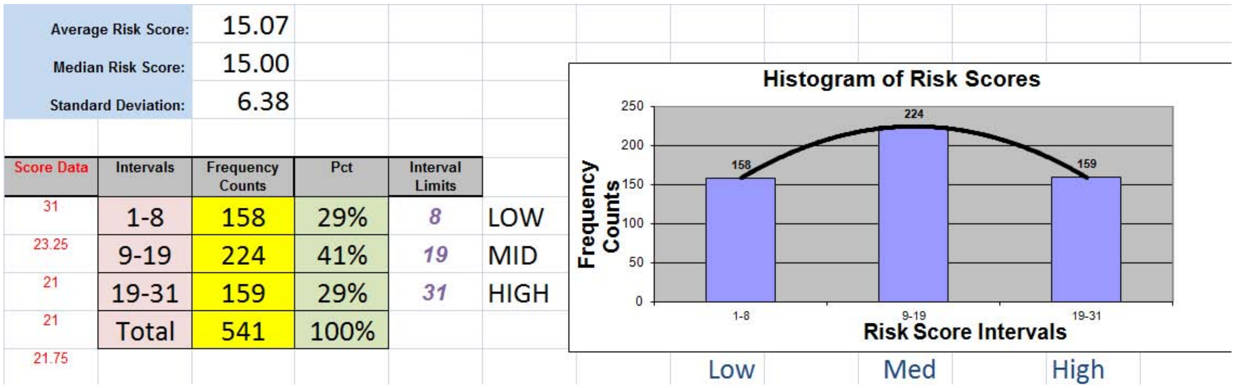
=IF (Business Impact=3, <Highest Score>, SUM (Combined Sum + Test Script Priority) * (Risk Multiplier))

Other variations on the formulas are possible, but the above three are all suggested ways you can weight your formula to arrive at the optimal Risk Score distribution for you organization.

2.4 Calculate distribution of test script risk scores across different tiers

After all the initial scoring data has been collected, use a spreadsheet with formulas to calculate the distribution of the risk scores into different tiers (e.g., High, Medium, Low) which can be set by changing the interval limits that the risk scores are distributed over (see Figure 5).

Figure 5: Distribution of Risk Scores into Tiers and Histogram



Average Risk Score:	15.07
Median Risk Score:	15.00
Standard Deviation:	6.38

Score Data	Intervals	Frequency Counts	Pct	Interval Limits	
31	1-8	158	29%	8	LOW
23.25	9-19	224	41%	19	MID
21	19-31	159	29%	31	HIGH
21	Total	541	100%		
21.75					

By using filters in Excel, you should review how all of the Risk Score data distributes based on the formula used, and set the Interval Limits appropriately to allow the data to fall into different tiers.

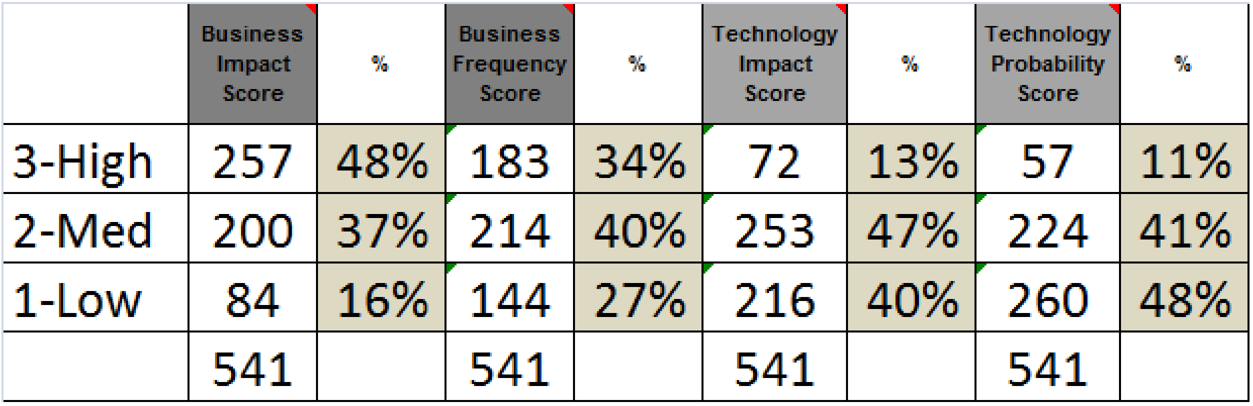
The formula to use for the Frequency Counts, which counts the number of tests that fall into the interval range specified, is the following:

= FREQUENCY (Score start range : Score end range, Interval Limit start : Interval Limit end)

By observing the distribution of risk scores, you can modify the Interval Limits to create different intervals, to obtain the desired tier distribution. Optimally, the distribution should try to emulate a Normal distribution (Bell Curve-like), but any rough approximation of this shape is acceptable. The important theme is you don't want too many tests falling in the High Tier, since your organization can only focus on so many high risk tests. By changing the formulas in section 2.4, and adjustment the tier intervals, you can achieve the best possible tier distribution, as desired.

Another useful method is to show tables on how each attribute was scored individually, so you can observe the raw data of how the Risk Scores were derived; see Figure 6 for a sample of how this is gathered.

Figure 6: Separate Attribute scoring data tables

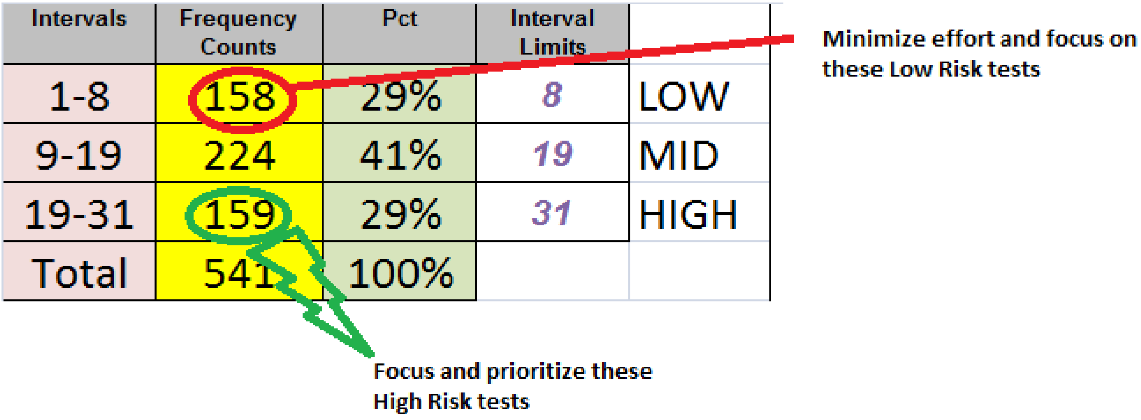


	Business Impact Score	%	Business Frequency Score	%	Technology Impact Score	%	Technology Probability Score	%
3-High	257	48%	183	34%	72	13%	57	11%
2-Med	200	37%	214	40%	253	47%	224	41%
1-Low	84	16%	144	27%	216	40%	260	48%
	541		541		541		541	

2.5 Make recommendations based on results

Once you have finalized the Risk Score formula and have a Tier Distribution that is accurate, decisions can then be made on which tests should be run earlier and more often (High Tier tests), and which ones should not be focused on (Low Tier tests), as shown in Figure 7.

Figure 7: Risk Score Distribution focus



Intervals	Frequency Counts	Pct	Interval Limits	
1-8	158	29%	8	LOW
9-19	224	41%	19	MID
19-31	159	29%	31	HIGH
Total	541	100%		

Minimize effort and focus on these Low Risk tests

Focus and prioritize these High Risk tests

As the figure emphasizes, use the results of how the risk scores were distributed into the High, Medium and Low tiers to determine which tests to focus on, and not focus on for your testing efforts going forward.

3 Risk Based Testing Approach Summary

Software testing as an exercise helps answer the basic question of when an organization should release their software to production at the highest quality with the least amount of risk. At some point in your testing career, management will inevitably ask your testing organization the question:

What is the minimum amount of testing effort that we can do now in order to maximize risk reduction?

By following the Risk Based Testing steps described in section 2 in an organized process, you will be able to easily quantify and categorize any set of existing tests to be run across easily identifiable risk tiers, according to the customizable testing risk attributes your organization chooses

Once your tests have been broken down into High, Medium and Low risk tiers, the decisions to make of what to test first, with more resources, and more often becomes simplified.

4 References

Web Sites:

Wikipedia definition of Risk Based Testing:

http://en.wikipedia.org/wiki/Risk-based_Testing

Wikipedia definition of Normal distribution:

http://en.wikipedia.org/wiki/Normal_distribution

Andy Tinkham, Modern Apps Live (MAL), QAT and Automated Testing of Modern Apps

http://www.slideshare.net/andytinkham/mal12-qa-tandautomatedtesting

Stephane Besson, A Strategy for Risk Based Testing

http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=ART&ObjectId=7566&tth=DYN&tt=siteemail&iDyn=2

Sample RBT Excel Spreadsheets:

GoogleDocs link to PNSQC folder

Some sample RBT spreadsheets will be uploaded for attendees to download and use for their efforts using different formulas I cover during the presentation.

How Did I Miss That Bug?

Peter Varhol and Gerie Owen

peter@petervarhol.com; gerie@gerieowen.com

Abstract

How many bugs have you missed that were clearly easy to spot?

Testers approach all phases of testing hampered by their own biases in what to look for, how to go about setting up and executing tests, and how to interpret the results. These biases cause less than outstanding testing performance and data interpretation in an environment where test teams expect results to be cut and dried. The data is good by itself, but what we measure, and what decisions we make about the data, are driven by our own understandings and misunderstandings of the project and the goals of testing.

This paper will give testers and their managers an understanding of how their own mindsets and biases influence their testing. Testers and test managers will learn how to use this understanding to develop ways of managing their thinking and improving their testing. Using principles from the social sciences, such as Kahneman's[7] framework for critical thinking and Chabris and Simons [2] findings on attention, perception and memory, we will demonstrate to you that you aren't as smart as you think.

We'll show you how to improve your test results by understanding how you think and the role that biases play in testing. We'll discuss the balance between scripted and exploratory testing and how to use each effectively, and we'll show how test managers can help their teams maintain their focus individually and as a team. Finally, we'll provide tips for managing your biases and focusing your attention in the right places throughout the test process so you won't miss that obvious bug.

Biography

Peter Varhol is a well-known writer and speaker on software and technology topics, having authored dozens of articles and spoken at a number of industry conferences and webcasts. He has advanced degrees in computer science, applied mathematics, and psychology, and is currently TestStudio Evangelist at Telerik. His past roles include technology journalist, software product manager, software developer, and university professor.

Gerie Owen is a Quality Assurance Consultant who specializes in developing and managing test teams. She has implemented various Quality Assurance methodology models, and has developed, trained and mentored new teams from their inception. Gerie manages large, complex projects involving multiple applications, coordinates test teams across multiple time zones and delivers high quality projects on time and within budget. In her everyday QA Life, Gerie brings a cohesive team approach to testing. She has also presented at several conferences and authored articles on testing and quality assurance.

1. Introduction

How did I miss that bug? We have often asked ourselves and our teams to consider that question. Software testers frequently miss bugs that are clearly easy to spot. This costs organizations financially, and negatively impacts an organization's reputation when these bugs escape into production. Testers approach all phases of testing hampered by their own biases in what look for, how to go about designing and executing tests, and interpreting results. These biases can result in incomplete testing and incorrect data interpretation in business environments where the results must be correct. The data is good by itself, but what we measure, and what decisions we make on the data, are driven by our own understandings and misunderstandings of the project and the goals of testing.

Daniel Kahneman and Amos Tversky [6] developed the idea of cognitive bias as a pattern of deviation in judgment from their research, which showed people's inability to think critically in complicated situations. People tend to use heuristics, or rules of thumb, to make decisions when the subject matter is complicated or when time is limited. Heuristics are usually focused on one aspect of the problem while ignoring others.

Our preconceived notions as testers about the application under test, the requirements, and the skill of the developers all affect our testing. Some of the biases that especially impact testers include the representative bias, the curse of knowledge, the congruence bias, the confirmation bias, the planning fallacy, and anchoring. We will discuss these in detail a little later.

What is going on when we miss the obvious bugs, the ones that are literally staring us in the face? This, too, is likely attributable to a bias — inattentional blindness, a psychological lack of attention, not associated with any vision deficits. Christopher Chabris and Daniel Simons [2] demonstrated this bias in their famous invisible gorilla test, in which subjects were so fixated on a specific task that many didn't see a person in a gorilla suit run across the screen. As testers, we become so focused on executing our test cases that we sometimes fail to see the obvious bug.

So how do we use the concepts of bias and preconceived notions to become top performing testers? Rather than allowing our biases to hamper our testing and cause us to miss bugs, we should plan and execute our testing in recognition of the fact that we do have these biases and preconceived notions. We should add additional time to our estimates and include negative test scenarios in our test planning. Although we "know" that one developer's code is "always" full of bugs, and the other developer's code "never" has bugs, we should not execute fewer tests on the presumably better developer's code.

As testers, we can help each other prevent biased testing by peer reviewing test results or running each other's test cases. In addition to executing our test cases, we should perform exploratory testing. Exploratory testing prior to running our test cases is especially useful because we have not yet made any assessments, and potentially developed biases, about the quality of the product.

How do we make sure we see the "Invisible Gorilla"? As testers, we need to approach our testing holistically; we need to focus our attention on determining if this is a quality product as opposed to just tracing our test cases to requirements and executing all of our test cases.

2. Relevant Research

In order to understand how we miss bugs, we must understand how we test. What is software testing? In its most basic form, software testing is making judgments about the quality of the software under test. It is a very complex task, involving not only objective comparisons of code to specifications but also subjective assessments regarding usability, functionality etc. It follows then that failure to find a particular bug can be viewed as an error in judgment by the tester. Therefore, to determine how testers miss bugs, we need to understand how humans make judgments, especially in complex situations. Here is a review of the most relevant research.

In “Maps of Bounded Rationality: Psychology for Behavioral Economics,” Daniel Kahneman [3] provided a model of bounded rationality. When people need to make a decision in a complex situation, they tend to create boundaries or condense the issue into something that they can handle within their thought processes. In his model of bounded rationality, Kahneman categorizes our thought processes into distinct components, specifically, System 1 and System 2.

- System 1 thinking is intuitive, quick and emotional and is applied during our initial reactions to situations.
- System 2 thinking relies on reason, rules and objectivity and is applied when more analysis is needed, for example when calculating the answer to a math problem.

System 1 and System 2 thinking often can be in conflict and lead to biases in decision-making.

In “Judgment Under Uncertainty: Heuristics and Biases,” Kahneman and Tversky [6] examine three heuristics that people use to make judgments. Heuristics are guidelines or “rules of thumb” that people use to make decisions quickly, especially in complicated situations. The authors describe how using heuristics can lead to predispositions in our decision-making. These heuristics include:

- Representativeness
- Availability of instances or scenarios, and
- Adjustment from anchor

These heuristics are used to explain many systematic biases in judgment in complex situations.

We use representativeness when we compare current situations to previous situations. Sometimes what has happened in the past is a good guide to future decisions, but we may not realize that the situations weren’t similar enough to make the same decisions.

We use availability of instances when we make decisions based on information from our own recent experiences or those of others. For example, we may make health and lifestyle decisions based on how many of our relatives have died of strokes.

We use the anchor bias when we compare to what we are given as standard, for example manufacturers’ suggested retail price. The anchor tends to invite a comparison with other values when we should be making a more informed judgment. Just because our last project had a certain number of defects, for example, we shouldn’t use that as a standard for judging the quality of future projects.

