TWENTY-SECOND ANNUAL PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

OCTOBER 12 - 13, 2004

Oregon Convention Center
Portland, Oregon

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

TABLE OF CONTENTS

Preface v

Conference Officers/Committee Chairs vii

Conference Planning Committee viii

Keynote Address – October 12

Agility for Testers 1
Elisabeth Hendrickson, Quality Tree Software, Inc.

Keynote Address – October 13

What Does a Tester Need to Know in 2005 and Beyond? 23
Sam Guckenheimer, Microsoft Visual Studio Team System

Metrics Track – October 12

Software Metrics: Don't just track – know when to act! 25
Diane Manlove and Stephen Kan, IBM Corporation

Proactive Data Validation in the Data Warehouse 37
Thomas O'Meara, Intel Corporation

Effective Metrics for Pragmatic Project Managers 41
Johanna Rothman, Rothman Consulting Group, Inc.

Guiding Software Quality by Forecasting Defects using Defect Density 57
Bhushan B. Gupta, Hewlett-Packard Company

Software Estimation Models: When is Enough Data Enough? 65
Tim Menzies, Portland State University

Management Track – October 12

Lean Cycle Time Reduction Techniques 73
Neil Potter and Mary Sakry, The Process Group

A Process for Very Small Projects 107
Dave Fleck, Wacom Technology Corp.

Keeping QA Job Skills Relevant in a Competitive Market 115
Peter Yarbrough and Claudia Dencker, Software SETT Corporation

Controlled Agility 131
Dr. Geoffrey J. Hewson, Software Productivity Center, Inc.

Becoming a Better Tester – Staying Relevant **145**
Kelly Whitmill, IBM Corporation

Testing Track – October 12

An Automation and Analysis Framework for Testing Multi-tiered Applications **161**
Kingsum Chow, Zhidong Yu, Lixin Su, Michael LQ Jones and Huijun Yan, Intel Corporation

Pairwise Testing: A Best Practice That Isn't **175**
James Bach, Satisfice, Inc., and Patrick J. Schroeder, Milwaukee School of Engineering

Design Patterns for Customer Testing **193**
Misha Rybalov, Quality Centered Developer

Using a Database to Automate and Manage Interdependent Functional Tests Across Multiple Platforms **215**
Jennifer Smith-Brock, ClearLake Solutions, LLC, and Jeff Pearson, Personality, LLC

Open Source Testing Framework for Handling a Multiple Product Stack **225**
Judith Lebzelter, Open Source Development Lab

Tools May Come, and Tools May Go … But Some Things Never Change **235**
Scott A. Whitmire, ODS Software, LLC

Process Track – October 12

How Quality is Assured by Evolutionary Methods **249**
Niels Malotaux, N R Malotaux – Consultancy

Demystifying Microsoft's Quality Assurance Process **263**
Ambrosio Blanco, Microsoft Corporation

Dynamic Design Analysis: Testing Without Code **273**
Scott A Whitmire

Retrospectives: The Impact to Process Improvement **289**
Debra Schratz, Intel Corporation

The Requirements Dilemma: It's Hard Work **305**
Debra Aubrey, General Dynamics C4 Systems

Metric Track – October 13

Estimating and Managing Project Scope for Maintenance and Reuse Projects **321**
William H. Roetzheim

Software Process Improvement Frameworks Perceived Impact on TQM Practices and Financial Performance of Software Organizations **329**
Arthur Oster

Instituting a 360 Degree Metrics Program at Cadence **349**
Mark Noneman, Cadence Design Systems, Inc.

Defect Analysis – A Tool for Improving Software Quality **365**
John Balza, Hewlett-Packard Company

Management Track – October 13

Effective Team Practices for Facilitative Leaders: Creating and Sustaining Project Momentum **381**
Diana Larsen, FutureWorks Consulting, LLC

Are You Hiring Yesterday's Testers? **385**
Johanna Rothman

Just Unit Testing **395**
Brian G. Smith, Xmcell.com, and Richard P. Vireday, Intel Corporation

Intercultural Needs Assessment Project **407**
Ms. Balbinder K. Banga, Portland State University

Can You Hear Me? ... Can You Hear Me Now? **425**
Yana Mezher, Software SETT Corporation

Leading a Global Test Team – From Terror To Bliss **435**
Kathleen Kallhoff, IBM Corporation

Effective Solutions for Global Software Development **447**
Robert Roggio, University of North Florida; Rahul Dighe, Inteliant Technologies, Inc.; and Sharada Bobbali, CACI Federal, Inc.

Testing Track – October 13

Simulators, or How to Turn Your System's World Upside Down **463**
David Liebreich, Independent Consultant

How to Make Your Bugs Lonely – Tips on Bug Isolation **471**
Danny R. Faught, Tejas Software Consulting

Agile Testing **481**
Bret Pettichord, Pettichord Consulting

Getting Started With Automated Testing **483**
Mike Kelly, Consultant for Computer Horizons Corporation

Test Improvement under Fire **491**
Nasa Koski, Microsoft Corporation

Process Track – October 13

Does Software Development come under the Total Quality Management umbrella? **509**
Ita Richardson, Norah Power and Kevin Ryan, University of Limerick, and Gerry Coleman, Dundalk Institute of Technology

Encouraging SME Software Companies to Adopt Software Reviews in Their Entirety **525**
Ilkka Tervonen and Lasse Harjuma, University of Oulu

Open Source Software Engineering Tools **537**
Les Grove, Tektronix, Inc.; Wells Mathews, Micro System Engineering, Inc.; Rodney Hickman, Intel Corporation; and Kal Toth, Portland State University

Inherent Risks in Object-Oriented Development **547**
Dr. Peter Hantos, The Aerospace Corporation

ROI for Software Development Operations **555**
Wolfgang Strigel, QA Labs, Inc.

Proceedings Order Form ..... **last page**

President's Welcome to the 2004 Pacific Northwest Software Quality Conference

Our theme for this year's conference is "Are you keeping up with emerging technologies and processes?" As the economy in 2004 slowly recovers from the doldrums that we have all suffered we hope that our conference will provide training and resources which will help ensure that your skills will keep you sought after and competitive in the work place. We selected our theme to focus our efforts on papers, workshops and some new venues that we thought would be important towards this goal.

We have two keynotes speakers, Elizabeth Hendrickson and Sam Guckenheimer, who will be talking about the changing role of testers into the future. Although the topic is testing they are also talking about some of the overall process changes affecting development and management of software. We hope that you enjoy these talks.

You will also find that many of our workshops reflect our theme. Here you will see a wide variety of topics ranging from interviewing techniques to managing global teams. These workshops cover a wide range of skill levels offering something for everyone, from the beginning tester to an advanced manager.

The conference theme and our push to expand our networking with potential authors resulted in more submitted abstracts than past years. To our delight, the quality of these abstracts was exceptional. Instead of rejecting many excellent papers, we decided to add an additional track to accommodate these fine papers. If you have a topic you feel passionate about and would like to speak at our next conference, please consider sending us an abstract.

We have a new venue at the conference called "Open Spaces." This is a natural extension to the "Birds of a Feather" lunchtime program that we have used in the past. In Open Spaces you, the conference attendee, pick the topics you would like to discuss and help lead the discussion. We have created this space as an all day track to give people a chance to really network with others about topics that they feel passionate about or where they would like to get some practical knowledge from others in the field.

We have noticed that many people are in a quandary over the use of traditional software processes and the newer XP/agile approaches to software development. Some people see these practices at odds with each other and are not sure which way to go. We decided to sponsor a panel on this topic in conjunction with the Rose City Software Process Improvement Network (SPIN) entitled, "Agile and Traditional Development: A Creative Synergy?" We hope that you enjoy this panel and that it answers any questions you may have on this topic.

You may be aware that PNSQC is a non-profit, volunteer organization. The board of directors and committee chairs have done an incredible amount of work in the pursuit of a conference that they believe in and being part of a team that works very well together. Being president of such a group has been an extremely rewarding and fulfilling role for me. I hope that you will consider being part of our team. Please talk to us at the PNSQC booth if you would like to do so.

I hope you enjoy the changes in this year's conference. We would very much like to hear your feedback, suggestions, and any other changes you would like to see at next year's conference.

Please feel free to talk to any of the conference organizers in person or through the web at www.pnsqc.org.

Paul Dittman
President,
Pacific Northwest Software Quality Conference

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Paul Dittman – PNSQC President and Chair

Debra Schratz – PNSQC Vice President
Intel Corporation

Richard Vireday – PNSQC Secretary
Intel Corporation

Cindy Oubre – PNSQC Treasurer & Exhibits Chair

Sue Bartlett – 2004 Keynote Chair
System Harmony, Inc.

David Butt – Program Co-Chair

Rick Clements – Workshops Co-Chair
Cypress Semiconductor

Esther Derby – Program Co-Chair
Esther Derby Associates

Rebecca Gee – Workshops Co-Chair
HOSTS Learning

Shauna Gonzales – Open Space & Birds of a Feather Chair
Nike, Inc.

Randy King – Communications Chair
Mentor Graphics, Inc.

Doug Reynolds – Program Co-Chair 2005
Tektronix, Inc.

Patt Thomasson – Publicity Chair
McAfee, Inc.

2004 CONFERENCE PLANNING COMMITTEE

Kit Bradley
PSC Inc.

Julie Fleischer
Intel Corporation

Manny Gatlin
Timberline Software

Cynthia Gens
Loui Canz - Louis XV

Brian Hansen
Oregon Health Science University

Kathy Iberle
Hewlett Packard Company

Diana Larsen
FutureWorks Consulting, LLC

Tim Menzies
Portland State University

Howard Mercier
Integrated Information Systems

Jonathan Morris
Unicru, Inc

Elizabeth Ness
Hewlett-Packard Company

PS Rao
Intel Corporation

Jean Richardson
BJR Communication, Inc.

Ian Savage

Eric Schnellman
JanEric Systems

Erik Simmons
Intel Corporation

Woldgang Strigel
QA Labs

Ruku Tekchandani
Intel Corporation

John Veneruso

Scott Whitmire
Odyssey Software & Consulting, Inc.

Agility for Testers

Elisabeth Hendrickson

Pacific Northwest Software Quality Conference
October 2004

Background

"Agile" is a great buzzword. In everyday language, "agile" means adaptable or able to move or respond quickly. Given the pace of software development today, everyone wants to be agile.

But it's more than a buzzword. The phrase "Agile Methods" has become an umbrella term for a collection of methodologies that increase agility, including Extreme Programming (XP), Scrum, Crystal, and Lean Development. The Agile Manifesto describes the values of the Agile community: individuals and interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation; and responding to change over following a plan.[1](The Agile Alliance, 2001)

Agile teams accept change as inevitable and tailor their processes accordingly. Short iterations mean that stakeholders can see steady progress and provide frequent feedback. Continuous integration means that if one part of the system isn't playing nicely with others, the team will find out almost immediately. Merciless refactoring, where programmers improve the code internally without changing its external behavior, prevents code from becoming fragile over time. Extensive automated unit tests ensure that fixing one bug won't introduce two more.

Agile teams test early, often, and relentlessly. Many Agile teams perform extensive unit testing and collaborate with users on creating automated acceptance tests. Some teams even write automated unit tests before writing the code those tests will exercise. Agile programmers hold themselves accountable for the quality of the code, and therefore view testing as a core part of software development, not a separate process performed at the end.

The original founders of the Agile community are programmers and the early literature had little to say about the role of independent testers. In fact, just a few years ago, some Agile proponents optimistically suggested that diligent early unit testing and automated customer-driven acceptance testing reduced the need for independent system testers.

These days, many teams, including one I have been working with for several months, are discovering that testers can indeed add value. Skilled testers have an uncanny ability to expose the flaws in software in a surprisingly short amount of time. Programmers are

---

[1] These values are expressed in the Agile Manifesto, reproduced in its entirety including the copyright notice and author names at the end of this paper. The Agile Manifesto is also available from the Agile Manifesto website: http://www.agilemanifesto.org.

Copyright © 2004 Quality Tree Software, Inc.

often dumbfounded by the speed with which a good tester can find a defect in supposedly completed code. "But I tested that!" they protest.

So the question is not, "Are testers needed on Agile projects?" but "How can testers contribute most significantly to Agile projects?"

In this paper, I contrast "Traditional"[2] and Agile testing, describing how independent testers can contribute effectively on Agile projects by shedding some practices that no longer fit and adopting others made possible in an Agile context.

What's Really Different about Agile

Unfortunately, as often happens with great buzzwords, some teams claim they're "going Agile" when all they're really doing is compressing the schedule, tossing out the documentation, and coding up to the last minute. As Abraham Lincoln said, "If you call a tail a leg, how many legs does a dog have? Four. Because calling it a leg doesn't make it a leg." Similarly, calling a project or team Agile doesn't make it so.

For teams that adopt Agile development practices, agility means delivering working code at frequent intervals. Each of the Agile methods has its own set of practices for doing that, but most seem to agree that short iterations, continuous integration, lots of team communication, and frequent feedback are critical. Agile methods spurn any activity that doesn't add value to the final product. In lean manufacturing terms, activities that don't contribute directly to the end product are considered waste.(Poppendieck & Poppendieck, 2003)

Agile teams are typically co-located in a team room where big, visible charts help convey information at a glance. Some use online collaboration tools like a Wiki (a web site with pages anyone could edit from their browser).[3] All seek frequent feedback from the business stakeholders.

XP lays down the most rigorous set of development practices of the Agile methods. In XP, programmers pair up to write code so every line is inspected as it's written. They develop the code test-first, so there is always a comprehensive set of automated unit tests that must pass before the code can be checked in.[4] Both XP and Agile programmers are sometimes called "test-infected," meaning that they've been infected with the idea that testing early and often (as opposed to debugging) will help them write better code.[5]

Agile developers have developed a shared vocabulary that expresses their programming ideals.[6] They avoid "BDUF" (Big Design Up Front) and instead Do the Simplest Thing That Could Possibly Work. When someone suggests adding code "just in case," they're reminded "YAGNI" (You Aren't Gonna Need It). The Once and Only Once (OAOO) principle aims at keeping duplication out of the code. This emphasis on simplicity is one

---

[2] The word "traditional" is somewhat ironic given the relative youth of the software industry. By traditional, I mean a formal test process that is focused on the independent verification and validation of the system under test.

[3] For more information on Wikis, see http://www.wiki.org

[4] For more information on XP, see Beck's *Extreme Programming Explained*.(Beck, 1999)

[5] As evidence of the growing popularity of testing among the programming community, there were several books on test driven development (TDD) published in the last couple of years.

[6] A good place to start investigating Agile lingo is http://c2.com/cgi/wiki?YouArentGonnaNeedIt

of the things that sets Agile practices apart from traditional heavy weight and code and fix software development.

Perhaps more important than any single Agile practice is that within the Agile community, these practices are considered the norm. Agile programmers truly think differently about software development.

Conflicting Values: Traditional Testing and Agile Development

In traditional testing, system testing is usually done at the very end of the project and is often the first time all the components of the system are integrated together. The predictable outcome is that system testing reveals numerous issues that must be resolved before release. Furthermore, because the system testing occurs at the end of the development cycle and the release date is often fixed, the time for system testing is likely to be squeezed when development dates slip.

In order to resolve these conflicting forces—too many problems to find and not enough time in which to find them—formal system test and quality practices emphasize preparation, documentation, adherence to plan, clean handoffs, and strict change management.

These formal practices arose from contexts in which the team relies heavily on system-level testing performed by an independent test group as the last step in the development cycle.[7] They make less sense for Agile teams with test-infected programmers.

Traditional Wisdom	**Agile Perspectives**
Strict change management	Change is inevitable.
Comprehensive documentation	Working software is more important. And face-to-face communication is better anyway.
Up front planning	Plan to the next iteration.
Formal entrance and exit criteria with signoffs	Collaborate, don't hand off.
Comprehensive system-level regression tests	Detect defects earlier with automated unit tests and continuous integration.

This conflict in values results in tension when testers attempt to apply the same tried and true techniques from heavyweight or code and fix contexts on Agile projects. As one programmer commented to me, "We've changed the way we do things completely, but our QA manager wants to keep doing the same old heavyweight things. They just don't have any value any more."

[7] There is a growing movement within the testing community toward Context Driven testing. For more information, see the Context Driven Testing Wiki, http://www.context-driven-testing.com/wiki/

An Agile Testing Story

I recently had the good fortune to participate in three successive projects with an XP team. I was on the projects part time, a member of the development team but with a distinct role as a tester. This is unusual for an XP team where individual team members are expected to be generalists without specific roles. Both the team and I were experimenting with how a testing professional could add value on an XP team.

My first day on the project, one of the programmers helped me install the development tools and connect to the source control system. My task was to begin exploratory testing on the features that were completed.

A traditional QA department wouldn't have touched the application at this point because it was nowhere near ready for system test. But by testing so early, I found issues while there was still plenty of time to address them. This practice paid off almost immediately. One of the first bugs I found required not just a simple code change but also a design change.

It's worth noting that I could not have been as effective testing early if the programmers hadn't been writing the code test-first. I came into the project in the second week and was able to be productive executing tests immediately, but only because the software already behaved predictably under most conditions. If it had not been so clean, I would have spent most of my time attempting to get the system to work at all.

Agile Implications for Testing

Before joining the XP team, I already had plenty of evidence that testing software developed using Agile practices would be very different from testing software developed in a code-and-fix or process-heavy context. The good news is that the software is easier to test because it is more reliable and testable. But I discovered that Agile methods pose three key challenges for testing:

- The software is a moving target
- Short iterations and frequent releases increase time pressure
- The risks are shifting

Software as a Moving Target

In traditional testing, system testers typically push for an early code freeze so they have time to test the feature-complete system.

This simply isn't possible in an Agile context. There may be some short period of time in which new stories are not being implemented, but it is not the month or more that system test teams are accustomed to having for a full regression cycle.

Agile testing must take that continuous change into account. That means finding ways to test earlier.

Short Iterations

Traditional test teams accustomed to having 4 – 6 weeks at the end of a release cycle to do nothing but end-to-end system or regression tests will discover that simply isn't possible in an Agile context. The iterations are too short.

Agile testing must provide feedback not only sooner but faster. The solution to these first two challenges is not to find ways to do the same type of testing only faster, but to re-evaluate the testing processes entirely.

Shifting Risk Profiles

For testers accustomed to working in a code-and-fix context where any small change can have an unanticipated ripple effect through the rest of the system, the lack of time for full regression tests may be frustrating. They perceive any change to the system as introducing new risks. Fortunately, the risk that such a change could occur, while not gone altogether, is significantly less with an Agile project where the programmers have created extensive automated unit tests.

Agile projects still have risks, of course. Agile practices, particularly XP programming practices, mitigate risks associated with late breaking change. Other risks remain. The Agile tester's challenge is to detect those new risks and determine how to respond to them.

For example, I realized on the XP projects I worked on that we had significant risks around:

- The parts of the system that stayed mocked out the longest.
- Our assumptions about production data we didn't control.
- Areas of the system that didn't have many automated acceptance-level tests.

By understanding the risks, I was able to focus my test efforts more effectively. I found a number of issues by testing around each area where we interfaced to other software and data that we didn't control.

Another example of shifting risks: one QA group was using manual smoke tests as a way of ensuring they didn't waste time testing software that didn't meet a basic level of quality. In the past, about half the builds had been Dead on Arrival (DOA) in QA. A few months after the programmers switched over to XP, the QA group realized that the builds always passed the smoke tests. Now they were wasting time by smoke testing—the software just worked.

This QA group discovered that they no longer had to worry about the risk that the build would be broken. So they abandoned their manual smoke tests and focused on running more interesting, informative tests.

Toward More Agile Testing Practices

I discovered that while everything I already knew about testing helped me work with the XP team, I had to adjust my style in order to work more effectively with the team.

Offering Feedback

I provided feedback to the team in various forms: sometimes with a side-by-side demonstration while pairing, sometimes by adding a new story card, sometimes by making a note on the Wiki, sometimes by creating an automated test to demonstrate the problem, sometimes by entering a bug record in our Bugzilla database. The form of the feedback was less important than the content.

At first I was a bit uncomfortable with this free form flow of feedback. I was accustomed to capturing items in a bug tracking system. I missed the ability to search through a central database of issues to see what else had been found and to review the issues I'd raised already.

My discomfort remained until I realized that providing the information in the most usable form was the most important contribution I could make.

As an example, toward the end of the project I discovered that a series of searches didn't work. When I showed one of the programmers the issue, he said "Oh, I know what that is. Hang on while I fix it." In this case going through the effort of filing a bug would simply have increased the overhead of the communication enough additional benefit to justify the cost.

Avoid the Customer Surrogate Role

Our customer for this software was not working in the same office. He came in to work with the team once a week. This time onsite enabled us to demonstrate the software, get his feedback, and have him clarify stories. Sometimes I did the demonstration. Other times one of the programmers did.

Because our customer was offsite, there were times when the programmers asked me customer questions. "Should we make the error message look like this or that?" they'd query. It was tempting to slip into a customer surrogate role. Because I was not very familiar with the domain, I was able to resist temptation. It turned out to be a good thing: my natural inclination turned out to be wrong in most cases.

One of the key aspects of the Agile movement is that the business stakeholder is responsible for the business decisions.

Minimize Documentation

Test documentation can account for a large percentage of the test effort. I've been conducting informal polls in my classes to understand exactly how much time test documentation takes. 135 testers across 57 organizations revealed that testers spend about one third of their time just documenting test cases.[8]

Knowing the cost of heavyweight documentation and keenly aware of the tight timeframes, initially I resisted documenting my testing. But I learned on the first project that the rate of change actually made it more important that I document what had and had

---

[8] It's worth noting that this is an unscientific poll where the data may be skewed by several biases, including selection bias (only people in my classes participated) and self-reporting bias (participants were reporting their time from memory). Despite the poll's flaws, the results are both interesting and consistent with my own experience working in document-heavy environments.

not been tested. As an example, at one point a programmer asked me, "Did you test for that?" I replied, "Yes" because I was sure I'd run that test sometime in the last few days. But the more important question was, "Have you tested for that since yesterday?" I would not have been able to answer that question.

Although we all realized that documenting my test effort was important, we didn't want that documentation to become burdensome.

We were already using a Wiki, a web site with pages anyone could edit from their browser. The Wiki software we used has a table plugin we used to keep tabular data (like story status) straight[9].

In order to track testing, I set up test status grids with brief test descriptions, date last executed, drop down lists for configuration (local v. staging), and test result (pass, fail, blocked). The result looked like this:

Test #	Category	Test Name	Last Executed	Result	Configuration
01	Home Page	All sections present	8/30/04	Pass	Staging
02	Search Results	Pagination Tests	8/30/04	Blocked	Staging
03	Search Results	Input Tests - special characters, etc.	8/30/04	Fail	Staging
04	Search Results	Ordering/Sort Tests	8/31/04	Pass	Staging
05	Search Results	No results returned	8/31/04	Pass	Staging
06	Contact Us	HTML Email	8/31/04	Pass	Local
07	Contact Us	Text-only Email	8/31/04	Pass	Local

The table control let me march down the table of test cases and select the values from the drop down. Tracking status became almost zero overhead, and everyone knew what I was testing and where.

Automated tests also become a source of documentation. They may even become the most up to date and accurate specification available. I recently needed to remember how some software I had tested worked. I was able to get the answer from the automated tests more quickly than from the official specification.

Sometimes the automated tests are not sufficient by themselves, but they can be made to generate the necessary documentation. In one case, we added logging that turned the automated test steps into human-readable documentation as the tests ran.

If you need additional documentation beyond what the automated tests provide, consider using a Wiki. It's public, visible, open, and archivable.

Don't Duplicate Automation

Although the team produced phenomenally good code, I was still finding an average of 5 issues a day. Many of these were small issues: typos or formatting discrepancies between our implementation and the customer's mockups. Some were unanticipated requirements such as the need for exception handling when the user manually edited the URL and made it invalid in the process. A few were serious bugs resulting from integrating our software with an existing framework we didn't control.

[9] We were using Twiki (http://www.twiki.org) with the Table Plugin (http://www.multieditsoftware.com/twiki/bin/view/TWiki/TablePlugin)

Copyright © 2004 Quality Tree Software, Inc.

Originally we thought it would make sense for me to write automated tests in jUnit with jWebUnit to demonstrate the bugs. Ultimately, we gave up on that idea: it made me too inefficient. It took me an average of about 10 minutes to log an issue as a bug or show it to a programmer. It took me half an hour to write the test. Since I was a part-time tester supporting a team of agile programmers, I couldn't afford the extra time.

I also tried to automate some of my end-to-end tests. We found that although my automated tests were a little different from what the programmers were already writing, they weren't different enough to justify the time I was spending on them.

We also discovered that we had some stumbling blocks attempting to automate tests to run in the fully deployed system (as opposed to the mocked out system). As an example, one of the bugs I found involved an email feature. Reproducing it involved setting a preference in another part of the system we didn't control, then forcing an email to be sent, then checking whether or not the email arrived. While it would have been theoretically possible to automate that, it was cost prohibitive: it took a few moments to verify manually as opposed to the several days it would have taken to get automation working with all the parts of the system I had to touch.

So we finally decided it made sense to let the programmers do what they do best (write code) while I did what I do best (test). The programmers continued to create unit tests test first. And in some cases, I paired with programmers to write acceptance-level automated tests. But for the most part, I focused on testing the software in ways they had not already.

Although I didn't automate as many end-to-end tests as I originally assumed I would, I did use scripts extensively to support my manual testing, particularly for data setup. When the application was in the early stages, there was no way to push data into the system without writing code to do so.

One alternative that we did not have the opportunity to try on these projects but that other Agile teams find works well for them: the open source test frameworks Fit and FitNesse[10] provide enable testers and programmers to collaborate effectively on test automation. With these frameworks, programmers can write support code to connect the tests to the system while testers can contribute test cases very quickly by adding data to a table.

Use Exploratory Testing

Exploratory testing is a highly disciplined form of testing in which the tester is simultaneously learning about the program, designing tests, and executing them. This kind of testing involves using rigorous testing techniques just as pre-planned testing does, but applying those techniques directly on the software being tested instead of writing about it first.

On the Agile-Testing list, Ward Cunningham said "…agile programs are more subject to unintended consequences of choices simply because choices happen so much faster. This is where exploratory testing saves the day. Because the program always runs, it is always ready to be explored."(Cunningham, 2004)

---

[10] See http://fit.c2.com and http://www.fitnesse.org

Integrate Testing into the Development Processes

In the last few months I discovered first hand what several Agile practitioners have been telling me for much longer: having everyone in one room speeds up communication by an astonishing amount. Questions that might otherwise take a couple days to be answered over email are answered within a couple of minutes. There were numerous occasions when I was able to save hours of my time by overhearing a conversation in the team room.

But I also learned that collocation is not sufficient to ensure that the testing is fully integrated with the efforts of the rest of the development team. Even though I was sitting right next to the programmers, there were occasions when I felt as though I was handed software over a wall.

In order to integrate my test efforts more completely with the programmers work, we learned to:

- Include test activities such as test data creation in each iteration's plan
- Share our test data—though not necessarily use the same test data
- Pair on creating test infrastructure code

The Evolving Role of Independent Testing

On traditional teams, testers often view themselves as a last line of defense. They may feel that they're protecting the unsuspecting users from the sub-standard software the developers would otherwise inflict on them. The result of this stance is usually some degree of tension between the testers and developers, where the degree of tension can range from mild to vitriolic.

Agile testing requires a new take on the role of the independent tester.

Testing cannot exist in a vacuum. Testing yields information about the system under test. But for whom? Who needs that information and what are they going to do with it?

In an Agile context there are two key sets of stakeholders who need the feedback that testing provides:

- The developers need information they can use to improve the code, including the unit tests.
- The business decision makers[11] need information they can use to guide the project.

Instead of taking the role of last line of defense between the users and the project team, testers on an Agile project do better to take a supporting role to these two sets of stakeholders.

[11] In XP, this is the Customer role.

Copyright © 2004 Quality Tree Software, Inc.

A traditional view of testing suggests that independent testers exist to assess the quality of the system. The argument for independent assessors is that they won't have the blind spots that the developers have. They can objectively reconcile what the users or business stake holders said they wanted (requirements) with what the developers delivered (implementation).

This independent assessment has less value for an Agile team working closely with a customer. The customer can assess for themselves whether or not the implementation meets his or her needs. Testers should focus on providing feedback and information rather than assessing quality, exposing bugs, or evaluating compliance. They may help facilitate discussions between programmers and business stakeholders.

Taking a Supporting Role

In addition to supplementing the team's test effort, Agile testers can support the team by:

- Asking "what if" questions of both the programmers and business stakeholders in the planning game. For example, "What if the migrated data has null values?" or "What if a user decides to…?"
- Analyzing risks and providing information early. For example, "A similar Web application that used a similar mechanism was hijacked by spammers. What safeguards do we have in place to ensure this won't become a spam machine?"
- Offering information about external dependencies or requirements that the team might not otherwise know about. For example, "I would expect this application to look and feel like this other related application that was just released. Will it?"

Copyright © 2004 Quality Tree Software, Inc.

It's important to remember that testing is as much about perspective as skill. Where the programmers are thinking about things like whether or not to use the Singleton Pattern[12], testers are thinking about the likelihood of a NULL sneaking into the database. Programmers have their heads wrapped around the emerging design. Testers have their heads wrapped around the emerging risks.

Supporting, not Servile

Taking a supporting role doesn't mean serving as a personal assistant to individuals on the team, however.

In traditional contexts, programmers have sometimes asked me to test their code before they check in. I almost always refused their requests because it would double my effort. If I tested the code on their machine before it was checked in, I would still have to test it in the build.

In light of my recent experiences with XP, I've been wondering if I was doing the right thing. I realized that in some cases, I could have saved everyone a lot of time if I'd helped the programmer earlier, especially since the time delay between code and test can be days or weeks on a traditional project. But in other cases, I would have been supporting an individual programmer at the expense of the team.

Consider the difference between two requests by two programmers, both when I was an independent tester in a non-Agile context.

One programmer asked me to review his code and watch it execute at his desk, with him present, before he checked it in. This was several years ago, before I'd heard the term Pair Programming, but that's what he was looking for: a pair, another set of eyes. Back then, I hesitated before agreeing. Today I wouldn't hesitate at all. This is the kind of team support we all benefit from providing and requesting.

Another programmer tried to hand me a floppy disk, saying, "Hey, can you test this?" He wanted to hand off responsibility for testing to the nearest available tester so he could move on to other tasks. He had no idea whether the code he'd written was any good and he thought it was someone else's job to find out. I refused his request and learned later that he had to rewrite almost the whole thing because he hadn't understood the interfaces. Doing that programmer's testing for him would have been more like enabling than supporting—enabling him to continue to do a sloppy job.

Conclusion

If programmers change how they create the software, testing needs to change as well. Traditional test processes are not Agile: heavyweight documentation, fragile automation, and extensive test tracking slow down the test effort and make it more difficult for the test process to adapt to extensive changes in the software.

We can become more agile, and thus adapt better to Agile methods if we:

---

[12] Although not explicitly an Agile technique, Design Patterns are well known by the Agile community as a whole. In fact there's a great deal of overlap between the Agile community and the Patterns community. For more about patterns, see the now classic Gang of Four (GOF) book, (Gamma, Helm, Johnson, & Vlissides, 1995)

- Streamline our test processes
- Focus on providing feedback not assessments
- Shift our role from last line of defense to team supporter

Without these changes, testers find themselves at odds with Agile programmers. But by becoming more agile in our approach, testers can have a huge positive impact, helping Agile projects be even more agile.

The Agile Manifesto

From http://www.agilemanifesto.org/

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn,
Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith,
Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin,
Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas

© 2001, the above authors
this declaration may be freely copied in any form,
but only in its entirety through this notice.

End Notes and Resources

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*: Addison-Wesley.

Crispin, L., & House, T. (2002). *Testing Extreme Programming*: Addison-Wesley.

Cunningham, W. (2004). "Re: [agile-testing] Summary of Position". *Agile Testing list* Available online: http://groups.yahoo.com/group/agile-testing/message/3881.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns*: Addison-Wesley.

Kaner, C., Bach, J., & Pettichord, B. (2001). *Lessons Learned in Software Testing*: Wiley.

Pettichord, B. (2003). "Where Are the Testers in XP?" Available online: www.stickyminds.com/se/S6217.asp.

Poppendieck, M., & Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit*: Addison Wesley.

The Agile Alliance. (2001). "The Agile Manifesto". Available online: http://www.agilemanifesto.org/.

Copyright © 2004 Quality Tree Software, Inc.

Acknowledgements

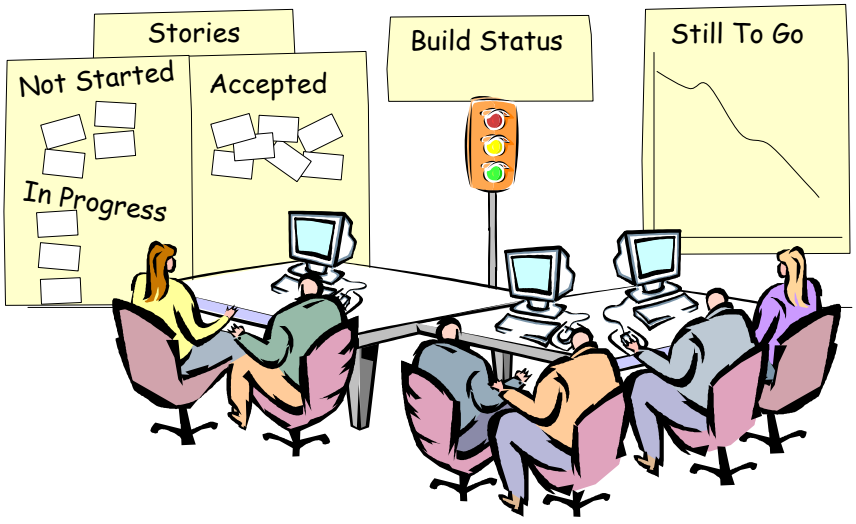
Many thanks to the following people for reviewing an early draft of this paper: Brian Marick, William Wake, Jonathan Kohl, Jeffrey Fredrick, Daniel Knierim, Marc Kellogg, Danny Faught, Ron Jeffries, Hubert Smits, Rob Mee, Sherry Erskine, Amy Jo Esser, Gunjan Doshi, Dave Liebreich, Janet Gregory, Chris McMahon

Agility for Testers

Presented at PNSQC 2004
Elisabeth Hendrickson
Quality Tree Software, Inc.
www.qualitytree.com

Copyright (c) 2004, Quality Tree Software, Inc.