In “The Invisible Gorilla: How Our Intuitions Deceive Us,” Chabris and Simon [2] describe their famous “invisible gorilla” experiment that showed how focused System 2 thinking could cause the inattentional blindness bias. Furthermore, they showed that people could actually be blind to their own blindness. In the experiment, subjects were asked watch a video of a basketball game and count the number of passes between players. During the game, a gorilla walked across the basketball court. Over 50% of the subjects were so focused on counting the passes that they failed to see the gorilla!

3. Prevalent Biases in Software Testing

There are several biases that most frequently influence testers, and specifically impair the ability of the tester to find bugs. Let’s take a look at some of those biases.

3.1 Representative Bias

The representative bias is used when we judge the likelihood of an occurrence in a particular situation by how closely the situation resembles similar situations. Testers may be influenced by this bias when designing data matrices, perhaps not testing data in all states or not testing enough types of data. For example, a tester might decide that if the code works for a new order and an order in process, why test it with a completed order?

3.2 The Curse of Knowledge

The curse of knowledge is when we are so knowledgeable about our subject matter that our ability to address it from a less informed, more neutral perspective is diminished. This affects software testers when they develop so much domain knowledge that they fail to test from the perspective of a new or novice user. Usability bugs are often missed due to this bias.

3.3 The Congruence Bias

The congruence bias is the tendency of experimenters to plan and execute tests on just their own hypotheses without considering alternative hypotheses. In software testing, this bias is often the root cause of missed negative test cases. Testers write test cases to validate that the functionality works according to the specifications and neglect to validate that the functionality doesn't work in ways that it should not. For example, if the specification is that a field should accept only alpha characters, the tester must also validate that the field does not accept numeric. If a tester does not create a test case for this, the congruence bias may be why this test case was missed.

3.4 The Confirmation Bias

The confirmation bias is the tendency to search for and interpret information in a way that confirms initial perceptions. In software testing, testers' initial perceptions of the quality of code, the quality of the requirements and the capabilities of developers can impact the ways in which they test. For example, testers may test a less experienced developer's code more thoroughly than a more experienced developer because finding more bugs in the less experienced developer's code will confirm the tester's expectations.

3.5 The Planning Fallacy

The planning fallacy is the tendency to underestimate how long it will take to complete tasks. We are all optimists in that regard. Software testers often fall prey to this bias when estimating test planning tasks and test execution cycles. Often, test managers will plan or agree to a schedule that barely allows time to complete all the test cases. When unexpected issues such as late code deliveries or environmental or access issues arise, the test execution falls behind schedule. Risk based testing may be implemented; testers may be rushed or encouraged to work overtime resulting in missed bugs.

3.6 The Anchoring Effect

The anchoring effect is the tendency to become fixated and rely too heavily on a piece of information which may cause us to reject other ideas or evidence that contradicts the initial information. Software testers do this often when they validate code to specifications exclusively without considering ambiguities or errors in the requirements.

3.7 Inattentional Blindness

Inattentional Blindness is the tendency to miss obvious inconsistencies when focusing specifically on a particular task or feature set. This happens in software testing when testers miss the blatantly obvious bugs like misspellings while concentrating on executing their test cases. Another example is when testers are so focused on validating the bugs delivered in a release that they miss regression defects.

Gerie did this. She was testing a Smart Grid application and the associated devices including programmable thermostats and load control devices. When a “critical event” was input, the devices were show a red signal. When Gerie tested a new release, she totally missed that the critical event didn’t activate the red signal and change the pricing because she was so focused on verifying the bugs fixes turned over in the release.

4. How biases and preconceived notions negatively impact software testing

Because test managers and testers are affected by so many biases and preconceived notions, this can negatively impact the effectiveness of the entire test process and decrease defect removal efficiency. When software testers, influenced by the planning fallacy, underestimate the time for the test process, the test process begins already compromised. In the previous section, we discussed the ways in which biases influence individual testers. In this section, we will examine how these biases impact the test process. In this paper, we review the test process within the waterfall methodology systems development lifecycle, however; the same biases apply to the test process within an agile framework.

4.1 Requirements Development Phase

In the requirements phase of the System’s development lifecycle, software testers commonly perform ambiguity reviews on the requirements documents. As testers perform their ambiguity reviews, preconceived notions about the design of the application may affect their ability to spot ambiguities. The **curse of knowledge** may come into play especially during requirements reviews for upgrade projects. Having already tested previous releases of the application, the testers may have become subject matter experts and miss obvious ambiguities. Also, testers may make assumptions about what is not specified as a requirement. Without realizing it, testers may fill in the missing pieces and their completions may not be correct.

4.2 Test Planning Phase

Test planning may be negatively impacted when testers are influenced by the **confirmation**, **representative** and **congruence** biases. When testers develop test cases according to their initial impressions of the application under test and the project in general, they are being influenced by the confirmation bias.

When test data matrices are being designed, the **representative** bias comes into play. Testers rely on their experiences with past projects when they make judgments when determining the data types, data states and integration points that should be included in the data matrix.

When testers miss negative test cases, it may be that the **congruence** bias is at work. If software works as we expect, we sometimes fail to consider circumstances where it shouldn’t work that way.

4.3 Test Execution Phase

When testers concentrate on executing their test scripts and test managers concentrate on test coverage, they become too focused and miss obvious defects. When schedules slip and risk-based testing is implemented, testers attempt to test what they, the project team or the stakeholders deem to be the most critical functionality with the highest risk. In doing so, they perform minimal exploratory testing and miss potentially critical defects. These misses may be attributed to **inattentional blindness** and **anchoring**.

4.4 Project Closure

In the project closure phase, testers have an opportunity to analyze what parts of the test process did and didn’t work well. In this phase, root cause analysis may be used to determine why certain bugs were not found in testing. The root cause analysis includes *why* bugs were missed, however, we don't usually focus on *how*. If we focus on *how*, we miss bugs, we may find that testers' biases had an impact.

5. Why we aren't as smart as we think; how we develop biases and preconceived notions

So why is it that we aren't as smart as we think? The works of Kahneman and Tversky,[6] Ariely, [1] and Chabris and Simon [2] have shown that we aren't as smart as we think because of our bounded rationality which is defined as our need to use heuristics and frameworks when faced having to make complex decisions.

But why is it that we develop biases? Does our level of intelligence matter? Surprisingly enough, West, Meserve and Stanovich [4] at James Madison University showed that the more intelligent people are, the more susceptible they are to biases. Intelligence matters, but in itself, does not explain why we develop biases.

Is it because we don't realize that we have biases and preconceived notions and that we are employing them in our decision-making process? In "*Thinking Fast and Slow,*" Kahneman [7] admits that all of his research on the topic has not enabled him to make unbiased decisions. So although we may or may not realize that we employ biases in our decisions, knowledge of our biases may not change our decision-making approach.

An answer to why we aren't as smart as we think comes from the research of West, Meserve and Stanovich [4] which showed that we can easily identify biases in the decision-making process of others but we neglect to factor in our own biases. We actually have a bias about biases! West et al. [4] called this the bias blind spot. We evaluate our own decision-making process differently than we evaluate how others make decisions.

6. How to design an approach to effectively manage the way we think during the test process

Now that we understand why we are so susceptible to biases and preconceived notions, how do we effectively manage the way we think throughout the test process? Although the research shows that we cannot prevent biases and preconceived notions through awareness and understanding, we, as individual testers, can attempt to manage our thought processes as we approach our test projects.

In their working paper, "How Can Decision Making Be Improved," Milkman et al. [4] review the work of several researchers which suggests various ways of invoking System 2 thinking (reasoning and objectivity) more heavily than System 1 thinking (intuitive and emotional). These include evaluating multiple options simultaneously rather than accepting or rejecting each option individually, making choices when under less cognitive load and making decisions in groups rather than individually.

Will an increased focus on System 2 thinking improve testers' effectiveness at finding bugs? We believe not. In software testing, our test methodology already focuses us and requires us to use System 2 thinking. Let's review the test process:

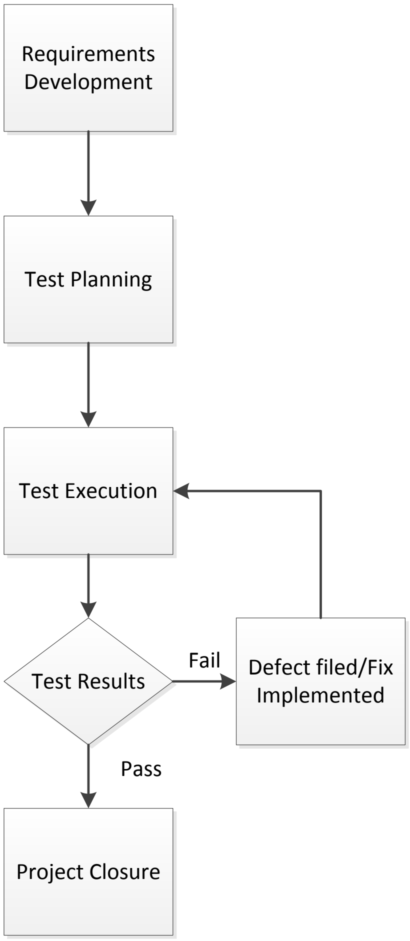


Figure 1. The testing lifecycle.

Requirements coverage, traceability and data matrices, and test case execution metrics are the tools of our trade that are designed to prevent us from missing bugs. These tools and processes are very useful in validating that the code is built to specifications and that all requirements have been coded and tested. Test methodology is the analytical framework of testing; it naturally invokes our System 2 thinking and places the tester under cognitive load. Haven't you left work exhausted after a day of writing or executing test cases?

We use the functional specifications to determine how the code should work and write test cases based on what should be the expected results. If we don't have functional specifications, we use business requirements, workflow training documents or even conversations with customers, developers or Project Management to develop our expected results. Then we compare the functionality of the application to our expected results. The determination of whether the actual results match the expected results becomes an objective assessment.

However, does validating that the application functions according to specifications mean that the application is bug free? Does 100 percent test coverage with all test cases passed mean that the application works as the business customer intended? We all know that the answer to both of these questions is not necessarily. Now the question becomes, how do we find the bugs that prevent the application from working as the customer intended. We believe this requires us to refocus on System 1 thinking (intuition).

Heuristics, when used with oracles, are quite valuable in invoking System 2 thinking. For example, if a typical user is our oracle, we might test workflows and find bugs that we might not find by executing our test cases. How many times have you reported a bug to which the developer responds: “The user would never do that!”?

Users make decisions about using applications based on look and feel as well as ease of use. Since users and potential customers use their System 1 thinking to make these decisions, it follows that to test effectively, we, as testers, must also use System 1 thinking and anticipate the emotions of the user. There is no better way to perform this testing than to consider our emotions and what they might be telling us about the application under test. For example if we are feeling frustrated and anxious, perhaps there is a performance or usability issue.

Finally, how can we find the obvious bugs, the ones we miss due to inattentional blindness? An answer becomes clear. At specific times during our testing, we need to focus less. In addition to our test cases, we need to perform exploratory testing and play with the application. We must use our intuition, in a controlled way, and when we see something we can’t believe we are seeing, we need to believe it and test it further.

7. How managers can increase their teams’ effectiveness by managing biases and preconceived notions

While testers can design an approach to manage their own biases and preconceived notions by using oracles and heuristics that employ more System 1 thinking, test managers can increase team effectiveness by fostering an environment in which the testers feel comfortable and empowered to use System 1 thinking.

Kahneman’s [7] research shows that people engage System 1 thinking in situations requiring intense mental effort, when they are emotionally engaged positively or negatively and when they feel empowered. Kahneman’s research [7] also shows that when people lack subject matter knowledge and when they believe in the power of intuition, they are more likely to use System 1 thinking in decision making.

Managers can create this type of an environment in many simple ways. Test schedules should be planned with time allotted to exploratory testing, preferably at the beginning of the test cycle before testers develop preconceived notions regarding the quality of the code.

Managers should provide a safe environment where testers are encouraged to take risks. Rather than requiring that each tester execute a predefined number of test cases each day, testers should be encouraged to test creatively, going beyond the predetermined test suite. Test managers might reward testers based on the importance or criticality of the bugs they find. Testers could be assigned to execute test cases that they did not write as an effective way of offsetting each other’s biases.

8. Summary

The quality assurance profession can help to prevent bugs that are missed due to testers’ biases and preconceived notions by promoting and encouraging the use of System 1 thinking throughout the test process. This may require a paradigm shift to our standard test methodology.

1. Testers need to understand that bias plays a large part in what we test and what we do not, and therefore what we miss, including those “obvious” defects. Understanding the types of bias will help us ensure we are considering more possibilities.

2. Management should not insist solely on automation/scripting to find the obvious bugs. Using exploratory testing is a valuable and necessary part of any testing function that wants to deliver a high quality product.

3. As an industry, we must shift our focus from requirements coverage based test execution to a more intuitive approach. Exploratory testing and business process flow testing should become the norm rather than the exception. To do this, we should experiment with new testing frameworks where risk-based testing is executed through targeted exploratory testing and is balanced with scripted testing to achieve a more effective approach to quality.

.

References

1. Ariely, D., 2009. *Predictably Irrational: The hidden forces that shape our decisions*. New York: Harper Colins Publishers.

2. Chabris, C and Daniel Simons, 2010. *The Invisible Gorilla: How Our Intuitions Deceive Us.* New York: Crown Publishers.

3. Kahneman, D. (2003). *Maps of Bounded Rationality: Psychology for Behavioral Economics*. The American Economic Review, 93(5), 1449-1475.

4. West RF, Meserve RJ, Stanovich KE , *Cognitive sophistication does not attenuate the bias blind spot.* J Pers Soc Psychol. 2012 Sep;103(3):506-19.

5. Milkman K, Clough, D., & Bazerman, M. (2008) *How Can Decision Making Be Improved?* Harvard Business School, 2008.

6. Tversky, Amos, and Daniel Kahneman. *Judgment under Uncertainty: Heuristics and Biases.* Science, New Series, Vol. 185, No. 4157. (Sep. 27, 1974), pp. 1124-1131.

7. Kahneman, Daniel. Thinking, Fast and Slow. MacMillan, 2011.

Towards Understanding and Improving Mobile User Experience

Philip Lew and Luis Olsina

philip.lew@xbosoft.com, olsinal@ing.unlpam.edu.ar

Abstract

Design and evaluation of traditional web applications cannot account for the particular features and usage contexts of mobile applications (MobileApps). MobileApps have several characteristics that pose challenges in their design and evaluation regarding current quality models and their included characteristics and sub-characteristics. For instance, user interface operability has a much different and greater influence when evaluating MobileApp usability and user experience due to the context of the user. Characteristics such as multi-touch gestures, button size, and widget usage have a magnified impact on task completion rates.

In this paper, we propose utilizing our previously developed ISO 25010-based quality models and framework so-called 2Q2U (*Quality*, *Quality in use*, *actual Usability* and *User experience*) as a basis to understand, evaluate and improve MobileApp user experience. Specific MobileApp task screens and attributes are illustrated in order to show our evaluation approach applicability.

Biography

After working in various management and technical positions in software product development and product management, today, Phil leads XBOSoft's (www.xbosoft.com) direction and strategy as their CEO. His Ph.D. research in software quality and usability resulted in several IEEE and ACM journal publications and he has been published in various trade journals as well. He has presented at several conferences including Software Test Professionals 2012, and the International Conference of Web Engineering-2009-10-11 on web application usability, user experience, and quality evaluation. In the past 20 years, he has helped hundreds of organizations assess the quality of their software, examine software quality processes and set forth measurement plans so that they can consistently improve software quality using systematic methods. He received his B.S. and Master of Engineering degrees in Operations Research from Cornell University. His current post-doctorate research areas are focused on user experience measurement frameworks and software learnability.

Luis Olsina is Full Professor in the Engineering School at the National University of La Pampa, Argentina, and heads the Software and Web Engineering R&D group. His research interests include software and web engineering, particularly, quality requirements modeling, web quality strategies, quality improvement, measurement and evaluation processes, evaluation methods and tools, and domain ontologies. He earned a PhD in the area of software engineering and a MSE from National University of La Plata, Argentina. In the last 17 years, he has published over 100 refereed papers, and participated in numerous regional and international events both as program committee chair and member. Recently, Luis and his colleagues have co-edited the book titled Web Engineering: Modeling and Implementing Web Applications.

1. Introduction

Web applications (WebApps), have become the most predominant form of software delivery today with users and businesses choosing to rent software rather than buy it. The concept of renting versus buying, and the availability of free trials and ease of switching software has led to increased and focused attention on WebApp quality and software quality in general. Users (customers) cannot easily switch and are not tied in with a large capital expenditure. Software quality models in conjunction with an evaluation strategy facilitate understanding, recommending, and especially improving their quality.

With respect to software quality models, ISO 25010 (ISO/IEC 25010, 2011) outlines a flexible model with product/system quality –also known as internal and external quality (EQ)-, and system-in-use quality –also referred to as quality in use (QinU). Product quality consists of those characteristics that can be evaluated in early development stages, for instance, design documents, code quality, etc., while system quality consists of those characteristics and attributes that can be evaluated in late stages, with the application in execution state. On the other hand, system-in-use quality consists of characteristics as evaluated by an end user when actually executing application tasks in a real context. An example would be a nurse or a doctor entering patient record and diagnosis information into an electronic health records system. A doctor even while doing the same task, will have different error and completion rates than a nurse. Doctors may also take longer and have less efficiency in completing tasks simply because they don't do as many. In addition, system QinU is heavily dependent on the context of the task and user. For instance, a user in a dim warehouse doing inventory will have a different viewpoint and user experience than a doctor in a well-lit hospital.

ISO 25010 also delineates a relationship between the two quality views whereby system quality 'influences' system-in-use quality and system-in-use quality 'depends' on system quality. We recently developed 2Q2U version 2.0 (Olsina et. al. 2012), which ties together all of these quality concepts by relating system quality characteristics and attributes with QinU and user experience (UX). By instantiating 2Q2U, evaluators can select the quality characteristics to evaluate and conduct a systematic evaluation using the 'depends' and 'influences' relationships in conjunction with a strategy (Lew, Olsina et. al. 2012).

Today, for MobileApps, more robust network infrastructures and smart mobile devices have led to increased functionality and capability thereby warranting special attention in comprehending how they are different from the UX point of view because user requirements, expectations, and behavior can be somewhat different. For instance, the quality design and evaluation of operability from a system viewpoint has a much different and greater influence for MobileApp usability and UX due to the size of the screen and context of the user. Characteristics such as button size, placement, contextual help, and widget usage for example have a much greater impact on task completion rates and task error rates (Apple, 2013; Budin et. al. 2013; Google, 2013; Nielsen, 2011; Olsina et al. 2012). Ultimately, UX characteristics are very often neglected in quality modeling or seldom placed appropriately in quality views and this is magnified in a mobile context (Nayebi et. al. 2012; Olsina et al. 2012).

Given this, there is a need for a characterization of MobileApps considering non-functional aspects in both UX and EQ. Consequently, the particular features of MobileApps –regarded both as a system and a system-in-use entity- pose new challenges regarding current quality models and their included and more relevant characteristics and sub-characteristics, as well as the particular attributes or properties to be measured and evaluated, e.g., by metrics and indicators. Additionally, for MobileApps, there is an increased emphasis on many contextual elements related to the task and therefore the QinU. So, starting with the task at hand, and applying our 2Q2U model, we can incorporate the importance of task and particular MobileApp UX factors into the design of MobileApps. Starting with MobileApp UX, we can outline practical guidelines to design a MobileApp with optimal EQ characteristics based on UX goals.

Following this introduction, Section 2 outlines our 2Q2U quality framework for better understanding where UX and usability fit in. Section 3, discusses relevant features of MobileApps (both as a system and a system-in-use entity category) useful for designing and evaluating UX and usability. In Section 4, we discuss the usefulness of the proposed framework, while examples of UX/usability attributes and screens for MobileApps are illustrated. Section 5 draws our main conclusions and outlines future work.

2. 2Q2U Quality Framework

Regarding quality models, we recently enhanced the ISO 25010 external quality and quality in use models, while maintaining many characteristics and the ‘depends’ and ‘influences’ relationships between both views. Figure 1 depicts the quality model enhancement named 2Q2U, which was used and instantiated in different case studies (Lew, Qanber Abbasi et. al. 2012; Olsina et. al. 2012).

For the QinU model we have added two main characteristics which are absent in the ISO standard viz. Actual User Experience, and Actual Usability. These concepts are related hierarchically as shown in Figure 1 and also defined in Table 1.

For the EQ model, we have also rephrased Usability as the “*degree to which the product or system has attributes that enable it to be understood, learned, operated, error protected, attractive and accessible to the user, when used under specified conditions*”. Ultimately, the rationale of these adaptations is in (Olsina et. al. 2012).

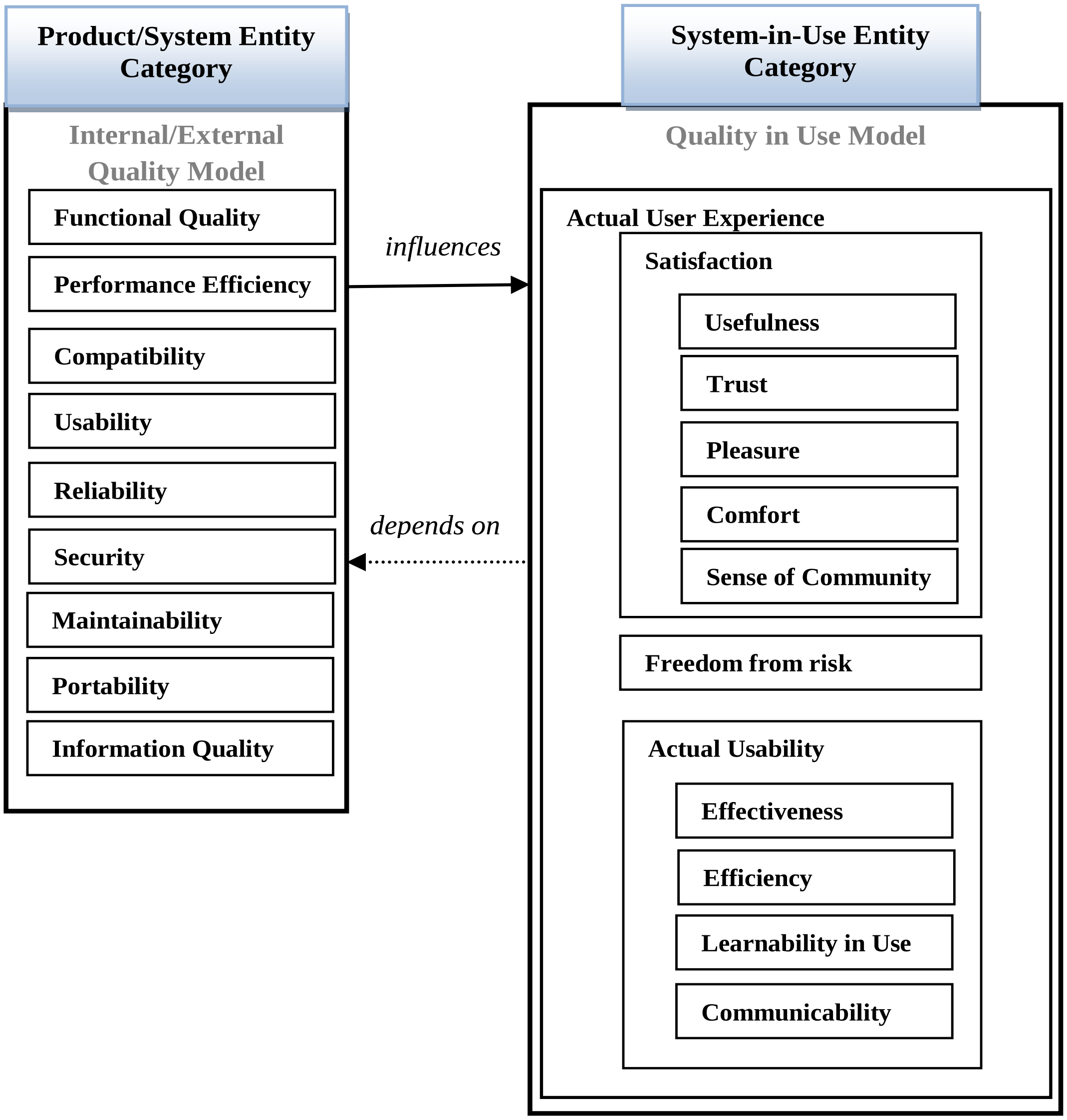


Figure 1: 2Q2U v2.0 model characteristics with some sub-characteristics and relationships

Table 1: Definitions of QinU characteristics and sub-characteristics that are absent in ISO 25010 or were rephrased in 2Q2U v2.0

2Q2U v2.0 QinU Characteristic or Sub-characteristic Definition	**ISO 25010 QinU Definition** (ISO/IEC 25010, 2011)
Actual User Experience*:* Degree to which specified users can achieve actual usability, freedom from risk, and satisfaction in a specified context of use	Note: Absent calculable concept
Actual Usability *(*synonym *usability in use):*Degree to which specified users can achieve specified goals with effectiveness, efficiency, learnability in use, and without communicability breakdowns in a specified context of use	Note: Absent calculable concept, but similar concept (i.e. *usability in use*) was in the ISO 25010 2009 draft
Effectiveness: Degree to which specified users can achieve specified goals with accuracy and completeness in a specified context of use	*Effectiveness*: Accuracy and completeness with which users achieve specified goals
Efficiency*:* Degree to which specified users expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use	*Efficiency:* Resources expended in relation to the accuracy and completeness with which users achieve goals
Learnability in use*:* Degree to which specified users can learn efficiently and effectively while achieving specified goals in a specified context of use	Note: Absent calculable concept
Communicability*:* Degree to which specified users can achieve specified goals without communicative breakdowns in the interaction in a specified context of use	Note: Absent calculable concept
Sense of Community: Degree to which a user is satisfied when meeting, collaborating and communicating with other users with similar interest and needs	Note: Absent calculable concept

In using the ISO 25010 quality models for MobileApps, it is assumed that the software quality models, definitions, and concepts in the standard were intended for application to software information systems as a whole and therefore are also applicable to a great extent to MobileApps, a type of software application. Moreover, we have built up 2Q2U on top of ISO standards in order to consider new features of new generation MobileApps and for instantiating quality models for either native MobileApps or web-oriented MobileApps, as we see later on.

Next, we examine several features and entities relevant for the quality design and evaluation of MobileApps with the ultimate goal to carry out evaluations and accomplish improvement in a MobileApp.

3. Featuring MobileApp Usability and UX

QinU, UX and Usability have all recently come to the research forefront due to a general shift in emphasis to satisfying the end user as part of the customer experience. For MobileApps (including native MobileApps and Mobile WebApps), they become even more important due to the significance of the user context. In particular the user's activity at the time of usage, location, and time, amongst other influencing factors have significant impact on the quality of the user's experience.

This section examines several elements relevant for designing and evaluating UX, Usability, Information Suitability, etc. for MobileApps.

These elements include sub-entities (e.g. widgets, menus, forms, etc.) and their associated attributes and characteristics (e.g. Operability, Understandability, Functional Suitability, etc.). Some of the typical sub-entities for MobileApps that should to be considered for quality design and evaluation are:

- *Typing/input*: which includes search bars, and other data entry fields whereby the users should be assisted as much as possible to reduce errors and the 'cost' of typing. This includes such measurable attributes as default values, default value removal and shortcuts.
- *Entry widgets* such as carousels, drop down boxes and lists. System designers need to prevent the need for typing and reduce error rates by using widgets.
- *Sort, search and filter*: Special considerations are needed for MobileApps in order to reduce the workload and typing input. In addition, the small screen size makes it easy for the user to lose context, so attributes like typo tolerance and predictive contextual help are desirable.
- *Menus* should be limited, simple and easily navigated with a clear breadcrumb path showing where the user has come from and where they can go. This is mostly applicable to Mobile WebApps where a small screen limits the users' context or field of vision in navigating from one place to another.
- *Forms and registration*: Forms need be clear with context sensitive help. The last thing you want is a user unable to complete a form because they didn't quite understand one particular mandatory field. Either defaults, or help within the entry field giving an example of what goes in the field should be provided.

Regarding the aforementioned sub-entities of typical MobileApps, we can now consider some examples of sub-characteristics of Usability, Information and Functional Quality that are particular to MobileApps, for example:

i) *Learnability*: Through its various entities listed above such as menus and widgets, learnability can be designed into the MobileApp through defaults, facilitating predictive actions, context sensitive help, and so on;

ii) *Navigation*: The MobileApp's ability to enable to a user to easily find the functionality or information that they need is critical. Not only do they need to easily find it, they need to do it fast. These are sometimes in alignment but not necessarily;

iii) *Operability*: This is a central sub-characteristic of usability. It means the degree to which a MobileApp has attributes that make it easy to operate and control. For example, controllability represents the degree to which users can initiate and control the direction and pace of the task until task completion. Easy to operate is related to those provided mechanisms which make entering data as easy and as accurate as possible while maintaining consistent usage and placement of controls even in different contexts and platforms of use;

iv) *Error handling*: Error prevention, error awareness, and error status are key attributes that need to be designed correctly to not only prevent errors, but enable the mobile user to recover with minimal effort. In addition, errors should be easily and quickly understood so that the user can move forward with their task and learn from the error;

v) *Understanding*: For MobileApps, because the screen is so small, it requires special consideration for understanding what the application is about, and what it does almost instantly. As mentioned previously, mobile users have a very short attention span so they must glance at the application and understand how it operates. For instance, an airline application, when they open it up, they already know they want to, either check a flight status or make a reservation, so the design must heavily consider this expectation so that there is reduced ramp up time;

vi) *Visibility*: Many factors determine whether or not the application is easily visible to a user. Depending on the context, different text colors and backgrounds can have a positive or negative impact. This sub-characteristic, while related to aesthetics is not identical. Remember that mobile users want to glance quickly and understand almost immediately and there may be glare on their screen if they are outdoors. This means appropriate usage and placement of text in appropriate format can impact the user's speed of comprehension greatly.

Table 2: Definition of Operability sub-characteristics (related to Usability) and potential attributes (in italic).Note that code numbers are only intended to show only hierarchical relationship dependences.