What's So Different About Agile Anyway?



Copyright (c) 2004, Quality Tree Software, Inc.

Conflicting Values

Traditional Wisdom	**Agile Perspectives**
Strict change management	Change is inevitable.
Comprehensive documentation	Working software is more important. And face-to-face communication is better anyway.
Up front planning	Plan to the next iteration.
Formal entrance and exit criteria with signoffs	Collaborate, don't hand off.
Comprehensive system-level regression tests	Detect defects earlier with automated unit tests and continuous integration.

Copyright (c) 2004, Quality Tree Software, Inc.

Why We Do What We Do

Given the problem of too many bugs and too little time, traditional answers for QA include an emphasis on:

- Preparation
- Documentation
- Adherence to plan
- Clean handoffs
- Entrance and exit criteria
- Signoffs
- Strict change management

Copyright (c) 2004, Quality Tree Software, Inc.

Agile Implications for Testing

- Constant Change
- Shorter Schedules
- Shifting Risks

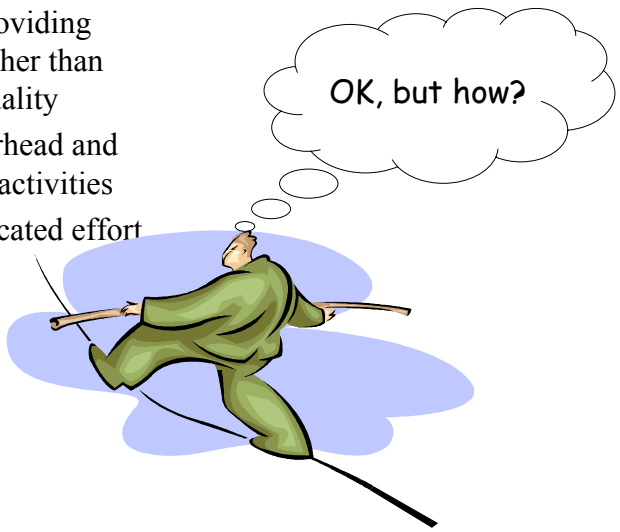
- Better Quality
- More Visibility
- More Testable Code

The solution is not to try to do the same thing only faster, but to re-evaluate our testing processes entirely.

Copyright (c) 2004, Quality Tree Software, Inc.

Toward More Agile Testing Practices

- Focus on providing feedback rather than assessing quality
- Reduce overhead and compliance activities
- Avoid duplicated effort



Copyright (c) 2004, Quality Tree Software, Inc.

Test Automation

- Look at what the programmers are testing. Avoid duplicating their unit tests at a GUI level—make sure system-level test automation adds unique value.
- Collaborate with the programmers on test infrastructure code.
- Use programming-assisted testing to enable exploration before the system is ready.
- Use Agile methods in developing test automation—it's programming too.

Copyright (c) 2004, Quality Tree Software, Inc.

Test Documentation and Status Reporting

- An informal poll of 135 testers across 57 organizations revealed that testers spend about one third of their time just documenting test cases.
- What might lighter weight test documentation look like?

Test #	Category	Test Name	Last Executed	Result	Configuration
01	Home Page	All sections present	8/30/04	Pass	Staging
02	Search Results	Pagination Tests	8/30/04	Blocked	Staging
03	Search Results	Input Tests - special characters, etc.	8/30/04	Fail	Staging
04	Search Results	Ordering/Sort Tests	8/31/04	Pass	Staging
05	Search Results	No results returned	8/31/04	Pass	Staging
06	Contact Us	HTML Email	8/31/04	Pass	Local
07	Contact Us	Text-only Email	8/31/04	Pass	Local

Copyright (c) 2004, Quality Tree Software, Inc.

Use Exploratory Testing

"...agile programs are more subject to unintended consequences of choices simply because choices happen so much faster. This is where exploratory testing saves the day. Because the program always runs, it is always ready to be explored."

Ward Cunningham on the Agile-Testing mail list

Copyright (c) 2004, Quality Tree Software, Inc.

Integrate the Test and Development Processes

- Co-locate testers and programmers. (But don't think that sitting side by side ensures communication.)
- Include testing tasks such as test data creation in the planning game.
- Have programmers and testers pair on test-related tasks.
- Track testing status and programming status together.

Copyright (c) 2004, Quality Tree Software, Inc.

The Evolving Role of the Tester

from last line of defense…

…to team support

Copyright (c) 2004, Quality Tree Software, Inc.

Skills for Agile Testers

Concentrate on building skills that will help you support your primary stakeholders.

To support business stakeholders

Domain skills: general business knowledge, understanding of specific business rules and relevant regulations

To support either

Testing skills: testing techniques, critical thinking, formal logic, and analysis

To support programmers

Technical skills: databases, networks, operating systems, programming, scripting, and/or testing tools

Copyright (c) 2004, Quality Tree Software, Inc.

Conclusion

We can become more agile, and thus adapt better to Agile methods if we:

- Streamline our test processes
- Focus on providing feedback not assessments
- Shift our role from last line of defense to team supporter

Copyright (c) 2004, Quality Tree Software, Inc.

Acknowledgements

Many thanks to those who reviewed the paper accompanying these slides: Brian Marick, William Wake, Jonathan Kohl, Jeffrey Fredrick, Daniel Knierim, Marc Kellogg, Danny Faught, Ron Jeffries, Hubert Smits, Rob Mee, Sherry Erskine, Amy Jo Esser, Gunjan Doshi, Dave Liebreich, Janet Gregory, Chris McMahon

Please see the paper for references and additional acknowledgements.

Copyright (c) 2004, Quality Tree Software, Inc.

What Does a Tester Need to Know in 2005 and Beyond?

Sam Guckenheimer

Group Product Planner

Microsoft Visual Studio Team System

Since the dot-boom went dot-bust, software testers have felt the impact of changing business practices. In a "do more with less" world, the skills and knowledge that software testers need to be successful and employable have been shifting. We'll identify the changes that testers are likely to encounter in the next five years.

Sam Guckenheimer is the Group Product Planner for Microsoft Visual Studio Enterprise. In this role, he is responsible for the end-to-end design of the product suite in its next release ("Whidbey") and the software development practices in the accompanying methodology. Prior to joining Microsoft in 2003, Sam was Director of Product Line Strategy at Rational Software Corporation, now the Rational Division of IBM. Sam has been developing of Automated Software Quality and Enterprise tools for eight years and has been a practitioner for 20, in capacities spanning tester, developer, product manager, and business manager. Sam is a frequent speaker at conferences such as Software Testing, Analysis and Review (STAR), Quality Week, and the International Conference on Software Quality.

Software Metrics: Don't just track – know when to act!

Diane Manlove and Stephen Kan
Software Quality Management, IBM®
dmanlove@us.ibm.com, skan@us.ibm.com

Diane Manlove is a Software Quality Engineer for IBM® in Rochester, MN. Her responsibilities include management of release quality during product development, system quality improvement, and product quality trend analysis and projections. Ms. Manlove is certified by the American Society for Quality as a Software Quality Engineer, a Quality Manager, and as a Reliability Engineer and by the Project Management Institute as a Project Management Professional. She holds a Master's degree in Reliability and an undergraduate degree in Engineering.

Dr. Kan is a Senior Technical Staff Member and a technical manager in programming at IBM in Rochester, Minnesota. He is responsible for the Quality Management Process in software development for IBM's eServer iSeries. His responsibility covers all aspects of quality ranging from quality goal setting, quality plans, in-process metrics and quality assessments, to reliability projections, field quality tracking, and customer satisfaction. Dr. Kan has been the software quality focal point for the software system of the iSeries since its initial release in 1988. He is the author of the book Metrics and Models in Software Quality Engineering, numerous technical reports, and articles and chapters in the IBM Systems Journal, Encyclopedia of Computer Science and Technology, Encyclopedia of Microcomputers, and other professional journals. Dr. Kan is also a faculty member of the University of Minnesota Master of Science in Software Engineering (MSSE) program.*

Abstract

Software metrics are an essential tool for project and quality management, but do you know when your metrics are telling you to take action? Can you identify significant trends and process changes or deviations? Do you know if process shifts are statistically significant? Tracking process and product metrics gives you insight into your development process. Comparisons to historical or industry data or well-defined targets bring you closer to understanding if current data is good news or cause for alarm. Even better are statistical limits which help to take the guess-work out of metrics analysis and empower software developers, project managers, and quality engineers to knowledgeably control development processes and achieve product quality objectives.

Statistical Process Control (SPC) techniques, such as process control charts, are common in manufacturing environments, but uncommon in the software industry due to many implementation problems inherent to the nature of software development. However, as in manufacturing, SPC techniques are extremely useful for understanding process capabilities, quickly identifying process deviations, and controlling product quality. In this paper, some of the challenges of implementing SPC for software processes are discussed, several methods for addressing the problems unique to SPC use within software development are described, and practical examples of SPC implementation across the software development lifecycle are explained. Other traditional quality tools, such as Pareto analysis, are used to augment metrics analysis.

Real-life examples from large and complex industry projects developed at IBM® Rochester are used to illustrate SPC use in development as well as the maintenance phase of the software lifecycle, while the caveats/limitations of applying control charts in software environments are addressed. Examples of the analysis of process outliers and actual implemented actions are also provided. The project examples given are based mainly on releases of the operating system of the IBM eServer iSeries.*

1. Introduction

While the use of statistical tools in software development has gained acceptance in recent years, practical examples of statistical techniques in general, and process control in particular, are still rare in the literature. In manufacturing production, the use of control charts is synonymous with statistical process control (SPC). For software, ingenuity in both methodology and application is required for tools such as control charts to be useful. In this paper, we discuss SPC in the context of software development, with real-life examples from large and complex industry projects developed at IBM Rochester. These projects are usually releases of the operating system of the IBM® eServer iSeries*. In the discussions, we attempt to provide examples covering the major phases of the software development life cycle: design, implementation, test, beta, and maintenance (field).

2. Challenges and Advantages to the Implementation of Control Charts within Software Development

Control charts have long been established as a tool for identifying potentially problematic normal manufacturing production. The implementation of similar statistical action limits for software production is logically sound, but is problematic in practice for several reasons.

- Software is produced by people and not machines.

  While machines don't object if we identify their output as 'abnormal', people often do. That makes the application of statistical control techniques open to suspicion and resistance from the beginning. Care must be used in the application of control charts within the software development community so that early efforts do not reinforce developer resistance. In some development organizations it has become a practice to control chart the amount of time spent reviewing designs or code. Action limits are set to identify when too little or too much time has been spent in a review. This is an example of control chart use (or misuse) which could have negative side effects within development organizations. Additionally, control charts, and metrics in general, should never be used to track individual performance.

- There are significant sources of variability within the software development environment that can be difficult to minimize or control.

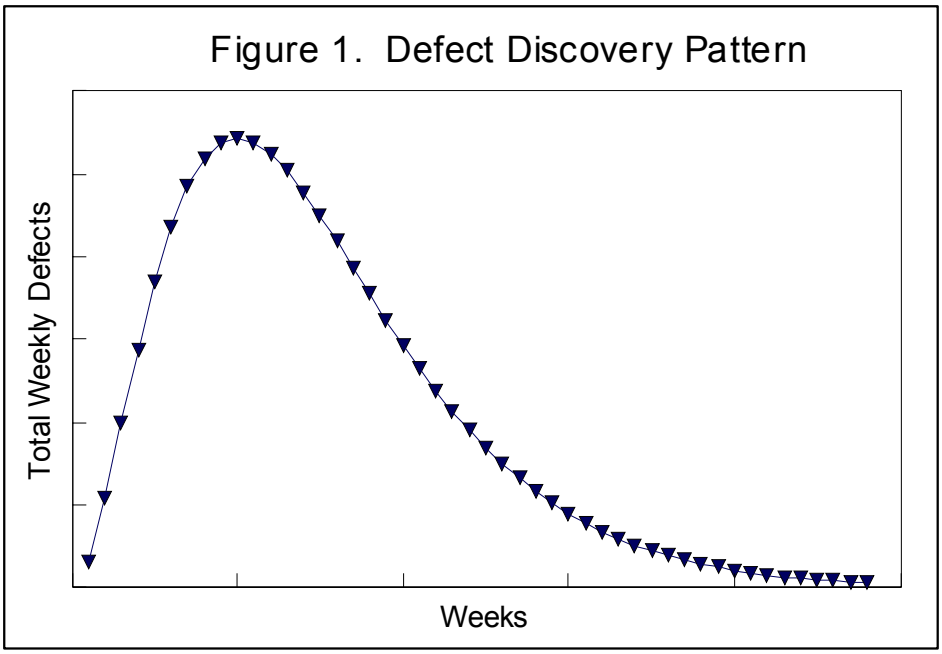
  A few examples of sources of process variability are code language differences, code size, design complexity, number of external interfaces, and developer skill and experience. Control limits are derived from the process average and standard deviation. Those limits get wider, and less useful, with every additional source of variability. Individual control charts can be created for each special variation cause within a process. However, at the system level these are of limited use for understanding the overall process or product under study.

- Process indicator behavior varies with the development cycle.

  Software measurement indicators tend to follow specific patterns in accordance with the “cycles” or “phases” of the development process. As an example, the Rayleigh model, as shown in Figure 1, has long been accepted as a typical pattern for defect discovery in software development

environments. Clearly, traditional control limits based on the overall process average would not be useful for this metric.

In addition to the above problems, software development indicators do not show a direct correlation to the final product quality. In manufacturing, production control limits, process capability, and the level of end-product quality are directly interrelated. This difference stems from the basic fact that software is development and manufacturing is production. Therefore, for the SPC experts who are purists, one might call the control charts within the software lifecycle "pseudo control charts".



Despite the many problems associated with the use of traditional control charts, SPC is useful in software development for several reasons. Control charts help us to identify process shifts and abnormal variations. With investigation and experimentation we can determine the sources behind these process variations and take the appropriate action to address them. Through early action we can prevent outcomes which do not meet our product objectives. We can also capitalize on process changes that create improvement trends. This is the most rudimentary use of control charts.

Once SPC is implemented, the process' capability can be understood. This facilitates planning and target setting. Data on past process performance allows us to implement process improvements knowledgeably. The results of those process changes can be monitored and evaluated.

Due to the previously stated challenges of implementing standard control chart methodologies, innovative approaches to implementation must sometimes be applied. What follows are examples of our implementations of SPC for a variety of software metrics along the software development cycle.

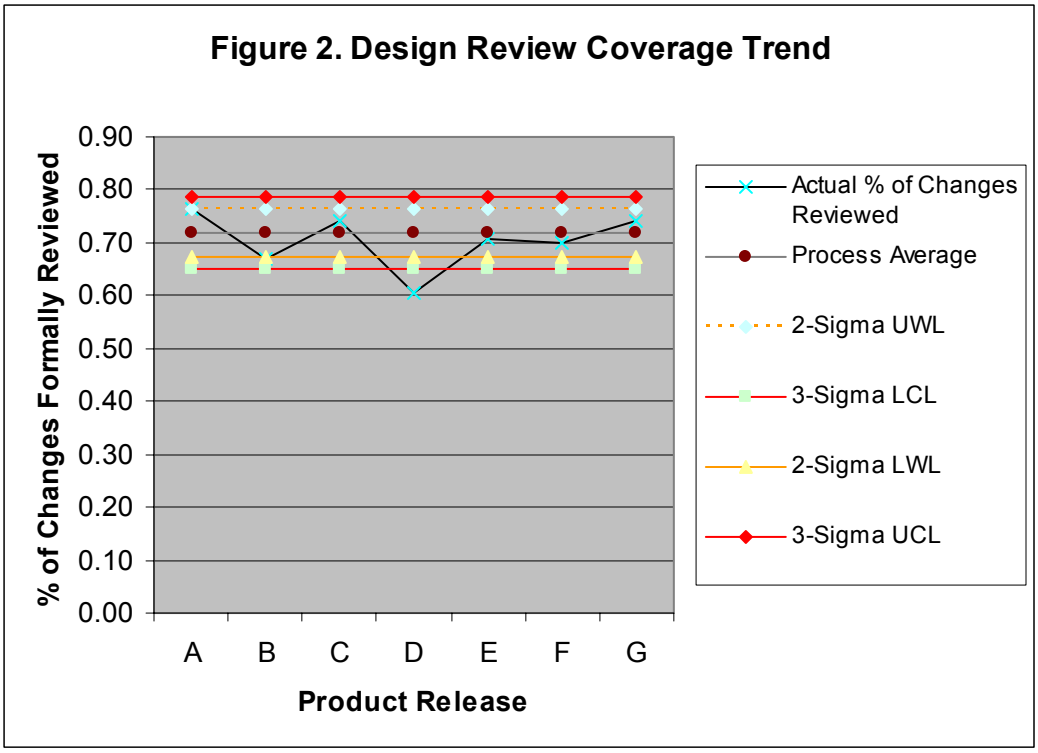
3. SPC in the Design Phase

It is well known that early development phases contribute significantly to final product quality. Preventative and corrective measures implemented in early development can significantly reduce overall development cost. Fortunately, some software metrics during the design phase lend themselves well to standard control chart methodology. One example is the number of design reviews held, expressed as the percentage of line items (requirements) for which formal design reviews are conducted.

In any software project or release of a software product, there are usually many requirements, or line items, or units of function. Formal design review coverage is not usually 100%. Within our design process, formal design reviews may not be held for several reasons. These reasons include minor code changes, removal of obsolete code, or ports of existing software. However, it is well known that formal design reviews contribute significantly to early defect removal. This is, therefore, an important metric to monitor and control. At the system level we want to understand if the percentage of planned design reviews is consistent with process history and will support our product quality objectives. To determine this we plot the percentages from past releases using a p-chart, as shown in Figure 2. The Upper Control Limit (UCL), Lower Control Limit (LCL), Upper Warning Limit (UWL), and Lower Warning Limit (LWL) are calculated as follows:

UCL = p+3*Sqrt [p*(1-p)/n]
LCL = p-3*Sqrt [p*(1-p)/n]
UWL = p+2*Sqrt [p*(1-p)/n]
LWL = p-2*Sqrt [p*(1-p)/n]

Where: p=process average, n=average sample size



From Figure 2 we can see that the planned design review coverage for the latest release (right most release on the graph) compares well to our historical data. Had the point fallen outside of a control limit, further investigation into the cause would have been initiated. For example, we could break the design review coverage out by system area, product, or component to identify areas of low coverage. We could also examine the list of line items where design reviews are not being conducted. If investigation uncovered areas where additional design reviews would be beneficial, they could then be scheduled during the planning phase. This is much better than reacting after the reviews are under way or when design defects are discovered.

Real-time SPC is also employed within our design phase. One example is the SPC used by our design control group (DCG) to identify outliers within the review process. Action limits are set for the number of

issues found during a review and the number of reviews held for a single design change. These action limits are based on a standard c-chart. The SPC limits for a c-chart are calculated as follows:

UCL= c+3*Sqrt(c)
LCL= c-3*Sqrt(c)
UWL = c+2*Sqrt(c)
LWL = c-2*Sqrt(c)

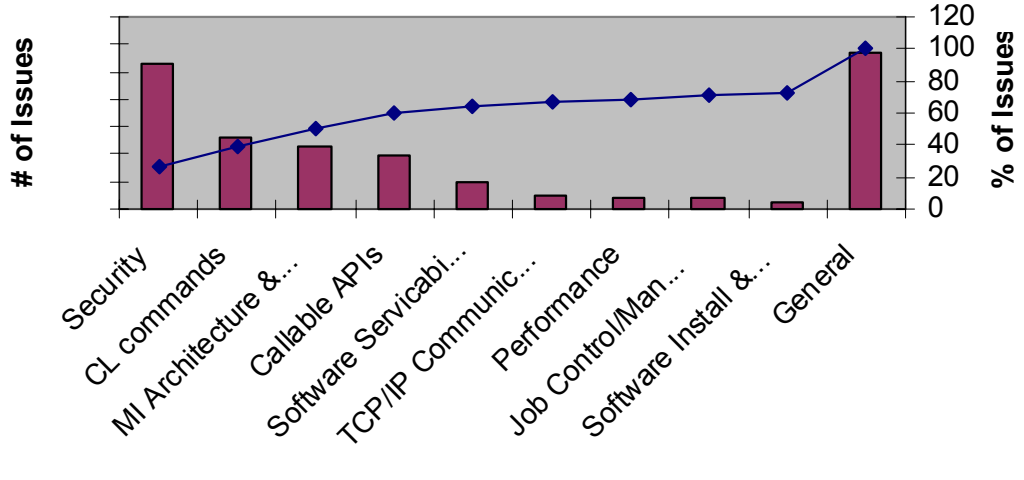
Where: c = process average number of non-conformities, Sqrt=Square Root

One of the basic assumptions for this chart is that the amount of material reviewed be held constant. Our DCG controls the content and length of reviews to the extent possible in order to maintain review consistency. If 8 or more issues are recorded for a single review, additional action, such as requiring a re-review of the change, is initiated. For this metric the warning limit of 6 or more issues is also closely monitored. If a review has 6 issues which are all non-trivial, action is initiated.

The DCG routinely reviews changes multiple times. Examples are when the change is funded (technical feasibility has been established) and when it is approved (code can be integrated). There are many reasons why a particular change may go through additional reviews. If the change reaches the review UCL, the DCG will investigate the source behind the outlier. Action will then be initiated as appropriate. These real-time control charts help our DCG to identify and act on potential quality problems early in our development process

Another quality tool employed within our DCG process is the Pareto chart. Issues are categorized for each release as shown in Figure 3. Once the top issue areas are identified, actions are brainstormed and initiated. For example, when command line (CL) issues were found to be a major contributor to the total DCG review issues, education seminars were put in place to better educate line item owners on the use of CL commands. Subsequently, more line item owners implement CL commands correctly in their designs. This resulted in fewer CL issues at DCG reviews. Documentation improvements for CL commands and security, another high issue area, have also been implemented with similar results.

Figure 3. DCG Issues



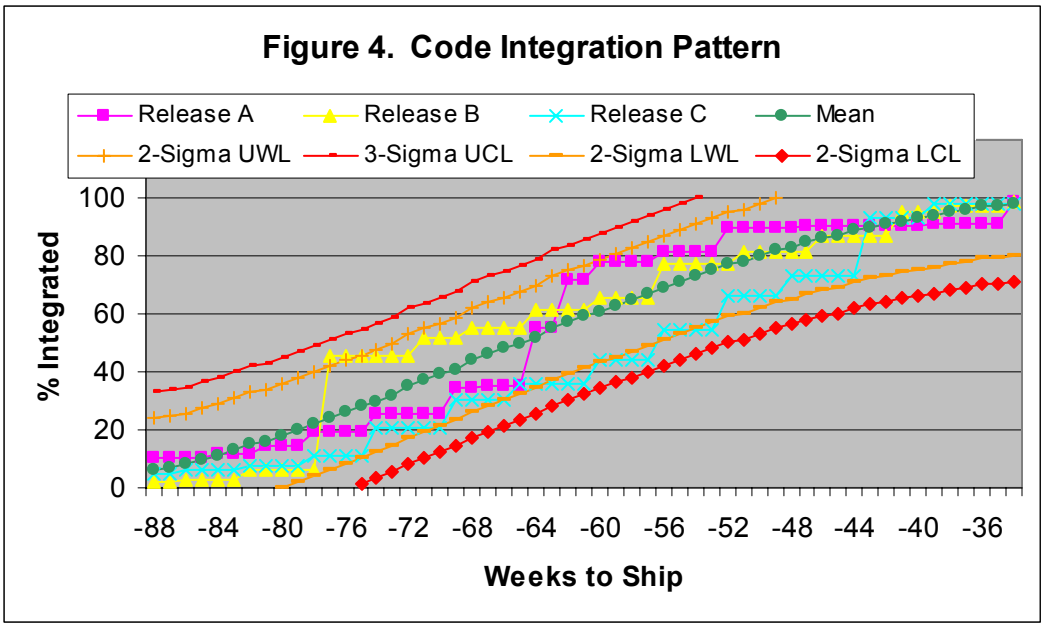
4. SPC in the Code Phase

Many metrics within software development follow an expected pattern, but do not remain within constant limits. As stated earlier, defect discovery has been found to follow a Rayleigh pattern (see to Figure 1). Another example would be the defect backlog, which typically grows when defect discovery increases and then decreases as product development approaches conclusion. We have developed several methods for creating actionable, statistical limits that address the unique problems of this type of data. Our example for the code phase is the cumulative code integration pattern expressed as a percentage of the total lines of code (LOC). This metric grows from 0% to 100% with an increasing pattern over time. Integration is the process of integrating code from a developer's desk top to the system library to build a temporary release of the product for testing.

The planned and actual integration patterns provide useful insight into the coding phase of development. The planned integration pattern can be examined to identify times where a large amount of new code is expected to integrate. With more churn due to the significant amount of new code, the system may be less stable at these times. It may therefore be useful to look for opportunities to redistribute the integrations over several periods. Additionally, we may want to make sure that these integrations are conducted when the developers who own the code are available to assist with debugging. We could also schedule additional testing before or after the newly integrated code is released to developers.

We can compare the planned integration pattern to historical data using a control chart format. If the pattern differs significantly, we can once again pursue actions early. As the code is integrated, we can compare our actual integration data to our plan and to our historical data. This facilitates real-time corrective action.

To derive pseudo-control limits for the integration pattern, we first examine our historical data. Figure 4 shows the integration pattern for 3 of our releases. Standard control chart limits will not adequately address the complexity of this metric, which is increasing in a slightly 's-curve' pattern. In order to create meaningful control limits for this metric we first developed a non-linear (polynomial) model of the form $y=a+b*x+c*x^2+d*x^3+e*x^4$ based on our historical data. In this model y is the cumulative percent of code integrated for a given week and x, x^2, X^3, and X^4 are the polynomials of the time unit (week). This model is an average of historical patterns. It can be used to represent the process average, as shown in Figure 4. An estimate of the standard deviation is obtained from the analysis of variance of the model. The mean square residual is used to estimate the variance. Control limits can then be derived from the model curve and the variance. As part of the modeling process, we also examine modeling assumptions such as normality and homoskedasticity before a model is accepted.

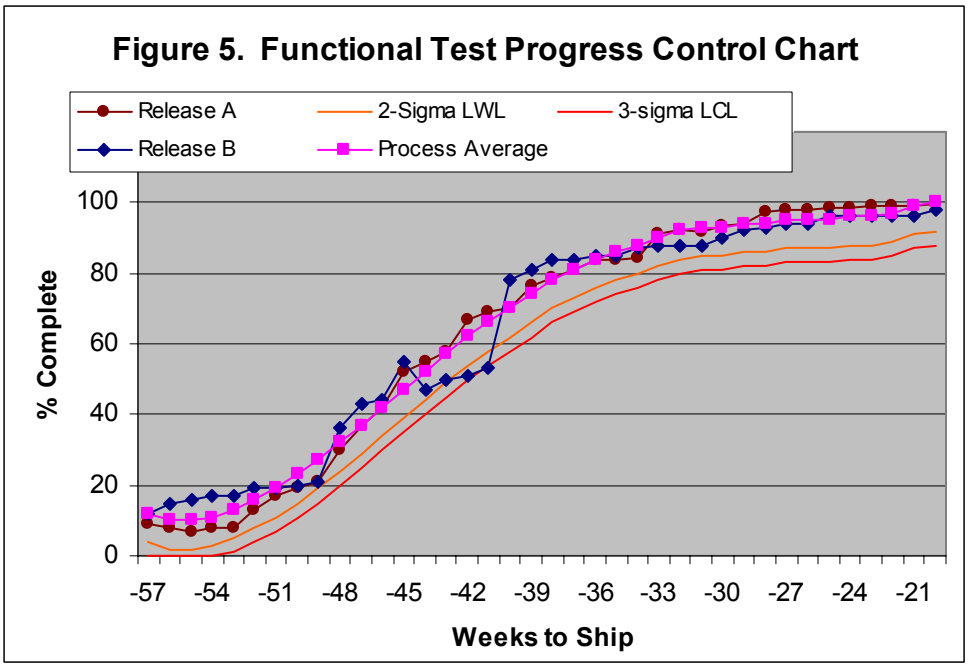


Weekly integration defect discovery is another code phase metric where we can employ this method of deriving control limits. The weekly integration defect metric provides insight into early code quality. It is also a predictor of code stability and readiness for functional testing. With the aid of a control chart format, we can identify newly integrated code which is a candidate for early quality improvement efforts. Problems with the integration process itself can also be found and corrected in a timely manner.

5. SPC in the Test Phase

Figure 5 is a graph of functional test progress (completion) by week. This is a critical test metric. The Release A and Release B lines shown in the figure represent actual past release historical data. Functional test progress normally follows an s-curve pattern. Control limits can be derived in the same manner as previously shown for the integration pattern. It can be seen in Figure 5 that the model conforms well to actual process performance. The resulting control limits are suitably sensitive for detecting process deviations, particularly during the ramp-up phase of testing.

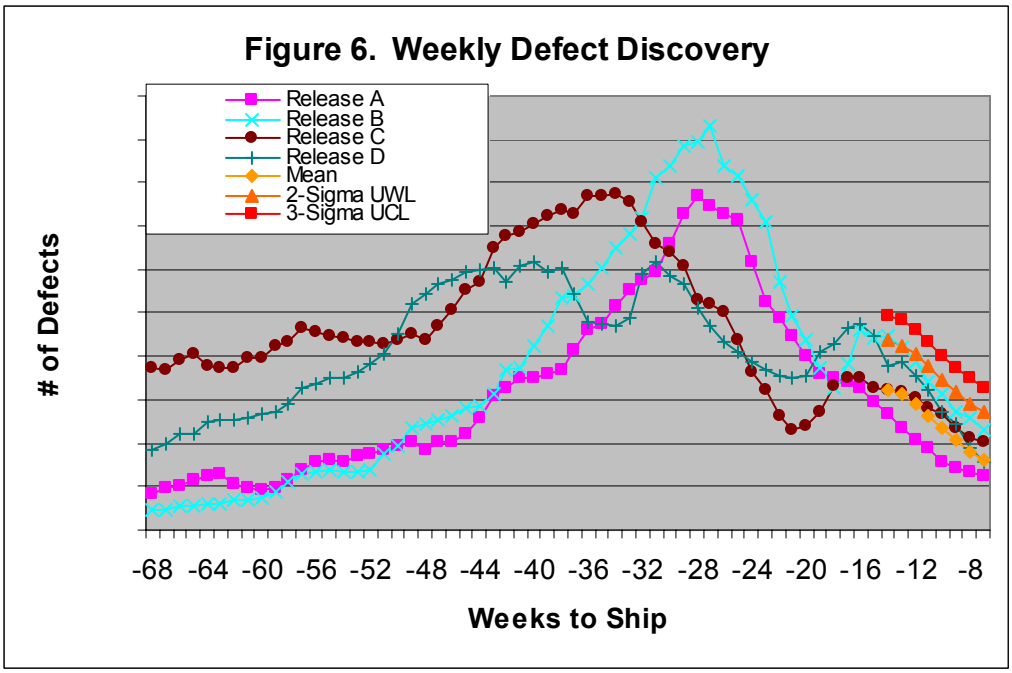
As with the integration curve, this functional test chart can be used to assess planned test curves as well as actual progress. The test plan curve can be plotted with the historical data and control limits. From this view we can determine if the planned test progress conforms well to historical data. Judgement or historical data can be used to factor in potential progress lag. Since potential progress problems can be identified beforehand, action can be initiated early. One example of early action would be to identify the critical path for testing. Actions to improve the critical path could include examining resource utilization, evaluating potential tools and automation, etc.



It is important to monitor the test progress curve along with the defect discovery curve during the test phase. Together, these curves can provide an insightful picture of actual test progress and effectiveness. This picture would be masked by examining only the test progress curve. However, release size and function, testing plans, the overall development schedule, and other factors create significant variability. This makes control limits for the overall defect arrival curve impractical.

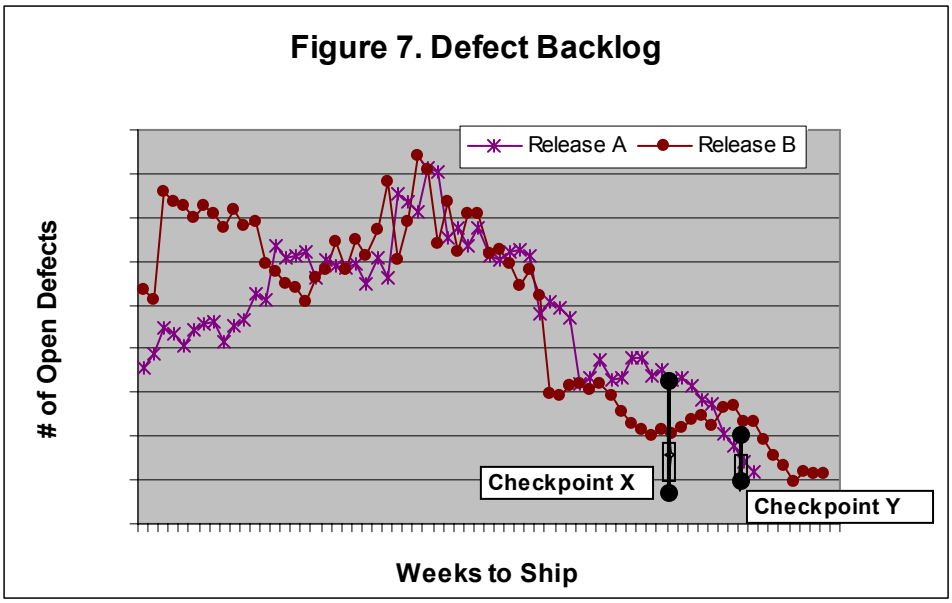
One approach to this problem is shown in Figure 6. In this example we have applied control limits only to the back end of the curve. Final process activities are more consistent, regardless of many release variables, during this time period. In particular, our Platform Evaluation Test (PET) is completed at this time. PET is a customer-like final test conducted by an independent test group.

It is logical to create a control chart just for this back portion of the defect discovery curve. We are judging the product's readiness to ship during this time frame. It is a critical measurement period for the defect discovery metric. Defect discovery historically tapers off as the development and test cycle nears completion, as shown by Releases A through D in the figure. Any different arrival pattern would give us cause for concern. Figure 6 shows the upper warning and upper control limits for the last part of the defect discovery curve. In this application of SPC, we have narrowed our scope of study to a period of reduced sources of variability.



Tracking the backlog of open defects logically follows the tracking of defect arrivals. This is particularly important in the test phase. However, once again variability due to schedule, test progress, resource constraints, etc. makes SPC for this parameter difficult at best. For cases such as the defect backlog, where the application of control chart techniques is not useful, other quality tools can fill the void. One good example is the simple box-and-whisker chart. Box & Whisker Charts graphically represent data so that the data distribution is easy to understand. These charts use a box to depict the lower and upper quartiles of the distribution. The median is a point within the box. Lines or 'whiskers' extend from the lower and upper quartiles to the distribution extremes.

Figure 7 shows the defect backlog curves for two releases. Box-and-whisker charts are used at 2 key checkpoints within the development process to provide a view of past process performance. These two checkpoints are associated with the dates for two critical milestones for the project. These mini-charts help us to set goals for future releases based on actual past performance and historical quality data. They are also used in a similar manner to assess current release metrics. It should be noted that there is a key difference between the defect backlog metric and the defect arrival metric. The defect backlog is a cross-sectional view of the defects that are open (to be fixed and closed). Therefore, setting control limits on specific dates or milestones is meaningful for this metric. For defect arrivals, it is the pattern (in addition to the level) that is meaningful. For that reason "point" controls may not be as meaningful.



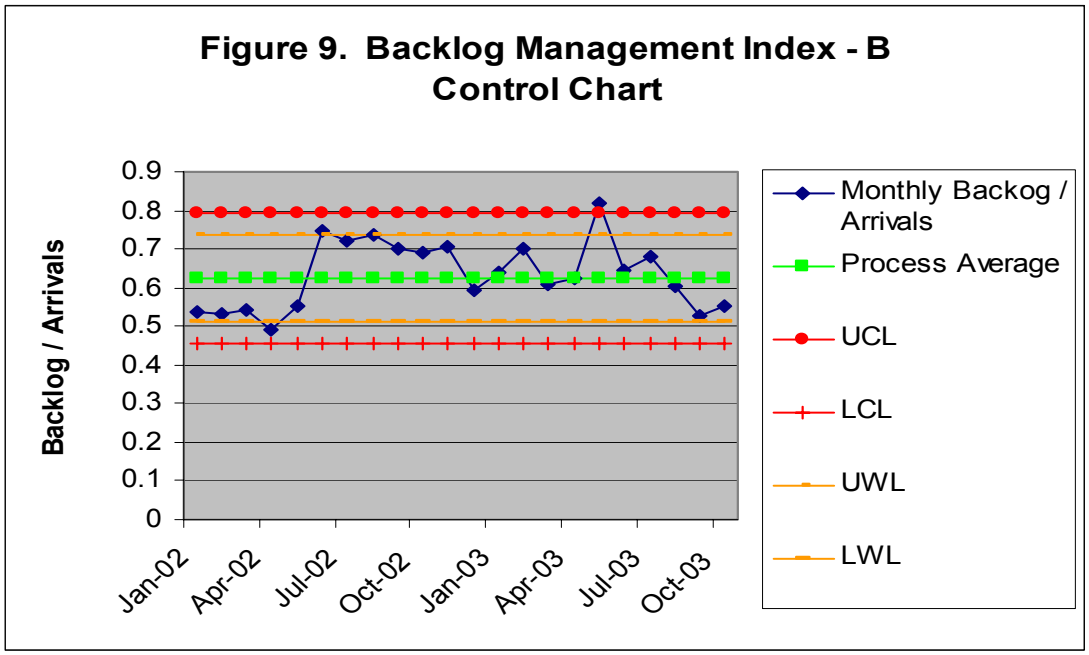
6. SPC in the Maintenance Phase

SPC is no less important during the maintenance phase of product life. At this time it can be used to identify unfavorable quality trends which could adversely affect customers. Our experience shows that measurement indicators during maintenance lend themselves more readily to the traditional concept of control charting, compared to measurement indicators during the development of the software project. One reason for the difference is that in-process metrics and indicators are subject to specific patterns and cycles associated with the development process. During the maintenance cycle, the operations of specific indicators are more on an ongoing basis. This does not mean that no innovation or data transformation is needed when applying control chart techniques.

A key metric for measuring the effectiveness of the maintenance process is how well the fix backlog is managed. The fix backlog is affected by the level of defect arrivals and the ability of the team to fix the defects (and therefore close out the defect reports). The level and pattern of field defect arrivals depend on how long the product has been in the field, its quality level, and how many products are shipped and when. In other words, the trend of the defect backlog measurement may not be linear, and may not even follow a specific pattern. Therefore applying SPC techniques to such data would be very difficult. In this case, our experience shows that both of the following transformations work:

- Backlog Management Index A (BMI-A) = (number of defects closed (fixed) during the month / number of defect arrivals during the month) x 100%
- Backlog Management Index B (BMI-B) = (number of defects in the backlog at the end of the month / number of defect arrivals during the month) x 100%

For BMI-A, the interpretation is that if the metric value is at 100%, the defect fix rate is the same as the defect arrival rate; if the metric value is more than 100%, then the fix team is ahead of the level of defect arrivals; if the metric value is lower than 100%, then the fix team is behind, or the backlog is growing relative to defect arrivals. For BMI-B, the metric value is the level of the backlog relative to the level of defect arrivals. Both metrics are valuable for backlog management, and both can be subjected to traditional control chart techniques. Figure 9 shows a control chart example of BMI-B. In this case, the process average is about 62%. This indicates that the process is stable and the team maintains a backlog at about 62% of the defect arrivals.



7. Summary

In this paper we discussed briefly the challenges of applying SPC techniques in the software development environment. We provided real-life examples covering the major phases of a typical software development process: design, implementation/code, test, beta, and maintenance. Through these examples we illustrated the approaches and techniques to make SPC work for software projects. Some of these techniques include:

- Applying traditional control charting techniques to parameters that fit
- Augmenting control charts with other quality analysis tools
- Using statistical modeling to establish control charts for parameters that usually follow the S curve, a pattern that is common in software development
- Using SPC on specific points that are related to milestones of the project, for relevant parameters
- Using a heuristic approach in setting control limits – tying experience with SPC and not being bound by traditional concepts of control limits
- Data transformation to make the metric of interest meaningful

In conclusion, to make SPC work in software development, ingenuity in both methodology and application is required; statistical techniques must be combined with common sense, experience in software development, and observations on how software parameters behave.

8. References

1. Curtis, Bill, Beth Layman, Joe Puffer, Charles Webber. "Solving the Challenge of Quantitative Management." Software Engineering Process Group 2002 Conference. Carnegie Mellon Software Engineering Institute. Phoenix, Arizona. 18 February 2002.
2. Ishikawa, Kaoru. Guide to Quality Control. Hong Kong: Asian Productivity Organization, 1982.
3. Kan, Stephen H. Metrics and Models in Software Quality Engineering. 2nd edition. Boston: Addison-Wesley, 2002.

**** IBM and iSeries are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.***

Proactive Data Validation in the Data Warehouse

by

Thomas O'Meara

Intel Corporation

Author Biography

Email Address: thomas.omeara@intel.com
Current Position: Database Architect
Technical Interests: Data Warehousing, Relational and Dimensional Modeling, MS SQL Server Database Administration

Nothing erodes data warehouse consumer confidence more quickly than bad data. Consumers will live with slow response times, complex querying, etc, but they will stop using the warehouse if they cannot rely on its quality. Therefore, ensuring data quality must be the primary mission of the warehouse. All warehouses receive bad data; that is data that violates a source system's own business rules, or which is inconsistent with data it will be combined with. It is inevitable. So, what is the best way to keep it out? This question eventually boils down into separate questions for the data warehouse designer; 'What is the best way to identify bad data?', 'What is the best way to deal with bad data once it is identified?', and 'What is the best method to use to keep it from infecting the good data already in the warehouse?'

I have been dealing with these questions for the five years I've been involved in data warehousing. I experimented with several of the traditional waterfall methodologies for creating data warehouses. Just last year, I took a position creating an Operational Data Store for a group of applications in my company. The development teams working on the applications that my ODS all employ XP software practices. They immediately began to try to convert me, and it took me a while to let my skepticism go and embraced the process. Later, while I was watching one of the developers refactor some code, I observed him employ 'Test First' practice, which involves writing test cases for an object before actually writing code that implements the object's behavior. Over the years, I'd coded numerous SQL statements to validated data as I staged it. At first, I'd thrown away these tests when I though I no longer needed them. Later, I learned to save them to help me track down bad data when production data issues surfaced. That day, it came to me in an epiphany that the tests I had, and the whole data validation process, could be used to during data processing to identify bad data, and provide an agile, testable, and decoupled solution to dealing with the bad data.

I continued to use the "Extraction, Transformation, and Loading", or 'ETL' paradigm to break the flow of data into the data warehouse into three discrete, sequentially coordinated processes. However, I stopped staging data using the methods I had used during Transformation processing. To show why coding data validation test into the staging process is better than the traditional methods we'll need to back up and discuss the choices available to you as a data warehouse designer.

Though it isn't obvious, the first choice facing the designer is whether to preserve bad data. Regardless of which option you choose, you begin at the same place, which is developing criteria to use to identify the bad data. Consider this example: "Your data warehouse receives data about your company's customers. Each Customer record has many attributes, including "CompanyName", "SalesAgent.", and "ContactPhoneNumber". Two criteria are defined that you must use to identify a bad customer record; the "SalesAgent" attribute must not be NULL and the "ContactPhoneNumber" must be a properly formatted US phone number that contains all numbers. Let's say that you coded your solution, in either case, to filter out Customer records that failed the first criterion. After your code goes into production, you happen to receive a record for Consolidated ACME, a new customer that placed its first order earlier this quarter, and the record happens to fail both criteria. If you chose to not to preserve the fact that the row failed, your code simply filtered out this record, excluding it from further consideration. The upside to making this choice is that it's very easy to code, effective, and cheap (without considering total cost of ownership). This code won't break. The downside comes when Bob Smith, from marketing, calls you up saying he can't find Consolidated ACME's sales figures, and he needs them for a really important report. The problem you've got is that you've never heard of Consolidated ACME, and you have no idea why their sales figures aren't in your warehouse. Now you're stuck. The only way you're going to track it down is by trying to recreate an issue that happened at run-time at some point during the past quarter. The situation is analogous to someone asking you to trace the exact path a bird flew in the air yesterday.

Your other option is to code your solution to also preserve the error and then filtered it out, preventing it from further consideration. The downside to this method is that it requires extra time to test and develop a system to capture the error row, and extra server resources to provide the preservation. But now you're in a much better situation when Bob calls. After some digging through a log, you can tell Bob that Consolidated ACME's sales figures can't be in your warehouse because the record was bad coming out of the Sales system. You can call up John Jones, the DBA for the Sales system, and tell him he has a problem. The eventual resolution of the problem may come too late for Bob, but at least it didn't take you three days to get to the root out the problem. I've lived with the consequences of making both choices, and, invariably, I spend more time (and therefore money) tracking issues down than I've spent testing and coding a logging system. Clearly, logging error rows is the superior choice.

Once you make the choice to preserve error information, you'll be faced with the choice of whether to write it to a log file or a database table. Writing to a log has some advantages. It is very easy, and it conservative with server resources. A bonus is that many ETL tools provide this capability out of the box. But suppose you're in that conversation with Bob, looking for the Consolidated ACME record. You are going to have to open up the log file, and search it. This is really not too bad, and doesn't take much time. You can find Bob's specific problem row very quickly. But, if you've chosen to write it to a database, you can also write a query that will pull up information about his issue. The great thing about the database comes when you bring the issue to John Jones. Once an issue is identified, it will like have occurred in the past. He will want to know about it, so he can have the Sales application support team address the issue. How are you going to identify all the times the same issue happened in the past by scanning a log file? It's easy in a database.

Another capability can be exploited with logging errors in a database. You can create an automated report that sends John a list of all of the Customer records that have failed for this reason, before Bob Smith ever becomes aware that an issue occurred. If Bob is the data owner for the Sales Department, he may want this information, too. No problem. Just add him to the report's distribution list. He'll be happy about you and confident in your warehouse. Using these techniques, I've taken my data validation practices, and leveraged them to *proactively* deal with issues. This is great, but can it get even better?

In the Consolidate ACME example above, we coded the solution to stop considering the Customer record when it failed the first criterion (it had a NULL value for "SalesAgent"). Let's jump into Bob's shoes, and consider his satisfaction with us when he receives a report that tells him his record failed, and why it failed. He brings the issue to the system administrator, who has his team fix it. Maybe they just fix the data, or maybe they need to fix some code first. They resend the record to you, and wham, it fails again, this time because, though it made it past the NULL value "SalesAgent" filter, it was caught by the "ContactPhoneNumber" filter. Whether it's in a log file, or in a database, John is in for more work, and both he and Bob are wondering why you didn't just tell them about both issues at the same time. You can't because you *didn't* know either. The better way is to apply as many tests as necessary to the data to determine all the known ways in which it is bad.

The method I have developed is to develop and apply tests to the data during the Transformation process, and immediately write the failures to a Transformation Error Log table. The table is supported by other tables that capture metadata about the ETL process, the staging table in which the error was identified, and a description of the error (In simple language, for communication). Since I logically divide my warehouse into 'sub-models', I include metadata about them also. Finally, I group the error metadata into categories. In our example, I look to see if I already have a Test description similar to the new test captured as metadata in my Transformation Error table. In this case, let's say I don't, so I create a new metadata row for it, and associate the new error with the "CustomerStaging" table, which belongs to the Customer ETL Stream, which supplies data to the Sales sub-model in my warehouse. Then I create some sample rows in a script file that I keep to determine if the test is working. Some of the samples I will expect to fail my test, while others I expect to pass my test. Next, I code my test, which asks each Customer row if the "SalesAgent" attribute value is NULL. If a row says, "Yes, my "SalesAgent" attribute value is NULL", it's staging table identifier, staging table row identifier, ETL Stream ID, Test identifier, and the data and time the error will be identified are inserted into the Transformation Error Log table. Then I repeat the process for the "ContactPhoneNumber" attribute, first creating a test for a NULL value.

Let's see what our example problem looks like if we implemented tests. Now I do know that the Consolidated ACME Customer row failed, and that it failed two tests. I also know exactly where in the Transformation process the failures occurred, when the failure occurred. I give this information to John and Bob in report form. I can also let them know that are 27 Customer records with the NULL value "SalesAgent" issue, 56 records with the "ContactPhoneNumber" issue, that the "ContactPhoneNumber" issue represents a 89% increase in that particular issue just in the past week. The additional information allows Bob and John to discover that Sue White entered the error records, and is new to the Sales Department. She hasn't been trained on the system, but has managed to uncover a bug in their software. Both these guys will love the level of customer service you provide. I can use this information, also. I can see from analysis

that we are beginning to have significant issues with the Sales application, and if trending continues, we are in for real problems ahead. I can take that information to my boss for action.

The Proactive data validation methodology also enables the staging process to be more agile. Our company wants to begin selling our products internationally. The Sales Department needs to respond quickly by reformatting the "ContactPhoneNumber" attribute to allow internationally formatted phone numbers. The warehouse will also need to be changed quickly to accommodate the change. Now I can simply change the existing tests that need to change, and add others to accommodate the new formats. Nothing else needs to change. Adapting to change becomes very simple.

So, great, you can identify all the ways that the Consolidated ACME Customer row can fail, you have logged all it's failures in the Transformation Error Log table, you responded to change with agility, and you proactively tell John and Bob about issues that come up. How do you keep the Consolidated ACME record out of the presentation database? This becomes very easy. You already have a record of the rows failures, and you have row in the "CustomerStaging" table. All you have to do is attempt to join each row in the staging table against the Transformation Error Log table using an existence check in your INSERT and UPDATE statements. When a match is found, it is filtered out of the Loading process.

Effective Metrics for Pragmatic Project Managers

Johanna Rothman

Rothman Consulting Group, Inc.

Puzzled about what's really happening with your project? Perplexed about whether the project will finish successfully? Perturbed about what and how to communicate what's happening to your team, your sponsors, and your management? Penchant for a crystal ball?

Unfortunately, not only do we not have silver bullets, we don't have crystal balls either. The good news is that a handful of key metrics can tell where your project really is. Once you know where your project is, you can choose how to steer the project to success.

In this presentation, we'll discuss what to measure and how those metrics can help you manage your project. Learn how to gather data in a way that will encourage your technical staff to help you measure and reduce the tendency to "fudge" the data. Finally, we'll discuss how to assess true project progress by measuring schedule, work completion, and how well the project staff is managing defects.
Learn how to use effective methods - based on measurements - to assess a project, communicate its status, and steer it to success.

Johanna Rothman consults, speaks, and writes on managing high-technology product development. She helps managers, teams, and organizations become more effective with her pragmatic approaches to the issues of project management, risk management, and people management. Johanna assists her clients in applying effective practices that create successful teams and projects.

A frequent speaker and author on managing high technology product development, Johanna has written over 100 articles and papers and is now a columnist for Software Development, Computerworld.com, and StickyMinds.com. Johanna is the author of Hiring Technical People and coauthor of Corrective Action for the Software Industry. Johanna is also a host and session leader at the Amplifying Your Effectiveness (AYE) conference.

Effective Metrics for Pragmatic Project Managers

Johanna Rothman

Rothman Consulting Group, Inc.

Author of *Hiring the Best Knowledge Workers, Techies, and Nerds: The Secrets and Science of Hiring Technical People* and coauthor of *Corrective Action for the Software Industry*

781-641-4046

jr@jrothman.com

www.jrothman.com

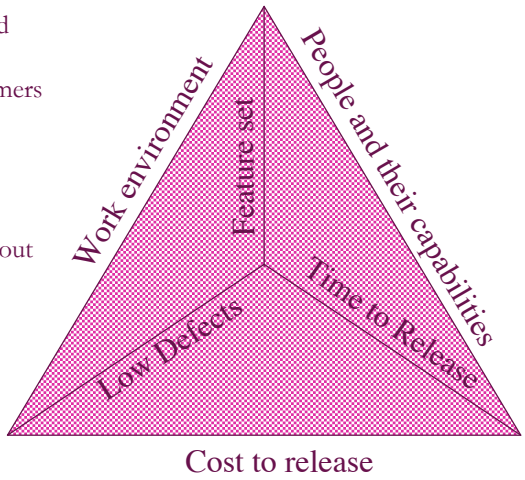
What Do You Want to Know About Your Projects?

- Where are we?
  - Is the project on time?
  - On budget?
  - Are we managing the critical path? (Do we know the critical path?)
  - Do we need more or fewer resources (people and other resources)?
  - Are we making progress? Are we tracking to the plan? (Do we have a plan?)
  - How good are our work products?
- How well are we working?
  - Do people know what they have to do?
  - How is morale?
  - Has the team reached the state of team flow?
- How effective are we?
  - Productivity measures (read Putnam and Myers' Five Core Metrics: The Intelligence Behind Successful Software Management)

© 2004 Johanna Rothman

Project Constraints and Requirements

- Every project has constraints and requirements
- Internal constraints: Your customers don't care about these. You do.
  - Cost to market
  - People and their capabilities
  - Work environment
- What do your customers care about the most?
  - Time to market
  - Feature set
  - Defect levels



© 2004 Johanna Rothman www.jrothman.com jr@jrothman.com 3

Pragmatic Measurements

- View all six sides of the pyramid
  - Time to release
  - Feature set
  - Defect levels
  - People and their contribution to the project
  - Work environment
  - Cost
- You (your management) care differently about each of these, depending on your context

© 2004 Johanna Rothman www.jrothman.com jr@jrothman.com 4

To Measure Project Completion …

- Project completion is a function of:
  - How accurate your initial estimation was
  - How much progress you've made (how many features completed and how good they are)
  - Schedule progress

© 2004 Johanna Rothman

Project Completion Measurements

- Earned value (Measuring what's been accomplished and comparing the accomplishments to the time used, rather than only measuring the time used for the project)
- Progress towards release criteria
- Estimates vs. actuals for major and appropriate minor milestones
  - The dates the project achieved the milestones vs. the planned dates
- Estimation Quality Factor
  - The histogram of the completion date over time: how good your estimation is

© 2004 Johanna Rothman

About Earned Value

- Earned value makes sense for products that can be created in self-contained pieces
  - You can create a piece, and measure how long and how much it cost you to create it
- Earned value makes no sense for products whose parts are interdependent
  - If each part is interdependent, how can you tell when a part is done?
  - If you can't tell when a part is complete, you can't measure earned value
- Software projects are composed of interdependent parts
  - Projects that have a software component are composed of interdependent parts
- Projects with tangible deliverables are great candidates for measuring earned value

© 2004 Johanna Rothman www.jrothman.com jr@jrothman.com 7

Schedule Metrics: Estimation

- How do you estimate now?
  - Range of dates (1/15-3/1)
  - "about" (About Mar 1)
  - Spiral in on a date (Q1, February, 2/14-2/21, 2/18)
- If you choose a date, you can use DeMarco's Estimation Quality Factor, a measurement of how good your estimation was during the project
- EQF is good for:
  - An early warning sign to see if events outside your project are consuming people when they should be on your project
  - A check against the initial estimations for your next project
  - If you have a chance of completing this project sometime in the next millennium

© 2004 Johanna Rothman www.jrothman.com jr@jrothman.com 8

Calculating Where You Are vs. Where the Schedule Says You Are

- Simple earned value discussion:
  - How much time have you spent?
  - How much money have you spent?
  - How much of the self-contained, tangible deliverables do you have?
  - If you've spent more time and money than you have deliverables to show than you will overrun the budget and/or schedule
- Inch-pebbles for software projects
  - Software projects suffer from the "90% complete" problem
    - It takes the first 90% of the time to complete 90% of the project. It takes the next 90% of the project to complete the other 90%.
  - If you can freeze or baseline requirements, you can avoid the 90% problem
  - Inch-pebbles help you create good estimates

© 2004 Johanna Rothman www.jrothman.com jr@jrothman.com 9

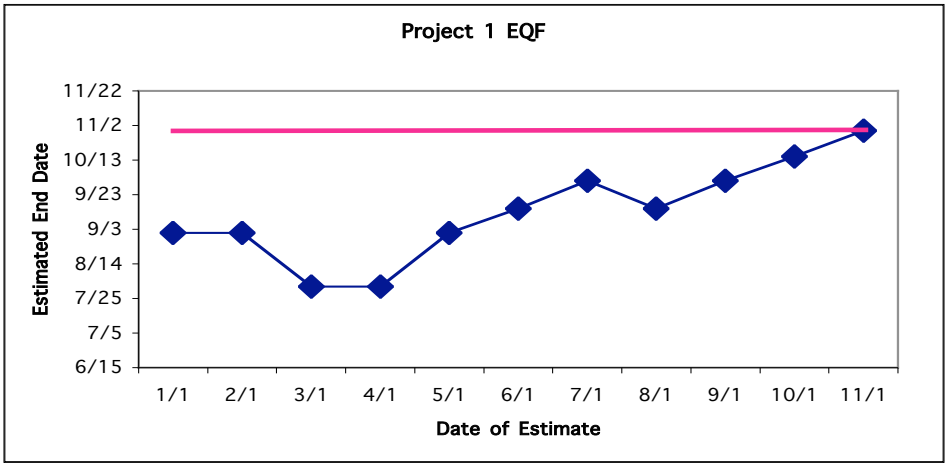
When You're Not on Track

- Ask your staff why
- Ask your staff to log their time for a week and let you know
- Are you holding things up?

- You have numerous possible actions when your project is not on track. However, you need to collect some data first, and then determine which actions to take.

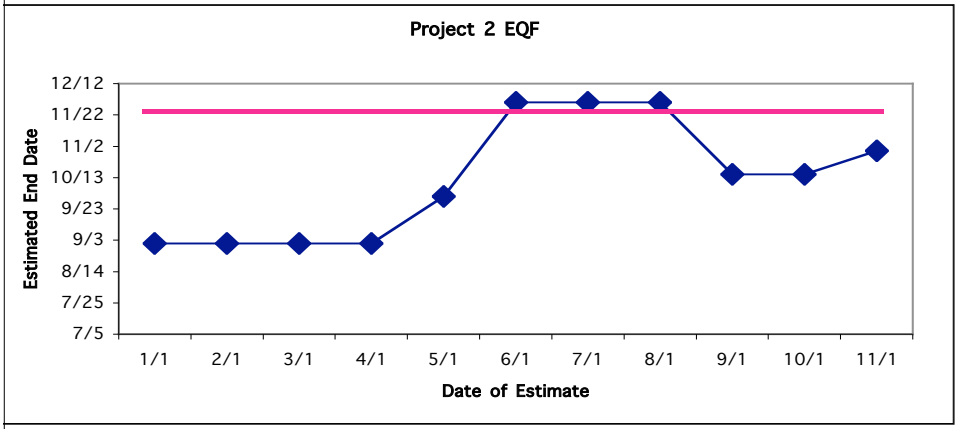
© 2004 Johanna Rothman www.jrothman.com jr@jrothman.com 10

Examples of EQF From 3 Projects



© 2004 Johanna Rothman www.jrothman.com jr@jrothman.com 11

Project 2 EQF



© 2004 Johanna Rothman www.jrothman.com jr@jrothman.com 12

Context Switching Is a Huge Potential Drain

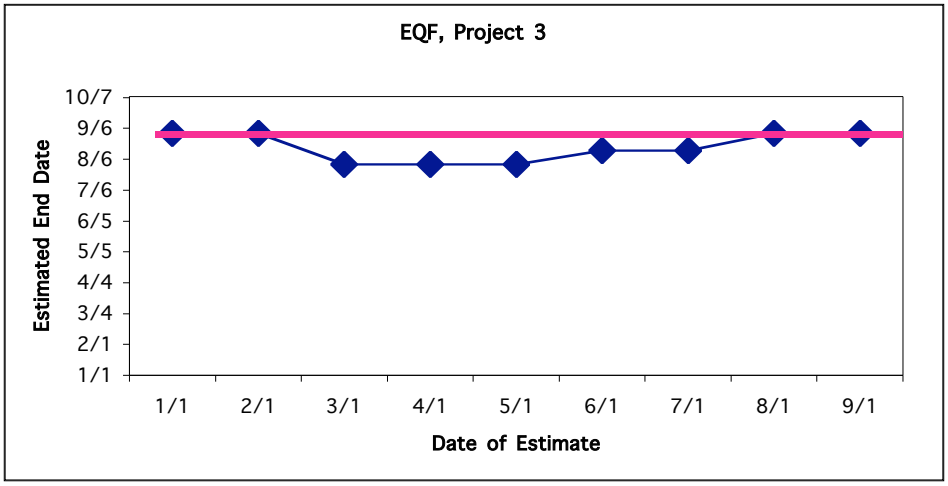
- Multiple parallel projects waste time
- If you want to get the work done faster, assign people full time without any other interruptions

Source: Quality Software Management, vol.1, Systems Thinking, Gerald M. Weinberg, Dorset House, New York, 1992.

Number of tasks	% Time on each task
1	100
2	40
3	20
4	10
5	5
More than 5	Random

© 2004 Johanna Rothman www.jrothman.com jr@jrothman.com 13

Project 3 EQF



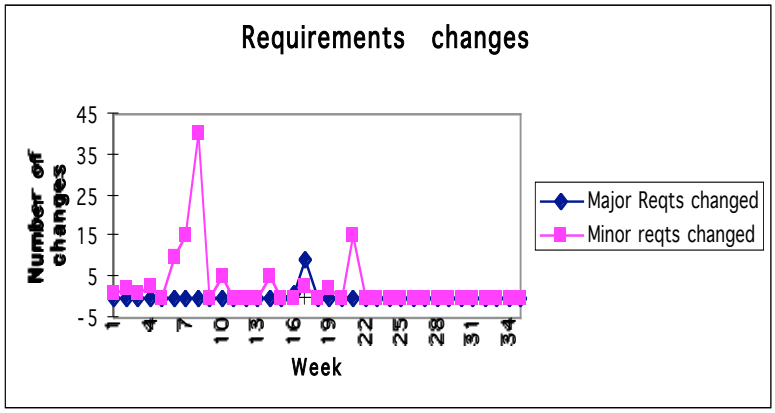
© 2004 Johanna Rothman www.jrothman.com jr@jrothman.com 14

Measurements You Can Start Early in the Project

- Do the requirements freeze? Ever?
  - How many requirements change over time?
  - Measure the number of requirements you have, the number of major/minor changes per week over the course of the project
  - Or, measure the number of requirements changes per iteration
- Do you have the people you need to complete the project?
  - Are they dedicated to the project, or are they working on other things, such as fixing problems from previous releases or other projects?
  - Measure the number of people assigned week by week. Show when you needed the people

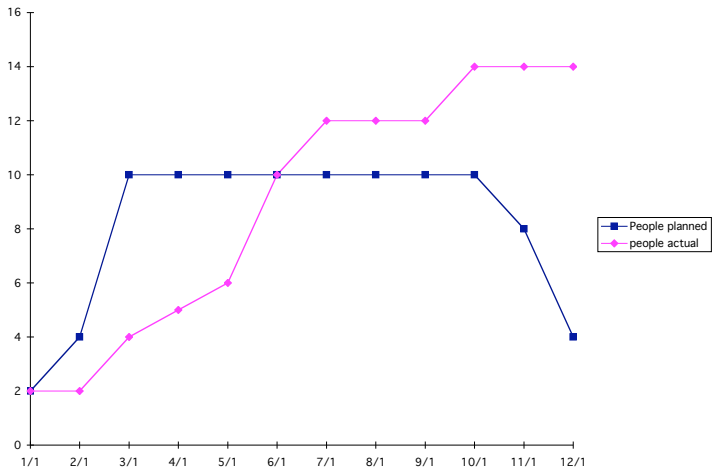
© 2004 Johanna Rothman

Example of Requirements Changes Chart



© 2004 Johanna Rothman

Example of Staff-Days Chart



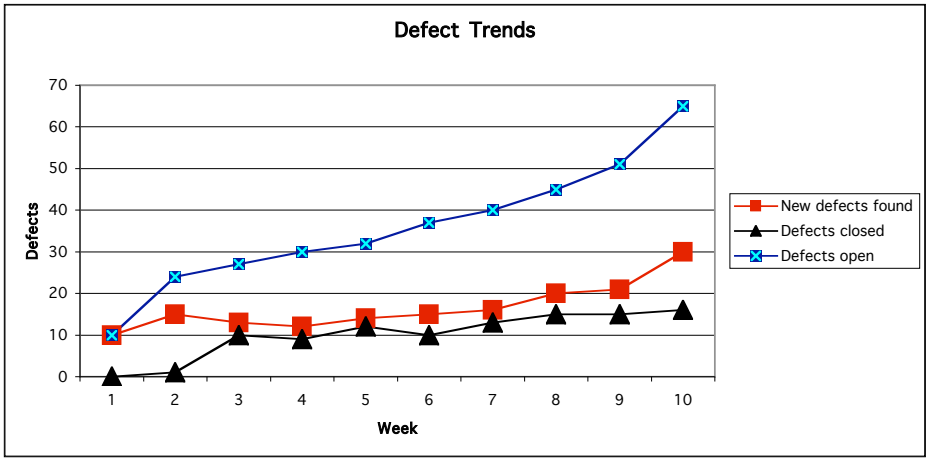
© 2004 Johanna Rothman

Middle of the Project Trends

- Defect trends
  - Defect find and close rates: Are you fixing defects as quickly as you find them?
  - Cost to fix a defect: How much does it cost you (at which time in the project) to fix a defect?
  - Fault Feedback Ratio: How many defects reappear?
- What is product performance, and is it good enough?
  - Do you know how to measure product performance?
- If reliability or performance are important to you, start measuring them now
  - Compare scenarios by build

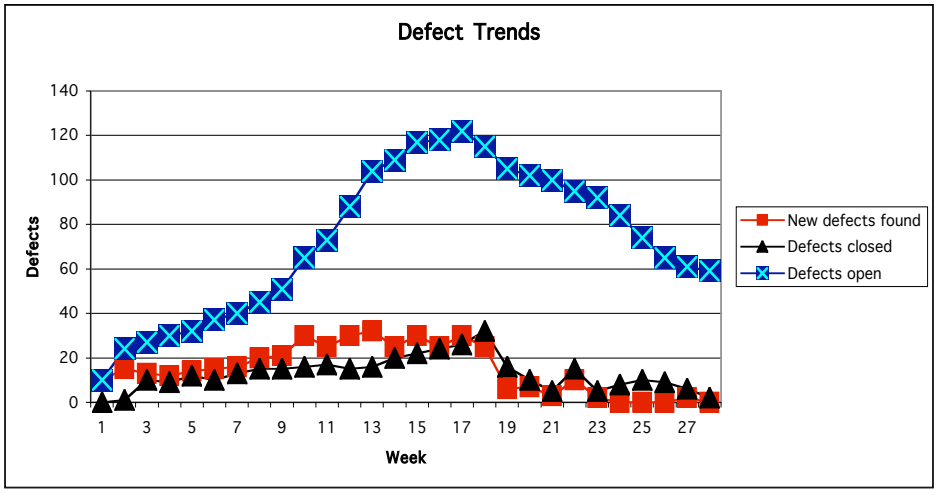
© 2004 Johanna Rothman

Defect Trends, Early in the Project



© 2004 Johanna Rothman

Defect Trends: Found/Closed, Full Graph



© 2004 Johanna Rothman

About Cost to Fix a Defect

- The more proactive you are, the higher your costs will be later in the project
  - The fewer defects you find later, the higher the cost of finding and then each defect
  - The lower the total overall cost
- Make sure you know what you're measuring
- NEVER measure cost to fix a defect by individual

© 2004 Johanna Rothman

Cost to (Find and) Fix a Defect, Initial

Company (for a specific release)	# of people	Cost per person -day	# of days	# of system test fixes	person -days	Average days to find and fix during system test	System test cost to fix a defect
Dan	5	$500	40	125	200	1.6	$800
Avery	10	$500	20	30	200	6.7	$3,333

© 2004 Johanna Rothman

Updated cost (Avery is Proactive)

Company (for a specific release)	# of people	Cost per person-day	# of days	# of system test fixes	person-days	Average days to fix during system test	System test cost to fix a defect
Dan	5	$500	40	125	200	1.6	$800
Avery	10	$500	20	30	200	1.0	$500

© 2004 Johanna Rothman

Cost to (Find and) Fix a Defect, Total Picture

Project	System test cost to fix a defect	# of System test fixes	System test total cost	pre-release total cost	Average time to fix post-release	post release cost to fix	post-release # of fixes	post release total cost	Total pre and post release defect fix cost
Dan	$800	125	$100,000	$100,000	15 person-days	$7,500	23	$172,500	$272,500
Avery	$200	30	$6,000	$37,250	5 person-days	$2,500	2	$5,000	$42,250

© 2004 Johanna Rothman

Fault Feedback Ratio

- What's the total picture about productivity?
  - You can't measure output without also measuring quality of output
    - People create output and they create defects
  - Best to measure this weekly
- FFR tells you how good the work products are initially, and how much wheel-spinning they are doing

Date	Size in LOC	FFR (this week only)	Average pre-release cost to fix a defect
1/1	425,000	14%	1 person-days
4/1	800,000	16%	3 person-days
7/1	1,500,000	12%	2 person-days
10/1	2,000,000	18%	3 person-days

© 2004 Johanna Rothman www.jrothman.com jr@jrothman.com 25

Late in the Project Trends

- Defect trends
  - How many problems are open?
    - The more open problems, the longer it takes to handle each one
  - Do people choose the easiest problems to fix first?
    - Then the remaining problems take longer to fix
  - How long does it take to fix a problem?
    - If problems take longer and longer to fix, the system becomes unmaintainable
  - How many new problems are introduced when fixing problems?
    - The higher the Fault Feedback Ratio, the longer the developers take to fix problems

© 2004 Johanna Rothman www.jrothman.com jr@jrothman.com 26

My Minimum Metrics

- Schedule estimates, actuals, and EQF
- Staffing
- Requirements changes
- FFR throughout the project
- Cost to fix a defect throughout the project
- Defect find/close trends throughout the project

© 2004 Johanna Rothman

Summary

- Choose what you'll need to measure
  - Start at the beginning of the project
- Act based on your measurements
  - Projects don't improve if you leave them alone
- No matter what, learn from your projects…

© 2004 Johanna Rothman

Resources and References

- Austin, Robert. Measuring and Managing Performance in Organizations , Dorset House, New York, 1996.
- DeMarco, Tom. Slack. Broadway Business Books, New York, 2001.
- DeMarco, Tom and Tim Lister. Peopleware, Productive Projects and Teams. Dorset House, New York, 1999.
- DeMarco, Tom. Why Does Software Cost So Much. Dorset House, New York, 1995.
- Weinberg, Gerald M. Quality Software Management, volumes 1-4. Dorset House, New York, 1992-1997
- Wysocki, Robert et al. Effective Project Management. Wiley, New York, 2000.
- Hal Macomber's weblog, weblog.halmacomber.com
- I have a number of articles www.jrothman.com, and my Managing Product Development blog: www.jrothman.com/weblog/blogger.html

© 2004 Johanna Rothman

Guiding Software Quality by Forecasting Defects using Defect Density

Bhushan B. Gupta
Hewlett-Packard Company

Abstract

Defect prediction using historical defect density can provide an early look into inadequate testing resources, ineffective test cases, improper code coverage, and sub–standard code quality. Better understanding of these indicators helps a development team focus on creating more effective test cases and test procedures, introducing code reviews, and balancing resources to get a better quality product. The author has utilized the historical defect density and defect partitioning to predict total number of defects on two projects. Although the success has been limited due to the lack of historic data, the technique has compelled to search answers to questions including: are we finding enough defects and why are we seeing so many defects at such an early stage.