Sub-characteristic / *Attribute*	**2Q2U v2.0 Definition**
1. Operability	Degree to which a product or system has attributes that make it easy to operate and control. Note: Same ISO definition (ISO/IEC 25010, 2011)
1.1 Controllability	Degree to which users can initiate and control the direction and pace of the task until task completion
1.1.1 Task workload reduction (synonym: Task brevity or conciseness)	Degree to which users can reduce the workload and cognitive effort by performing actions to complete a task in an abbreviated manner.
1.1.2 Task interruption support (synonym: Task operation cancel-ability)	Degree to which the task steps can be interrupted without harmful effects to the normal operation.
1.1.3 Undo action support	Degree to which the action can be undone without harmful effects to the normal operation. Note: This could also include timeouts where the user can return to their original location in the application.
1.2 Data Entry Ease	Degree to which mechanisms are provided which make entering data as easy and as accurate as possible
1.2.1 Defaults	Degree to which the application provides support for default information.
1.2.2 Mandatory entry	Degree to which the application provides support for mandatory data entry.
1.2.3 Control (widget) appropriateness	Degree to which the application provides the appropriate type of entry mechanism in order to reduce the effort required, i.e. reducing number of keystrokes and data entry through drop down list boxes and radio buttons which reduce typing needed.
1.3 Visibility (synonym**: Optical Legibility**)	Degree to which the application enables ease of operation through controls and content which can be seen and discerned by the user in order to take appropriate actions.
1.3.1 Control(widget)size appropriateness	Degree to which size of controls enable user to easily determine and understand their function as well as use the control.
1.3.2 Text size appropriateness	Degree to which text sizes and font types are appropriate to enable user to easily determine and understand their meaning.
1.3.3 Contrast visibility appropriateness	Degree to which text and controls colors provide contrast with background color so that they are easily visible. Note: This can be gauged considering background and foreground colors.
1.3.4 GUI widgets placement appropriateness	Degree to which the placement of widgets and text is appropriate and visible in any context.
1.4 Consistency	Degree to which users can operate the task controls and actions in a consistent and coherent way even in different contexts and platforms of use.
1.4.1 Control label consistency	Degree to which the labels of widgets/controls correspond to the actions they represent.

Table 2 shows for the Usability characteristic one of its sub-characteristic, i.e. the Operability sub-characteristic as prescribed in the ISO standard (ISO/IEC 25010, 2011) from the EQ standpoint. We have elaborated Operability, adding and defining new sub-characteristics and potential measurable attributes. Attributes can in turn be measured by means of metrics and evaluated and interpreted by means of indicators. In (Lew et. al. 2010) we have evaluated some Learnability and Operability attributes.

Finally, among other sub-characteristics and attributes to take into account for evaluating MobileApps is Information Conciseness, which in few words means shorter is better. In (Olsina et. al. 2012) Information Conciseness is an attribute of the Information Coverage sub-characteristic, which in turn is related to Information Suitability, and is defined as "*degree to which the information coverage is compactly represented without being overwhelming*". Note that some of these attributes are identified in the screens shown in the next section.

In alignment with the ISO 25010 'influences' relationship (see Figure 1), each of these system sub-characteristics/attributes can have an influence on the system in use for both the do-goals (Actual Usability characteristic and its sub-characteristics) and be-goals (Satisfaction characteristic and its sub-characteristics). Depending on the context, user, and task, the influence can impact the user's ability to operate, navigate and use the application efficiently (do-goals) or comfortably and with pleasure and utility perception (be-goals).

Thus, in designing a MobileApp, one must also take into account the decision to make a native MobileApp versus a Mobile WebApp. For instance, navigability, as described above, and congruence and consistency with the regular WebApp are significant. On the other hand, many widgets and other features of the mobile phone are not accessible to WebApps so they inherently start at a disadvantage regarding widget usage which, depending on the application, can impact system in use greatly.

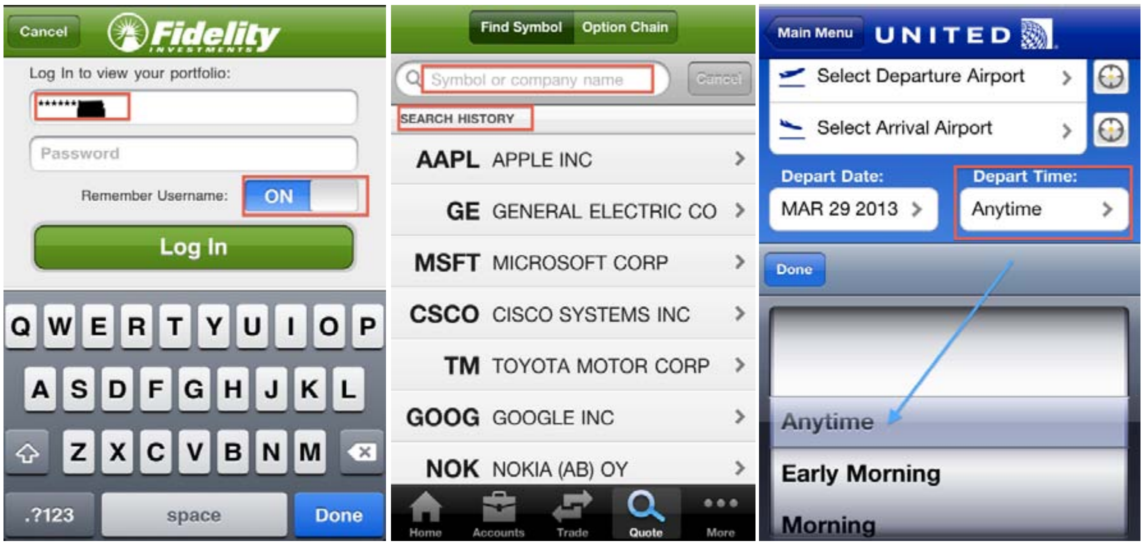
4. Illustrating Quality Design and Evaluation for MobileApps

Given the aforementioned 2Q2U framework and mobile UX and usability concepts we can apply them to MobileApps. This section builds upon Section 3 and demonstrates the importance of a few of the factors through some examples whereby the context, system, and system in use come into play for MobileApps.

When designing MobileApp tasks for QinU, for example for evaluating Effectiveness and Efficiency characteristics, content and functions are embedded in the task design itself rather than as attributes of the application. And, as mentioned previously, UX do-goals for a MobileApp-in-use are heavily influenced by the workflow of the task. Because the most prevalent mobile tasks include deadline oriented, or time sensitive information, the workflow should be short in order to be effective and efficient to make up for the small screen size and user context (in a hurry with short attention span). User's are often executing tasks 'on the spot' and 'in the moment' so a delay, or a mistake is critical at that moment. Can you imagine getting stranded at an airport, and then looking up all the flights going out of an airport and not being able to search effectively because the search menu was designed only with one search parameter?

From this simple example, it is easy to see that QinU depends on many system sub-entities having particular sub-characteristics and attributes depending on the context, user and task. Note that when executing a particular task, there should be very few controls that could lead the user astray or toward a mistake (see for example the *Fidelity Investment* MobileApp screenshots in Figures 2.a and b). They purposely design the widgets and controls to be large, thumb friendly, so that the user can complete the task given the information needed as effectively and efficiently as possible. There is contextual help, or a simple drop down menu with limited choices in the fields that need to be filled in.

As mentioned for the typing/input sub-entity, defaults are extremely important to reduce the workload of the user. Defaults can be based on what the user has typed or submitted in the past (e.g., zip codes, names, addresses). Defaults are bound to be wrong sometimes and you don't want to have a default that takes unnecessary effort to change. For example don't use 0 as the default for a telephone number or zip code and don’t make users manually erase the text field character by character, by clicking the Delete key. Instead, erasing defaults with a single button that clears the entire field can save several seconds.



AAPL	APPLE INC
GE	GENERAL ELECTRIC CO
MSFT	MICROSOFT CORP
CSCO	CISCO SYSTEMS INC
TM	TOYOTA MOTOR CORP
GOOG	GOOGLE INC
NOK	NOKIA (AB) OY

Depart Date: MAR 29 2013

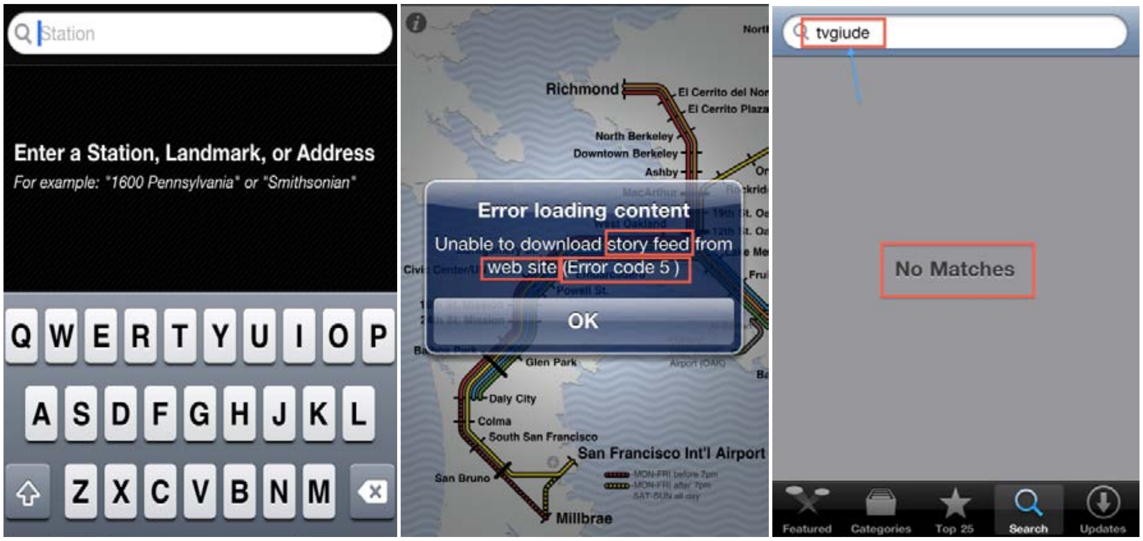
Depart Time: Anytime

Anytime
Early Morning
Morning

Figure 2: a) Fidelity Investment MobileApp login screen; b) Fidelity Investments MobileApp find a ticker symbol task; and c)United Airlines MobileApp finding a flight

In Figure 2.a, the MobileApp shows the default user name as well as the option to save it or not, giving the user a sense of controllability. Also, for typing/input and widget sub-entities, the *United Airlines* MobileApp shown in Figure 2.c, shows a carousel for data entry rather than typing. It also gives the user another option for input. Additionally, the screen is very clear for the task at hand. Notice that there is no other input or information not directly related to the task of Finding a flight.

Examining Figure 2.b, shows the *Fidelity Investment* MobileApp for the task of “Finding a stock ticker symbol”. Notice the context sensitive help in light grey color symbol or company name, which aids the user. Also, there is a list of past searches which helps the user as well. Both the ticker symbol and company name are shown to eliminate any confusion and prevent using the incorrect ticker symbol. The information is also presented in a clear and concise manner with no other information (information conciseness) to distract from the task at hand while features have been designed to reduce errors and increase efficiencies and task completion rates. The last thing Fidelity wants is for a user to not complete this task which may affect executing a trade.



Enter a Station, Landmark, or Address

For example: "1600 Pennsylvania" or "Smithsonian"

Error loading content

Unable to download story feed from web site (Error code 5)

No Matches

Figure 3: a) and b) Embark Washington DC Metro MobileApp; c) Apple App Store MobileApp

Figures 3.a and b show the *Embark* application for the *Washington DC Metro* train system. Notice that (Figure 3.a) for the task of “finding a station”, the user is given context sensitive help below the search bar. There is also light grey letters in the search bar to make sure the user is not lost, reminding the user what they are trying to do. Also notice that the search bar is quite long, thus providing the user with ample space to type in their desired location. On the other hand, for the same entity, you can see from Figure 3.b that the errors are not comprehensible thus very difficult for the user to recover and move forward with their task. Lastly, Figure 3.c shows a search function implemented in the Apple App Store. For the App Store, a user in search of 'tvguide' accidently typed 'tvgiude' and got no results. Thus in terms of typo tolerance and error prevention, it would rate poorly for that particular task.

These examples demonstrate MobileApp sub-entities such as forms, menus and search boxes and why design principles, i.e. mobile system factors can significantly influence the UX and other system in use factors. With a screen so small, it is easy to see why concise information, contextual help, etc. are critical attributes.

Note that a systematic and specific instantiation of quality models for MobileApps -as we did in (Lew, Olsina et. al. 2012) for the JIRA WebApp's QinU and EQ- will be documented in a separate manuscript.

In (Lew, Olsina et. al. 2012) we elaborated SIQinU (*Strategy for understanding and Improving Quality in Use*), which is a specific-purpose and context-sensitive strategy to incrementally and continuously improve a WebApp-in-use’s QinU by means of mapping actual usage problems to measurable EQ attributes –that are inherent to a WebApp-, and then by performing improvement actions that enable evaluators assessing the gain both at EQ and QinU levels. SIQinU is a specific-purpose strategy because it can be used to evaluate the quality for only system-in-use and system entity categories by using their instantiated QinU and EQ models respectively.

Basically, the strategy collects user task data from log files (that were derived through for example adding snippets of code in a real WebApp-in-use that allow recording user activity), with the aim to derive attributes, metrics and indicators for QinU, thus leading us to understand the current QinU satisfaction level met. Then, by performing a preliminary analysis, EQ requirements that can affect QinU are derived, followed by proposed recommendations for improvement. After performing the changes on the WebApp, and after conducting studies with the same user group in the same daily environment (context) with the new (and improved) system version, an assessment of the improvement gain can be gauged.

5. Conclusions and Future Work

In summary, we have characterized relevant features of MobileApps with regard to usability and UX analyzed them using our previously proposed 2Q2U v2.0 quality models. In the process, we looked at MobileApp usage context and its impact in MobileApp design when considering quality characteristics and attributes for system and system-in-use quality evaluation. Lastly, we illustrated some example MobileApps to show their design features could have a significant influence on the system quality and the user's perception of quality, i.e. system in use and UX.

Our proposal shows the usage of 2Q2U v2.0 as an integrated approach for modeling requirements for external quality and quality in use (i.e., actual usability, satisfaction and user experience) for MobileApps.

Another manuscript will expand the work further including the definition of attributes, metrics, and indicators for user group types performing specific tasks in real MobileApp context of use. Ongoing research is focused on further utilizing the 2Q2U v2.0 framework for systematic instantiation of quality models for MobileApps –as we did in (Lew, Olsina et. al. 2012)- in order to provide foundations when modeling and understanding the relationships among EQ, QinU, actual usability and UX. In doing so, our end goal is improvement. That is, to be able to directly use the system-in-use quality evaluation to directly improve the design of the MobileApp system. This concern has often been neglected in the literature, but may help improve quality design recommendations and ultimately to increase MobileApp UX as a whole.

References

Apple. 2013. "iOS Human Interface Guidelines", http://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html, (accessed April, 2013).

Budiu, R. and Nielsen, J. 2013. "Usability of Mobile Websites", http://www.nngroup.com/reports/mobile, (accessed April, 2013).

Google. 2013. "User Interface Guidelines" :http://developer.android.com/guide/practices/ui_guidelines/index.html, (accessed April, 2013).

ISO/IEC 25010. 2011. "Systems and software engineering. Systems and software Quality Requirements and Evaluation (SQuaRE). System and software quality models".

Lew P., Qanber Abbasi M., Rafique I., Wang X., Olsina L. 2012. "Using Web Quality Models and Questionnaires for Web Applications Evaluation". *In: IEEE proceedings of QUATIC*, Lisbon, Portugal, pp. 20-29.

Lew P., Olsina L., Becker P., Zhang, L. 2012. "An Integrated Strategy to Understand and Manage Quality in Use for Web Applications". *Requirements Engineering Journal*, Springer London, Vol.17, No. 4, pp. 299-330.

Lew P., Olsina L., Zhang L. 2010. "Quality, Quality in Use, Actual Usability and User Experience as Key Drivers for Web Application Evaluation". In: *LNCS 6189, Springer, 10th Int'l Congress on Web Engineering (ICWE2010)*, Vienne, Austria, pp. 218-232.

Nayebi F., Desharnais J-M., and Abran A. 2012. "The state of the art of mobile application usability evaluation". In: *25th IEEE Canadian Conference on Electrical Computer Engineering*, 2012, pp.1-4.