By assessing the number of lines to be written for a product and using a shop's historic defect density, one can predict the total number of defects. Based upon an organization's past experiences, the prediction can be fitted to a pattern that typically represents defect arrival rate in calendar time during a product development. The prediction depends upon accuracy of estimated lines of code and the similarity of the development environment for which the historical density was calculated to the current environment. It is imperative that the counting mechanism for all systems be the same. If it is not, the estimated lines of code will differ substantially and the total estimated defects will be off by a wide margin.

Introduction

By far, the defect density is the least used data point to monitor the progress of a software product under development. Defect density can help in predicting the possible total number of defects injected into a product. This information can be leveraged to assess the overall resources needed for product validation, ongoing software quality, test and code coverage, and the additional resources needed for delivering a better quality product. Typical defect related metrics are: find rate, fix rate, and overall open and closed defects. Some organizations make an effort to track the defect by software components. Very rarely, the organizations develop a defect profile based upon their historical data or empirical formula to guide their quality activities. David Card [1] has suggested developing a defect profile that can be used as a budgeting tool for estimating the quality efforts needed for product development. He has emphasized the use of historical data to develop a defect profile throughout the lifecycle.

Basic Concepts

Defect Density

"Lines of code" is a simple measure of productivity that forms the basis for defect density. It can be measured with relatively low effort and can be accurate within reasonable limits, but the number can be misleading if a development environment uses multiple languages such as assembly, C, perl, and html. For example, in a development environment such as C++, one can achieve the same functionality as a Visual Basic module by using an existing object with fewer lines of code. One should also clearly distinguish a defect from an enhancement request when establishing the defect density. The measurement can include all the lines in the code-base including comments (normally known as commented Source Lines of Code - SLC) or exclude the comments (Non-commented Source Lines of Code – NSCL). Although, NSLC provides better accuracy, SLC is simpler to estimate.

The defect density is defined as the number of defects per thousand lines of code (KSLC). It varies from company to company, division to division, and team to team. The variation is caused by the development methods used (adhoc vs. structured), programming practices, and the effectiveness of the testing group. Some organizations are very end-user centric and their testing is aligned with the customer's use of the system. In those circumstances, the number of defects found is higher leading to a higher defect density. In such an instance, defect density still forms a sound basis for defect prediction as it truly reflects the end-user environment. For a well established development team the defect density is often consistent across projects unless the nature of projects changes drastically. The prediction accuracy can be refined over time as the team completes more projects and gathers additional data

You can predict the total number of defects before the onset of a project. This enables you to estimate your quality resources. Additionally, you can predict the defect profile for the entire development cycle and adjust the profile as the project progresses and deploy resources accordingly. If you were to predict the number of defects at the onset of the project, you would need to have a reliable defect density value and a fairly reasonable count of lines of code to be written. A sound estimation of the total number of defects for a similar project can be a reasonable approach. A defect density value based upon the historic data of the team is the best representation. The historic data takes into account the team dynamics, development environment, and quality practices. In the absence of historic data it is possible to use a representative value form industry. David Card [2] has cited a defect density value of 5 per KSLC based upon the industry trends. If one wishes to predict the total number of defects once the project is in progress, Steve McConnell [3] has provided a method based upon the defects found so far during the development. The method involves dividing total defects found into two or more groups and computing the total number of defects for the project based upon an empirical formula.

Analytical models can also be used to predict the overall defects and track the quality status of a product under development. One such model has been developed by Grafney [4] based upon the Rayleigh dispersion curve. The profile is defined by:

$$Vt = E\,(1 - \exp(-B(t^{**}2)))$$

Where:
Vt = Number of defects discovered by time *t*.
E = Total number of defects inserted.
B = Location parameter for peak.

At a given time t, it is possible to apply the linear regression technique to determine the values of B and E. These values may vary as the qualification goes on.

The challenge is to apply this technique in an iterative development where the software functionality is built incrementally, especially when some iterations have different development scopes. It is also necessary to consider the definition of time in this model; execution time can give the best results, but can be hard to measure for many teams. Calendar time is far easier to measure, but can bias results to some degree when test efforts vary during a period. Simmons [5] has described a six step process to use this technique for predicting defect arrival rate in calendar time.

Case Studies

We applied the defect density concept to predict the total number of defects and thus manage the quality activities using our estimates for two projects. Both projects were web-based applications, but had different environments. Project I had a code base that was purchased from a third party and the Hewlett-Packard team customized it as a Web printing application. The code was developed by a globally-dispersed team. The development work for Project II was carried out by a Hewlett Packard team. The application of these concepts on a third project (referred to as Project III) is in progress. The remaining paper describes how we utilized the defect density concepts.

Project I
As stated earlier, this project was to build additional functionality on a code from a third party. The developers who built upon this code were not intimately familiar with its initial design and the documentation was sparse. The original code was developed by globally-dispersed teams. Certain components of the code base were well tested, but others were not tested to the same extent. The ultimate goal of the project was to update the code-base to support web

publishing. The effort included changing some existing functionality and writing some new code.

The group did not have any of its own historical data to base the defect density estimate for the new code. Early Web application development used new languages, technology, and concepts. Since it was a new environment, we decided to use the defect density value of 8 per KSLC for this project, which we borrowed from another HP group. To compensate for code complexity we chose to vary this number between 7 and 9; 7 for the simple code and 9 for the complex code. The code complexity was based upon how engineers felt about a particular function and was not supported by any statistical calculations (Although one can apply the concepts of amount of data processed, input/output, database interaction, and code logic as a measure of complexity). We simulated the defect profile by dividing the defects among iterations and distributing them over the product development lifecycle. Figure 1 shows the predicted defects.

We tracked the defects during the development and compared the actual rate with our prediction all along the development. As we started to test the code, we soon realized that there were far more defects than we predicted. Our analysis lead to the following explanation:

- The code modules that were not well tested before were now being utilized by the new code and were being tested. In addition, the new functionality added to the code required additional testing of the existing functionality that resulted in more defects.

- There were issues with the alignment of the old code with the overall value of the solution to the customer. This lead to the revision of some of the original code and subsequently more defects.

- Since the original code was designed and built by an external company and the amount of code was rather large (over 1 millions lines), the new developers did not have sufficient time to comprehend the integration of the new code with the old code. The lack of documentation made this task even more difficult.

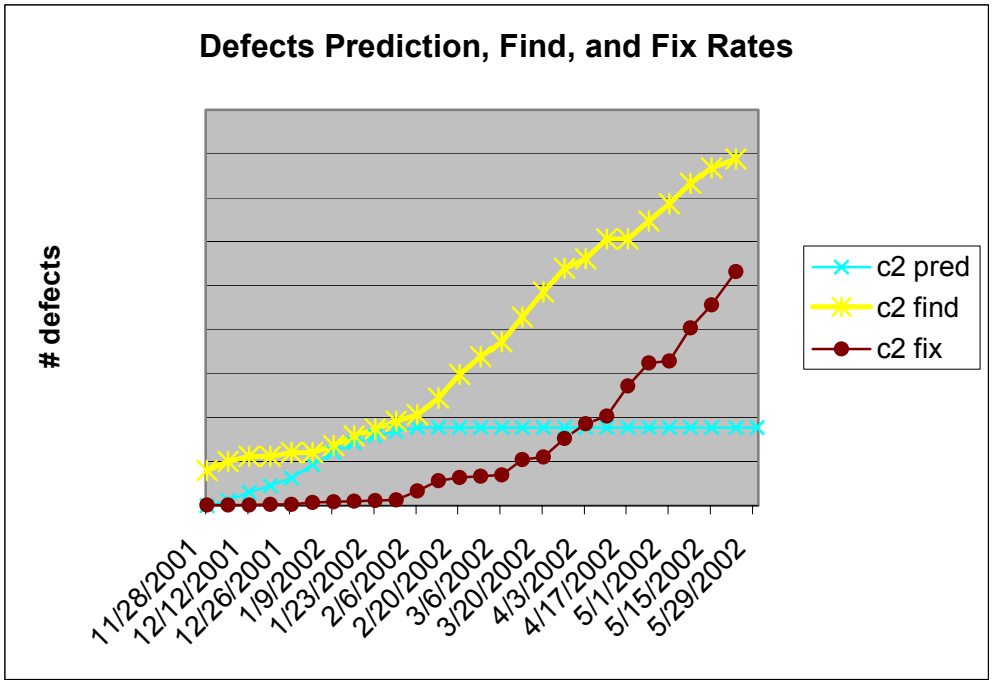


Figure 1. Predicted and Actual Defects Found in Project I

As we watched the gap between the predicted and actual defects grow we began to realize that one needs to be aware of the differences that can exist when the entire code is not developed by the same organization. The knowledge and understanding of the two environments plays a big role in this exercise. We also did not verify the number of lines that were added to the code, which would have been a good measure of our prediction.

Project II

Project II was being developed in parallel with project I in its entirety which differentiated it from Project I. The same team gathered the requirements, designed the system, and implemented and validated it. Therefore, the team had a sound understanding of the system. The team used the same defect density as for Project I and followed the same process to calculate the total number of defects. The results were more encouraging as shown in Figure 2. We did not use the Rayleigh dispersion curve and kept the profile linear. A Rayleigh dispersion would have more closely matched the actual find rate. In retrospective, we feel that a Rayleigh curve is more useful in assessing the adequacy of the quality resources and progress being made towards the quality goals.

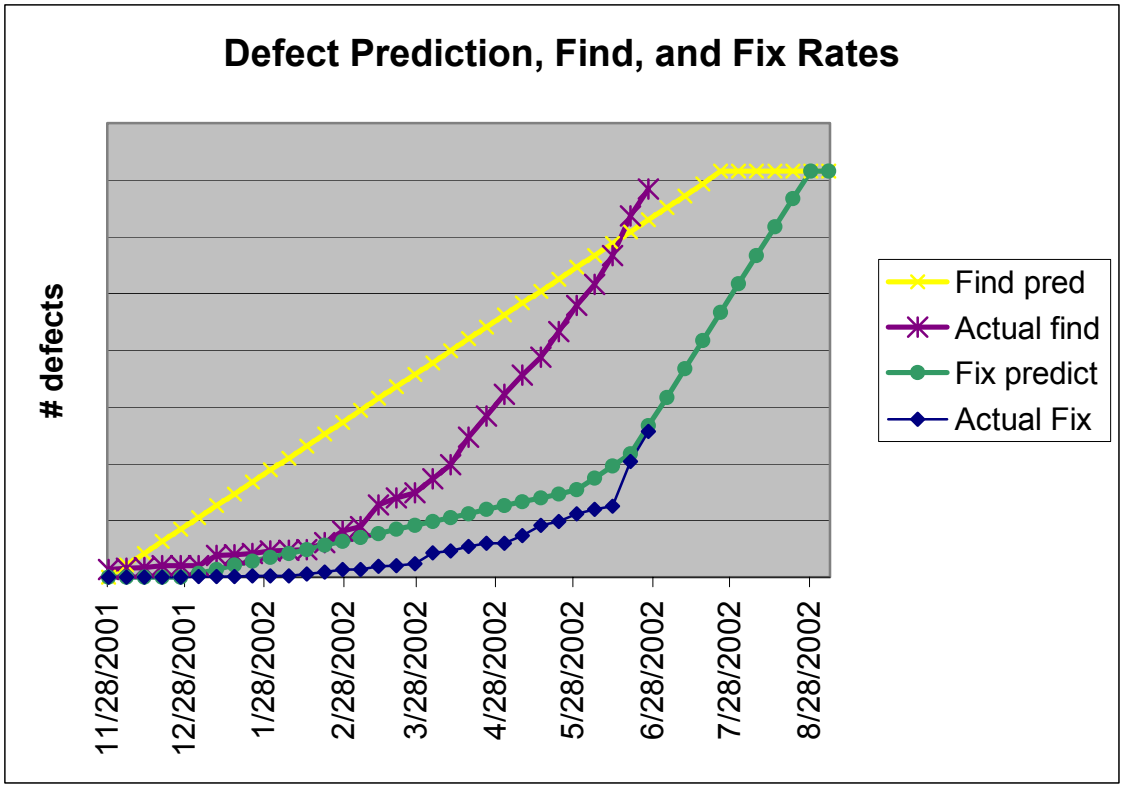


Figure 2. Predicted and Actual Defects Found in Project II

Project III

The two projects gave us a baseline and some direction in the use of defect density and defect profile. The same team that worked on Project II is now working on Project III. Project III is similar in scope with the exception that it has a larger UI component. Since the actual data from Project II came close to our prediction, the defect density from this project, 6 per KSLC, made a good candidate for Project III. We estimated the lines of code for Project III and predicted the total lines of code and generated a defect profile.

Since our prediction was based on a historic data and the development conditions were similar to Project II, we have had a higher level of confidence in our prediction. In the early stages of the development, we saw fewer defects than predicted, which lead us to think that we don't have a good test coverage. A further analysis confirmed our belief as the percentage test coverage was less than half. We were not running all the tests as some of the functionality was not yet turned on.

As the development progressed, we began to see a higher number of defects. The trend continued and we eventually passed the predicted number of defects mark. The testing conditions stayed the same. At that juncture, we were beginning to speculate on the defect density. We re-predicted the defects using

alternate approach suggested by McConnell [3]. We also tried to evaluate the validity of defect density estimate we had used. Simple research at other HP sites revealed a defect density of 12 per KSLC. This data point was quite interesting and was based upon the fact that HP, in general, approaches testing from the customer scenario. This testing approach yields additional defects that may actually be categorized as enhancement requests and may inflate the defect density for future estimates. The use of this defect density doubled the earlier predicted defects. Using McConnell's partition method, we have predicted a new total and are waiting to see how it holds up

As we proceed towards the last phase of the project, we are watching the defect trend. If we exceed our new prediction, that will alert us to look into our development practices. We will use the defect causal analysis techniques we have frequently used [6] and employ some practices such as pair programming and code review to achieve higher levels of code quality. We would also study the defects to identify the ones that are not actually defects, but enhancement requests.

Utilizing the concepts of defect density and defect profile

We have begun to deploy the defect density and defect profile as program management tools to directly support testing decisions, testing activities, and quality processes. In the early stages, finding less than the expected number of defects resulted checking our test coverage. The overall result was – better attention to test coverage.

At a later stage, when we reached the predicted number of defects, we were still testing at our peak and the defect find rate was still increasing. When the number of defects exceeded 30% of our predicted defects well before the predicted time, we started to investigate the reasons for finding more defects than expected. When we determined that our quality processes are sound, we started to delve into defect density and adjusted our prediction using McConnell's technique. At a later date, it will be prudent to count the lines of code and do a reality-check on the defect density.

The total defects and defect profile information is being used to get a feel for test coverage, managing open defects, resolving defects to minimize the backlog, and other quality activities. The information has helped us guide the defect fix needs, worried us about the overall product quality, and helped us make decisions about the overall quality resources.

Conclusion:

It is often difficult to predict the total number of defect within a reasonable accuracy (<10%). However, using a proven prediction method with the historic

data could minimize the estimation error. Like any other estimation, this also is a subjective analysis and one should use these results cautiously.

Although the known methods to predict the overall defects and defect profiles are primitive, the information can effectively guide the quality resource estimation on the offset of a project. The defect profile can guide test coverage and code coverage. The defect find trend during the early stages of the development can be used to establish the defect profile and a reasonable estimate of the total number of defects using an analytical model. The defect profile can help in day-to-day test resource requirements and streamline the resources for the upcoming long and short term needs. The program managers can be warned of inadequate defect discovery and resolution rates and have a clear understanding of the overall product release-quality. Although the testing is never complete, knowing that we have found all anticipated defects provides a higher level of confidence, especially once the defect density predictions are backed by field data supporting their accuracy. The consequences of decisions such as "reducing inspection and testing effort to meet the release date" are better understood. Unintended departures from planned quality activities can be detected and addressed.

References:

1. David N. Card, Managing Software Quality with Defects, CROSSTALK, March 2003. http://www.stsc.hill.af.mil/crosstalk/2003/03/card.html

2. David N card, Published Sources of Benchmarking Data, Software Productivity Consortium, March 2002, http://www.software.org/library/pubBenchmarking.asp

3. Steve McConnel, Gauging Software Readiness With Defect Tracking, *IEEE Software*, Vol. 14, No. 3, May/June 1997, http://www.stevemcconnell.com/bp09.htm

4. Gaffney, John. "Some Models for Software Defect Analysis." Lockheed Martin Software Engineering Workshop, Gaithersburg, MD, Nov. 1996.

5. Simmons, Erik., When Will We Be Done Testing? Software Defect Arrival Modeling Using the Weibull Distribution, 18th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2000

6. Bhushan B. Gupta, Defect Causal Analysis - A Grass Root Approach, Sixth International Conference on Practical Software Quality Techniques, March 2000, Austin, Texas

Software Estimation Models:When is Enough Data Enough?

Tim Menzies
Department of Computer Science, Portland State University, Oregon
tim@menzies.us;http://menzies.us

August 29, 2004

1 Overview

Do you want to be the manager of a canceled software project?[1] Do you know if your budget is adequate to the task at hand? If a project's costs are under-estimated then developers will be forced into many quality-threatening cost-cutting measures as the project develops. Sometimes, "cost-cutting" becomes "project-canceling":

> To gain control over its finances, NASA last week scuttled a new launch control system for the space shuttle. A recent assessment of the Checkout and Launch Control System, which the space agency originally estimated would cost $206 million to field, estimated that costs would swell to between $488 million and $533 million by the time the project was completed.
> –Computer News:Wed June 11, 2003

One reason for poor cost estimation is that, all too often, software managers don't have enough relevant data to make accurate estimations. For example, consider how much work was required to tune the standard COCOMO-II-2000 cost estimation model [2]:

- An initial regression analysis of 83 projects (this generated the COCOMO-I-1981 model [1]);
- Further data collection on 78 more projects;
- A DELPHI panel where experts offered their best judgment on factors controlling software costs;
- A Bayesian tuning phase that integrated the DELPHI results with data from the 83+78 projects.

Most industrial sites lack the resources to repeat the above process. Data collection from industry is notoriously slow and the above process took nearly two decades to complete [1, 2].

To shortcut the development time of an effort estimation model, the COCOMO team offer certain *tuning parameters* a and b in their model. They recommend

> ...having at least 5 data points for local calibrating the multiplicative constant a and at least 10 points for calibrating both the multiplicative constant a and the baseline exponent b.[1, p175]

While the comment comes from a definitive source, the COCOMO teams offers no evidence for the claim that data from 5 to 10 projects is enough. Curiously, the COCOMO team felt the need for 161 projects to generate COCOMO-II-2000. Perhaps there may be some drawback to tuning based on only 5 to 10 projects. To check this speculation, we conducted two studies using data from 62 COCOMO-I-1981 projects and 60 projects from

[1] *To appear, Pacific Northwest Software Quality Conference, Portland, Oregon, Fall, 2004;* http://www.pnsqc.org/

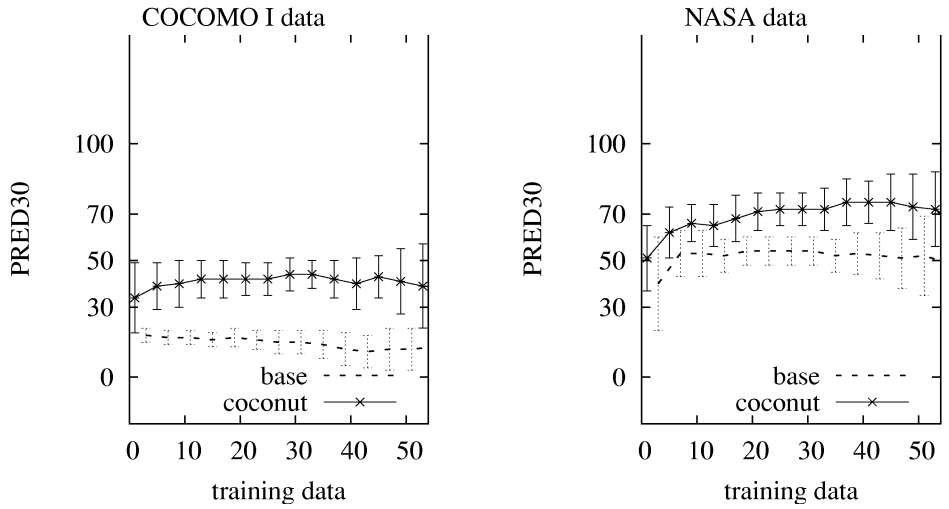


Figure 1: Sequence tuning.

60 NASA projects from the 1980s and 1990s [2] *Sequence tuning* experiments were performed where some learning device tuned an effort estimation model using more and more data. The learning was then disabled and the estimation model was tested on data not seen during training.

Sequence tuning was used since (a) it tests the theory on data not used in training (which is good experimental design); and (b) it emulates standard business practice where managers learn their own estimation models based on the projects seen to date.

In order to study the variance in the effort predictions, this procedure was repeated 30 times, each time with a different randomly selected ordering of the projects. The tuning method was the COCONUT tool described later in this paper[3] (COCONUT is short for "COCOMO, Not Unless Tuned"). As seen in Figure 1 for the NASA data, as sequence tuning proceeds down the x-axis, more data is available for training and performance improves. Performance is measured in terms of PRED(N). For example, at a PRED(30) of 69%, effort estimations for 69% of the projects in the test set are within 30% of the actual value. PRED(30)=69% was the best results achieved by the COCOMO-II team This was a *mean* figure seen in 15 of their test sets.

From Figure 1, several conclusions follow. Firstly, statements such as "10 projects are enough for tuning" need clarification. Certainly, the NASA results from Figure 1 stopped growing sharply after 10 projects but significant changes occurred up to 40 projects. Hence, we would replace claims like "N projects are enough" with predictive accuracy growth curves like Figure 1. Managers *might* want to stop at 10 projects but they may also decide to put in the extra effort to get more data (and this would be a decision driven by the available funding).

Secondly, it may not be enough to report the results of a calibration using just the mean PRED(N) levels. The whiskers in Figure 1 show ± 1 standard deviation around the mean. If this variance gets large enough, then while the *mean* might change, it may not longer be true that it changes *significantly* (i.e. satisfies a t-test that the new

[2] COCOMO data from [6], and downloaded from `http://www.vuse.vanderbilt.edu/~dfisher/tech-reports/raw-TSE-95`. NASA data courtesy of Dr. Jairus Hihn. Both data sets are available from `http://scant.org/2/eg/arff/cocomonasa.arff` and `http://scant.org/2/eg/arff/cocomo81.arff`.

[3] Available from `http://scant.org/tunes/tunes.html`

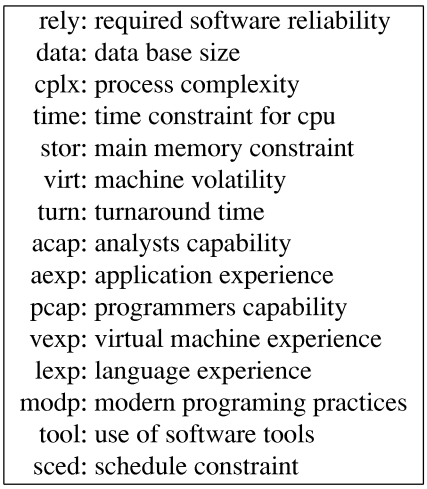


Figure 2: COCOMO-I-1981 multipliers.

mean is indeed different to the last mean). Our view is that calibration should continue till the variance minimizes which in Figure 1 is not till ≈ 30 projects.

The rest of this paper described exactly how Figure 1 was generated. That description starts with more details on COCOMO, then some notes on the NASA and COCOMO I data used in these experiments. This is followed by a description of the COCONUT tool.

2 COCOMO

The COCOMO project aims at developing an open-source, public-domain software effort estimation model. The project has collected information on 161 projects from commercial, aerospace, government, and non-profit organizations[3]. As of 1998, the projects represented in the database were of size 20 to 2000 KSLOC (thousands of lines of code) and took between 100 to 10000 person months to build.

COCOMO measures effort in calendar months where one month is 152 hours (and includes development and management hours). The core intuition behind COCOMO-based estimation is that as systems grow in size, the effort required to create them grows exponentially, i.e. $effort \propto KSLOC^x$. More precisely:

$$months = a * \left(KSLOC^b\right) * \left(\prod_j EM_j\right) \quad (1)$$

where a and b are domain-specific parameters; KSLOC is estimated directly or computed from a function point analysis; and EM_j is one of a set of *effort multipliers* shown in Figure 2. In COCOMO-I, the exponent on KSLOC was a single value ranging from 1.05 to 1.2. In COCOMO-II, the exponent was divided into a constant, plus the sum of five *scale factors* which modeled issues such as "have we built this kind of system before?". This COCONUT study used the COCOMO-I model since the data available to this study did not contain scale factors

The numeric values of the effort multipliers are shown in Figure 3. These were learnt by Boehm after a regression analysis of the projects in the COCOMO-I data set [1]. The last column shows $\frac{EM_{max}}{EM_{min}}$ and shows

	very low	low	nominal	high	very high	extremely high	productivity range
acap	1.46	1.19	1.00	0.86	0.71		2.06
rely	0.75	0.88	1.00	1.15	1.40		1.87
cplx	0.70	0.85	1.00	1.15	1.30	1.65	1.86
pcap	1.42.	1.17	1.00	0.86	0.70		1.67
aexp	1.29	1.13	1.00	0.91	0.82		1.57
tool	1.24	1.10	1.00	0.91	0.83		1.49
virt		0.87	1.00	1.15	1.30		1.49
vexp	1.21	1.10	1.00	0.90			1.34
modp	1.24.	1.10	1.00	0.91	0.82		1.34
turn		0.87	1.00	1.07	1.15		1.32
time			1.00	1.11	1.30	1.66	1.30
data		0.94	1.00	1.08	1.16		1.23
sced	1.23	1.08	1.00	1.04	1.10		1.23
stor			1.00	1.06	1.21	1.56	1.21
lexp	1.14	1.07	1.00	0.95			1.20

Figure 3: COCOMO I effort multipliers.

the overall effect of a single effort multiplier. For example, *acap* (analyst experience) is most important and *lexp* (language experience) is least important.

There is much more to COCOMO that the above description. Equation 1 is only the effort model and COCOMO comes with a schedule model as well. The COCOMO-II text [2] is over 500 pages long and offers all the details needed to implement data capture and analysis of COCOMO in an industrial context. For example, that text includes the templates, definitions and training material needed to deploy COCOMO in an industrial setting. That text also describes numerous details such as how to apply COCOMO very early in the life cycle; and how to handle multi-component systems or systems with significant amounts of automatic code generation. Further, Chapter Five of that text describes numerous emerging extensions of COCOMO including models for rapid prototyping, COTS integration, quality and risk assessment.

This current study is very relevant to those emerging models. When those COCOMO extensions are validated and calibrated to local sites, it will be vital to know how much data is required for that validation and calibration.

3 The Data

The tuning results on COCOMO I in Figure 1 are far less impressive than the results gained from the NASA data. This is an example of the *stratification effect*. When data is *stratified* into similar projects, then calibration occurs on projects that are more closely related. For example, Chulani et.al report that stratifying data can improve PRED(30) by anywhere from 5% to 12% [2]. Similarly, tuning works better on NASA data than on COCOMO I since the NASA data is more specialized. The COCOMO I data was collected from multiple industries (financial, telecommunications,...) while the NASA data is more *stratified* since it comes just from the aerospace industry. Since there is less variance in the NASA data, it is easier to tune.

For example, Figure 4 plots source lines of code vs development effort (in months) for our two data sets. Note that the *intra-organizational* NASA data has much less variance than the *inter-organizational* data from the COCOMO I data set (e.g. at KSLOC=10, the COCOMO-I efforts vary by an order of magnitude which is much more than the KSLOC=10 NASA efforts). COCONUT works better on intra-organizational data since it is less of a struggle to tune a and b to data with less variance.

This stratification effect has an important lesson for calibration: calibration to a particular site is beneficial when there is local data from that site to assist in the calibration.

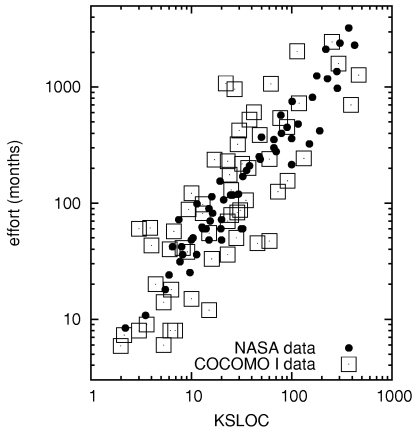


Figure 4: LOC,effort for NASA & COCOMO I.

4 The COCONUT Tool

Our previous experiments with COCOMO [4, 5] allowed for variation in all the effort multipliers of Figure 3, and the scale factors. This is a large space of options.

This large problem can be converted to a much simpler problem as follows:

- Ignore the scale factors; i.e. use COCOMO-I and not COCOMO-II);
- Freeze the effort multipliers;
- Just seek values of a and b that minimize the difference between the $actual$ the $estimated$ development effort; i.e. $(actual - estimated)/actual$.

This simpler problem is small enough to enable a search over the entire range of a and b.

Given N projects, the `test` function of the COCONUT tool shown in Figure 5 inputs a and b parameters and calls COCOMO-I for the projects numbered $start$ to $stop$ ($start \leq stop \leq N$). The `learn` function calls `test` for projects 1 to i ($i \leq N$) to find the a and b parameters that minimizes the `test` failure count. These best a and b values are then `test`ed on the projects numbered $i + 1$ to N. All the plots in this paper and reports of the test run, repeated 30 times (with the project numbering randomized before each repeat).

COCONUT is a very simple system and could be improved. For example, Equation 1 shows that a and b effect $effort$ monotonically and continuously. Hence, the exhaustive search on lines 5 and 6 of `learn` might be replaced with a binary search. Secondly, COCONUT is implemented in an interpreted C-like language (awk) and a reimplementation in "C" would make it run faster. Nevertheless, given that this sub-optimal exhaustive search implemented in an interpreted language takes just a few minutes to generate Figure 1, we are not motivated to explore optimizations. Further, the exhaustive nature of the search makes it hard to argue that some other method might do better than COCONUT.

Figure 6 show how COCONUT changed a and b for the PRED(20) runs on the NASA and COCOMO-I data. For all the experiments reported here, the a values were explored from 2 to 5 in increments of 0.2; i.e. $a \in \{2, 2.2, ..., 4.8, 5.0\}$. The b values were explored from 0.9 to 1.2 in increments of 0.02; i.e. $b \in \{0.9, 0.92, ..., 1.18, 1.2\}$. The whiskers in Figure 1 are 2 standard deviations wide, centered on the mean value.

```
   function test(start,stop,a,b,pred) {
1.   failures=0
2.   for(i=start;i<=stop;i++) {
3.     est   = cocomo(i,a,b)
4.     act   = actual(i)
5.     delta = (act-est)/act
6.     if (abs(delta)>pred) failures++}
7.   return failures}
```

```
   function learn() {
1.   Repeats=30
2.   while(Repeats--) {
3.     randomizeOrderOfProjects()
4.     for(i=2;i<=N;i += 3) {

         #Training stage
5.       for(a=AMin;   a <=AMax; a += AInc) {
6.         for(b=BMin; b<=BMax;  b += BInc) {
7.           sum=test(1,i,a,b,Pred)
8.           if (sum < least) { least=sum
9.                              BestA=a
10.                             BestB=b }}}

         #Testing stage
11.      failures = test(i+1,N,BestA,BestB,Pred)
12.      print Repeats " " i " " failures}}}
```

Figure 5: The COCONUT tool: pseudo-code.

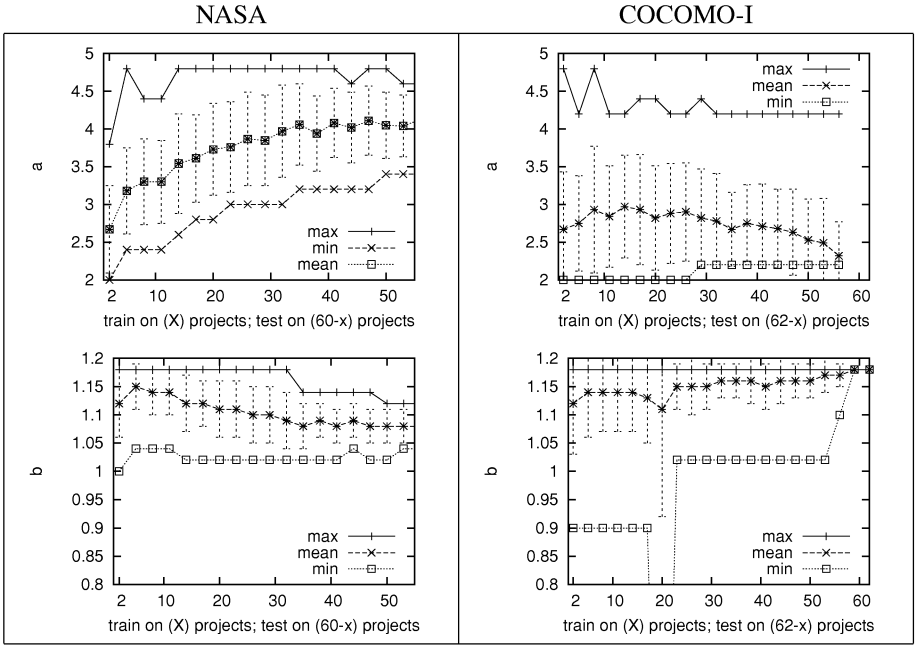


Figure 6: Tuning a and b in a PRED(20) run on NASA data (top) and COCOMO-I data (bottom).

As seen before in Figure 1, the variances tend to decrease as more training data is used. Note that the NASA runs converge to significantly much higher a and b values that offered by standard COCOMO-I and COCOMO-II (those standard values are $\langle a, b\rangle = \langle 2.94, 0.91\rangle$ respectively). Hence, NASA would be advised not to use un-tuned COCOMO-I for its effort estimations.

5 Conclusions

In summary, and in answer the question in the title of this paper, COCONUT-based calibration can improve the performance of an effort estimation model. That improvement rises sharply to ten projects, but the improvements continue till 40 projects. Further, after 10 projects, the variance in the estimates keep shrinking to around 30 projects.

It would be an error to summarize this study as (say) "here are new a,b parameters for COCOMO-I". The large variances in the *inter-organizational data* (COCOMO-I) of Figure 6 show that we can't just offer a single point estimate for COCOMO-I tunings (at least, if those tunings are learnt by COCONUT). The conclusions of this paper have to be interpreted within the context of COCONUT. Figure 1 is the preferred format and such plots can be summarized as follows:

> Based on sequence tuning experiments on N past projects, we predict that we can estimate effort on future projects within such-and-such intervals.

(those intervals being read off the min-max curves of Figure 1). Further, a sequence tuning experiment might also conclude that:

> Based on the history of projects seen to date, we predict that the variance in our effort estimations will reduce by *this much* if we can collect data from *these many* more projects.

Lest this reports appears too critical of COCOMO, it is important to note that COCONUT is an extension to COCOMO-I-1981 and could not work without it. As to COCOMO-II-2000, the COCONUT results offer a simpler method of obtaining the same, or even better, results:

- One of the main motivations for the Bayesian analysis of COCOMO-II was the regression results from the 83+78 projects had slopes that contradicted certain expert intuitions. For example regression of the COCOMO data concluded that building reusable components *decreased* development costs. Most experts believe that the extra effort required to generalize a design actually *increases* the cost of building such components. This anomaly was explained as follows: the 83+78 projects did not contain enough samples of projects that make heavy use of reuse. To DELPHI panel and the subsequent Bayesian tuning was used to fill in the gaps in the project data with expert knowledge. This combination of DELPHI+Bayesian methods proved successful: COCOMO-II-2000 had much higher PRED(N) levels than COCOMO-I.
- Given the COCONUT results, a much simpler method for incorporating expert knowledge is possible. Recall that COCONUT can find reach good PRED(20) after 30 to 40 projects. This number of projects could be artificially generated from, say, 4 experts each asked to describe 10 exemplar projects showing the range of projects typically done at their company. If those descriptions were made in terms of the COCOMO-I parameters, then COCONUT could then tune an effort model to that expert option using those 40 expert-generated examples.

Acknowledgments

Dan Port found a bug in an earlier version of this system and this paper has benefited enormously from the care he applied to this work. Jairus Hihn suppled the data that enabled this study and this paper would not have been possible without his help. This research was conducted at Portland State University, partially sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

References cited

[1] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[2] B. Boehm, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, A. W. Brown, S. Chulani, and C. Abts. *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.

[3] S. Chulani, B. Boehm, and B. Steece. Bayesian analysis of empirical software engineering cost models. *IEEE Transaction on Software Engineerining*, 25(4), July/August 1999.

[4] T. Menzies, D. Raffo, S. on Setamanit, Y. Hu, and S. Tootoonian. Model-based tests of truisms. In *Proceedings of IEEE ASE 2002*, 2002. Available from `http://menzies.us/pdf/02truisms.pdf`.

[5] T. Menzies and E. Sinsel. Practical large scale what-if queries: Case studies with software risk assessment. In *Proceedings ASE 2000*, 2000. Available from `http://menzies.us/pdf/00ase.pdf`.

[6] K. Srinivasan and D. Fisher. Machine learning approaches to estimating software development effort. *IEEE Trans. Soft. Eng.*, pages 126–137, February 1995.

Lean Cycle Time Reduction Techniques

How do you reduce cycle time? Skimp on quality? Add more staff? Have people work overtime? These are the classic, short-term solutions to schedule problems. They sometimes work -- but they do nothing to head off the next schedule crisis, and often make it worse. The solution boils down to three fundamentals: reduce variability, simplify processes and achieve smooth flow.

Introduction
* Why Cycle Time is Important
* House Video and Discussion
* Symptoms and Causes of Cycle Time Problems
* Definitions - Basic Terms and Concepts
* How Cycle Time is Improved (3 basic steps)

Step 1 - Reducing Variability

Step 2 - Reducing Process Complexity
* Value Added Analysis
* Process Flow Analysis
* Removing Queues and Waits
* Process Reengineering
* Managing White Space
* The Importance of Optimizing the System Rather Than the Components

Step 3 - Reducing Work in Process
* Smooth Flow
* Identifying and Removing Constraints
* Utilization vs. Productivity
* Observations and Caveats

Neil Potter is co-founder of The Process Group, a software engineering process improvement consultancy. He has 19 years of experience in software and process engineering. For six years Neil was a Software Engineer in Texas Instruments. For two years he managed a TI Software Engineering Process Group spanning America, England and India. Neil is an SEI authorized assessor for CBA-IPI and SCAMPI appraisals and Intro to CMMI instructor. He has a B.Sc. Computer Science from the University of Essex (UK) and is the co-author of *Making Process Improvement Work - A Concise Action Guide for Software Managers and Practitioners*, Addison-Wesley.

Mary Sakry is co-founder of The Process Group, a software engineering process improvement consultancy. She has 28 years of experience in software and process engineering. For 15 years she was a Software Manager within TI. In 1989 she lead TI's process assessment effort and was a member of an SEPG. Mary is an SEI authorized assessor for CBA-IPI and SCAMPI appraisals and Intro to CMMI instructor. She has an M.B.A. (St. Edwards) and B.S. in Computer Science (Minnesota). She is the co-author of *Making Process Improvement Work - A Concise Action Guide for Software Managers and Practitioners*, Addison-Wesley

THE PROCESS GROUP

Lean/Cycle Time Reduction Techniques

Neil Potter, Mary Sakry - The Process Group
Dennis J. Frailey - Senior Fellow, Raytheon; Adjunct Professor, SMU
P.O. Box 700012
Dallas, TX 75370-0012

Tel. 972-418-9541
Fax. 972-618-6283

E-mail: help@processgroup.com
Web: http://www.processgroup.com

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Workshop Outline

- **Introduction to Cycle Time Reduction**
  - A Cycle Time Reduction Process
  - Defining Cycle Time
- **Simplifying Complex Processes**
  - Value Added Analysis
  - Reducing Rework
  - Making Processes More Streamlined
  - Managing the White Space
  - Reducing Hot Lots / Priority Jobs
- **Achieving Smooth Flow**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Workshop Objectives

- **Explain cycle time concepts**
- **Think about your current project for cycle time reduction opportunities**
- **Develop ideas to address cycle time reduction opportunities**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Our Mission

Work **collaboratively** with customers worldwide to improve their software engineering process capability. Provide assessments, training, and consulting to meet their specific needs, resulting in improved software quality and delighted customers.

help@processgroup.com

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Introduction to Cycle Time Reduction

Cycle time is the time required to execute all activities in a process, including actual processing time AND all waiting time

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Why Reduce Development Cycle Time?

- Improve the organization's capability to develop products quickly
- Achieve competitive costs
- Start development later in the program cycle
- Allow less time to change requirements

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

How can Cycle Time be Improved?

- The following video illustrates how to improve cycle time
- As you watch, think of ideas that might be applicable to software development

What did they do to reduce cycle time?

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved. www.processgroup.com Version 1.5.3 7

THE PROCESS GROUP

How is Cycle Time Improved?

- Doing every process step faster?
- Working longer hours?
- Piling up work?

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved. www.processgroup.com Version 1.5.3 8

THE PROCESS GROUP

How is Cycle Time Improved?

- Doing every proc[illegible] step faster?
- Working lon[illegible]urs?
- Piling up [illegible]ork?

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

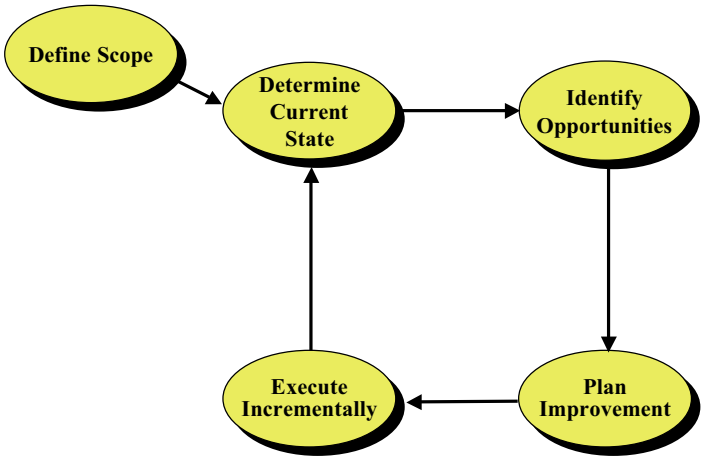
Cycle Time Improvement Helps People Work Smarter

- Reduce the critical path (the longest path)
- Eliminate waits, queues, bottlenecks
- Increase cycles of learning
- Reengineer the process (starting over)
- Achieve smooth flow

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Cycle Time Reduction Process



© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

What is Our Cycle Time?

- The time required to execute all activities in a process.
- Cycle time includes actual processing time AND all waiting time

Consider a "10 minute" oil change

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

How do You Measure Cycle Time?

STATIC CYCLE TIME

The average of the actual cycle times (CT) experienced by some number (n) of products

$$\text{Cycle Time} = \frac{CT_1 + CT_2 + CT_3 + CT_4 + \ldots + CT_n}{n}$$

- **But this is not always easy to measure when many of the products are only partway through the process**

  … so we need a dynamic measure

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Dynamic Cycle Time

DYNAMIC CYCLE TIME

The total work in process divided by the throughput of the process

$$\text{Cycle Time} = \frac{\text{WIP (products being developed)}}{\text{THROUGHPUT (products produced/unit time)}}$$

WIP = Work in Process

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

A Complex Process Can Lead to Long Cycle Time

- **Typical software development processes are more complex than they need to be**
- **Process simplification includes such techniques:**
  - **Value** added analysis
  - Reducing **rework**
  - Making the processes more **streamlined**
  - Managing the **white space**
  - Reducing **hot lots / priority jobs** that disrupt normal activity

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Process Simplification

- **Value added analysis**
- **Reducing rework**
- **Making the processes more streamlined**
- **Managing the white space**
- **Reducing hot lots / priority jobs that disrupt normal activity**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Value Added Analysis

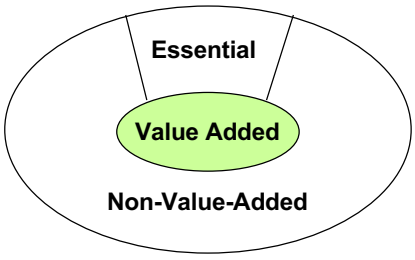
- **A process step adds value if it does ALL THREE of the following:**
  - **Changes the product**
  - Provides something that the **customer wants** done
  - Executes correctly the **first time**
- **Examples of non-value-added:**
  - **Waiting** time (in queues)
  - **Rework** and debugging
  - **Gold plating** (extra, unnecessary work)
  - **Reviews, inspections, testing and approvals**
  - **Measurement**
  - **Management**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Non-value-added Essential

- **Because our processes are not perfect, they must include certain non-value-added steps:**
  - Management
  - Measurement
  - Inspections and reviews
  - Testing



- **These are known as "Non-Value-Added Essential"**
- **They are very hard to eliminate, but are good long-term targets**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

First You Eliminate the Non-Essential, Non-Value-Added Process Steps

For example:

- **Reuse instead of reinventing**
- **Simplify procedures**
- **Simplify product designs**
- **Unnecessary project reporting (internal)**
- **Unnecessary project reporting (external)**
- **Redundant builds**
- **Redundant testing**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

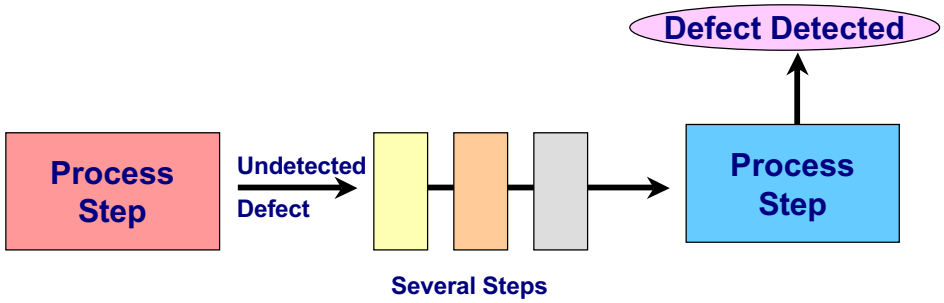
Process Simplification

- **Value added analysis**
- **Reducing rework**
- **Making the processes more streamlined**
- **Managing the white space**
- **Reducing hot lots / priority jobs that disrupt normal activity**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

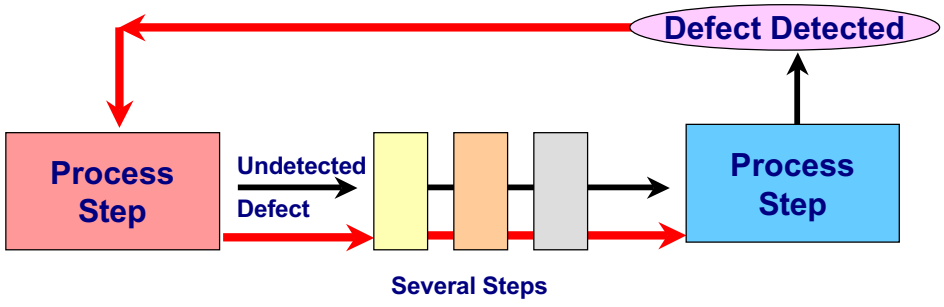
Reducing Rework - The Impact of Defects on Cycle Time



© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved. www.processgroup.com Version 1.5.3 21

THE PROCESS GROUP

Doing it Over Again

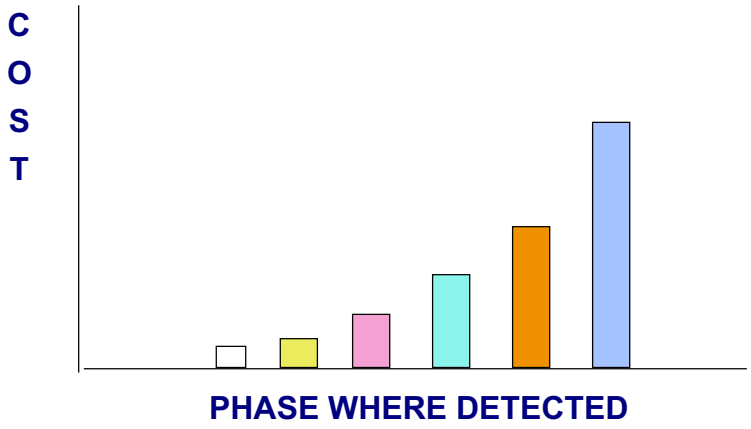


Rework costs money and time

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved. www.processgroup.com Version 1.5.3 22

THE PROCESS GROUP

The Longer it Takes to Detect, the Higher the Cost

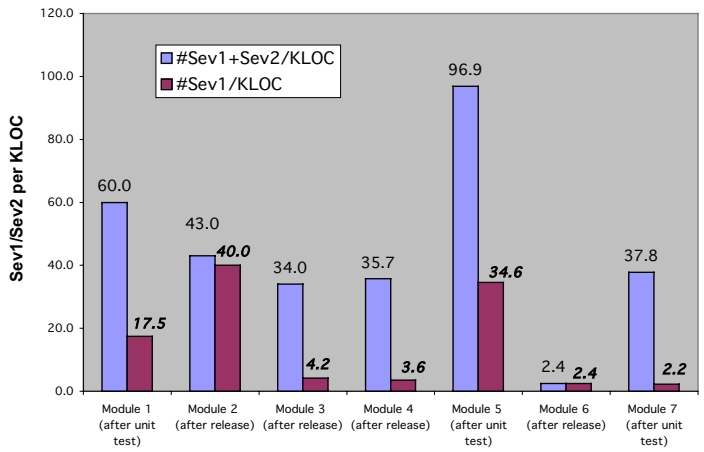


© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved. www.processgroup.com Version 1.5.3 23

THE PROCESS GROUP

Defects Released are Expensive

Java/C++ Inspections – Severity 1 + Severity 2 Defects per Thousands of Lines of Code



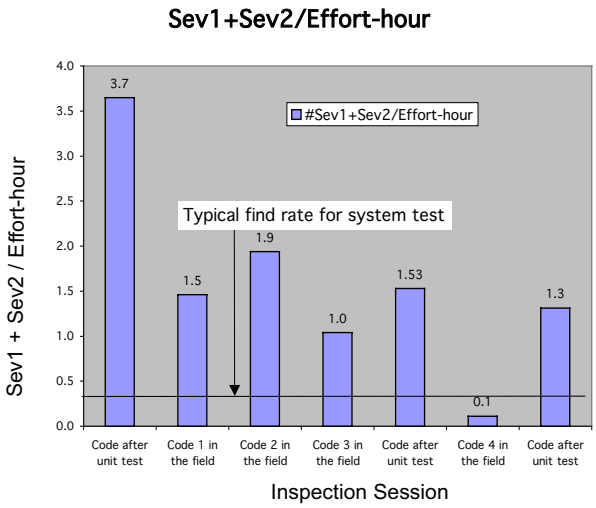
These defects delay the next release, and cause the current one to derail

KLOC = Thousands of Source Lines of Code (not including comments)

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved. www.processgroup.com Version 1.5.3 24

THE PROCESS GROUP

Finding Defects Quickly



Inspection found the defects at an average rate of 1.6 defects / effort-hour.

Test found some of them at 0.3 defects / effort-hour.

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved. www.processgroup.com Version 1.5.3 25

THE PROCESS GROUP

Look for Rework

- **REWORK is anything you do because you didn't do it right the first time**
  - debugging
  - correcting documentation
  - correcting designs
  - correcting requirements
  - retesting
  - responding to customer complaints
- **SOME rework is necessary, but most is not**
- **Total rework is a measure of process efficiency**
- **You probably have a lot more rework than you think**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved. www.processgroup.com Version 1.5.3 26

THE PROCESS GROUP

Reduce Rework - 1

- **Train people to do their jobs:**
  - Process education, common errors checklist, standards, tools
- **Focus on understanding the customers' needs**
- **Prototype high risk features / algorithms**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Reduce Rework - 2

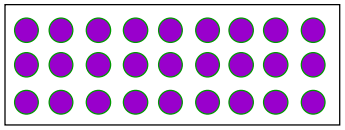
- **Focus defect identification techniques on areas that are:**
  - **The most critical to the program's operation**
  - **The most used section in the product**
  - **The most costly if defects were to exist**
  - **The most error-prone**
  - **The least well known**
  - **The most frequently changed**
  - **Just before a bottleneck**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

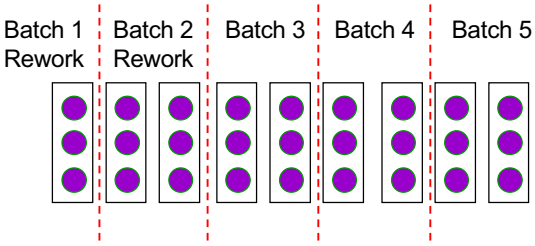
THE PROCESS GROUP

Reduce Rework - 3

- **Use the small batch principle**



The work to be done



Small batch principle

- allows for finding errors and requirements changes without large amounts of rework

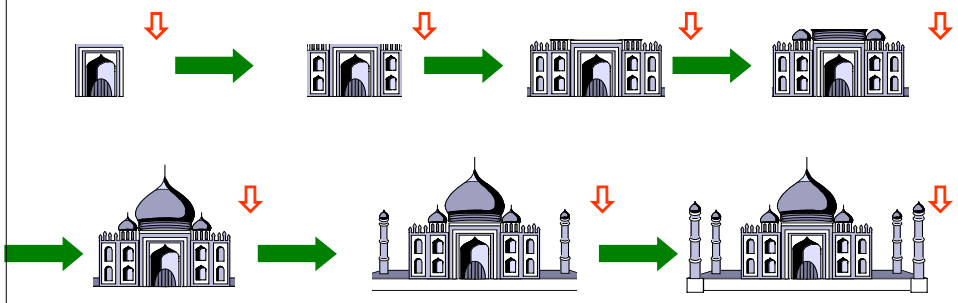
© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Reduce Rework - 4

- **Use evolutionary development to manage the changes of unstable requirements, algorithms and interfaces**

Changes are received and processed here (⇩)

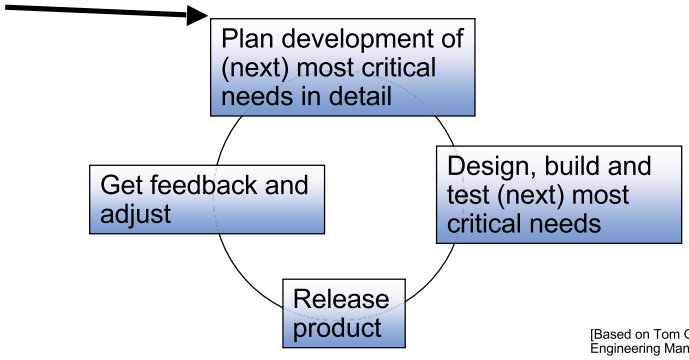


© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Evolutionary Development

- Determine customer needs
- Determine priority of customer needs
- Plan evolutionary development (high-value, low-cost items first)



[Based on Tom Gilb, Principles of Software Engineering Management, chapter 7]

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Reduce Rework - 5

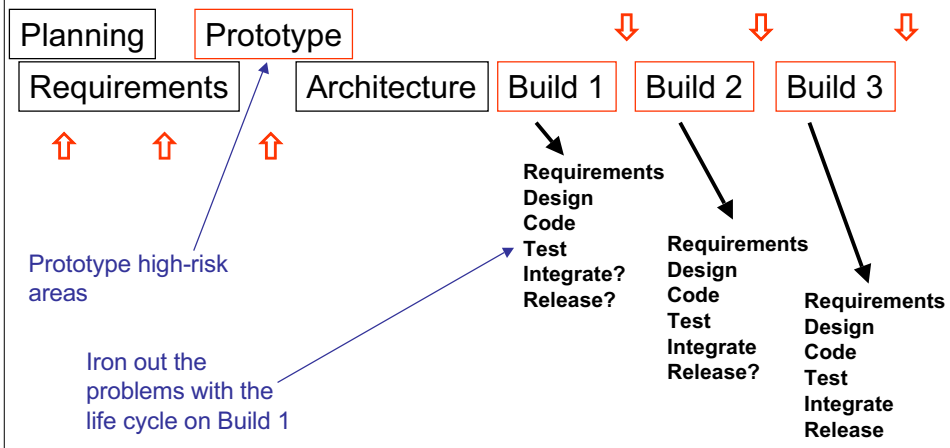
- Add a cycle of learning for new / critical / high-risk activities
  - Try new compiler or configuration management tool tool on one module
  - Write one module using the new language
  - Take one subset of the features through the whole process flow first to work out the quirks
  - Test server performance using dummy data

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Planning for Cycles of Learning

Requirements changes are processed here (⇩)



© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Process Simplification

- Value added analysis
- Reducing rework
- Making the processes more streamlined
- Managing the white space
- Reducing Hot lots / priority jobs that disrupt normal activity

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Understanding Streamlining

Exercise - The Passing Game

- **Input: Six objects piled on the end of a long table, with six people sitting on one side of the table**
- **Output: Six objects piled on the other end of the table**
- **Rules:**
  - Each item must be picked up before it is moved
  - Each person must hold each item for a minimum of 1 second -- if they do not, you get a penalty - the item moves back to the starting point
- **Goal: Shortest cycle time**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

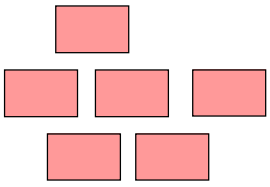
THE PROCESS GROUP

Re-engineering

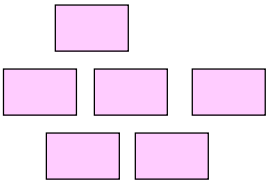
an Example of Streamlining the Process

- **Organizations usually begin by designing themselves to fit the needs of the customer, the business environment and the available technology**

Organization / Process



Customer, Environment, Technology



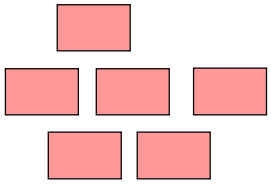
© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

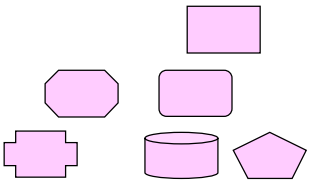
But Things Change

- **After time, the organization (or process) no longer fits the customer and/or the available technology**
- **Getting the job done becomes cumbersome**

Organization / Process



Customer, Environment, Technology



© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Re-engineering Example

Requirements process (version 1)

MRD

What is needed

ERD

What will be done

- **Develop Marketing Requirements Document**
- **Develop Engineering Requirements Document**

Requirements process (version 2)

MERD

What will be done

What will NOTbe done

- **Develop Marketing Requirements Document**
- **Review/revise with Engineering**
- **Add Engineering/Marketing priorities**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

How Often do you Need to Re-engineer?

- **Re-engineering is disruptive, so you should not do it too often**
- **Conditions that call for re-engineering:**
  - Changed customer environment
  - Significant new technology
  - Significant new competitive circumstances
  - Need to make the process more efficient
  - Significant new but different business opportunity

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Rethinking The Passing Game

- **Analyze what happened**
- **Try it again**
- **Hint: it can be done in under 2 seconds!**

- **What assumptions were invalid?**
- **Why did we make them?**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Examples of Unstreamlined Processes

- Tools do not share data
- Individuals do not understand each others' work, e.g.,
  - Step A does not consider the needs of step B
  - Step B spends time reworking the output of A
- Excessive time and effort spent on interfaces between different individuals and organizations
- Excessive movement of programs or documents or material between workstations or systems (including hand-offs)
- Special cases that require holds and delays
  - Custom installations, configurations, special design formats, special hardware and drivers

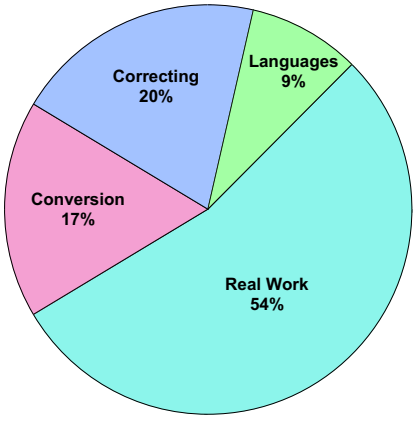
© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Impact of Unstreamlined Processes

On one project, roughly 50%:

- Converting documents and software from one tool/format to another
- Correcting problems due to different design styles
- Handling interfaces between computer programs written in different languages



© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Areas to Focus on

- **Processes and methods**
  - Are they working **effectively**?
  - Are they **interfacing well** with each other?
- **People**
  - Are they using their **time efficiently**?
  - Are they spending **too much time waiting** between tasks?
- **Computers and software**
  - Are they **available and effective**?
  - Is too much time spent **getting them to work**?
- **Materials/specifications/other inputs**
  - Are they **available** when they are needed?

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Process Simplification

- **Value added analysis**
- **Reducing rework**
- **Making the processes more streamlined**
- **Managing the white space**
- **Reducing hot lots / priority jobs that disrupt normal activity**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

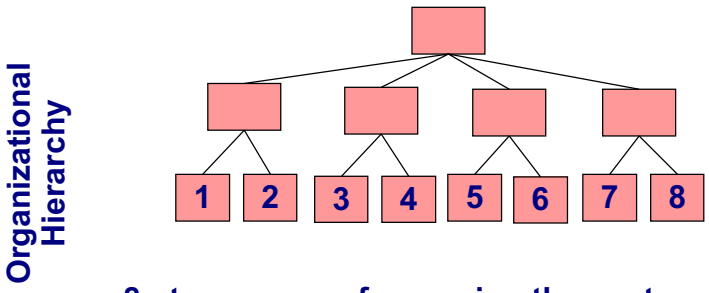
Managing the White Space

- **The "white space" represents all of the parts of your process that nobody is responsible for**
- **Or where the responsibility is too far from the day-to-day process**
  - **handoffs** between organizations or activities
  - changes in **responsibility**
  - senior **managers acting passively** as product managers
  - senior managers acting passively as system engineers
  - etc.

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

The Classic "White Space" Situation

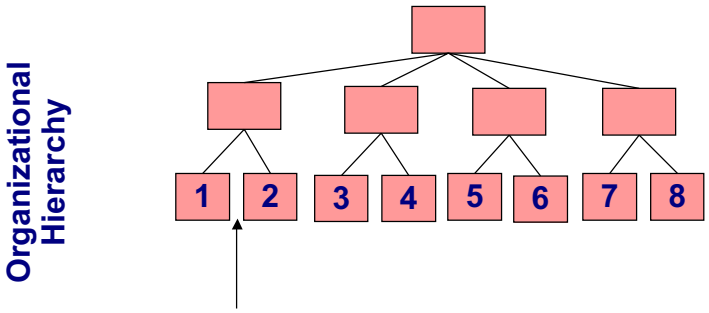


8-step process for serving the customer

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

The Classic "White Space" Situation

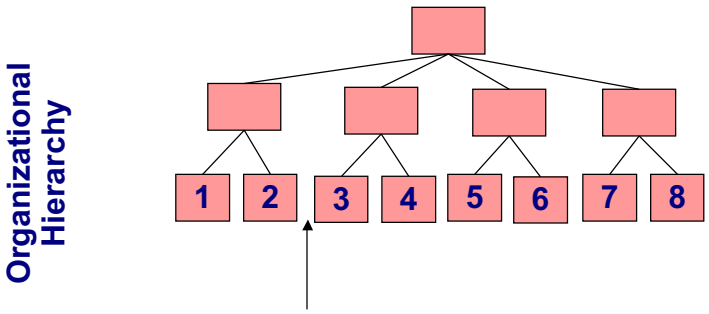


Who is responsible for this handoff?

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved. www.processgroup.com Version 1.5.3 47

THE PROCESS GROUP

The Classic "White Space" Situation



Who is responsible for this handoff?