Nielsen J. Sep 26, 2011. "Mobile Usability Update", *Jakob Nielsen's Alertbox*, http://www.nngroup.com/articles/mobile-usability-update/, (accessed April, 2013).

Olsina L., Lew P., Dieser A., Rivera B. 2012. "Updating Quality Models for Evaluating New Generation Web Applications", In*: Journal of Web Engineering, Special issue: Quality in new generation Web applications*, S. Abrahão, C. Cachero, C. Cappiello, M. Matera (Eds.), Rinton Press, US, 11 (3), pp. 209-246.

Thom-Santelli, J. and Hedge, A. 2005. "Effects of a multi-touch keyboard on wrist posture, typing performance and comfort". *Proc. of the Human Factors and Ergonomics Society*, 49th Annual Meeting, Santa Monica, USA, pp. 646-650.

Using FMEA to Improve Software Reliability

Kraig Strong

Kraig.Strong@Tektronix.com

Abstract

Failure Mode and Effect Analysis (FMEA) is a methodology widely used by hardware designers to model and avoid field failures. Increasingly, this methodology is being adapted to modeling software systems for improving reliability. Among the methods, tools, and practices for improving software reliability, FMEA is arguably the least costly, easiest to learn, and most effective.

The presentation will review the essential elements of the FMEA methodology as it is applied to systems that involve software components. These essential elements include the Functional Block Diagram (FBD) and the FMEA worksheet. The FMEA worksheet is the final work product in the process and culminates in a prioritized list of recommended actions.

Examples, lessons learned, and recommendations will be offered to reinforce the FMEA concepts and provide helpful guidance for those interested in applying FMEA on their software projects.

Biography

Kraig Strong is a software quality engineer at Tektronix in Beaverton, Oregon. Prior to joining Tektronix, he spent a few years as a manufacturing test engineer where he used FMEA on a daily basis. Kraig is bringing over his quality-focused background in order to help deliver more reliable software and products at Tektronix.

Kraig has a B.S. in Electrical and Computer Engineering from Oregon State University and is currently pursuing an M.S. in Computer Science from Portland State University.

1. Introduction

Failure Mode and Effects Analysis (FMEA) is a methodology to find potential failures before they occur. While FMEA identifies individual failure modes, its primary benefit is the early identification of system failure modes so a solution can be designed to mitigate the potential failure. It is a methodology to design reliability into a system. In a FMEA, numerical weights can be applied to the likelihoods of each failure, as well as the severity of the consequences. FMEA is a very cost-effective, easy to learn, and productive way to design a more reliable system.

Although this method was not originally created for software systems, we can translate the principles over to software and take advantage of the many benefits that FMEA has to offer. These benefits include:

- Facilitates early identification of failure points and system interface problems
- Yields a better understanding of planning/scheduling by revealing additional work efforts
- Enables early test planning
- Provides a single worksheet summary of analysis
- Requires minimal training to participate, or even lead a FMEA
- Increases reliability in products

The outcome of this process is a single FMEA worksheet containing a list of failure modes prioritized by risk in the system under study. This prioritized list identifies the most valuable places to focus extra design and test efforts to produce a more reliable system.

Section 2 of this paper presents the key elements of FMEA. Section 3 is an introduction to a Manufacturing Process FMEA for those that are unfamiliar with basic FMEA principles. Section 4 relates to FMEA for software systems.

2. Key Elements of FMEA

A typical FMEA is a team activity, accomplished in one or more meetings. The owner of the system/sub-being analyzed, project leads, QA, and at least one domain expert typically attend the meeting(s). Anybody with even moderate knowledge of the system can also attend and be of great benefit to the outcome. During the meeting, these seven essential steps provide guidance through the process:

1. Define Failure Modes – *What can go wrong here?*
2. Define Effects – *What will happen then?*
3. Describe Targets – *Who will suffer from the failure?*
4. Find Root Causes – *Why will that happen?*
5. Prioritize the Risks – *What is the likelihood?*
6. Define Solution Actions – *How can this be prevented?*
7. Define Current Prevention and Detection Methods – *What is currently being done?*

These steps are repeated throughout the meeting, resulting in the FMEA worksheet with a prioritized list of risks and their associated failures in the system.

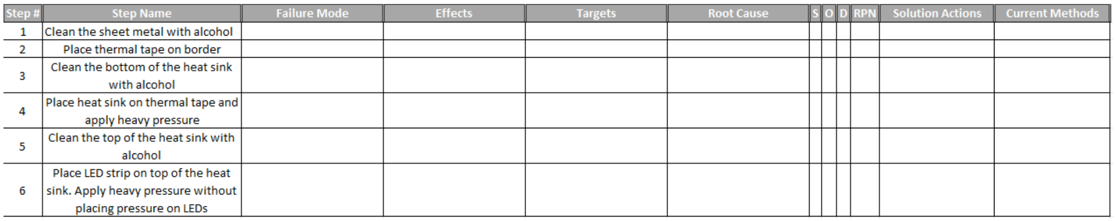
3. Manufacturing Process FMEA

There are three main types of FMEA:

- Process: Used to analyze manufacturing and assembly processes. This is arguably the most straightforward, and easiest to understand type.
- Hardware: Used to analyze hardware systems both before (concept design) and after (detailed design) the hardware design is completed.
- Functional: Based on a functional breakdown of a system. Used to analyze high-level functional blocks of a system. This is what is used for software evaluation. This paper will use the term ‘Software FMEA’ to represent this type.

Since the Process FMEA is the most straightforward, an example will be provided to clarify the concepts and steps to perform a successful FMEA. Section 4 will provide a software FMEA example. The example used below is a simple process of attaching a printed circuit board (PCB) to a heat sink to sheet metal using thermal tape. This is done to help the PCB stay cool to increase the lifespan of the components.

Before jumping into the steps, first define what is to be analyzed. This is where the Process FMEA shows its simplicity. A work instruction is already a broken down linear list of step-by-step tasks needed to build the product. These steps are be copied into the FMEA worksheet in the ‘Step’ column.



Step #	Step Name	Failure Mode	Effects	Targets	Root Cause	S	O	D	RPN	Solution Actions	Current Methods
1	Clean the sheet metal with alcohol										
2	Place thermal tape on border										
3	Clean the bottom of the heat sink with alcohol										
4	Place heat sink on thermal tape and apply heavy pressure										
5	Clean the top of the heat sink with alcohol										
6	Place LED strip on top of the heat sink. Apply heavy pressure without placing pressure on LEDs										

Table 1 – FMEA Worksheet

The columns in Table 1 will be referenced throughout this section.

3.1 FMEA Steps

To perform a Process FMEA, simply follow these steps:

3.1.1 Define Failure Modes – *What can go wrong here?*

For every step previously defined, ask the question, “What can go wrong here?”. The answer to this question needs to be focused directly on this step alone; assume everything else in the manufacturing process was completed correctly. An important note is that not all manufacturing process steps have a discernable failure mode. If no possible failure modes are present, skip to the next row of the worksheet.

Step: Clean the sheet metal with alcohol

Possible Failure: The sheet metal may not be fully cleaned

Possible Failure: The sheet metal could be bent during the cleaning process

These two possibilities are the failure modes of this step. They go under the “Failure Modes” column on the FMEA worksheet.

3.1.2 Define Effects – *What will happen then?*

The next step is to ask, “What will happen then?” for every failure mode listed. Look exclusively at only one failure mode at a time and focus on any direct results from the failure.

Failure Mode: The sheet metal was not fully cleaned

Direct Result: the thermal tape to be applied here could not fully adhere, leading to a breakdown of the thermal path which could lead to the PCB to overheat or to become dislodged entirely.

Failure Mode: The sheet metal could be bent during the cleaning process

Direct Result: an air gap between the heat sink and the sheet metal, resulting in a breakdown of the thermal path that could result in the PCB overheating.

Direct Result: a bend was severe enough to prevent installation at a later step. The result would be to scrap or rework that part.

These results are the effects on the system and go into the “Effects” column on the FMEA worksheet.

3.1.3 Describe Targets – *Who will suffer from the failure?*

The third step is to define the targets for each effect. This is done by asking the question, ”Who will suffer from the failure?”.

Effect: Tape not fully adhered, thermal path breakdown, PCB overheating

Who Suffers: End customer will suffer due to overheating generally taking a good deal of stress to cause issues and is likely not to be caught at manufacturing test. There is a good chance this defect would ship to the customer.

Effect: Air gap in thermal tape, thermal path breakdown, PCB overheating

Who Suffers: End customer will suffer due to overheating generally taking a good deal of stress to cause issues and is likely not to be caught at manufacturing test. There is a good chance this defect would ship to the customer.

Effect: Severe bend caused part to be unusable during install causing a scrap or rework of that part

Who Suffers: The company would suffer since this part would no longer physically fit into the system. The part would need to be reworked or scrapped, resulting in lower yields and higher operating costs.

These targets are captured the “Targets” column on the FMEA worksheet.

3.1.4 Find Root Causes – *Why will that happen?*

The fourth step is to define the root causes by asking, “Why will that happen?”.

Failure Mode: The sheet metal was not fully cleaned

Root Cause: A contaminant that is insoluble with alcohol

Root Cause: Operator error

Failure Mode: The sheet metal was bent during cleaning

Root Cause: Operator error

The root causes are captured in the “Root Causes” column on the FMEA worksheet.

3.1.5 Prioritize the Risks – *What is the likelihood?*

The fifth step is to prioritize the risks associated with the failure mode. Every failure is assigned three numeric ratings in a scale of 1 to 10:

- Severity (S): 1 (insignificant) to 10 (catastrophic)
- Likelihood of Occurrence (O): 1 (unlikely) to 10 (inevitable)
- Detectability (D): 1 (guaranteed to be detected) to 10 (undetectable).

These three numbers are multiplied to create the Risk Priority Number (RPN). Thus, S x O x D = RPN.

Effect: Tape not fully adhered, thermal path breakdown, PCB overheating

Severity: 6. Shortened PCB component lifespan.
Occurrence: 2. With proper training, this is unlikely to happen.
Detectability: 3. This can be easily detected in production.
RPN: 6 x 2 x 3 = 36

Effect: Air gap in thermal tape, thermal path breakdown, PCB overheating

Severity 6: PCB overheating would shorten the lifespan of components.
Occurrence 2: With proper training this is unlikely to happen.
Detectability 1: Very easily detected in production.
RPN: 6 x 2 x 1 = 12

Effect: Severe bend makes part unusable causing a scrap or rework

Severity: 3. Wasted time or resources. This will not directly affect the customer.
Occurrence: 2. With proper training, this is unlikely to happen.
Detectability: 1. Easily detected in production.
RPN: 3 x 2 x 1 = 6

The severity, occurrence, detectability, and RPN go on the FMEA worksheet.

3.1.6 Define Solution Actions – *How can this be prevented?*

The next step is to define solution actions. These are steps that can be taken to mitigate the chance of a particular failure occurring.

Failure Mode: The sheet metal may not be fully cleaned

Solution Action: Operator training on proper cleaning techniques.

Solution Action: Additional step in work instruction to check for contaminants.

Failure Mode: The sheet metal could be bent during the cleaning process.

Solution Action: Operator training on proper cleaning techniques.

Solution Action: A fixture may be needed to hold the sheet metal during the cleaning process.

Enter the solution actions into the “Solutions” column on the FMEA worksheet.

3.1.7 Describe Current Prevention and Detection Methods – *What is currently being done?*

The final step is to describe what is currently being done to prevent or detect the failure.

Failure Mode: The sheet metal may not be fully cleaned.

Current Methods: Operator training prior to working on the production line. Work instructions are displayed during production.

Failure Mode: The sheet metal could be bent during the cleaning process.

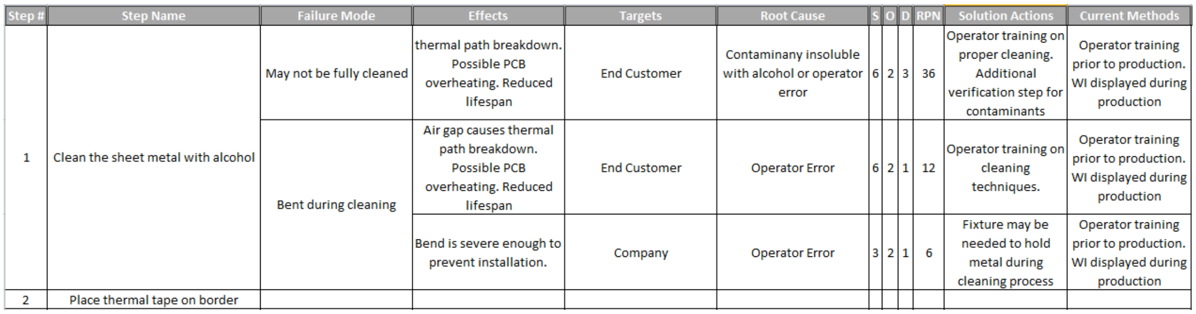
Current Methods: Operator training prior to working on the production line. Work instructions are displayed during production.

Enter the current methods into the “Current Methods” column on the FMEA worksheet.

3.1.8 Repeat

Repeat these steps for every row of the worksheet until it is complete. After the process is complete, the highest risk areas of the system can be identified by sorting on the RPN column. From this information, it is possible to identify where the most benefit will be realized from implementing mitigations or redundancy into the system.

Table 2 shows the completed FMEA worksheet for the first step of the manufacturing process. The above steps should be repeated for every row in the worksheet until it is completed.



Step #	Step Name	Failure Mode	Effects	Targets	Root Cause	S	O	D	RPN	Solution Actions	Current Methods
1	Clean the sheet metal with alcohol	May not be fully cleaned	thermal path breakdown. Possible PCB overheating. Reduced lifespan	End Customer	Contaminany insoluble with alcohol or operator error	6	2	3	36	Operator training on proper cleaning. Additional verification step for contaminants	Operator training prior to production. WI displayed during production
		Bent during cleaning	Air gap causes thermal path breakdown. Possible PCB overheating. Reduced lifespan	End Customer	Operator Error	6	2	1	12	Operator training on cleaning techniques.	Operator training prior to production. WI displayed during production
			Bend is severe enough to prevent installation.	Company	Operator Error	3	2	1	6	Fixture may be needed to hold metal during cleaning process	Operator training prior to production. WI displayed during production
2	Place thermal tape on border										

Table 2 – Completed Process FMEA Worksheet for Step 1

4. FMEA in a Software System

To perform a FMEA in a software system, the Functional FMEA model is used. The main difference from a Process FMEA is that for the Software FMEA there is no pre-defined, linear flow that can be copied from a work instruction. In the example in Section 3, the work instructions are the starting point for the FMEA steps. In a software system decisions are made at run-time to determine the appropriate path. This complicates the definition of the steps needed to begin the FMEA worksheet. Another fundamental difference between Process and Software FMEA is that, for software, it is not suggested to cover 100% of the design. Instead, focus efforts where failures are most likely or where the consequences are severe. These should be limited to a particular sub-system or even a new feature of a system.

In order to define steps, first break the system down into Function Blocks to create a Functional Block Diagram (FBD). A function block is a high level description of a piece of the software system. The level of detail that goes into the function blocks is a large decision and can be influenced by a variety of factors such as:

- New technologies or hardware present – The team may want to spend more time on areas where new technologies or hardware are present, so more detailed blocks should describe these areas.
- Overall confidence of the subsystem – Was there any re-used code in the system? These parts can be described with larger, less descriptive blocks, so more energy is focused where problems are more likely.
- Safety concerns – When safety is a concern, functional blocks should be more detailed so there is less chance for an oversight or assumption.