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved. www.processgroup.com Version 1.5.3 48

THE PROCESS GROUP

Process Simplification

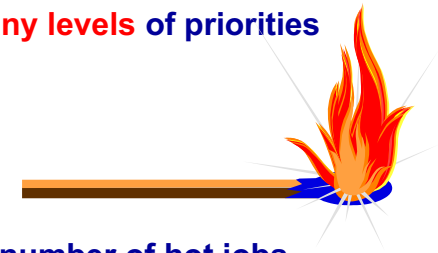
- Value added analysis
- Reducing rework
- Making the processes more streamlined
- Managing the white space
- Reducing hot lots / priority jobs that disrupt normal activity

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Hot Lots and Priority Jobs

- Tendency is to establish many levels of priorities
  - HOT
  - SUPER HOT
  - IMMEDIATE
  - PER THE BOSS



- Pressure exists to raise the number of hot jobs
- The list always grows

• Managing priorities consumes many resources
• And it delays other jobs

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

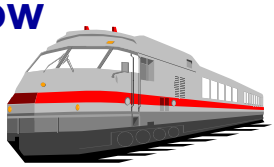
A Strategy for Managing Priorities

- **Allow only two priorities**
  - Hot
  - Not hot
- **Control the maximum percentage of "HOT"** (less than 10%)
  - If you add a hot job, you must subtract another
  - This requires DISCIPLINE (and faith that it works!)

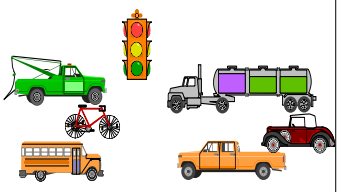
© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Achieve Smooth Flow



- **The ideal process flows smoothly, like a train running on tracks**
  - Note: tracks are empty most of the time
- **The typical process runs unevenly, like vehicles on a city street**
  - Lots of entrances and exits
  - Vehicles of different sizes and speeds
  - Some drivers uncertain of what they want to do
  - Lots of stoplights to "control" the flow (mainly to prevent collisions)
  - Note: streets are usually crowded

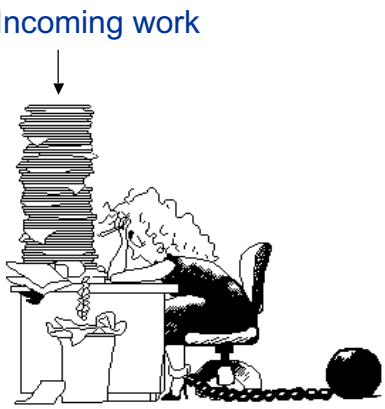


© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

What Prevents Smooth Flow?

- **Bottlenecks and problems that lead to:**
  - Queues and waits
  - Work in process
- **For example:**
  - Work piling up
  - Machines or software not being available
  - Excessive approval requirements
  - People pulled off projects
  - Excessive rework
  - Product stuck in test



© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

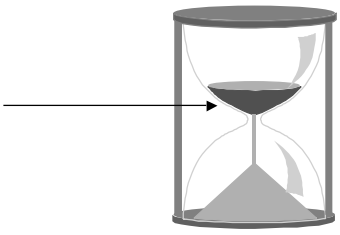
THE PROCESS GROUP

What is a Bottleneck?

- **A bottleneck is any resource with capacity less than the demand placed upon it**
  - **This could be a person, a computer, a network, a machine, etc.**
- **The bottleneck regulates the output rate of the entire process**

These symptoms reveal bottlenecks

- Code waiting to be tested
- Designs waiting to be coded
- Specifications waiting to be inspected
- Change requests waiting for approval
- Extreme multi-tasking



© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Identifying Queues and Waits

Think of yourself as the product being developed.

Take yourself through the process.

At what points are you just waiting with no useful work being performed?

These are bottlenecks, waits or queues that should be removed

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Eliminating Queues and Waits

Problem: You take a long time to complete test because everyone needs the test facility at the same time

Options:

- **Stagger development schedules to even out use of test system**
  - costs more up-front planning
  - requires people to be flexible in their schedules
- **Obtain more test systems**
  - costs more money? borrow for free? rent?
  - but may be worth it
- **Selective testing**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

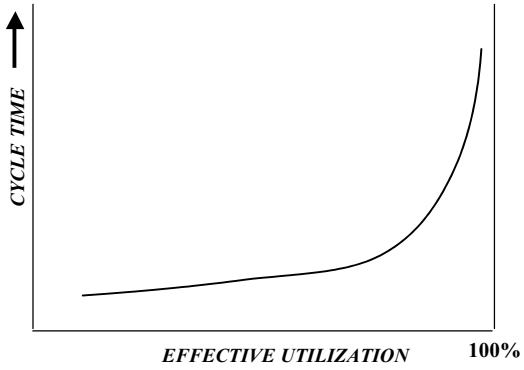
Bottleneck Improvement
Utilization is NOT Productivity

- **WEBSTER'S DEFINITIONS:**
  - **UTILIZE:** To **make use** of
  - **BUSY:** **Constantly active** or in motion
  - **PRODUCTIVE:** **Yielding** or furnishing **results**, benefits or profits
- **We tend to measure utilization by how busy we are, BUT utilization tells us little about how productive we are**
- **We should work to increase productivity rather than increase utilization**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Improving Bottlenecks



Observations

- High effective utilization will cause cycle time to increase greatly as output is increased marginally
- Reducing effective utilization is especially critical at the bottlenecks
- To minimize effective utilization, we must make our assets more productive

per Erlang, 1917 (see Gross and Harris, *Fundamentals of Queueing Theory*, Wiley, pp 10-11, 101-102)

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Example Cycle Time Problems

- Excessive paperwork, signatures, and reporting
- Poorly qualified subcontractors
- Reuse of products or software not designed for reuse
- Attempts to use the latest tools and methods -- without adequate support and integration
- Defective work entering a bottleneck (e.g., SCM & test)
- Not allocating resources to start the project

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Examples of Cycle Time Improvement

- "Just-in-time" training
- Plan testing and test equipment well in advance
- Rethink the detailed design process
  - Do you need to maintain detailed design documentation?
  - Do you need to do detailed design at all?
- Use on-line requirements and design models instead of paper documents and specifications
- Use focused product teams or Integrated Product Teams

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Examples of Cycle Time Improvement

- **Electronic approvals**
  - Web/email based approval
- **Customer access to development environment**
  - Providing customer with a password and i.d.
- **One location for product team members**
  - Or very good network access and tools
- **Standardize formats and tools**
  - PDF, Word, Excel, plain text

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Where are the Opportunities?

- **~30% of the improvement comes from technical changes**
  - Process changes
  - Tool changes
  - Changing rules and operations
- **~70% of the improvement comes from organizational, cultural and environmental changes, such as**
  - Education
  - Communication
  - Management
  - Teamwork

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Summary

- **Introduction to Cycle Time Reduction**
  - A Cycle Time Reduction Process
  - Defining Cycle Time
- **Simplifying Complex Processes**
  - Value Added Analysis
  - Reducing Rework
  - Making Processes More Streamlined
  - Managing the White Space
  - Reducing Hot Lots / Priority Jobs
- **Achieving Smooth Flow**

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Appendix

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

THE PROCESS GROUP

Recommended Reading

- Goldratt, Eliyahu M. & Jeff Cox, *The Goal*, (North River Press, 1984.) Also, *Theory of Constraints* and *It's Not Luck*.
- Hammer, Michael & James Champy, *Reengineering the Corporation, - A Manifesto for Business Revolution* (Harper Collins, 1993.)
- Womack, James P. & Jones, Daniel T. *Lean Thinking,* (Simon Schuster, 1996)

© Copyright 2000 The Process Group and Dennis J. Frailey. All rights reserved.

A Process for Very Small Projects

Dave Fleck
VP of Engineering
Wacom Technology Corp.
1311 SE Cardinal Ct.
Vancouver, WA 98683
dfleck@wacom.com

Abstract

Managers often find themselves attempting to properly manage very small projects, where very small projects are defined to be two weeks or less, and usually involve a single person. The requirements for these projects are notoriously incomplete. Project management often consists of a manager directing an engineer to "just do the work" with no guiding process. Development time is often charged to another larger project or to overhead. But for all the reasons that larger projects need formal management, smaller projects need managing too. Take for example a "quick" product release for a special customer. It might be a simple matter for a developer to make a few changes and do a release. But how will the quality group prepare to test a release that they have no requirements document to test against? Without a plan with start and end dates, how will it be possible to assure resources will be available at the right time? This paper presents a process scaled to apply sound practices to very small projects.

Biographical Sketch

Dave Fleck is Vice President of Engineering at Wacom Technology Corp. where he has been employed for the past 13 years. Under Dave's leadership, his department won the Software Excellence Award at the Pacific Northwest Quality Conference in 2002. He has over 25 years experience as an electrical engineer, software engineer, and engineering manager. He earned a B.Sc. in Electrical Engineering at the University of Cincinnati, a M.Sc. in Computer Science at Johns Hopkins University, and an MBA at Loyola College in Maryland. He is a member of ACM and IEEE and is an ASQ Certified Software Quality Engineer.

CMM® is registered in the U.S. Patent and Trademark Office
Capability Maturity Model is a registered service mark of Carnegie Mellon University

First published in the Pacific Northwest Software Quality Conference Proceedings, 2004

Introduction

Proper management of projects helps ensure within time and within budget completion. There are a multitude of published processes and standards on project management. However, nearly all of the literature describes management of projects that are tens or hundreds of person-years. Due to the nature of our work at Wacom Technology Corp., we have been forced to develop processes that work for small projects. Once we developed these processes, we were still left with projects of an even smaller size that needed an even lighter process. We call these very small projects "mini-projects". Mini-projects are projects which take two person-weeks or less and are usually carried out by one staff member. This paper describes the process we developed for mini-projects and the history behind its development.

Background

Industry standard processes are document heavy

If you follow IEEE or ISO standards for a small project, the documentation could take much longer than the project. Documents generated for larger projects could include documents for Concept Exploration, Requirements Elicitation, Software Requirements Specification, Risk Management Plan, Project Management Plan, Project Charter, Business Justification, Configuration Management Plan, Communications Plan, Design Specification, Post Process Analysis and more. We needed a process with documentation requirements that match the size of our projects.

Derived from our existing process

The software engineering department at Wacom Technology Corp., responsible for driver development, has a total of seven team members. It has been a challenge developing processes for such a small group. In the literature, teams of fewer than 50 are considered small and teams of ten or fewer are not considered. A typical project at Wacom is staffed with three or four engineers and has an average duration of three months. In response to this environment, Wacom developed a process tailored to small groups and small projects. Details of this process were presented at a previous PNSQC[i]. The existing process works well for approximately three quarters of our work. However, the process was burdensome for the remaining one quarter of our work, which consists of very small projects of roughly two person weeks. The process developed for mini-projects was derived from the process already in place.

Based on CMM®[ii]

Although the proposed process is a "lightweight" process, it is based on CMM® principles. Some CMM® activities are required by the process while others are optional, yet the process includes reminders to promote inclusion of the optional activities for those cases where appropriate. For example, the process considers risk, but there is no requirement to do a risk analysis and mitigation plan unless the responsible engineer determines that significant risks exist.

The Process

Overview of the original process

The standard project process within the engineering department at Wacom creates only one new engineering document per project. We call this document an Engineering Project Description (EPD). Having a single document reduces the number of documents created and updated. An EPD includes the following sections, chosen to cover CMM® Key Process Areas for level 2 and 3:

1. Preface and Summary
2. Requirements (Specifications)
   The majority of the document is specifications. The requirements are usually derived from (and traced to) a Product Requirements Document (PRD) from the marketing department. Standard sections common to most Wacom projects are in the document template. These sections aid in elicitation of requirements by reminding the responsible engineer to investigate these areas for potential requirements.
3. Design
   Some effort is made at a high level design to help improve the accuracy of time and effort estimates and to help implementation go more smoothly.
4. Plan
   The plan includes assumptions, dependencies, constraints, risks, schedule, and resources.
5. Process
   This section includes reporting, risk management, schedule management, and project retirement.

The level of detail in each section is kept low without losing the important points. Too little detail and the document is worthless, too much and it is a burden to develop and maintain it. The requirements coming from marketing have too little detail for proper planning and implementation. The process normally consists of adding detail to marketing requirements to the point where there is sufficient information to implement and test the features requested. Before the EPD is formalized, it is reviewed with marketing at a walkthrough meeting to verify that marketing's requirements were correctly understood.

The Mini-Project Process

Objectives

The main objective of creating the mini-project process was to control requirements. Very small projects are rarely initiated with a formal document from marketing (a PRD). Instead, the requirements are in the form of a verbal request from the boss, note of a discussion on the back of a napkin, or quite frequently, they are buried in an email thread with a product manager. Someone needs to extract these requirements and document what is to be done. In our case the developer, writing the mini-project document develops the requirements as part of the document. This avoids forcing someone else to write a PRD to derive the requirements from. At the same time, the requirements can be written in a fashion and to a level of detail that meets the developer's needs. Once the document is written, the requestor (usually product management)

can review the document to verify that the plans implement what they wanted. Software Quality Assurance (SQA) can also use the document for their resource and test planning.

The other main objective for this process was to avoid charging effort against other projects or no projects at all. Our first attempt at better tracking of miscellaneous effort was to write EPDs as placeholder projects to charge time against. For example, an EPD was written for an open-ended project to support third party developers. This worked well for ongoing activities we wanted to track, but still left us with a substantial number of very small untracked projects of short duration. Our response was to develop the mini-project process. The process consists of a master document, containing the generic practices that apply to all mini-projects. Separate simple documents are written for each mini-project. The contents of these documents are described below in the section "The Documents". The master document represents the project to which all mini-project time is charged to. Each mini-project is considered a task within the master project.

The goal is to keep the process lightweight while still achieving its objectives. Considering the size of the projects, overhead had to be kept to a minimum for the process to be practical. This was partly achieved by reducing the number of reviews and keeping the documentation under semi-formal document control.

Change Control

Wacom has two levels of formality to document change control, informal and formal. No matter what the level, any time a document is distributed after a change it must have a new version identifier.

During informal control the version identifiers are sequential numbers. Informally controlled documents can be changed and distributed by the author without any change control or approval. This level of control is useful when developing drafts of documents.

Normal project documents start under informal control while they are developed, then move to formal control when a project begins (implementation begins) and remains there during the life of the project. It is signed by all department heads before formal release and has usually been walked-through with stakeholders. Once a document is put under formal control its version identifier becomes sequential letters. Formal documents must have detailed change orders written in a precise and traceable from/to format, and approved by all department heads, before the document is modified.

Mini-projects are always kept under informal control. This eliminates the overhead of formal changes. The mini-project's responsible engineer is not only responsible for the work, but is also responsible for the project and document management. Even though the control of a mini-project is informal, each version of the document in placed in a designated area of the source control database. VSS (Visual Source Safe) is the source code control database Wacom uses to store all code and controlled documents.

Mini-projects also have limited approval requirements. The only approval occurs before work begins. This eliminates the overhead of stakeholder walk-throughs and obtaining all department head's signatures. The engineering manager approves all mini-projects, but SQA and marketing

only approve projects where they are stakeholders. For example, researching a new source code control system would only require the engineering manager's approval, but a software release with a slight variation for a special customer would require all three approvals.

The Documents

There are two documents used in the mini-project process. The first is a common process document, which applies to all mini-projects. The second is a document derived from a document template and is created for each and every mini-project. This later document could be likened to an EPD for a mini-project without the process details.

Common Process Document

The first document used in the process contains the mini-project process description. This document is under formal control and only changes if the overall mini-project process changes. It includes both the process for mini-projects (such how to initiate a project, how to properly fill out the document template, etc.) as well as the process followed for the project. Since the process that is followed for all mini-projects is usually the same, that process is included in this document. This avoids repeating the same process documentation in every individual mini-project document.

Document Contents

1. Preface
   The preface includes a sentence or two for each of the Purpose, Scope, Audience, Process Owner, Reference Documents (if there are any), and Glossary (if there are terms that need to be defined).

2. Process
   The process section describes the process used for and during mini-projects.
   a. Project Log: Mini-project numbers are sequentially assigned by entering them in the logbook.
   b. Document Control: This describes the change control and archiving process.
   c. Project Definition: This section describes how to properly capture the essential information about a mini-project. The majority of this section details how to fill out each item in the document template below.
   d. Project Approval: This section describes who needs to approve the project based on the type of project. For example, researching a new tool for engineering only requires the engineering manager's approval. Another example would be that creating a modified software release for a customer would require engineering, marketing, and SQA signatures. Marketing approval is an indication that they agree with the requirements as stated in the document. SQA approval is an indication that the requirements are testable. SQA uses the approved document for test planning.
   e. Reporting: Schedule status and issues are discussed at a weekly staff meeting.
   f. Schedule Management: This section describes what actions to take in the case of schedule slippage. The standard response is to drop defect corrections in reverse priority order (if there are any defect corrections) or re-planning.

g. Project Retirement: When the work is complete and delivered to SQA the project is retired. At this time the document is revised one last time to add the metrics data. These metrics include actual milestone dates and actual effort as determined from time sheets. Then the version of the document is changed from a number to OBS to indicate the document is now obsolete. The obsolete status indicates the project is complete. However, all obsolete documents are retained in the document control system for future metric retrieval.

Individual Project Document Template

Document Attributes

Every project starts with a standard document template. Using a template makes it easier to define a project without overlooking important information. The document template is very light on process. With the exception of the planning details or project specific exceptions, all of the processes for a mini-project are the same, so the Common Process Document (above) provides common process information without repeating it in this document. "Boiler plate" may be easy to insert in the template, but bloating the project document with it detracts from the important details of the project.

A high percentage of the document is requirements. These are the essence of the project, and their communication is essential. This is the main purpose of the document.

The document maintains its CMM® heritage. As you can see from the common process and individual project documents the level 2 key practice areas of Requirements Management, Project Planning, and Project Tracking are part of the process.

Document Contents

A mini-project document consists of the following sections:

1. Title Block
   The title block includes the Title, Date, Document Number, signature blocks for the Engineering, SQA, and Marketing managers, and revision number and history. If a manager's approval is not required, the corresponding signature block is signed by the engineering manager. In this way all blocks are always filled in and it is easy to tell at a glance if a project is ready to commence.
2. Project Objectives
   The purpose of this project is described here.
3. Project Deliverable
   The project deliverables, such as software and reports, are described here.
4. Requirements

The requirements breakdown is selected to match common requirements for the type of projects we work on. This section may be totally rewritten for some projects, for example a research project. Our template has a predefined breakdown as follows[1]:

a. Requirements.Base Version: This section lists the initial software version the project is starting with if the project is a corrective or adaptive maintenance release.
b. Requirements.Product: This section lists supported products or differences in supported products from the base version code.

c. Requirements.Platform: This section lists non-product hardware or operating systems to be supported or differences in support from the base version.
d. Requirements.Interfaces: This section lists any new component interface (such as API) or interface differences from the base version. These are usually transparent to the user but may affect third party developers. Inter-component interfaces are not included here.
e. Requirements.Behavior: This section describes the behavior or differences in behavior from the base version. These are behavioral differences that are externally observable.
f. Requirements.Look and Feel: This section describes any new user interface, or changes in the user interface from the base version. This is mainly the look and feel of the control panel and not the installer, which is detailed separately below.
g. Requirements.Installation: This section describes the installer details, or any differences from the base version installer, including look and feel.
h. Requirements.Other: This section lists any other requirements that are not included in the sections above.

5. Software Design: This section contains design at a level of detail determined by the responsible engineer. This could be as complex as a UML diagram for a newly implemented feature, or more of a "strategy" for simpler changes (such as "modify the layout of the XYZ dialog" or "search for occurrences of a similar product and add the new one to the case statements found"). This is done before starting the project to help improve time and effort estimates.
6. Testing Issues: Any special testing issues are detailed here. As an example sometimes an area of code will change that is not obvious in the plans. In this case this section would include a suggestion to SQA to test that area. Another example would be a note to test certain dialogs on both landscape and portrait monitor positions.
7. Technical Support Issues: Any issues that technical support should be aware of are detailed here. As an example, the technical support group should be aware of a new location for preference files.
8. Plan:
    a. Plan.Schedule: Project milestones are listed here. At a minimum, the end date will be listed here. Actual dates are inserted at project conclusion. Example: Initial release to SQA 12/09/03 (Actual 12/07/03)
    b. Plan.Resources: Staffing requirements are specified by employee name and man-hours required. Actual man-hours are inserted at project conclusion. Example John Doe 25 hours (Actual 32 hours). Any other required resources are also listed here, such as equipment or off the shelf software.

---

[1] Please note the numbering system we have adopted. Rather than numbering sections 1, 1.1, 1.2, and so on we use a "dotted" textual notation. Therefore "Requirements.Base Version" is a subsection of "Requirements". Numbers are only used within a section or subsection. This method makes intra and inter document references easier to maintain in the presence of document modifications. With abbreviations to the notation a document reference becomes "MP-17 rq.bv 3" for document number MP-0017, section Requirements, subsection Base Version, item 3. These are extensively used between marketing, engineering, and quality assurance documents for traceability.

c. Plan.Other: If other relevant planning issues need to be included, such as assumption, dependencies, constraints, risks, etc. they are included here.

9. Process: Simply states that this project will follow the standard Mini project process as detailed in the master document. If there are any risks under "Plan.Other" the mitigating activities are detailed here.

Conclusion

Results

To date, 15 projects have used the mini-project process. The process has achieved its two main objectives of improving requirements and tracking effort. In the past requirements arrived as a verbal request or email. Now the requirements from these requests are always enhanced and written as part of the mini-project and approved before work begins. The document template is helping in requirements elicitation. So for example, where in the past it was common for installer changes to be overlooked, those changes are now documented and become part of the effort estimation. Time sheets now have work charged to mini-projects instead of miscellaneous. There have not been any issues with overhead since very little time is required to fill in the document template and get it approved. SQA is now made aware that a project is underway, what its completion date is, and what the requirements are. Previously a cc on the initial email request was all of the information they received.

Challenges

A number of project documents were not properly controlled by placing them and their updates into the source control system. Also, most of the projects were not properly retired. The lightweight and informal nature of the process may be contributing to the lack of follow through. All of the engineers should be familiar with the process since the development of the process was a group effort. However, the process has very little oversight, relying on the responsible engineer to perform the process. The only managerial interfaces are the approval to begin work on the project and weekly verbal progress reports. A potential solution is to add columns to the logbook used to assign the project numbers to indicate who the responsible engineer is and a check for completion. That would allow the engineers and manager to see the open/complete status of projects at a glance. Another issue is the weekly reporting. Since the projects are two person-week or less (which is usually also two calendar weeks) the average project has two verbal reports. One report is an arbitrarily timed intermediate report and the other occurs at or after completion. Given the size of the projects the intermediate report could be eliminated.

---

[i] Ritchie, K., "Climbing CMM Levels in a Small Organization", Pacific Northwest Software Quality Conference 2002 Proceedings, October 2002

[ii] Paulk, M., Curtis, B., Chrissis, M., Weber, C., Capability Maturity Model for Software, Version 1.1, Technical Report CMU/SEI-93-TR-024, ESC-TR-93-177, February 1993 http://www.sei.cmu.edu/pub/documents/93.reports/pdf/tr24.93.pdf.

Keeping QA Job Skills Relevant in a Competitive Market

How to Find, Keep and Advance your QA Career

Claudia Dencker and Peter Yarbrough
Software SETT Corporation
15750 Winchester Blvd., #203
Los Gatos, CA 95030
408-395-9376
www.softsett.com

Abstract

Staying relevant is key to workplace survival, especially over the past few years in which high tech has seen many of its traditional jobs disappear or become transformed. Individuals who have had great job experience, good salaries and steady employment have had to suddenly re-evaluate what it means to be employed in high tech. This paper is a collection of thoughts and experiences of two Silicon Valley, software QA professionals, one an industry veteran and the other a dot com newcomer. In addition, an informal survey was conducted among the membership of a Silicon Valley Software QA association (SSQA), and some of their responses are included as well. [1]

This paper touches on topics ranging from formal and informal training, networking through local and national QA and software testing organizations, and Internet resources [2]. We also cover the importance of flexibility and willingness to try something new and getting out of one's "comfort zone". Two other traits will be explored: adaptability and providing value (ie., going the extra mile). Finally, we'll touch on how to seize the moment and not allow an opportunity to pass. This paper will include information from traditional and non-traditional sources as well as examples from the authors' own experiences.

[1] As a result of the authors' experiences and the respondents of the SSQA survey, many cited references are "Silicon Valley" centric. These could be used as a basis for further research into local resources in your area.

[2] The authors do not endorse the links listed in this paper but rather present them so that the reader can draw their own conclusions and use these links for further personal research.

Bios

Claudia Dencker, cdencker@softsett.com. Ms. Dencker is President of Software SETT Corporation, a company specializing in the global delivery of software testing and QA solutions. She has taught classes worldwide through the IEEE, University of California and Software SETT. Ms. Dencker received a Bachelors degree and a teaching credential from San Jose State University.

Peter Yarbrough, petey@softsett.com. Mr. Yarbrough has worked in Software QA and IT for Software SETT for the past seven years. During that time he has worked on many large-scale IT projects for Fortune 500 companies as QA lead or manager as well as testing web applications and shrink-wrapped software products. He studied Engineering at Santa Clara University and holds a degree in Technical Communications from De Anza College.

Copyright © 2004, Software SETT Corporation All rights reserved.

Getting Started

Revitalizing a QA career is never easy. In fact, it can be quite painful. You may find yourself pushed towards change at a pace that is uncomfortable or at time when it is inconvenient. But, start you must. It is always good to begin any career building activity with some self-assessment. How you think of yourself can affect your attitude, your perception and your opportunities (or the lack thereof).

Objectivity is important in assessing your career, but it must be tempered with interest and fueled by desire. Presumably, most people's goal is to find interesting, but not all-consuming, steady work. Others move from one short-term assignment to another. There is room in the world for both types, but keep in mind that business climate and needs may tend to prefer one type over another at any given time.

In kick-starting any career change or revitalization, you must ask yourself, "Who am I?" and "What do I want?"

Who Am I?

While many seek the answer to this question throughout much of their lives, in our context, the search should be a bit more straightforward, but nonetheless, just as deeply inquiring. You should first start by looking at your personal and professional traits and style. What managers and HR departments refer to as strengths and weaknesses can also be thought of as simply components of one's working or professional character.

Whether a trait is advantageous or otherwise is somewhat subjective, as different situations may be best served by one trait over another. Knowing when to express or engage a trait is a critical part of succeeding as a QA professional in high tech and the business world. The ability to stifle that same trait may keep one out of trouble.

Some traits can be unhelpful or detrimental in certain work situations. In a fast moving environment such as that of extreme programming not being a proactive, extremely communicative person could be a problem. On the other hand, in a more rigid, top-down bureaucracy, this same style might be seen as overly aggressive and disruptive.

Finding types of work and environments that are equitable to your demeanor might be a good approach, but could be ultimately unrewarding. Fitting types of work to your personality can also lead you away from enriching challenges. As we are often brought to achieve our best through adversity, so having work that is too comfortable or easy can lead to complacence and lack of professional growth.

So, let's look at your strengths and weaknesses, but not forget the shaping influence of "office politics" which will expose these strengths and weaknesses.

Recognize Your Weaknesses

As part of your self-assessment, you'll probably start with what you do not want to do. This could be something such as test automation with a new or different tool, lack of interest in a subject such as new testing methodologies or test strategies driven by technologies in which you have no interest, or a job that just isn't a good fit personally. Before narrowing

your options, it is important to understand what you know and don't know. Lack of information or knowledge is one of the biggest and most constant hurdles that a QA professional or knowledge worker will face.

Knowledge requirements for today's QA professional have increased dramatically, and consist of knowledge of both technical advancements and the business world. Technical knowledge can be something as simple as information in a specification or testing information in general. Business knowledge consists of knowing the business goals for a project and understanding the customer profile, needs and context. Common sense or 'street smarts' will help, but knowledge is still needed.

A common weakness for QA professionals who have been in the same domain or same job for some time is losing touch with current development and technology trends, especially in areas outside of their current domain. This technical weakness is the first challenge you must overcome. In fact, many QA professionals today recognize this and have taken steps to learn new test tools, take more programming classes and otherwise become more technical. This is no small undertaking, as it is a challenge to keep up with the job, family or outside interests and still have the time, focus and energy to keep up with software development and IT trends. The pace of technological advancement is much faster than it was in decades past, so this is now more of a necessity than in times past.

A second weakness is not having enough knowledge about a given business situation. Just as CEOs are generally unaware of what is happening at a workgroup level (except through statistics and business intelligence), so a QA professional on an IT project might be unaware of the company's objectives. The project partners and principals may themselves be unaware of the real purpose of the project and its place in the corporate IT ecosystem. Closing this knowledge gap is an area of weakness that can sometimes be overcome by diligent study and research.

In addition to technical advancements and business requirements, a third area of weakness that needs to be addressed is 'soft' skills. These skills are a combination of effective interpersonal communications, the ability to read body language and the flexibility to change one's approach based on awareness of whether the current methods of communication are succeeding or not.

These skills can be much more difficult to demonstrate and are often hidden from recruiters or new hiring managers. Fit of an employee or team has become a major consideration in many hi-tech jobs, as soft skills are harder to hone. It is relatively easy to learn a new test methodology or tool in comparison to learning how to communicate and work productively with others, but realizing your weakness in soft skill areas and then working on them is one of the most important things that you can do. Succeeding in improving your soft skills will make you a more versatile contributor and improve your chances in the technical and business workplace.

Play to Your Strengths

While figuring out what you don't want to do may be fairly easy, figuring out what your strengths are and what you want to do is considerably more difficult. We all have a general idea of our strengths, but during your self-assessment, articulating and defining your strengths will help. If you are a hands-on person, then hopefully your work involves actual testing. If you are a hands-off, big picture person, then test plans and analysis may be a suitable focus. If you are a leader who likes to delegate, then hopefully you have a team to lead and people to whom you can delegate tasks.

Unfortunately, finding jobs that leverage off our strengths in the way that we have them "packaged" is close to impossible. Jobs and career opportunities are more blended and may need to leverage simultaneously off our strengths AND weaknesses. Someone with attention to detail may be overly concerned with some specific unmet, but ultimately unimportant criteria of a project's overall design and usability goals. Strict adherence to exit criteria is considered a good trait in a QA person, but in a larger view this may represent a potential lack of flexibility.

For example, it may be impossible to close all of the outstanding issues on a project. There may be longstanding issues and constant releases that may make product stability elusive. There may be faults in the original specification or some technology limitation may make a completely successful product release unlikely. In many cases you must be prepared to simply declare conditional victory and be satisfied that you did your best. This is an area in which many dedicated QA professionals have problems in part because they are so dedicated and detailed-oriented. The customer is not well-served by an unyielding attitude towards quality by the QA professional.

In another example, if you are well-versed in automated testing, you might believe it to be the best solution for nearly every testing or QA work. This may make it more difficult for you to analyze correctly a situation in which test automation is not the best solution. You may also find yourself in a position in which automation is a good, viable solution, but for some reason it will not be used. Acting as a naysayer because your preferred solution was not used is not going to do you or anyone else any good. What might be considered a strength in one's professional methods can become a weakness if its application overrides consideration of other methods or approaches.

Heads Up!

Making rapid progress in a new job (or staying relevant in your current one) can be greatly helped by being observant and perceptive. Identifying principals who might be more (or less) supportive of your efforts can be a good survival skill. Large corporations can be battlegrounds between rival teams of consultancies, local management, contractors and company employees. Being unaware of the players and their agendas can be dangerous if you hope to remain effective on a project or in a department.

There is a temptation that when things are in flux to keep your head down and try to be inconspicuous. However, it is at times like these that having knowledge of what is going on may be critical for success or survival. You do not have to act like a spy or be a gossip, but it is important to be aware of any managerial redirection or political changes. This will allow you to chart a course of action in potentially hazardous waters.

Everyone reacts to unknowns and change differently, but being adaptable will help. The ambitious tester may need to rein himself in a little and not do the testing of five other team members, and the less ambitious test lead should perhaps aspire to do more and turn that test plan around more quickly. If you are quiet or passive, it may be important for you to speak up in a meeting in which you have concerns that have not been addressed even if you would rather not. At other times it is more important to smoothly blend ambition and deference to achieve a more balanced approach to work.

Adaptability in your response to change and potential unknowns will be greatly aided by how observant and perceptive you are. Use these important human qualities to help pick a path that challenges you, but also one that fits with your strengths and weaknesses. Using skill and personality profiling such as Meyers-Briggs (www.advisorteam.com, www.teamtechnology.co.uk/tt/t-articl/mb-simpl.htm) is a good way to understand yourself and what types of communication and collaboration best suit you, as well as areas that could be improved.

What Do You Want?

Many QA professionals work in jobs that they may only be marginally happy with, but may be loathe to leave due to concerns that they will not like a new job or they just fear the unknown. There is also the comfort of the predictable rut, but over time, unfulfilling jobs, no matter how comfortable, become barriers to career and personal growth.

So, what is your next step? What do you want to do? Unfortunately, HR departments are best at hiring individuals for what the candidate can already do, not what the individual would like to do or has the potential to do. If you are tired of the type of work you have been doing, it may require more effort on your part to convince an employer to take you on in some capacity that is new to you. This is where outside training and perseverance will help.

Outside Training

There have been scores of books, articles and seminars regarding career directions and opportunities, so this area will be mentioned only briefly in this paper. Suffice it to say that there are plenty of assessment tests and career counselors who are willing to apply their methods and theories to your career plans. For self-directed assessments, check http://www.quintcareers.com/career_assessment.html, http://www.rileyguide.com/assess.html, and http://www.hotjobs.com/htdocs/tools/tests/Career_Assessment_Tests__20021112-141-us.html.

However, there is a lot to be said about researching different fields and career paths for yourself in a more hands-on manner so you can get a feel for the subject. Getting hands-on exposure to an unfamiliar technology or process is very important, particularly if the job itself will be hands-on. However, sometimes professional career assessment and counseling services are warranted.

Learning more about software development methods, techniques and processes will help anyone with a job that is associated with software development. Learning a language such as java can give a non-programmer great insight into the design, coding and debugging of an application. Generally you either remain a generalist and rise through the hierarchy, or specialize and eventually become an Architect or some other sort of “wizard.”

Perseverance

Once a new course of action has been chosen it is important to realize that a great deal of dedication and patience may be required. If you have entered a new field, you may have to take a junior position at lower pay, take an internship to 'break in', or work graveyard shift. If you have gone back to school to get a degree or certification you will not only need to have the discipline for hundreds of hours of homework, study and lectures, but you will also have to learn, or relearn the importance of timely delivery of assignments and projects, and to give the teacher (customer) what they want.

Perseverance is also important in a job search. Your ideal job could be advertised months from now, but you will never see it if you have already given up your search. Many people become desperate and 'settle' for some job that is not want they really wanted to do. In order to persevere when pursuing a new career, it is usually necessary to have a strong belief or goal; otherwise, you will just settle for whatever work comes your way. This generally does not make for a satisfying career, although some people manage to 'fall' into good, fulfilling positions.

Jumpstarting Change

The pain of change can be eased through learning, although learning to learn again can be challenging. By proactively embracing learning through reading, classes, Internet research, peer support and more, you open up new opportunities, new possibilities and new directions. You will go down paths that you otherwise might never have imagined. In this manner you can turn an uncomfortable situation into a more pleasant journey.

Through the self-assessment that we discussed earlier in this paper as well as your keen observation of technological and business changes, your learning options will be shaped. For example, a tester today might need to do basic program maintenance or might need to read/understand the code because technical documentation wasn’t available. Conversely a developer might need to test because the QA department was laid off. Since testing efficiencies are sought through the use of automated tools, knowledge of automated testing and

its role is also important. As a result there are higher expectations for the QA professional. He or she must now have more technical knowledge and skills in programming and test automation.

So what are the key focus areas for today's QA professional? Let's look at the top three areas: technical skills, business research and globalization.

Technical Skills Worth Developing Today for the QA Professional

Based on a recent survey of QA professionals in Silicon Valley, technical skills were the most important to develop. One respondent put it well by indicating that QA professionals today must have a deep understanding of a developer's skills and process even if they don't develop code in their job. Additional technical knowledge helps give a view of testing from the "other" side and should really open a QA professional's eyes. However, it just isn't easy to stay current. In this instance, focusing on "best of" ideas would help.

Deciding what tools and processes to learn about can be tricky if you are just looking for any QA job that is available. You could study Oracle and find that you needed to know MS-SQL, or study Peoplebase and find a job that requires SAP. Fortunately, learning a single application in a class of applications should result in knowledge which can be applied to quickly come up to speed on others in that class.

Project management and reporting tools are perhaps the most ubiquitous tools for nearly any development project. Good knowledge of this application area combined with communications skills will rapidly allow you to get up to speed on almost any new assignment and help position you as a test lead or manager when looking for a new position.

Once you have identified an area of interest, it is then necessary to discover and understand the underlying processes, tools and methodology that are used in that field. By understanding this, you can best ascertain the quality issues inherent in that particular field and plan accordingly. Development methodologies and the form of the final product and its deployment have a great deal of effect on the type of QA and the testing that will be required.

Consider the suggested areas of study of general QA and software development methodologies and tools that were listed in the recent survey*:*

- Test automation from Mercury Interactive www.mercury.com, Segue www.segue.com, Rational http://www-306.ibm.com/software/rational/
- New technical and topical developments in testing, such as agile testing www.testing.com/agile and open source www.sourceforge.net
- Programming languages or environments, such as Java http://java.sun.com, www.javaworld.com, www.java.net; Perl www.perl.com, www.perl.org; c# http://msdn.Microsoft.com/vcsharp, www.c-sharpcorner.com; .net www.asp.net

From the recent survey most other suggestions fell into technological domains. Depending on the field you want to work in will determine the technical skills worth developing. While many of the examples prepare one for a development position, keep in mind that for the QA professional, this knowledge could be applied to the testing job. For example, with respect to scripting, it would be useful to not only know the language for casual

purposes, but to be able to construct libraries and more sophisticated tests that can only be done by scripts.

Consider the suggested areas of study by technical domains that were listed in the recent survey:

- Emergent environments, such as Linux www.linux.org, www.linux.com
- Databases including query languages such as SQL www.w3schools.com/sql/default.asp
- Security – application security www.securityfocus.com, www.esecurityplanet.com, www.cgisecurity.com and secure computing environments http://www.sun.com/software/whitepapers, www.hp.com/security, www.ibm.com/security, www.dell.com/security
- Networking http://www.comptechdoc.org/independent/networking, understanding IP routing (BGP, PSPF, ISIS), VPN technologies, MPLS, LDP, RSVP/TE, etc.
- Web technologies http://www.comptechdoc.org/independent/web/

Several individuals also cited more general knowledge, such as:

- Six Sigma practices www.isixsigma.com
- General development methodologies http://www.sysprog.net/index.html, like object oriented programming and design www.oopsla.org/2004/ShowPage.do?id=Home, www.ootips.org; patterns http://www.microsoft.com/resources/practices/default.mspx; UML www.uml.org

Consider the following sources to learn more about QA-related matters:

- www.soft.com/Institute/HotList/index.html. The oldest and most comprehensive resource site for professional software testers, hosted by Software Research, Inc. (San Francisco, California)
- www.stickyminds.com. "Better Software" online magazine and resource site for professional software testers, hosted by SQE (Orange Park, Florida)
- www.qaforums.com. Resources for professional software testers, hosted by BetaSoft. (Campbell, California)
- www.testinghotlist.com. Resources for professional software testers, hosted by Bret Pettichord (Austin, Texas).
- www.qalabs.com/resources/index.htm. Opt-in monthly newsletter on various software testing topics with past newsletters posted and easily accessible on website.
- http://www.asq.org/pub/qualityprogress/. "Quality Progress" magazine (from ASQ)
- http://www.computer.org/computer/about.htm. "Computer" magazine (from IEEE Computer Society)
- Local QA organizations that may or may not be affiliated with national organizations: SSQA (affiliated with ASQ) http://www.ventanatech.com/ssqa, Seattle Area Software Quality Assurance Group http://www.sasqag.org, Society for Software Quality http://www.ssq.org
- National quality organizations: PNSQC www.pnsqc.org, ASQ and their local chapters www.asq.org, www.asq-silicon-valley.org, IEEE www.ieee.org, ACM www.acm.org
- SQA authors: Cem Kaner www.kaner.com, Brian Marick www.testing.com, James Bach www.satisfice.com to name a few. Search on "software testing authors" for more.

Consider the following sources to help increase your technical depth:

- C|Net Tech News First www.news.com
- News for Nerds. Stuff that Matters. www.slashdot.org
- Open source site: www.sourceforge.net
- Developer sites: www.webmonkey.com, www.msdn.com
- RedHerring www.redherring.com

- IT world http://www.itworld.com
- eWeek http://www.eweek.com
- Addison Wesley technical series www.awprofessional.com
- Steve McConnell, author of http://cc2e.com/, www.stevemcconnell.com
- Local non-profit forums in Silicon Valley: SD Technology Forums www.sdforum.org
- Local newspapers: tech, business, science sections, although these are often simplified representations of previously reported subjects.

Forewarned Is Forearmed - Research the Business World

In the recent survey, the majority of respondents identified a need to develop more technical skills to augment their software testing and QA experience. Most of this advice is in response to job requirements that are posted today. For future jobs, there is another area worth exploring, developing your business knowledge. This is a critical area that many engineers, testers and competent technical people in the software development industry fail to appreciate or understand.

Understanding project business goals and being able to relate project technical issues and progress to management is expected by the customer or employer. Large consultancies often prosper in IT and development despite being spotty at the hands-on part of their business due to business-savvy managers and a sales force who will tell the customer (usually a manager) what they want to hear in a manner that they can understand.

Be conscious of the business environment you are in and your project's role in that environment. If you have a clear vision of this, then you are better able to serve your employer, as well as yourself. Always try to keep in mind the business objective of the QA task you are performing.

If you are not currently in a business environment, it will be necessary to dig deeper to get useful information as to IT and QA practices, issues and trends as they pertain to that business. It is necessary to go beyond the superficial mainstream business and hi-tech press to get useful information regarding IT trends and practices.

The IT industry in general and Fortune 500-type corporations in particular are major drivers on the software QA profession due to the sheer cost and scale of their projects and initiatives. This doesn't discount that small companies are technological innovators, but in terms of employing large numbers of professionals, large corporations are it. The collective actions of these companies and other players must be considered in terms of predicting the future of the IT job economy and the type of IT jobs that will be available.

Here are some good business-domain related links:

- Biting the hands that feeds IT www.theregister.co.uk
- Harvard Business Review Online http://harvardbusinessonline.hbsp.harvard.edu/b02/en/hbr/hbr_home.jhtml
- Business Week www.businessweekonline.com
- Wall Street Journal www.wsj.com
- Economist (more in-depth than the newspapers. The Economist is especially insightful.) www.economist.com
- Local for profit and non-profit forums: San Jose Business Journal Forums www.bizjournals.com/sanjose
- CIO insight http://www.cioinsight.com

It's a Global Village and You're In It!

Today many projects have a global component or are fully global. This adds an extra burden to synchronizing teams, bridging cross-cultural communication gaps and meeting deadlines. In fact, with good reason, many California colleges require some form of intercultural communication classwork in order receive an academic degree. As part of the Pacific Rim, the US west coast has a very highly mixed cultural environment. Distributed teams and international corporate operations add to that diversity and add to the project management challenges of managing scope, budget and time.

While enhancing one's skill set with more or deeper technical skills, the skills associated with effective cross-cultural communications and global staff management should not be overlooked. When describing complex business and software processes accurate but straightforward technical language is required. It is a rare person who can understand something complex and then be skilled enough to communicate to someone who is non-technical or unfamiliar with the situation (or the language). Good written or verbal communication is often scarce in the workplace.

Written communication often suffers in informal environments, but it is written communication that is often the only tangible evidence of what was actually noted, proposed and agreed to and is "de rigeur" when working on a global team. Global staff management presents its own unique challenges. Keeping a project moving forward with disparate members separated by geography, time, language, customs, and sometimes business practices, who you may never meet and seldom speak with is quite a task.

Generally, globalization and project management training tailored for the software QA professional is minimally available outside of the workplace (with the exception of workshops such as those offered through Software SETT) and requires resourcefulness to acquire, if on-the-job experience is unavailable.

Here are some useful links that address globalization:

- www.outsourcing-offshore.com/. Interesting source for offshoring but you have to "sign-in" in order to get to the white papers.
- www.executiveplanet.com/. Great site on doing business with over 60 countries covering topics like gift giving, business dress, appointments, negotiations and more. The offshore destinations such as India, China, Philippines and Russia are covered.
- www.cia.gov/cia/publications/factbook/. Good site for facts on over 250 countries. Facts include information on geography, people, government, economy, communications, transportation, military and transnational issues.
- www.languageinindia.com/junjul2002/baldridgeindianenglish.html. Fabulous site on the history of India through the linguistic transformation from Arabic, Persian, Hindu to English via the British East India Company.
- www.rttsweb.com/services/outsourcing/info_center.cfm. A collection point of articles on offshore outsourcing. But you have to "sign-in" to get the white papers. Other articles are free. Pull from Business Week Online, C|Net, and other publications for their articles as well as posting their own.
- www.softsett.com/InClass.jsp?pageN=3&tabN=4. A collection of links to offshore outsourcing information.

Essential Sources for Lifelong Learning

Based on a wide consensus of QA professionals in Silicon Valley, sources for learning fall into the following major categories:

- Internet
- Classes (formal instruction, in-classroom, some online training)
- Peer learning (informal and formal through networking opportunities)
- Communities of practice

The most important feature of a resource was identified in our survey as accessibility. This correlates well to the Internet which provides current information and immediate accessibility at work, from home or while on travel. Other important features were cited:

- human interaction (networking and personal guidance) which supports well peer learning and classes
- flexibility as evidenced through powerful search engines and convenience in time of getting to the information

Internet

Not surprising the most often cited resource for learning in our recent survey was the Internet. This is by far the primary source of information that QA professionals go to in order to keep current and/or gain new knowledge. Respondents had the following to say about the Internet:

~ "My favorite! Someone is bound to have something on anything some place."
~ "This has become a huge help. I'm now able to find info on almost any topic."
~ "I use the Internet a lot to get technical information relevant to projects on which I am working."
~ "I use the Internet to get information on company technology platforms, and related information on telecommunications, security and standards."
~ "I can get current technical information; it is up to date."
~ "Internet resources are good, due to ease of access and search capabilities."

Use of the Internet as cited by survey respondents was quite creative. For example, some read white papers from IT information sites, participated in RFCs (Request for Comments), read published papers on testing or even used sample code that was posted online. If you are currently working, it is possible that your company may have purchased access to white papers from which you could gain knowledge. It may just be a question of asking the company IT analysts, managers, or the corporate librarian.

Classes

Another popular source for learning is classes. In our recent survey respondents agreed that classes were a good method for learning, in addition to reading books and doing Internet research. Classes tied with Peer Learning as the second most popular method for staying relevant. A few respondents noted that classes, while good in theory have some negatives: they cost too much when a student is on a tight budget, there is no company benefit to help pay, and classes take too long when an

immediate return is needed (i.e. specific skill set is needed on the job resume.)

For some, classes provide a needed spark in terms of requirements and delivery dates such that they actually complete the work and gain the intended knowledge. It takes a very well-directed and motivated individual to learn as much on their own as they would from a class.

In Silicon Valley the following schools are available for addressing technical and business domain knowledge:

- University of California extensions www.ucsc-extension.edu, http://www.unex.berkeley.edu/
- University of Washing extension www.extension.washington.edu/ext/
- Local California, Bay area community colleges www.foothill.edu/index.shtml, www.deanza.edu
- Silicon Valley community-based adult education www.scae.org
- National, professional organization courses www.pnsqc.org, www.ASQ.org, www.ieee.org, www.acm.org
- Online training/self-study offered to the public www.capella.edu
- Online training offered by employer

Peer Learning

A third resource for learning is from one's peers. If you are exposed to people who are doing work that you are interested in pursuing, you can learn by discussing issues with colleagues, listening to their technical conversations, or just being a 'fly on the wall'. It helps to be observant and do further research at home and then wait until some basic knowledge or understanding has been gained before speaking up. Networking via professional groups, listservs and online communities is another important way to keep in touch with what is current in QA.

Community of Practice

A fourth resource for learning is joining a community of practice where you are exposed to general knowledge of techniques and issues and where you can network and meet new colleagues. A hidden benefit of joining an organization and becoming an active member is the morale boost one gets. Talking with colleagues and sharing experiences generally helps when looking for new opportunities, trying to find solutions to thorny problems or just meeting like-minded individuals.

In the recent survey this resource was considered a mixed blessing for many because participating in special interest groups invariably takes time out of one's personal life or just isn't considered valuable. Those who do attend and participate cited the primary benefit as networking among colleagues. Here are some additional thoughts from our survey in response to the question: "Which of the following have helped you grow professionally?"

~ "Learn from colleagues – I have discussions with practitioners who are bright, creative and articulate; attend meetings and lectures to learn; pump developers."

~ "Learn from vendors – I have them come in and provide a seminar or give a technical presentation on their product. Invite employees to listen and learn. Get hands-on experience with the product. Do likewise for your customers."

[Alternatively, if you are not in a position to have vendors come to you, join their newsgroups so that you can get invited to their local informational events]

~ "Attend workshops though the ones cited are by invitation only (LAWST, AWTA, STMR, WOPR) or start your own."

Don't Forget. Soft Skills are Important!

Earlier in this paper, we mentioned the importance of soft skills. Let's briefly expand on this here. If you are stronger in technical areas than in your soft skills, and the decision is made to improve yourself, the choice will often be towards the already strong set of skills. In the interest of being well-rounded and giving yourself that all elusive "edge", it is better instead to attempt to improve in the area to which you are less adept. This requires some fortitude, as we are often uncomfortable doing the things at which we feel we are not so good.

For example, many otherwise competent QA professionals have difficulties communicating with others - particularly in non-technical areas such as business communications. Giving presentations, speaking up in high-level or large meetings and interacting with non-technical management are all things that do not come naturally to some and are anticipated with dread by many.

Aside from the more obvious traits of being detail-oriented, methodical, and possessing excellent writing skills, we have identified the following additional skills and traits of the successful QA professional:

- Public speaking ability
- Flexibility (and adaptability)
- Providing added value

Public Speaking

To become more comfortable speaking before large, or intimidating groups, practice is required, just as an actor must rehearse his scenes. If there are no suitable family gatherings or impromptu events at which to practice speaking, then Toastmasters International (www.toastmasters.org) speaker's group is an excellent way to gain confidence through public speaking.

Speaking well and confidently is only half of the issue here - knowing your audience and the nature of the occasion is important in that you must also choose the right things to say and determine the level of detail and technicality required to communicate your message. What you say about the internal details of your project to your test team is going to be very different than what you would say to a group of managers. You would use different language with each of these groups even if you were saying the same thing to both.

Flexibility (Adaptability)

With the recent downturn in IT, some QA professionals have had to consider not only beefing up their technical expertise and business knowledge, but perhaps to consider related technical fields such as support, analyst or project management positions. Some have even left high tech completely, but for those wishing to remain, there are other options. In fact, in this paper we have suggested three important directions a QA professional

should consider: adding technical knowledge and skills, increasing your business know-how and understanding globalization. A QA professional should not only consider other "traditional" careers in IT, such as in software development, technical writing, technical support, IT support help desk, education/training, and business analysis, but perhaps a blend of several of these areas.

Consider adding business domain knowledge to your strong QA skills so that you can become a subject matter expert (SME) tester. Or, what about having technical skills as strong as the developer so you can participate more fully on the team during design and troubleshooting. In this case flexibility is key and allows you to make connections in ways never thought of before.

If you are unable or unwilling to change your job, you may be able to grow in your present position. Try to learn and understand the process in which you are involved. After learning to solve process problems at your level, you may be able to become a strategic contributor at a higher level. You will then have greater impact on the successful pursuit of the quality goals of the organization. If you cannot grow, then you should consider an exit strategy. If you decide to remain at a job, you should always be able to find something that you can do to improve the way you do your job.

If you are unemployed, develop your skills by volunteering for a non-profit organization or get involved with professional groups. This will help keep your skills fresh. An excellent volunteer option is working with Open Source projects as a tester, writing documentation or even writing code if you have the coding ability. Finding the right Open Source projects to become involved with can give you the chance to define the nature of your contribution such that it grows your skills in some new desired direction.

As with many other life transitions, searching for a new job provides us with a chance to evaluate where we are and where we are headed. In order to succeed, especially in a changing landscape, sometimes you must step out of your comfort zone and challenge yourself to gain the knowledge that will put you in the running for the new types of jobs. This may involve going back to a community college and taking an introductory class in some emerging technology (java, database, web applications, wireless, embedded devices, etc.).

Provide Added Value

A big part of providing added value is going beyond the task that has been assigned to you and being able to appreciate what your lead or manager needs and wants. Therefore it is important to seek out needs and provide solutions. This requires confidence to go beyond what is expected. When you do this, unexpected opportunities may be exposed. If you can help a colleague without a lot of fanfare and share tips, techniques, short-cuts and ideas with others, your contributions will be appreciated. Learning to work collaboratively with others and to help build teams will take you far.

And, Remember …

Sometimes opportunities come disguised as problems. It is up to you to be open and willing to explore and try to find the best solution to any problem with which you are confronted.

Many people become focused on a single 'perfect' or ideal solution, but it is important to be able to entertain multiple solutions in order to account for different contingencies. Being flexible allows you to see these various solutions. The other soft skills of adding value (seeking out problems to which you can provide a solution without being asked), perseverance, and communication skills (especially public speaking) will also help.

Conclusion

As with much of life, it helps to have a realistic plan in order to keep one's skills relevant. Knowing whether your plan is realistic is the hard part. This is where career counseling, industry knowledge and peer groups are helpful, as well as being open to new ideas and changes in direction. Opportunity is sometimes a matter of chance and we are not always presented with the opportunities for which we have prepared ourselves. Therefore, we must do our best and enjoy the journey of self-discovery.

Although IT spending is on the rise again, the landscape is quite different than it was in the dot-com era and the DP era before it. With organizations trying to keep costs down through integration, consolidation and outsourcing as well as things like utility and grid computing and web-based services, new skills and knowledge are required. New job categories are being created daily in the world of IT. It is up to you to figure out your role in it.

Once proper research has been done, the obvious step towards keeping technical skills and knowledge current or forward-looking is to gain more detailed knowledge in those areas through whatever method is deemed the best. Knowledge of the international business community and its practices and direction is also crucial in order to be forwardly positioned as the business technical landscape continues to evolve and change. In tandem with this, the distributed, multicultural workplace brings its own set of challenges.

Finally, communications ability and sensitivity to the 'soft' skills can be an important differentiator between you and the other qualified technical people with whom you are competing. Assuming you develop technical depth, business breadth (local and international) and better soft skills, you will enhance your QA career and give employers incentive to invest in you. You, in turn, will assure them of a solid ROI (return on investment) in you.

Controlled Agility

Controlling the Scope of Agile Projects With Change Management Best Practices

Dr. Geoffrey J. Hewson
Software Productivity Center Inc.
geoff.hewson@spc.ca

Initially published at the Pacific Northwest Software Quality Conference, 2004

Abstract

Like many other developers of commercial software, the Software Productivity Center's (SPC's) product development team is challenged to provide enough flexibility in its development process to accommodate shifting requirements due to market pressures. At the same time it is expected to maintain enough control of its projects to ensure the regular on-time delivery of product releases. The difficulty is to satisfy these conflicting needs and still deliver a meaningful feature set that satisfies both internal and customer business requirements. While most agile methodologies deal very well with shifting requirements, they have a tendency to be somewhat open ended with respect to predictable delivery dates. This is fine if you are engaged in exploratory development, or your software is deployed in environments where it is cheap and easy for you to make frequent changes to the code as you refine your product's functionality. Unfortunately, for most developers of commercial software, this is not the case. The open-ended nature of these agile methods presents problems to management when you try to constrain development within predictable delivery schedules.

This paper presents an experience report explaining how SPC's approach married an agile-like development process with change management best practices, allowing us to accommodate an evolving feature set during development, and to consistently deliver regular on-time updates to our products.

About the Author

Dr. Hewson is Chief Knowledge Officer and Principal Consultant at Software Productivity Center where he is responsible for articulating and delivering SPC's vision for integrated software process and application lifecycle management solutions. Geoff's 16+ years of IT industry experience include senior positions with Microsoft, Canada's Department of Justice, and Ford Motor Company. In his career, Geoff has applied his expert knowledge to assist 40+ organizations improve their ability to successfully deliver timely and quality software through business focused solutions addressing diverse problem areas. He has also contributed to a number of industry methodologies, including the Rational Unified Process and the Microsoft Solutions Framework, and Geoff was the chief architect of SPC's SD Productivity Award winning methodware tools. Dr. Hewson holds a Ph.D. in Physical Chemistry from the University of Sheffield, UK.

Introduction

This paper presents an experience report describing the Software Productivity Center's (SPC's) approach to product development used in the development of its ESTIMATE Professional ™ product line. SPC's approach is designed to address the challenge of providing enough flexibility in its development process to accommodate shifting requirements due to market pressures, yet at the same time maintaining enough control of its projects to ensure the regular delivery of commercial-grade software product releases within fixed delivery date targets.

SPC's approach exhibits many characteristics in common with the recently emerging "Agile Methods" for software development [1], but it also uses formal change management and scope control practices to ensure that it is the most relevant set of features that are delivered in each release, time-boxed to fit within the budget and schedule constraints.

Agile Methods and Commercial Development

The agile methods for software development are becoming increasing popular these days. While there are many success stories describing their application in various types of software development project, there are also many stories of failure too.

Typically the agile methods thrive in situations where requirements are uncertain or volatile, developers are motivated and responsible, and the customer understands and is willing to be actively involved in the project [1]. However, the leading proponents of agile methods are agreed that, in their pure form, agile methods do not work well in all situations.

Mostly, the risks and shortcomings of agile methods arise from mismatches in corporate culture, team size, and the (lack of) experience of team members. [2, 3]. Of particular relevance to commercial software developers (like SPC) is the dependence of agile methods on the ability to rapidly deploy new, updated, versions of the software to get quick feedback from customers on how well the software meets their needs [3]. This is fine if you are engaged in an exploratory in-house development, or if your software is deployed in environments where it is cheap and easy for you to make frequent changes to the code as you refine your product's functionality (e.g. corporate IT, ASP).

Unfortunately, for most developers of commercial software this is not feasible. Releasing new product versions to the public can incur significant costs for updated packaging, documentation, distribution media etc. There are also likely to be increased customer support costs due to the need to support many different product versions in the field.

Also, corporate business objectives often drive the need to be able to reliably deliver a meaningful product (i.e. the right feature set) within a predetermined product release schedule. Usually there are marketing activities and revenue expectations that depend on the availability of product updates tied to a predetermined product release schedule.

The Need For Control

Despite these mismatches, developers of commercial software can (and do) still benefit from many aspects of the agile methods. These methods allow commercial development teams to clarify their understanding of requirements as the project proceeds and evolve feature sets in response to changes in the market. But, for development teams to maintain focus on their need to provide a well coordinated feature set in time for scheduled public product releases, more control over the planned feature set is necessary than is usually prescribed for the exploratory style of development that the experts consider to be most appropriate for agile methods [4].

Background

About ESTIMATE Professional ™

The Software Productivity Center (SPC), based in Vancouver, British Columbia, works with software development organizations worldwide to help them more effectively meet their business objectives through the optimization of their software development processes and practices, aligning them with overall corporate business objectives. SPC's solutions typically combine consulting, training, and enabling tools to provide a complete solution addressing individual software development practices, or complete organizational process.

SPC initially introduced ESTIMATE Professional into its portfolio of software process improvement tools in 1998. ESTIMATE Professional is a fairly typical Windows-based desktop productivity tool that is used by project managers during early stage software project planning and estimation to:

- Prepare estimates for new software projects by using the COCOMO 2.0 and Putnam calculation models
- Evaluate the potential tradeoffs between different manpower, schedule, or cost options
- Quantify project risk using Monte Carlo simulation to determine the statistical range of likely project outcomes
- Time-box project scope to fit schedule, effort, or cost constraints
- Provide comprehensive reports

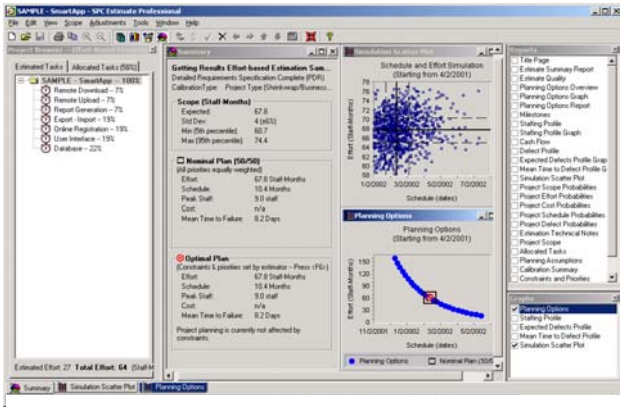


Figure 1 – SPC's ESTIMATE Professional

- Increase estimate accuracy through calibration based on your own historical project data

ESTIMATE Professional was written in Microsoft Visual Basic and Visual C++. It is designed primarily for standalone use, though it has the capability to share its database via a file server. Over the two-year period discussed in this paper, SPC released 4 versions of ESTIMATE Professional (2.0.x quality release, 3.0, 4.0, 5.0).

About the ESTIMATE Professional Development Team

The ESTIMATE Professional development team was very small (3-4 staff) but highly experienced (averaging more than 15 years in the industry). All the traditional team roles were represented: Project Management, Development, Testing, Product Management/Marketing (representing the customer and the business). Because of the size of the team, the formal unit and system testing burden was shared between the Development and Product Management roles, with additional beta testing by Marketing.

SPC's Development Strategy

Beginning from a nucleus of source code acquired from a previously available freeware tool, SPC's development team focused on incrementally adding additional value-added features into the product and correcting customer-reported defects. The focus of the development strategy was to identify required changes to the product's feature set and implement these changes on a feature-by-feature basis, packaged into a series of projects each of which delivered a commercial product release. Major version upgrades were released approximately twice a year. Additional "quality releases", which corrected severe product defects, were made available for download from the SPC web site periodically as necessary.

Business Goals Influence Development Strategy Principles

There were several over-arching corporate business goals that had a direct influence on the development strategy for ESTIMATE Professional.

As a new venture for SPC, it was important to SPC management that we deliver an initial commercial release of the product quickly to test the market needs. Then, in relatively quick succession, we planned to follow up with additional value added releases based on customer feedback. Also, the company wanted to be sure that product releases would be delivered on time to support corporate revenue growth targets that were directly tied to the planned ESTIMATE Professional releases. Lastly, as a small company, it was also important to SPC that the development and support costs for the product be minimized to provide the best return on investment.

These business goals meant that SPC's development strategy needed to support the following principles:

- Emphasize delivering acceptable business value within the time-to-market window as the top priority
- Be highly responsive to changes in customer needs
- Ensure that requested changes to the product are not misplaced or overlooked
- Make the most efficient use of the development teams efforts
- Deliver a stable, high quality product

The flexible nature of agile software projects means that project control takes on a different meaning than the "conformance-to-plan" thinking of traditional plan-driven approaches. The traditional approach is fine for environments where customer needs are relatively stable, but where they are volatile project control requires a different way of thinking.

For agile projects, the traditional project management scope-cost-schedule triangle still applies. However, cost and schedule need to be considered as constraints rather than targets [5] and scope is continuously adjusted within those constraints. In our case, our business goals required schedule to be a fixed constraint, so we had to be willing to adjust both scope and effort to ensure on-time delivery.

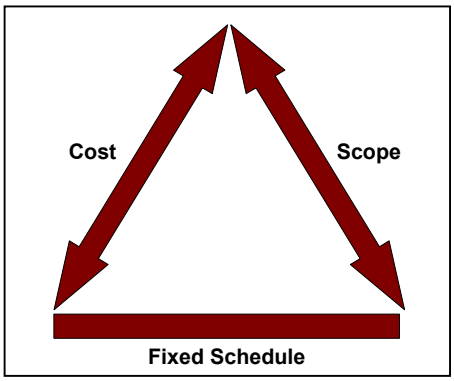


Figure 2 - The project management triangle – SPC worked with fixed schedule

Our primary goal from the standpoint of agile control was to steer the project so that the product delivered best meets the customer and SPC business needs. This meant delivering the most commercially viable product that could be made available on time, even if the final feature set differed substantially from that originally planned. It was also our philosophy that agile control is not how to control/prevent changes, but rather how to embrace change in such a way that we were still able to deliver a finished, functional product release when we wanted/needed it.

Planning for Change

Change management practices were fundamental to the success of SPC's ESTIMATE Professional development projects. We planned our projects assuming that the desired content for each product release was certain to change as the project proceeded. In such a fluid environment it was very important for us to keep track of all the various requests for new features, feature enhancements, bug fixes etc. Particularly, we didn't want anything to be lost when prospective product changes are added or removed from the planned content for each release. To do this we considered everything as some form of a change,

and we captured and managed all the changes as Change Requests (CRs) in our Change Management System. We classified the changes as new features, feature enhancements, or bug fixes, and we used a selected subset of the CRs recorded in our change management database as the basis for defining the technical scope of our projects.

Typically, the SPC development projects were organized into a series of short iterations (usually between 2 and 4 weeks in duration) each ending at a milestone. To set an overall decision-making and scope control framework we decided on an overall theme for each project. This would guide us in prioritizing and selecting appropriate CRs to be included in the project plan. Later on, as the project proceeded, we would also use this to guide us in deciding whether newly arising CRs should be considered for inclusion in the current project, or whether they should be deferred for consideration in a future product release (see Figure 3 below).

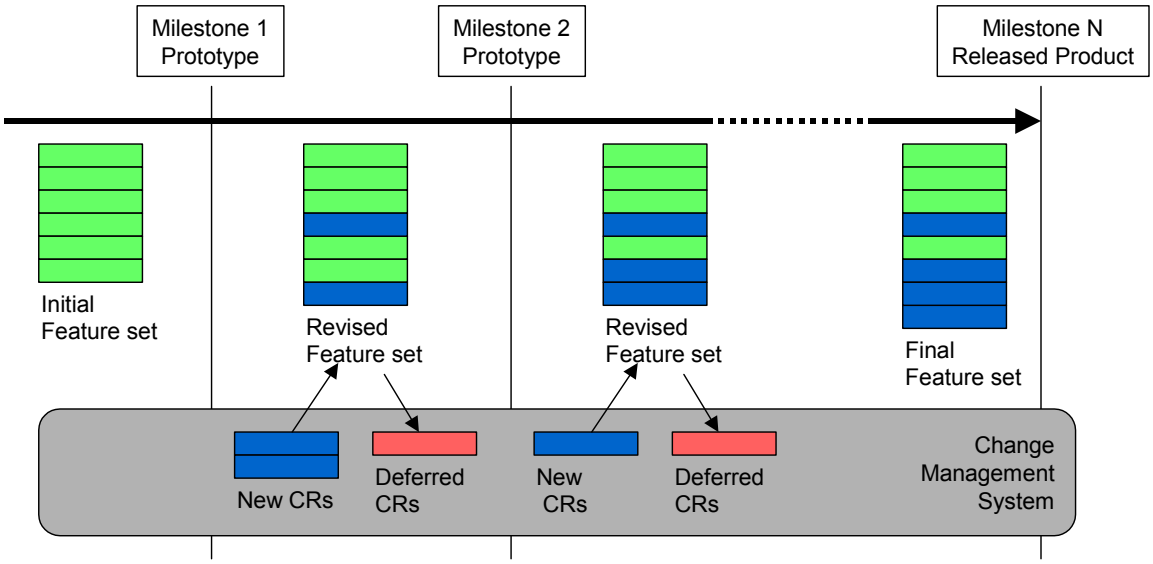


Figure 3 - Feature set evolves during development via change management

During the project planning process, the implementation effort for the highest priority CRs would be estimated, and CRs would be selected for inclusion in the project, working down the priority list until development capacity was reached. CRs would then be explicitly assigned to each of the milestones.