A natural approach might be to use the FBD from the design specification, but this can often lead to extensive and complicated analysis, so a higher-level FBD may need to be created. This can be accomplished by combining individual blocks of the FBD to make a more generalized version.

Once the generalized FBD is created for the system under analysis, it is likely that the diagram will not be linear and will not fit nicely into a worksheet. There may be a need to break up the FBD into smaller, more linear pieces so it can be easily referenced on a worksheet. Figure 1 provides an example of this.

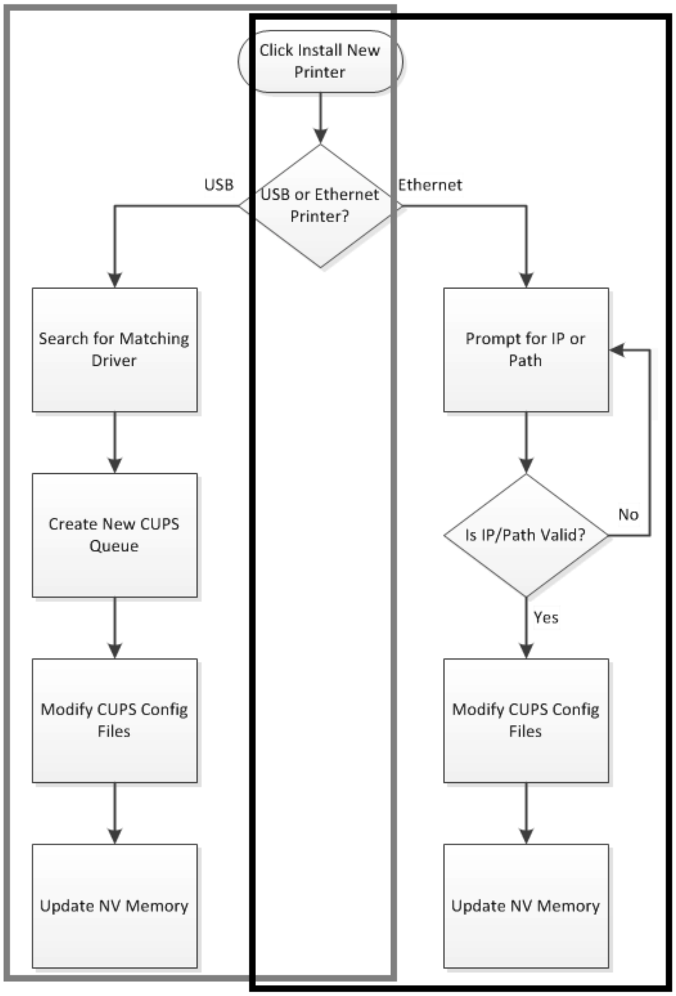
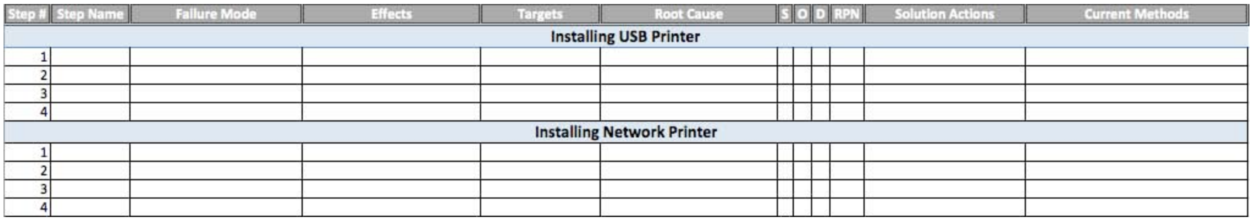


Figure 1 – FBD Splitting Example

Once the system is broken into smaller, more manageable FBDs, each can be placed into the FMEA worksheet. The FMEA worksheet will have a small section for each FBD so they can be easily referenced throughout and after the meeting. In the above example, there would be two sections for the FMEA worksheet. One would be titled “Installing USB Printer”, and the other would be titled “Installing Network Printer”. An example of this is below in figure 2.



Step #	Step Name	Failure Mode	Effects	Targets	Root Cause	S	O	D	RPN	Solution Actions	Current Methods
Installing USB Printer											
1											
2											
3											
4											
Installing Network Printer											
1											
2											
3											
4											

Figure 2 – Split FMEA Worksheet

Once the blocks are entered into the worksheet, go through the same FMEA steps (with one key difference) as the previous example, which were:

1. Define Failure Modes – *What can go wrong here?*
2. Define Effects – *What will happen then?*
3. Describe Targets – *Who will suffer from the failure?*
4. Find Root Causes – *Why will that happen?*
5. Prioritize the Risks – *What is the likelihood?*
6. Define Solution Actions – *How can this be prevented?*
7. Define Current Prevention and Detection Methods – *What is currently being done?*

The key difference between these steps in a Software FMEA versus the Process FMEA is that the root cause is often identical to the failure mode. There could be a root cause column in the FMEA worksheet, but it is not required when it is identical to the failure mode.

Note, while defining failure modes in a software system, it is important to not have ‘software bugs’ as a failure mode. Since any piece of code can contain bugs, this is a meaningless failure mode and should not be considered during this analysis. With that said, the FMEA meeting is a good time to see which blocks may be more likely to have bugs, yielding great candidates for an in-depth code reviews.

4.1 Software FMEA Example

As a Software FMEA example, consider a simplified purchasing algorithm for a website. The FBD is displayed in Figure 3 below.

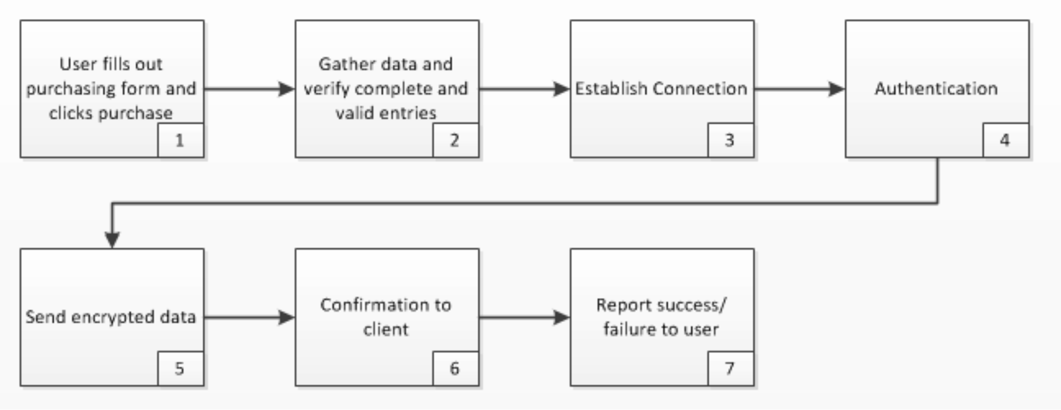
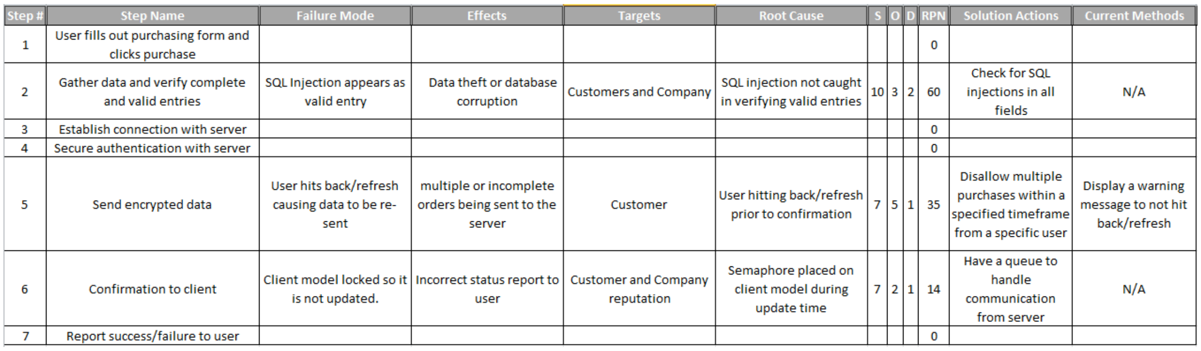

User fills out purchasing form and clicks purchase
1
Gather data and verify complete and valid entries
2
Establish Connection
3
Authentication
4
Send encrypted data
5
Confirmation to client
6
Report success/ failure to user
7

Figure 3 – Software FBD Example

A completed FMEA worksheet for the above FBD is located in Table 3 with highlights noted below.



Step #	Step Name	Failure Mode	Effects	Targets	Root Cause	S	O	D	RPN	Solution Actions	Current Methods
1	User fills out purchasing form and clicks purchase								0		
2	Gather data and verify complete and valid entries	SQL Injection appears as valid entry	Data theft or database corruption	Customers and Company	SQL injection not caught in verifying valid entries	10	3	2	60	Check for SQL injections in all fields	N/A
3	Establish connection with server								0		
4	Secure authentication with server								0		
5	Send encrypted data	User hits back/refresh causing data to be re-sent	multiple or incomplete orders being sent to the server	Customer	User hitting back/refresh prior to confirmation	7	5	1	35	Disallow multiple purchases within a specified timeframe from a specific user	Display a warning message to not hit back/refresh
6	Confirmation to client	Client model locked so it is not updated.	Incorrect status report to user	Customer and Company reputation	Semaphore placed on client model during update time	7	2	1	14	Have a queue to handle communication from server	N/A
7	Report success/failure to user								0		

Table 3 – Software FMEA Worksheet

Highlights from the above table:

- Two user-caused failure modes, and one possible development oversight.
- RPN column strongly reflects what would be most severe failures. Data theft at the top, and wrong confirmation at the bottom.
- Step 5 has a failure mode and root cause as being the same, which is acceptable in a Software FMEA.
- A few steps have no known failure modes. This is acceptable. Not all function blocks will have any discernable failures.
- Solution Action column can be added to software requirement specification to verify functionality.
- Test development is aided in steps 2 and 5 by performing or writing tests to attempt to recreate the failure modes.

4.2 Appropriate Time for a Software FMEA

The most benefit of a Software FMEA will be realized in early phases of design, ideally before any code has been written but after a majority of the requirements are defined. When performed early, the FMEA can reveal weak points of the system. Solutions can be designed in hardware, software, or both that can help avoid costly design changes in later project phases. Having the flexibility to determine the most cost-effective and most reliable solution sooner rather than later is the largest benefit of performing the FMEA.

Test development and planning can also be aided when performed early by knowing additional testing efforts up-front. These extra efforts may come from:

- Verification needed for all implemented solution actions
- Test cases can be created where the “Current Prevention/Detection Methods” column is empty.
- More testing efforts can be directed towards high RPN values.

While the most benefits of doing a Software FMEA are realized in the earliest design phases, many benefits can still be had in the later phases as well. The riskiest portions of the design will still be identified, and many solutions can still be developed. However, this typically comes at a cost of schedule or budget, and the flexibility to solve problems in hardware may be lost.

As an example, in an Anti-lock Braking System (ABS), there is a significant amount of hardware and software working together to keep the occupants of the vehicle safe. If a FMEA was performed on this system at any time, it may be realized that a single point of failure is in the sensor that detects when ABS needs to be applied. If this was caught early, there may be time to add in a redundant sensor and design a fail-safe of mechanical braking if both sensors fail. If caught too late, the redundant sensor may not be an option due to schedule constraints, and the only option would be to revert to mechanical braking if the sensor fails. Hence, the risk of this failure occurring would be reduced if caught early.

4.3 Mechanics of the Software FMEA Meeting

Since the FBDs to be used for the analysis are not created by the entire team attending the FMEA meeting, it is likely that there will be some modifications during the meeting. For this reason, it is recommended to display the FBDs in a way where they can be quickly re-arranged during the meeting if it is desired. A group of blocks may be grouped together, or one block may be split into several, depending on the level of detail desired. A software tool, such as Microsoft Visio can be used, or simple post-it notes can be placed on a wall or whiteboard to display the FBDs during the meeting.

The meeting discussion should be maintained at a high level. When a group of engineers get into a discussion about issues and continue asking questions that begin with "what if", the conversation often drills down to a very low level very quickly. Although these discussions are useful, and they should be had, during the meeting they should simply be noted and then continue with the meeting at a high level. Keep in mind that this is not a code or design review.

5. Conclusion

FMEA is a widely used and accepted method of reliability engineering. Its purpose is to identify possible failures, evaluate their effect on the system, and propose solutions to mitigate these effects. FMEA is most commonly used on the process and hardware levels, but is becoming more commonplace in the software industry. Some benefits of using FMEA on a software system include:

- Early identification of single failure points and system interface problems
- Getting a better idea for planning/scheduling
- Assisting in early test planning
- Catching issues early enough to be solved in hardware of software
- Single worksheet summary of the analysis

There are also fringe benefits to 'what if?' thinking. Having a dedicated time to get a group of engineers together to think about ways the system could break down leads to valuable test cases, and eventually, more reliable software.

Some common mistakes made during Software FMEAs include:

- Assuming that all failure modes are caused by hardware.
- Attempting to cover 100% of the design instead of focusing on where the design is most likely to cause serious failure.
- Neglecting to follow through with solution actions discovered during the process.
- Allowing the meeting to get into a low level (or code level) conversation.
- Not having appropriate domain experts attend the meeting.

A FMEA is a cost-effective way to guide you to a more reliable design by highlighting the most valuable places to insert fail-safes, redundancies, or other elements that increase the system's overall reliability.

References

Softrel, LLC. “SFMEAs and SFTAs” FMEA. http://www.softrel.com/fmea.htm (accessed 7/12/2013).

Michael Barr “Building Reliable and Secure Embedded Systems” Embedded. http://www.embedded.com/electronics-blogs/barr-code/4238429/Building-reliable-and-secure-embedded-systems (accessed 6/17/2013).

Mike Silverman and George de La Fuente. “Software Design for Reliability” Ops A La Carte LLC. http://www.opsalacarte.com/pdfs/Tech_Papers/Software_Design_for_Reliability_-_Paper.pdf (accessed 7/10/2013).

Turabian, Kate. 2007. *A Manual for Writers of Research Papers, Theses, and Dissertations: Chicago Style for Students and Researchers*. Chicago: The University of Chicago Press.

Alternatives: A New Dimension In Software Quality

Murugan Sundararaj

Murugan_Sundararaj@McAfee.com

Abstract

There are just umpteen numbers of things that go into building ”quality” software. Be it following good design principles, running static code analysis, conducting peer code reviews, testing, etc. But we barely think about “Alternatives”.

Alternative? Yes, finding alternate software code - one that can improve the fundamental quality of the software. It is not necessarily about changing the software implementation language but looking outside the traditional quality radar. Think of adapting to latest programming standard of the existing programming language in which the software is implemented. Or if only a subset of heavy-duty software is used then think of replacing it with lighter ones.

We put more time and energy into staffing up QA team instead of investing in adapting to latest programming standards up front to let developers develop better code. It is time to shed some light on this new approach in building quality software.

Today your software might be of better quality. But improving or introducing new quality processes may alone not sustain the current software quality. The core of software is the programming language in which it is implemented. Replacing the age-old code with simple, efficient and safer new code will boost the quality of the software. It does not stop there. The newer code will reduce the “mean time to repair”.

The out dated programming standard might make a developer write staggering amount of code to achieve a given functionality. But if there are simpler alternatives available in the new programming standard it is good to go for it. Remember bulky codes are one of the major sources of bugs.

This paper explains the various steps involved in finding alternatives and how to successfully apply them to improve the overall quality. We will also look at the challenges involved in multi platform environment.

Biography

Murugan Sundararaj is a principal engineer at McAfee, currently working in the McAfee India center in Bangalore. Over the past eight years he has been developing enterprise software both for windows and non-windows operating systems.

Prior to working for McAfee, Murugan worked as a senior software engineer for Yahoo!

Murugan holds a Bachelor’s degree in Electrical and Electronics Engineering from Periyar University, Tamilnadu, India.

1. Introduction

This technical paper focuses on software written in C++ programming language. But most of the ideas and methods, presented here, can be applied to software written in any programming language.

There are a number of ways to improve the quality of the software. This paper attempts to focus on a few key things that will make a huge impact on the overall quality.

Adopt new programming standards and see if the new facilities (language features) can solve your existing problems in a simple, efficient and safer way.

Stop toiling with the home grown platform abstraction library. Try the standard libraries and you might appreciate integrity of the code.

Software, over a period of time, tends to carry unnecessary baggage. This will result in increased footprint and redundancy. Find ways to get rid of the unused or under used code.

If throughput of the software is vital to your business, then it is time to think about non-blocking programming. The quality of the software might improve when the blocking functional calls are safely replaced by asynchronous ones.

2. Adopting New Programming Standards

We code day in and day out. Find bugs. Fix them. Tweak the performance by replacing dumb algorithms with intelligent and efficient ones. We try to solve problems with limited tools on hand. Run into design issues and often solve them the hard way. Hardly try something outside the traditional methodologies including but not limited to increasing the window of various phases like design, implementation, test, and automation.

Did we ever think adopting new programming standards could improve the quality and maintainability of the software?

The C++ Standards Committee has released a new standard – C++11, which is approved by ISO (The International Standards Organization). The new standard has got lot of useful features to make the software cleaner, safer and faster. If your software is based on older standards (say C++ 2003 or C++98) it is time to think about adopting new standards. Agreed, there are challenges in adoption. Starting from upgrading your compiler to learning what is new in the latest standards and how to apply them. But this transition is vital to see how one can benefit from the new standards.

Even if you have achieved the functionalities provided by the new features in your own way, it is worth considering what the standard library has to offer. Remember standard library is better than handcrafted specialized code.

I have picked up select features, which I feel, is going to be pervasive in the coming days. At first glance, new techniques may look esoteric, but they are simple, efficient, and extremely useful.

Motivating Example -1

Everybody likes “clean” code. It enhances the maintainability as well. Replace lengthy programming statements with short yet understandable constructs.

For example, to store sum of two elements of vector of type ‘U’ and ‘V’ in a variable, say ‘sum’, and use ‘sum’ at some point later, we would write something like this

```
template <class U, class V>
void f(const vector<U>& v1, const vector<V>& v2) {
   vector<U>::const_iterator v1_iter = v1.begin();
   vector<V>::const_iterator v2_iter = v2.begin();
   // ...
   int sum = *v1_iter + *v2_iter; // possible loss of summation result!!
   // ...
   // use 'sum' here.
}
```

If U is of type integer (say 10) and V of type double (say 20.5), then "sum" is 30. The summation result is lost. The type of "sum" should be double to hold the correct result. In other words the type of "sum" should be the one that you get from adding U with V.

Changing the type to "unsigned long" will not work out if either one or both of the vectors contain negative numbers. "long double" may not be desired if none of the vectors contain irrational numbers.

One has to solve this problem the hard way today. This leads to hacks and workarounds.

The new standard has introduced "auto"(The C++ Standards Committee 2013) to solve this problem. "auto" deduces the type of a variable from its initializer. Now the code will look like

```
template <class U, class V>
void f(const vector<U>& v1, const vector<V>& v2) {
   vector<U>::const_iterator v1_iter = v1.begin();
   vector<V>::const_iterator v2_iter = v2.begin();
   // ...
   auto sum = *v1_iter + *v2_iter; // type of sum is what you get from adding U with V
   // ...
   // use 'sum' here.
}
```

As you can see, it has not only solved the problem but also made the code cleaner and understandable.

Note: Type deduction happens at compile time. So, summation might lead to overflow (because the actual value of *v1_iter and *v2_iter are known only at run time), which is not solved by auto.

Few more examples:

```
auto x = 10 + 20;            // x will be of type int
auto x = 10 + 20.5;          // x will be of type double
auto x = INT_MAX + INT_MAX;  // overflow!! x will be of type int.
```

Motivating Example -2

We make all possible efforts to save the runtime overhead of a program. Today we use either "assert" macro or #error preprocessor directive for testing software assertions.

For example, if applications were intended to run only on 32-bit processor, we would write

```
assert(sizeof(long) >= 8);
```

But the assert macro tests assertions at runtime, which is far later than would be desired, and which implies a runtime performance cost. The #error preprocessor directive is processed too early to be of much use.

The new standard offers "static_assert" (The C++ Standards Committee 2013) to save the runtime performance cost. The expressions are guaranteed to evaluate at compile time.

```
static_assert(sizeof(long) >=8, "64 bit code generation not supported");
```

Library developers use assertions extensively in their program. Having something that will assert at the compile time will help in reducing the runtime cost.

Motivating Example -3

Creation of temporary objects is one of the pain points every developer is battling with. Temporary objects slow down a program. Temporary objects are optimized away by the compilers (the return value optimization, for example). But this is not always the case, and it can result in expensive object copies. Consider the following

```
 template <class T> swap(T& a, T& b) {
   T tmp(a);    // now we have two copies of a
   a = b;       // now we have two copies of b
   b = tmp;     // now we have two copies of tmp (aka a)
}
```

But, we didn't want to have any copies of a or b, we just wanted to swap them. C++11 provides “move” (The C++ Standards Committee 2013) semantics to overcome this.

```
template <class T> swap(T& a, T& b) {
   T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

Move semantics allows us to avoid unnecessary copies when working with temporary objects that are about to evaporate, and whose resources can safely be taken from that temporary object and used by another.

These are some of the features added in the latest standards. There could be lot many which are not covered here but might be useful to your application. Please read the latest standards for more information. The key thing which needs to be stressed upon is, look out for newer standards and try to solve problems that are present due to lack of features. This thinking can be applied to any product, independent of the implementation programming language.

3. Platform Agnostic APIs (Application Programming Interface)

Different OS(Operating system) provide different APIs to acquire and use system resources (creation of threads, for example). Software running on multiple OS has to handle system resource and concurrency in different ways, each varying from other OS. But writing cross-platform software takes time. We will end up writing huge amount of code to build platform abstract library.

Maintaining and updating this library on addition of support to new platforms or on upgrade of target platform may be expensive. Resource allocation or any OS specific operation is outside the boundary of software’s business logic. We should try to keep the direct dependency on OS specific API as minimal as possible.

You could either see if the newer standards have got ways (features) to address this issue or use a third party library that provides cleaner abstraction.

The new C++ standards, C++11, for example, added a memory model and provided ways to help the programmers to create threads and achieve synchronization using facilities such as mutexes, condition variables, and atomics. It will make the application development easier as the OS specific operations are blended with the programming language. It will lead to better error handling and debugging. Programmers may now get rid of their platform abstraction library and focus on the solving the business problem.

Here is an example of using C++11 thread (The C++ Standards Committee 2013) to sum up the elements of a huge vector, in parallel.

```
#include <iostream>
#include <vector>
#include <thread>

using namespace std;

class Accumulator {
private:
    vector<double>::iterator begin_, end_;
    double *sum_;
public:
    Accumulator(vector<double>::iterator begin, vector<double>::iterator end, double *sum) :
begin_(begin), end_(end), sum_(sum) {}
    void operator() () {
        while(begin_ != end_) *sum_ += *begin_++;
    }
};

int main()
{
    double sum_1 = 0.0;
    double sum_2 = 0.0;
    double sum_3 = 0.0;
    vector<double> v;

    for(int radius = 0; radius < 1500000; ++radius)
        v.push_back(3.14*radius*radius);

    std::thread t1{Accumulator(v.begin(), v.begin() + 500000, &sum_1)};
    std::thread t2{Accumulator(v.begin() + 500000, v.begin() + 1000000, &sum_2)};
    std::thread t3{Accumulator(v.begin() + 1000000, v.begin() + 1500000, &sum_3)};

    t1.join();
    t2.join();
    t3.join();
    std::cout << "sum = " << sum_1 + sum_2 + sum_3 << '\n';
}
```

4. Heavy But Under Or Un-used Code

Software when started from the grounds up stays lean and focused on the solving problem at hand. But over a period of time it might tend to attract more functionalities and features than required. This extra baggage goes unnoticed as it usually appears in smaller scale in every software release.

How to identify and safely remove the unnecessary code?

Get the latest design document. Form a team with the people that have had worked on most part of the software. Trawl through the source code. Brainstorm among the team and check with the design document and find the chunks which are no longer needed.

What to do with the unnecessary code?

You can either remove the code or replace it with some lighter ones.

How to verify if the software is still safe to use? In other words, how to verify the sanity of the software, post the above mentioned operation?

It is very important to see that the software's business logic functions the same way before and after carrying out the operation. You can follow the steps (this list not complete but are necessary) detailed below for the same.

1. Test the software thoroughly. Developers should be executing all the unit test cases, including the modules that are not impacted. This is to ensure the integrity of the software still remains intact. If needed add new test cases or modify existing ones around the impacted module.
2. Make sure there are no memory leaks by running the software through static analysis tools. There are various static analysis tools (for example, Coverity and Fortify) which give in-depth analysis of the software ranging from reporting "uninitialized variable" usage, dead code to buffer overflow and memory leaks. It is better to address all the issues reported by the tool.
3. Involve the QA and get all the black box testing (Wikipedia 2013) done. QA must be maintaining a repository of all the test cases executed since the birth of the software, meaning all the test cases created and executed across all the releases of the software. This is usually called Regression testing (Wikipedia 2013)
4. There are few bugs which will surface only under certain conditions, say after few hours or days of run or under heavy load. It will be difficult to unearth these kinds of bugs either in unit testing or black box testing. The software has to be soak tested (Wikipedia 2013).

The initial exercise of finding and removing or replacing the unnecessary code is just a starting point. Spend good time in this phase and make sure your changes do not have bearings on the business logic. The entire operation is said done only after QA gives thumps up.

But to successfully complete this operation we require good amount of time and effort. It is where the software test automation (Wikipedia 2013) gives us a big helping hand. If you have enough automation built around testing the complete functionality of the software, you could save a lot of time and effort.

Here is a use case on how replacing the mammoth code base with lighter and "to the point" one benefits us.

1. How do you store the runtime data?

Most of the software have to deal with the data (could be either input data or runtime data) in one or other way. The way the runtime data is handled will go a long way in improving the software quality, especially if your software churns out a lot of data in the runtime.

Study the kind and amount of data used and produced by the product. Make a wise choice before settling for a particular procedure. For example, using a full-blown sql database is not suggested unless your software produce huge amount of runtime data that requires structured query language for retrieval. You could consider storing them in a flat file (encrypt if necessary).

When to write the data in memory into the disk also matters as the disk I/O are expensive. Writing the data as and when created results in frequent disk I/O but reduces the data loss when the software goes haywire (crash or hung state). Frequent disk I/O hampers the software speed. Delayed writing means minimal disk I/O and good runtime speed, but might increase the probability of data loss. Strike a tradeoff between them to improve the quality of the software.

2. Get rid of unnecessary and accidental complexities:

A real world example from author's software development cycle:

We were developing a task scheduler module in which we formulated a state machine. The state machine captures the state of scheduler and the various tasks held by the scheduler. We started with few states and finished our first sprint (we follow scrum methodology for software development). As the scheduler evolved, more and more states were added. The resulting code became too big to understand, debug, and maintain. We ended up exceeding the runtime allocated and the footprint of the software. This is a classic example of introducing accidental complexity into the software.

Lesson learned: get rid of the accidental complexity before it is too late.

How?

The key here is the design. It is supposed to be modular and loosely coupled. Lack of which had led to accidental complexity. But achieving such a quality design at each and every level might be challenging and time consuming. The suggestion is to sense the moment a module gets clumsy and invest some time redesigning that particular module and/or the dependent modules. This is process of restructuring an existing body of code is commonly called Code Refactoring (Wikipedia 2013)

One way to go about in this particular use case is to segregate the state machine from the scheduler. The state machine knows what action to be taken for a particular state. Now all the state handling mechanism happens in one place, there is no code duplicity. Apparently, it will be lot more easier to clean up the states that are no longer valid or add new states as the scheduler evolves over the time.

5. Paradigm Shift: Blocking To Non-blocking

Producing responsive software is a challenge. A responsive software is one that let the users to continue submitting requests before the earlier requests were done.

One of the weakest links in the software is the blocking function call. It could be a blocking networking call, file system call, or individual I/O operations that take long time to complete. I/O operations can be extremely slow compared to the processing of data.

Implementation strategies such as thread-per-connection can degrade system performance, due to increased context switching, synchronization and data movement among CPUs. With asynchronous operations it is possible to avoid the cost of context switching by minimizing the number of operating system threads and only activating the logical threads of control that have events to process.

Many Operating systems offer a native asynchronous I/O API for developing high performance network application. These asynchronous mechanisms handle the blocking system calls without requiring programs to use concurrency models based on threads and explicit locking.

Long-duration operations are performed asynchronously by the implementation on behalf of the application. Consequently applications do not need to spawn many threads in order to increase concurrency. Consider using them and see the difference in performance.

The next time you create a socket think about making it a non-blocking one. Windows provide I/O completion port (IOCP) to achieve asynchrony. UNIX based system offer select, poll, kqueue, etc to achieve the same. Boost asio library (Kohlhoff Christopher, 2013) provides a good abstraction over windows and non-windows platform.

6. Conclusion

Adopting new standards may not be a panacea for all the software problems. But one can definitely improve the current quality of the software at various levels. The new standards might provide cleaner, safer, and faster ways to solve your problem. Use caution while adopting the latest standards.

Trawling through the thousands of lines of source code and finding the chunks which are no longer needed warrants careful examination of the source code. Run through all the business case scenarios and check the changes you made do not impact the flow. Revisit your unit test cases. Run static analysis tools to check if there are still dead codes out there in the product. Remember dead codes are revealed only by good unit test cases. Ideally a product should have all its functions covered by the unit test cases.

It is more difficult to develop applications using asynchronous mechanisms due to the separation in time and space between operation initiation and completion. Debugging asynchronous applications may also be harder due to the inverted flow of control. But the overall throughput of the software will rise sharply.

References

The C++ Standards Committee, 2013, http://www.open-std.org/JTC1/SC22/WG21/

Wikipedia, 2013, https://en.wikipedia.org/wiki/Test_automation

Wikipedia, 2013, http://en.wikipedia.org/wiki/Regression_testing

Wikipedia, 2013, https://en.wikipedia.org/wiki/Test_automation

Wikipedia, 2013, http://en.wikipedia.org/wiki/Code_refactoring

Kohlhoff Christopher, 2013, http://www.boost.org/doc/libs/1_54_0/doc/html/boost_asio.html

Performance Benchmarking an Enterprise Message Bus

Sumant, Pramod, Anurag

sumant_vashisth@mcafee.com, pramod_sharma@mcafee.com, anurag_sharma@mcafee.com

Abstract

An enterprise message bus is used for designing and implementing the interaction and communication between one or more interacting software applications or services. A messaging architecture is a combination of a common data model, a common command set, and a messaging infrastructure to allow different software services to communicate through a shared set of interfaces.

One of the challenges in the messaging architecture is to monitor and measure the system performance, scalability bottlenecks and reliability in real time operation. Performance metrics, for example, expected event notification latency, utilization and throughput of various components (like event broker), handling of messages of different sizes, are essential to determine the bottlenecks in the design, capacity of the system, optimal configuration and other parameters. Also, the systems need to be continuously monitored and fine-tuned based on the changing requirements in real-time scenarios.

In this paper we will present the process, method and test suite for evaluating the performance, reliability, and quality of a real world implementation of the messaging architecture using different messaging patterns. This paper will cover the following topics:

1. Provide an overview of the messaging architecture implementation in the security domain with different messaging patterns (pub-sub, request-response) and communication modes (in process, out of process, out of box)
2. Key factors to be considered for performance benchmarking.
3. Automation for the benchmark setup, configuration and execution.
4. Example test scenarios and results for the performance benchmarking test.

Biography

***Sumant Vashisth** is Director of Engineering, Security Management Business Unit at McAfee. Besides, managing and leading teams across various industries during his long career, Sumant is a techie at heart. You can often see him chatting with engineers about the latest in technology and day to day technical challenges.*

***Pramod Sharma** is a Software Architect at McAfee, with more than ten years of industry experience. An acknowledged technologist, Pramod has eight patent pending inventions to his credit. Passionate about software quality and process improvements, Pramod is a vocal champion of adopting changes, and learning from the industry. His interests span security management for embedded and mobile devices, and scalable architectures for distributed systems.*

***Anurag Sharma** is a Principal SDET Engineer at McAfee, with around eight years of industry experience in software testing and quality assurance.*

Introduction

The enterprise message bus is a lightweight framework that provides flexible, loosely coupled integration across a broad range of enterprise applications. The key to dynamic interconnection across security firewalls, application protocols and languages is the messaging backbone. This architecture provides guaranteed delivery of messages among services and endpoints, performing security checks and protocol and data conversions on the fly. The message buses are designed to be fault tolerant and fail-safe. It would be safe to say, the enterprise message bus architectures will drive the growth of many businesses in coming future.

Motivation

Enterprise level message-bus architectures are increasingly becoming common in various domains across IT industry from banking and financial sector to transportation, supply chain management and healthcare sector. Newer and novel applications using message buses are being designed every day. Most of the message buses are deployed in highly data intensive and distributed networks. Predicting the message bus throughput, configuration, topology and workload in real-life scenarios is always a challenge. McAfee is in forefront in adopting and implementing message bus architecture for security applications. Looking at the current landscape, there is no standard method or benchmark to understand and compare the performance of various message bus architectures. This information can be necessary to design the most efficient system. This information may also help the enterprises to decide between an existing open source message bus frameworks or to develop of their own.

Key Considerations

As we start developing the benchmarks for validating the performance and scalability of our message bus or any messaging architecture, in general, following points must be considered –

1. Benchmarks must be based on real world applications but not to be biased by any particular domain. This dilutes the usefulness across applications (*Financial applications were the early adopters of event based architecture. Current standards slightly lean towards these*).
2. Benchmarks standards and tests should measure the performance of the architecture, as a whole, not the various components in it.
3. Benchmark standards should not be implementation specific i.e. based on any particular platform, software, language of implementation, topology etc.
4. Tests should be reproducible, scalable and exhaustive.

Benchmark Categories

Configuration

The machine configuration used for benchmarking tests, is the single most important factor. If the tests are performed on a high end machine with 8 core processor, 8 GB RAM etc., the performance results are certainly going to be better, but, they will be misleading too. Not everyone in enterprise will have such a machine, therefore, the following points should be considered while selecting the configuration for running benchmark tests to

1. Identify the configuration and document it in advance. It is better to select a set of three configurations – one high end (best), one average and one low-end (worst) configuration and execute the benchmark test on all three configurations.

2. For out-of-machine (network) communication, network configuration parameters like network topology, the network bandwidth etc. have important impact on end results therefore; these parameters should be identified in advance and documented.

3. Platform parameters like – processor (speed and number of cores), operating system, architecture, physical memory etc. should also be documented.

4. No other processor or network hogging tasks should be running while the benchmark tests are conducted.

Communication Modes

The messaging architecture can have two communication modes – 1) Client-Server or Request-Response mode, where, the client sends a message to be server and then, receives the response either synchronously or asynchronously. 2) Publisher-Subscriber mode – In this mode, the subscribers for some particular “Topic” of interest and then, gets notified when topic of interest is available on the bus. The enterprise message bus may have different performance results for both communication modes, and more often than not, tuned for one specific communication mode, therefore, considering both communication mode while doing performance benchmarking is very important.

Message Reach

In enterprise message bus architecture, the message consumers can be within the same process, in a different process on same machine or out on network. The message delivery latency, throughput and reliability vary depending on the relative location of client and server. Therefore, this factor needs to be considered while designing the benchmark test suite.

Identifying Bottlenecks

Often, the overall efficiency of the messaging architecture depends on the performance of one or two entities in the architecture for example, the broker or the packet forwarder or the connection manager. These entities drive the limits of message bus scalability and throughput. Therefore, identifying such bottleneck nodes and targeting the tests, which push the limits of these nodes, is very important.

Latency

Latency is the time taken for a message round trip i.e. the time taken for a request from client to reach server and corresponding response to reach client. For a publisher-subscriber type communication mode, only one way time is measured. Latency is an important measure for the performance of message bus. Following factors must be considered while designing the tests for targeting the latency factor –

1. For within the process communication, the latency is negligible therefore, the in-the-process communication configuration is not a good choice for latency tests.

2. The out of process and out of machine, tests should be considered. In the out of machine tests too, the machines or nodes which are distant (IP hops wise, not physically), should be considered for communication.

3. The server or receiving node should perform some minimal processing before sending the response. Generally, to reduce the latency, the tests are designed such that the receiving node doesn’t perform any operation on the received message and just returns the response, which is not a practical scenario. Therefore, some minimal processing at receiving node should be performed.

4. Gradually, increase the number of simultaneous connections and message size and record the average time taken for response.

Throughput

Throughput is the measure of the average number of successful messages delivered and responses received over a communication channel. The throughput is an important factor while designing message bus or choosing any open source message bus for enterprise usage. Less throughput can be unsuitable for some highly data intensive networks. Following factors must be considered while designing the tests for targeting the throughput factor –

1. The machine configuration has a major impact on the throughput, therefore, the standard machine configuration should be used while designing and executing the throughput tests.

2. More number of threads spawned at the client and server sides can also increase the throughput, therefore, the throughput values should also correlate with the number of parallel threads of execution.
3. Moreover, some processing at the receiving node should also be taken into account while measuring the throughput.

It is also important to note here, that the throughput, reliability, latency are not independent factors, they should all be considered together while designing the test scenarios.

Thread Pool vs. Thread IO

Thread Pool is where a number of threads are created to perform a set of tasks. Typically, there are many more tasks than threads. As soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed. The thread can then terminate, or sleep until there are new tasks available. Thread IO is where a single worker thread performs the tasks sequentially. Thread pool provides better performance and throughput compared to creating a new thread for each task. For thread pool, the number of threads used is a parameter that can be tuned to provide the best performance. Therefore, if the message bus architecture supports thread pool and is configurable between the thread pool and thread IO, benchmark tests should be conducted for both configurations.

Reliability

Reliability in this context is measured in terms of number of packets lost in a given period of time. Some applications like financial applications require highly reliable networks. Following factors must be considered while designing the tests for targeting the reliability factor –

1. Push the messages continuously without delay and determine the number of messages dropped. Also, monitor the parameters like the CPU and memory utilization. Note that some message bus can crash in this scenario.
2. Run the tests for a large duration for example, two to three days continuously and monitor the messages dropped and other vital characteristics.
3. Varying the network configuration i.e topology and number of processes/machines communicating, can also generate significant reliability information.

Resource Utilization

In every test scenario, monitoring the resource utilization is very important. The following important resources should be considered –

1. CPU utilization – By general principle, more than 50% CPU utilization is not considered good.
2. Memory usage – Should be kept to minimum.
3. Number of threads – Always a tradeoff between the throughput and reliability.
4. Disk I/O – Disk operations are expensive and increase latency. They should be kept to minimum.
5. Number of open sockets/pipes – Expensive and potential security threats. Try to keep it to minimum. Consider intelligent designs concepts like reusing the sockets/pipes, opening connections on demand etc.

Security

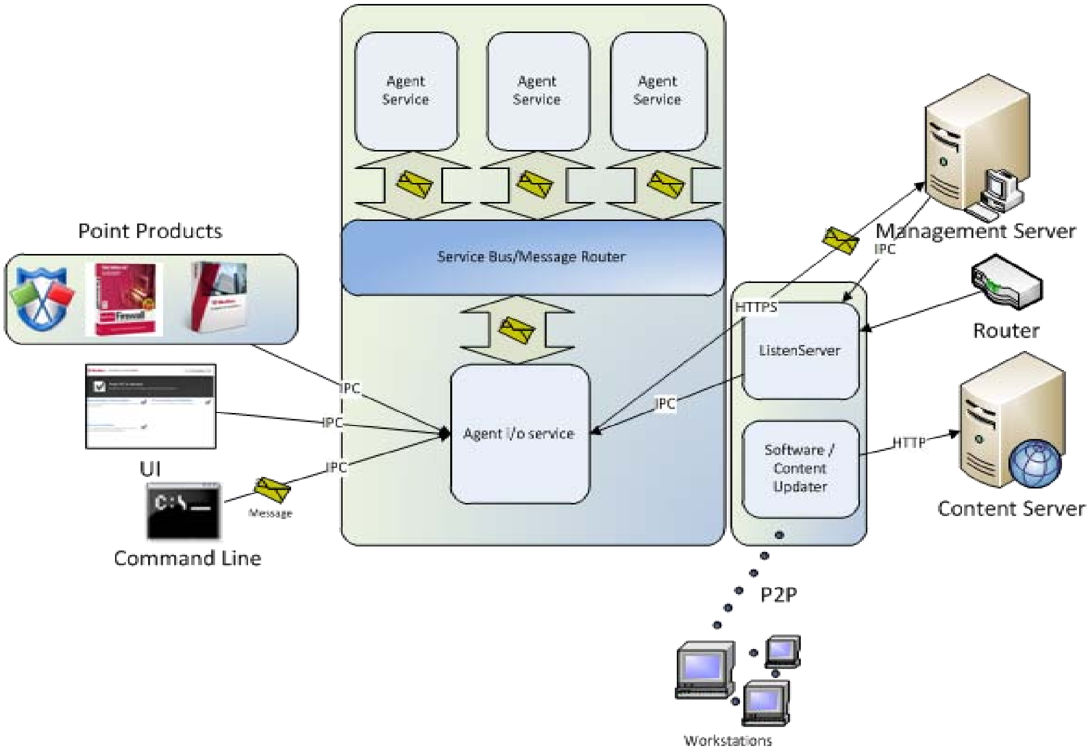
Most message bus architectures deploy some level of security either at message level or at connection level. Throughput can go down significantly when the message security aspect is introduced. Various security mechanisms like connection based authentication (connection based or message based), encryption, integrity check (hash algorithm), authorization etc., slow down the message bus at varying levels. Considering this, benchmark tests should be conducted for both secure and non-secure environments.

A Practical Example

McAfee Agent Messaging Architecture – A brief overview

The role of the McAfee Agent is to provide remote manageability of computing devices connected to a network, in particular to synchronize and enforce polices set forth by a management server. The agent is also responsible for sending device properties, events and operational data collected from device to the management server for reporting and other purposes. The agent will typically manage one or more security point products, each providing some kind of protection technology or ‘business value’ on the device. The Agent acts as a conduit for data between such point products and the remote management server, allowing exchange of predefined data types as well as private data formats. In turn, the management server typically manages several agents and associated point-products. The management server maintains a logical representation of each endpoint and allows the security administrator to configure policies for individual or groups of endpoints. The server also provides extensive reporting capabilities allowing the administrator to monitor the security status of the managed endpoints.

Endpoints and their associated management server can be deployed in an almost endless number of different network topologies. From the simplest, closely connected LAN environments to laptops and mobile devices travelling in and out of different network boundaries, possibly communicating via proxy servers, firewalls or even environments where devices communicate intermittently over low bandwidth but expensive satellite or dial-up connections. As you can see, Service-oriented Architecture (SoA) based on messaging architecture is the backbone of this enterprise application.



Example Test Scenarios

Some performance test scenarios are listed below. During each test, the resource utilization like CPU utilization, no. of threads, memory usage etc. should be monitored. Please note that this is not an exhaustive list of tests. Once the performance test setup is designed for automated test execution and monitoring resource utilization periodically, more test cases can be designed for message bus architecture performance fine-tuning.

1. Varying message size – Perform the communication between single synchronous server and client for a fixed amount of time varying the message size from 1 byte to 1 Gigabyte and count the number of successful. Perform similar test for asynchronous communication.

2. Varying number of clients – Gradually increase the number of clients communicating (fixed size messages) with a single server and measure the number of successful communications in a pre-determined fixed amount of time. Perform this test in both thread pool and thread IO mode.

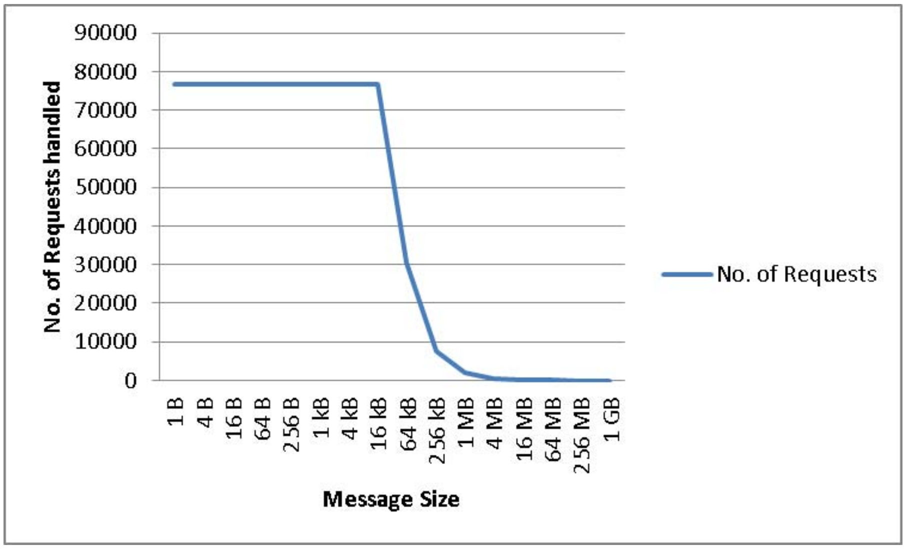
3. Implication of security - Perform the communication between single synchronous server and client for a fixed amount of time varying the message size from 1 byte to 1 Gigabyte and count the number of successful. Now, enable the message security and perform the test again. Observe the decrease in throughput when the message security is in place.

4. Thread Pool and Thread IO - Observe the number of clients that can be served by a single client in thread IO and thread pool configurations.

Example Performance Results

Following graph illustrates the behaviour of one server and one client process on the same machine communicating in the synchronous manner. The test is performed for 10 minutes duration where the client sends 1 request per millisecond. The message size is varying from 1 byte to 1GB. It is observed that the upper threshold for message size is 64MB, above that; messages could not be reliably transmitted. On the other side, the maximum ~78000 messages could be transmitted during the test duration. The throughput and latency parameters can be easily derived from this to data and performance can also be fine-tuned (Operating Platform – Intel ® Core ™ i7-2600 CPU @ 3.40GHz (8 CPUs), 8GB RAM running Windows 7 Professional 64-bit operating system. Network characteristics are not relevant for the out of process tests).

Approximate throughput calculation – 16 KB message size X 8 bits/byte X 2 (both request and response) X 78000 messages/ (10 X 60) seconds = ~32 Mbits/second



References

Zero MQ performance results at - http://www.zeromq.org/area:results