In addition to the effort associated with the implementation of the selected CRs, we allocated additional effort to the project for process overhead (project management, configuration management, code reviews, builds, system testing, unplanned work, etc) as percentages of the planned work. This process overhead was distributed across the milestones in proportion to the quantity of planned CR effort assigned to each milestone. The percentages for each component of the allocated tasks were derived from historical data from previous SPC projects. See Table 1 for an example from the development of ESTIMATE Professional version 4.0.

CR Id	Description	Effort (hours) Minimum	Expected	Maximum
Milestone 1 - Quality-Related defects; Default Specification				
CR0429	Warning message incorrect in "Constraints and Priorities"	0.5	0.5	0.5
CR0432	"Set Defaults" values for Function Points get lost.	5	8	14
CR0148	Default Specification Support	9	12	17
Milestone 2 - User Interface Enhancements				
CR0304	Combine scope type and scope data configuration	20	40	55
CR0413	View Enhancements	30	63	81
Milestone 3 - Project Data Import/Export				
CR0434	Project Data Import/Export	18.5	26.5	35.5
Milestone 4 - Time-based Estimation; Toolbar				
CR0400	Time-based Estimation	68	84	105
CR0363	Toolbar Support	12	15	20
Milestone 5 - Historical Data Import/Export				
CR0123	Historical Data Import/Export	18.5	26.5	35.5
System Testing				
CR0427	Regression Test Plan	8	12	16
	System Testing	73	96	116
	Defect Resolution	58	76	93
	Usability Testing (3 expected)	6	9	12
User Documentation				
	Getting Started Manual Updates	4	8	15
	Getting Started Manual Review	1	2	4
	Brochure Updates	1	2	3
	Brochure Review	1.5	2	2.5
Project Overhead				
	Project Management	35	40	45
	Code Reviews	6	11.5	15
	Configuration Management (CR Management, etc.)	8	10	12
	Release note creation	2	3	4
	Install set creation (1 per milestone + 4 RC builds)	6	7	9
	Interim Status Reports (1 per milestone = 5)	5	7	9
	Unplanned Enhancements	17	23	28
	Project Postmortem	16	18	20
	Total estimated effort (hours)	**429**	**602**	**767**

Table 1 - Milestone CR Assignments and Overhead Allocations (ESTIMATE Professional 4.0)

Controlling the Agility

To fulfill our desire to accommodate changing customer/business requirements, our scope management strategy provided a mechanism that allowed us to incorporate high priority changes as they came up, but also allowed us to remain focused on the fixed delivery date. Sometimes this meant removing some originally planned features to accommodate new higher priority ones. Sometimes we also swapped out some more

effort intensive features to accommodate more easily implemented ones. In all cases, the objective was to deliver the best set of features in the time available so that the product release was a worthwhile upgrade.

Each project milestone provided a critical review and recalibration point that allowed us to:

- Evaluate the status of the CRs planned to be included in the prototype delivered at the milestone
- Reprioritize the planned and yet-to-be-completed CRs in relation to any newly arising CRs
- Re-evaluate the planned list of CRs to be included in the project, adjusting the planned content based on the new priorities by adding, removing, or replacing CRs as appropriate
- Adjusting the assignment of the updated CR list to subsequent milestones
- Committing to the CRs to be implemented in the next milestone

At each milestone, the decision making criteria for determining which CRs should be included in the rest of the project hinged on the overall development goal of delivering the maximum business value achievable within the project constraints. In other words, was this the best list of features to go forward with? Is this what customers are asking for? Will it make customers want to buy/upgrade? Can it be delivered in the available time (SPC value)?

Using these criteria, newly arising CRs would be included where appropriate, and typically 40-75% of the final list of CRs implemented in a released product were not included in the list of initially planned CRs. Table 2 below provides an example of the final planned CR list from the development of ESTIMATE Professional version 4.0, and a detailed listing of the new CRs incorporated into the development that were not part of the original plan (the unplanned enhancements) is provided in Table 3.

By adopting this iterative approach we were able to develop each product release through a series of evolutionary prototypes, allowing us to adjust the planned feature set in response to shifts in market needs/priorities. At the same time, product stability was maintained by delivering a stabilized evolutionary prototype at each milestone. In terms of our overall business goals, this allowed us to satisfy our need to respond to changing market needs, use development resources efficiently (by reducing rework), and maintain a high level of product quality throughout the development effort.

CR Id	Description	Actuals (hours)
Milestone 1 - Quality-Related defects; Default Specification		
CR0429	Warning message incorrect in "Constraints and Priorities"	0.2
CR0432	"Set Defaults" values for Function Points get lost.	0
	Unplanned Enhancements (CRs 0185,0440,0441,450,451,452)	4.4
Milestone 2 - User Interface Enhancements		
CR0304	Combine scope type and scope data configuration	72
CR0413	View Enhancements	116
	Unplanned Enhancements (CRs 0464,0467,0499,0508)	7.7
Milestone 3 - Project Data Import/Export		
CR0434	Project Data Import/Export	41
	Unplanned Enhancements (CRs 0516,0519,0540,0541)	2.3
Milestone 4 - Time-based Estimation; Toolbar		
CR0400	Time-based Estimation	133.5
CR0363	Toolbar Support	16.2
	Unplanned Enhancements (CRs 0553,0554,0561,0566)	8
Milestone 5 - Historical Data Import/Export		
CR0123	Historical Data Import/Export	41
	Unplanned Enhancements (CRs 0583,0587,0618,0639)	3.2
System Testing		
	System Testing	97.2
	Defect Resolution	119.2
User Documentation		
	Getting Started Manual Updates	13.3
	Getting Started Manual Review	3.5
	Brochure Updates	
	Brochure Review	1.2
Project Overhead		
	Project Management	89
	Code Reviews	0
	Configuration Management (CR Management, etc.)	25.1
	Release note creation	3
	Install set creation (1 per milestone + 4 RC builds)	14.5
	Interim Status Reports (1 per milestone = 5)	5
	Project Postmortem	20
	Total effort (hours)	**836.5**

Table 2 - Delivered CRs and Actual Effort (ESTIMATE Professional 4.0)

CR Id	Description	Actuals (hours)
Unplanned Enhancements		
CR0185	Comma acts as home	0.9
CR0440	Rt error 13 using calendar with non-English	0.4
CR0441	"OK" should be default in popup calendar	1.1
CR0450	Font incorrect in Estimate Wizard	0.3
CR0451	Make "overconstrained" text red in Summary	0.3
CR0452	Upgrade to VB6 common controls	1.4
CR0464	Click on edit box should not select text (also CR0463)	7.2
CR0467	<Esc> not functional in Regn. Wizard	0.1
CR0499	Estimate Inputs reports generates divide by zero	0.2
CR0508	Agree mustn't be default in License Agreement	0.2
CR0516	Estimate Wizard only meaningful for whole-project estimates	0.2
CR0519	Estimate Wizard truncates project name at 25 characters	0.7
CR0540	Runtime Error '5' when re-calibrating to Historical	1
CR0541	Historical wizard new project - blank drop down list in "Method by which..."	0.4
CR0553	Prefix "Later, " to Estimate Wizard hints.	0.2
CR0554	Incorrect word use in "Size Estimate" dialog of Wizard	0.1
CR0561	Default to last measurement chosen	1.8
CR0566	Copying Reports too slow	5.9
CR0583	Need hourglass for "New Project"	0.2
CR0587	About box copyright should include 1999.	0.4
CR0618	Run-time error '5' on open with no projects	0.2
CR0639	Negative values appear for std. dev in some reports	2.4
	Total unplanned changes effort (hours):	**25.6**

Table 3 - Unplanned Enhancements incorporated into the ESTIMATE Professional 4.0 project

An Agile Process

Comparing SPC's development principles with those articulated by the Agile Alliance (the group responsible for the Agile Manifesto) [7], we find several common characteristics between our approach and the published agile methodologies:

- Emphasis on delivering overall business value - not just the originally planned feature set – this was key to achieving customer satisfaction
- Development in short increments, delivering working prototypes at regular, short intervals
- Change is embraced, not avoided, allowing flexibility in the final feature set of a release in response to shifting customer and business needs
- Integration of SPC business staff in the project team (Product Marketing, senior management)
- Use of time-boxing strategies to control project scope to ensure on-time project delivery

The concept of "just enough process" is one that is not consistently promoted by agile proponents, yet this was one of the keys to achieving the best development team efficiency in SPC projects. The concept is fairly straightforward, applying enough formal

process overhead to mitigate the significant project and technical risks, but no more. In our optimized process we emphasized the following practices:

- Lightweight project planning – schedule the milestones, but use the CR lists for detailed work planning within milestones
- Detailed requirements and design specifications were prepared for risky or complex features only. Straightforward CRs were implemented directly from the CR details (of course this requires well written CR descriptions)
- Prototypes developed for interim milestones – used to validate implementation vs. requirements, validate estimates, re-evaluate project scope, delivery target etc. We also depended on the prototypes to ensure that we had a potentially shippable product at each milestone. On several occasions we took advantage of this to release "quality releases" that fixed customer reported defects that required fixing urgently
- Well commented code to support light documentation

Our change management system was the key enabling tool for project control, which we used to track the status of the CRs being implemented in each development milestone, and to capture new CRs as they arose. Completion of work was tracked in the change management system, with scheduling done at the milestone level and a predefined CR lifecycle ensured that key process steps were not skipped.

Of the published agile methodologies, our project management approach is probably closest to SCRUM [6]. Each of our development milestones is analogous to a SCRUM sprint. Our change management system is effectively a repository for the various Scrum backlogs – the project backlog in Scrum is equivalent to the planned CR list for an SPC project, while the sprint backlog in Scrum is equivalent to the CR list for each SPC milestone. The complete list of open CRs in our change management system could be considered a product backlog. Looking at it this way, our development strategy is basically Scrum, but with the addition of formalized change management for the integrated management of the various Scrum backlogs.

There are also parallels in our technical development strategy with DSDM [8] and Feature-Driven Development (FDD) [9]. Like DSDM we evolve our product through a series of working prototypes, and rely on a limited amount of formal documentation (high-level plans, functional specifications for the more complex CRs) to mitigate early project risks. Similar to FDD, new functionality is implemented on a CR by CR (or feature by feature) basis.

Results And Observations

Over a two-year period, SPC delivered four major releases of ESTIMATE Professional, each time delivering within 10% of the initially planned development schedule (see Table 4). It is interesting to note that 47-75% of the delivered CRs were not included in the original plan. This high degree of change is attributable to a number of shifts in market priorities and the need to swap certain effort intensive CRs for less onerous ones in order to deliver the most appealing product within the available time-box.

Release	Features Implemented (#CRs)			Effort (hours)		Schedule (weeks)	
	Initial	Unplanned	Deferred	Expected	Actual	Expected	Actual
2.0.x	19	19	0	175	218	13.7	12.6
3	22	12	4	250	543	19.6	18.3
4	9	22	2	602	836	16.1	17.1
5	47	16	32	1005	553	16.9	18.0

Table 4 - Project results for ESTIMATE Professional versions 2.0.x through 5.0

For the first 3 releases (ESTIMATE Professional versions 2.0.x through 4.0), additional CRs were added to the total scope of the project, balanced by commitment of spare development team capacity. In these projects a few CRs were deferred late in the projects to ensure we hit the schedule targets.

For ESTIMATE Professional version 5.0 we faced some difficult decisions. An additional programmer had been added to the development team for the development of version 4.0, and we had planned the feature set for version 5.0 based on this increased development team capacity. Very early in the project this new programmer left the team unexpectedly. At the same time, critical features providing integrations with third party applications were proving to be much more difficult than originally anticipated. If we stuck to the initial plan, it was clear that we would be forced to slip a product release date for the first time. Since this would have jeopardized SPC's ability to achieve corporate revenue growth targets, we reviewed the prioritization of the planned feature set in the light of the new project circumstances. Based on this review, we refocused the project on a much smaller number of the most important CRs, again incorporating several additional low effort CRs to increase the market appeal of the product release. In the end, with these changes, we were again able to deliver the product release close to the original schedule target.

Conclusion

Our experience following the development strategy described in this paper was very positive. The ability to predictably and consistently deliver high quality product releases that had the highest possible market appeal enabled SPC to achieve important product revenue and customer satisfaction goals. In achieving these goals, several critical success factors stand out that made this possible:

Change Management System

Using a change management system as the master repository for all committed and potential product features gave us the confidence that nothing could fall through the cracks. New CRs could be safely captured and set aside for later consideration when CR priorities and project scope was reviewed at each milestone, while deferred CRs wouldn't be forgotten and overlooked for consideration in future product releases. Being able to rely on the change management system in this way allowed the development team to remain highly focused, without distraction, on the currently in-scope content for each milestone.

Team experience and discipline

SPC's development team was highly experienced, averaging greater than 10 years of industry experience. Two factors in particular stand out: the discipline of working on only the CRs committed for each milestone (without gold plating), diligently tracking progress in the change management system, and the experience to know when additional specification details were required and when the basic CR description alone was sufficient.

Management buy-in

SPC management was willing to focus on the delivery of the best possible feature set deliverable at the planned release date rather than fixating on the original plan. Management was highly involved in the CR prioritization and project scoping and re efforts.

Based on our experience, we believe this change-driven development approach to be universally applicable to most forms of software development. Fundamentally, software development is about implementing a series of well-defined "units of work" that represent the changes you wish to make to your product that can be tracked as change requests in a change management system. Depending on the methodology you follow, these changes may implement user stories, use cases, requirements, features, UI screens, reports, bugs, enhancements etc.

Where this approach excels in particular is in projects that incrementally add new or enhanced features to existing software within a long-term versioned release strategy, and where the exact requirements are fluid. This is exactly the model in which developers of commercial software commonly operate. While the concepts of time-boxing combined with the accommodation of evolving requirements are not particularly new, (in fact they

have been addressed by both traditional [10] and agile methods [11]), we found that the CR focused management of project scope can bring some formal rigor to agile projects without adding significant process overhead. This is important because it enables developers of commercial software to reliably deliver value in a fixed delivery date mind-set.

References:

[1] The New Methodology, Martin Fowler, April 2003 (www.martinfowler.com/articles/newMethodology.html)

[2] Methodological Agility, Larry Constantine, Software Development, June 2001

[3] When Shouldn't You Try XP?, Chapter 25, Extreme Programming Explained, Kent Beck, Addison-Wesley, 2000

[4] What Is Agile Software Development? Jim Highsmith, Crosstalk, October 2002

[5] The Principles of Agile Project Management (Part 1), Jim Highsmith, Agile Project Management Email Advisor, 13 May 2004, Cutter Consortium (www.cutter.com)

[6] Agile Software Development with SCRUM, Ken Schwaber and Mike Beedle, Prentice Hall, 2001

[7] Manifesto for Agile Software Development, Agile Alliance (www.agilemanifesto.org)

[8] DSDM is managed by the DSDM Consortium (www.dsdm.org)

[9] A Practical Guide to Feature-Driven Development, Stephen R Palmer, John M Felsing, Prentice Hall, 2002

[10] Software Project Survival Guide, Steve McConnell, Microsoft Press, 1998

[11] Agile Software Development Ecosystems, Jim Highsmith, Addison Wesley, 2002

Becoming a Better Tester
Staying Relevant

By Kelly Whitmill
IBM Printing Systems Division
May 2004

whitmill@us.ibm.com

© IBM Corporation 2004

Author
Kelly Whitmill has 20+ years experience in software testing. Most of that time his role has been that of a team lead with responsibility for finding and implementing effective methods and tools to accomplish the required tests. He is particularly interested in practical approaches that can be effective in environments with limited resources. He has a strong interest in test automation.. He currently works for the IBM Printing Systems Division in Boulder, Colorado.

Acknowledgements
This paper is a result of observing respected testers. Though I have added my own interpretation, I am not the originator nor the primary advocate of many of the concepts presented. I have tried to credit the source of the concepts appropriately. Thanks to these great testers for having such a positive impact on the test industry. Thanks to all the local testers here in Boulder whose expertise has taught me a great deal and will surely continue to teach me how to become a better tester.

Thanks to David Curtis for his help in reviewing this paper and for making significant contributions to help make it more readable.

Abstract

By becoming better testers we increase our worth to the business. It is a good time to be valuable to the business. This paper makes some practical suggestions for becoming a better tester. These suggestions are based on my observations of people I respect for their test expertise. These suggestions will help you learn how to increase your effectiveness and efficiency. You can pick up some practical ideas of things you can do right now to become a better tester. You will learn about:

- Fitting the test to the context.
- Using heuristic models to focus on the important aspects of the test without missing key elements. (In other words, learn how to be the respected tester that seems to have a real grasp of the test challenge and what to do about it.)
- Getting the skills and techniques you need to advance your expertise.
- Understanding that having the right questions may be more important than having the right answers.

Introduction

What separates novice testers from the experts? What keeps you relevant in your job, valuable to your company, and up-to-speed in the industry? This paper explores some of the critical elements in becoming a better tester, the kind of tester your company needs to keep.

Better Testing is Context-Driven: Better testing requires thinking. You must base your testing on what is best in your context, not simply repeating a previous test or blindly following a best practice. A best practice in one context may be a worst practice in another!

Better Testing is Model-Based: Models trigger ideas for ensuring the best possible coverage of the essential elements of testing. Expert testers have good models that lead them quickly to good tests. Your expertise as a tester depends largely on your ability to quickly form an effective model for the test at hand.

Better Testing Requires Familiarity With Test Techniques: If the only tool you have is a hammer then everything looks like a nail. Every job is easier if done with the correct tools. Learn practical suggestions on how to obtain familiarity with test techniques.

Better Testing Requires Practice to Increase Skills: Our children go to sports practice to get better at their sport. Without constant practice they wouldn't develop the skills to perform well and be competitive. So it is with testing! We need to practice constantly to perform at our best and be competitive. Skill is essential for staying relevant. Practice is essential for improving skills.

The Right Questions Are More Important Than the Right Answers: One of the real dangers in testing is we spend too much time answering the wrong questions. If you are asked "Can you test this just like last time?" it is easy to say "Yes." However, it may be

more productive to ask "Can I test this better than I did last time?". Learn to question everything - at least in your own mind!

Better Testing Requires More Than Technical Testing Skills: It is not enough to just know how to test. We work with people and organizations. To be an effective team member we need to develop our people skills as well as our ability to communicate and organize. Learn to work well with people and it will smooth the way for everything else you need or want to do.

Better Testing Results From Knowing Our Customers Better: The more we know about our customers and the domain in which our customers operate the better able we are to ensure our software is fit for their use

1. Better Testing is Context-Driven

Better testing requires thinking. You must base your testing on what is best in your context, not simply repeating a previous test or blindly following a "*best practice"*. A *best practice* in one context may be a *worst practice* in another. A context-driven approach to testing is essential if you want to get beyond mediocre testing and become a better tester. You need to consider available resources, skills, time, goals, and people. If you run every test the same with a one-size-fits-all approach you will have done your project a disservice. Most one-size-fits-all items don't fit any individual item very well. Consider the following:

Which is better?

- A) A regression test that builds on itself from release to release.
- B) A minimalist regression test that obtains coverage with the fewest number of test cases.
- C) A regression test that targets just the change area.

Which is better?

- A) Automated Testing or
- B) Manual Exploratory Testing

Which is better?

- A) Operational Profile Testing (Software Reliability Engineering)
- B) James Whittaker's Software Attacks

Which is better?

- A) An exit criteria of no high severity defects and all low severity defects have an action plan. All test cases have been attempted and 80% are successful.
- B) An exit criteria of the where the implementation team will make an exit decision based on ODC data, Defect Data, and other relevant information.

In each case the answer is "It depends." Which type of regression test is best may depend on:

- How much time you have to run the test?
- What test cases already exist?
- What is the scope and complexity of the changes?
- When will the next test be run?
- Who will use the code after it is tested?
- What are the history, importance, and intended use of this code?

It is not possible to accurately determine the value of a technique, practice, or process without its context.

As James Bach suggests [1], a key to becoming a better tester and to assessing the "goodness" of a test practice is to ask context setting questions [2]. The following is not the definitive list of context setting questions. Rather, the list provides examples of the types of things you may want to consider.

Context Setting Questions
• What are the constraints? (people, budget, resources, skills, schedules, processes, etc) • What are the goals, purposes and expectations? • What does "success" mean? • What does "quality" mean? • What is the project environment? (product, tools, equipment, experience level, attitude, team, etc.) • Who is available to do the work? How can I utilize them best? • Who is involved and how do they matter? • What testing processes and techniques would work best? • What are the givens? • What are the choices?

If you observe carefully, you will probably be surprised at how often we jump into testing without adequate consideration of context. Likewise, you may be equally surprised at how a few context setting questions at the beginning can yield a more productive test approach.

2. Better Testing is Based on Models

There are more ways to test software than you can possibly utilize on any particular project. A good model is an effective means of quickly and effectively pinpointing which strategies and techniques are of most value for your context. A good model will help you

to give adequate focus to all pertinent aspects of the test. It reduces the chances of giving undue focus to hobby techniques or spending more time than necessary on testing that is not well thought out and well founded.

Models provide a mechanism for quickly guiding a tester to a set of questions that trigger ideas for testing the various aspects of a product and ensuring better coverage of the essential elements of testing. Expert testers seem like experts because they have good models that lead them quickly to good tests. James Whittaker uses the software fault model which utilizes software attacks.[3][4][5]. James Bach uses the Satisfice Heuristic Test Strategy Model [7]. The data model, class diagrams, and state transition diagrams are combined to form an effective model for many testers [8]. ODC, Orthogonal Defect Classification, provides a testing model based on coverage of various characteristics such as triggers, activities and so forth [9]. John Musa uses the Software Reliability Engineering model [6]. There are other models available with varying degrees of effectiveness. Your expertise as a tester will depend in large part on your ability to quickly form an effective model for the test at hand. Those testers who can effectively use models will become the elite testers. Those who can utilize multiple models will stand out even more as experts and have increased value to the business.

Whether you realize it or not, you utilize some mental model to formulate your strategy for testing. It may be as simple as following Glenford Meyers' model of equivalence class partitioning and boundary value analysis [11]. It may be a model that is uniquely your own. It may be a popular model like that of James Bach or James Whittaker. The important thing is that you realize you use a model. If you can't identify what model you follow then take the time to identify the model that you use. Once you have identified it then you can examine it for strengths and weaknesses. You should be able to justify why you do what you do. How does your model (method of formulating tests and test strategy) compare with some of the examples mentioned above? Here are some questions about models that, once you understand the answers, will help you become a better tester.

- What model(s) do you use for testing?
- What are the strengths and weaknesses of your model(s)?
- For each model, what context is it best suited for? (For example, Whittaker's model is much better suited for exploratory GUI testing. Bach's model is more of a general purpose model but may be less effective at triggering ideas for more specific tests.)
    - Is this a good model for requirements testing? Unit testing? Function testing? System testing? Beta testing?

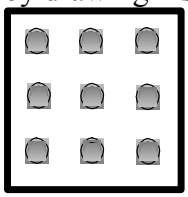
As James Bach points out, models are meant to be guides, not task masters. Don't be a slave to the model! A model is there to guide your thinking, not replace it.

It is worth noting that using models to guide testing may be an important step in moving testing towards an engineering discipline. Models provide a more systematic and predictable approach to testing. Models provide a basis for repeatability and incremental improvement. Models provide a degree of discipline to software testing.

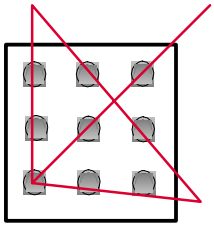
3. Better Testing Requires Familiarity with Test Techniques

If the only tool you have is a hammer then everything looks like a nail. Every job is easier if done with the correct tools. Trying to reuse the same handful of test techniques for all of our test responsibilities year after year limits our effectiveness as testers. Testers all over the world are finding better ways of doing things every day. Many of these testers are sharing what they have learned through conferences, articles, books, magazines, and on the internet. If you want to become a better tester you need to become familiar with what is working. We need to become more familiar with all the test techniques available.

To illustrate the point consider the following diagram. See if you can connect all 9 dots by drawing 4 straight lines without your pencil ever leaving the page.



Many of you are familiar with this little brain teaser. If so, you know the solution and can readily draw 4 lines to connect the dots. If you are not familiar with the technique for solving this problem it may take you some time to figure it out.



You face a similar situation in testing. If you are already aware of the proper technique you can save a lot of time and be much more effective. What is a reasonable approach to raising your awareness level of what techniques are being used successfully in the industry?

- Attend conferences and symposiums.
  Conferences and symposiums are great opportunities because the conference organizers have already taken the time to sift through the many options and have selected some of the best representatives. The presenters have already taken the time to net out the important information into a concise format. You have an opportunity to ask questions and make contacts should you need more information or assistance in the future.

  When you go to a conference, don't sit back and judge presentations based on their entertainment value. Be a detective. For every presentation you attend, identify the two things from that presentation that could help you in your job. Even poorly presented presentations have something to offer. Find it!

For each day of the conference, identify the two things you learned that are most likely to be of value to you. Likewise, identify the two things from the entire conference that are of most value to you.

After the conference, go back to your department and your organization and share every item that you noted was of worth. The process of organizing your thoughts and presenting what was useful and why, will help you understand what you have learned and you will have a better recollection of those items when you need them.

If you can't attend the conferences, do the next best thing. Glean what you can from conferences by reading the articles and talking to the people that were able to attend. Even looking at the brochure will help you to understand the hot topics and directions in testing. It can make you aware of the types of techniques people are claiming to be successful with.

- Use the Internet.
  There are web sites that already do a lot of the homework for you. One example is "stickyminds.com". These sites gather pertinent information so you can obtain it quickly. A minimum amount of time spent each month on such sites will go a long ways towards keeping you in the know and up to date.

  Seek out the experts on the web. Discover who in the test industry you respect. This can come from conferences, papers, other presentation, books, associations, and so forth. Search for additional information from them on the web.

  Seek out information for specific aspects of testing on the web. If your challenge is requirements testing, automation or regression test or whatever, search the web and see what the current knowledge is on that topic. Don't believe everything you read, but don't be afraid to learn something new.

  An hour or two a month spent on focused and diligent research on the web can significantly broaden your awareness of the test industry in general and of some key techniques.
- Observe your peers.
  Look around and see what the people in your own area are doing. You will be amazed at how much knowledge is hidden within your own organization. Who is the most effective tester in your department? In your organization? Find out who they are and why they are effective. Learn from them.
- Read books and articles
  Obviously book stores are a ready source for books. But, don't forget your peers at work have books. Check the local library and libraries at nearby technical schools and universities.

4. Better Testing Requires That We Increase Our Skills [12]

Each test technique requires some skill. Some techniques require more skill than others. Our kids go to sports practice week after week to get better at their sport. Without

practice they wouldn't develop the skills to perform at their best and compete with the other kids and the other teams. So it is with testing. We need to practice. We need to build our skills to perform well and to be competitive. The more skills you acquire, the more fun the sport is. Testing is no different. Practices can be interesting and engaging. However, the purpose of practice is not to entertain but rather to increase capacity and skill. With more skill comes more enjoyment from the testing career. Going to a camp once a year is not enough to become good at a sport. Going to a class once a year is not enough to get good at a career. We need to have practice sessions every week.

There are those who have been in software test for a couple of years and feel like they have learned what there is to learn and are bored and ready to move on. There are others who have spent years in the industry and feel they have barely scratched the surface of what there is to know. There is an explosion of ideas, techniques, and strategies bursting into software test. Do you know what they are? Do you have the knowledge and skill to implement them? Do you know how to assess their usefulness for your context? We need to get on a regular schedule of learning about and practicing the use of these ideas, techniques, and strategies so that each day, each week, each year we know a little more and are better able to do more and do it better. Every day we are asked to do more with less. Here is a way to work towards that end and increase our capacity, effectiveness, and efficiency

What should you be practicing?

- Skills needed right now to meet the immediate demands of your job.
- Skills recognized by the industry as important
- Skills that seem to have some prevalence in the industry that you may not be up-to-speed with

One suggestion is to schedule a one-hour practice each week.

Practices are just that, practicing, not sitting and listening. Whatever you do keep your practices simple enough that you can sustain the effort. Here is some information that may be helpful if you decide to get together and start practicing as a team of testers.

Why Practice?

Why do "test practices/training"? What's in it for me?

We are squeezed for resources and so we must learn to do more with the time and resources that we already have. Are you convinced that you are doing the most effective and efficient test possible for your context?

As we become better testers we will understand which test techniques to apply to a particular context and be able to apply those techniques effectively. The purpose of these practice/training sessions is to:

1. Increase your awareness of testing techniques that are available
2. Increase your skill level in employing test techniques
3. Increase your ability to know which techniques to apply when.

In so doing you should expect to become a better tester, add more value to the business, be harder to replace, and to get more enjoyment out of your career.

Doing the same old test routine time after time can become tiresome and monotonous. Attacking each project with the intent of utilizing the best techniques and expertise possible and demonstrating that those techniques are the best value-add for the business can be interesting, energizing, and career building.

Your level of test expertise can be highly influenced by a few skills. For example, if you give three people a document to review, will each find the exact same problems with the document? Or, are some people much more effective at discovering important problems? Is the other person smarter than you or do they just have a better technique and more skill at reviewing documents? Could you become the "respected" reviewer if you just picked up a skill or two?
Could these skills be the key to being a better reviewer?

- Active Reading with breakdown categories
- Use case analysis
- Ambiguity Checklist review
- Modeling (data flow, state diagram, class diagrams)

We are a professional team of testers. We are among the best in the world. But showing up each day and going after it with intensity is not enough. Without practice we won't be winners. Without practice, how can we even expect to be competitive? Like all other professional teams, if we want to compete and to win we must practice.

How Do You Prepare For and Conduct a Practice?

- Generate initial interest
  Be a salesman. Sell what is in it for tester. Use the information in this paper as a starting point of selling the "why" to your test group.
  Decide on a time and place and frequency.
- How many people do you need
  You can practice with one, two is better, at least three is optimal. Anything more than three is a plus.
- Find topics and materials for practicing
  Sources for topics include:
  - Business needs
  - Local employees with respected expertise
  - Conferences and symposiums
  - Magazines
  - Articles from anywhere including the internet (e.g. Stickyminds.com)

  Examples might include:

	Possible practice session topics
	Topic: Bug Fest What: Bring a bug and describe how you found it, etc. Identify bug fault, its failure symptoms, and what test technique found or could have found it.

Why: Learn bug hunting skills from others, learn bug habits and habitats.
Topic: Attacks What: One by one, practice each of James Whittaker's Attacks Why: Improve expertise in finding bugs
Topic: Issue Entrepreneur Why: Develop expertise on important issues (don't become a substitutable commodity) Possible issues: • Getting user input into test • Deciding what to test (If you can't test everything what do you test and why) • Deciding how to test (There are numerous techniques, practices, and approaches. What are they? Which are best for you and why?) • What are the top 10 items for Regression Testing? (Everyone regression tests. Who does it well? What are the keys to doing it better?) • Metrics (What metrics are truly meaningful? Which are the best? What makes a good metric?)
Topic: Scripting What: Practice basics of Perl, Python, Jython Why: The industry considers these to be essential tools of the trade. We need to know them and we need to be more marketable.
Topic: Book Club What: Pick a Book, Read/skim it for 5 minutes, Try to answer critical questions about the book. If we can't answer the questions then go back to the book and find the answers. Why: Practice rapid learning. Increase awareness of industry
Topic: <Your Topic Here>

- Format of a practice session:
  5-10 minutes: introduce and explain the skill to be practiced
  40 minutes: do exercises to practice the skill (don't let practices become lectures)
  5-10 minutes to wrap up
- Keep a journal
  Keep a copy of the references and sample of the output of the exercises

5. Better Testing is More Dependent on Better Questions than Better Answers

Take the following "pop quiz" regarding testing. Time yourself and take no more than 2 minutes to take the quiz.

POP QUIZ

1. Have you ever had formal training in software test?

2. In what year did you begin your test career?

3. What operating system are you most familiar with?

4. What phase of testing do you usually focus on?
__ Pre-Unit __ Unit __ FVT _SVT _Post SVT

5. How many people are in your test organization?

6. List as many of the industry recommended test practices as you can.

7. What are the five maturity levels in CMM?

8. What does CMM stand for?

9. Does your test organization have a process for continual improvement? If yes, briefly describe it.

10. List 3 software test experts that don't work for your company.

11. List 3 test strategies for function test.

12. What is the most important bug that you have ever found?

13. Have you ever participated in a software test conference?

14. How many test organization have you worked for?

15. How many companies have you worked for?

20. Who is the most effective tester in your organization and why?

25. What methodologies and techniques have been the most effective bug finders in your organization?

50. What are your strengths and weaknesses as a tester?

100. What are the three most important considerations in deciding what and how to test?

Score yourself on the quiz. Questions 1-15 are each worth 1 point. Question 20 is worth 20 points. Question 25 is worth 25 points. Question 50 is worth 50 points. Question 100 is worth 100 points.

Most of our test efforts have a lot of similarities to this pop quiz There are more questions than we can possibly answer in the time given. Some questions count a lot more than others. We need to make sure we are spending our time on the questions that count most.

One of the real dangers in testing is not that we come up with the wrong answer but that we spend too much time answering the wrong questions. For example, I may competently derive a comprehensive set of regression test cases for an item. However, maybe I should have first asked - Does this need regression testing or would a system test be more effective? Or, does it need a performance test or should I prepare an end-to-end test by a domain expert and so on? Every minute we spend on one test activity is time being taken away from another activity; we should be convinced our efforts are focused in the right places.

If you are asked "Can you test this just like last time?" It is easy to say "Yes." It may be more productive to ask "Can I test this better than I did last time?" Which question you work on answering will have a strong influence on how you test. Don't just dive into answering the questions that are put forth. Spend adequate time to ensure that the questions are the right ones to be answering.

The reason better testing is based on models is that models are just an effective way of leading us to ask the correct questions about what we are testing. Not all questions are of equal value. Work on the ones that count the most.

Learn to question everything - at least in your own mind [1]. With experience we learn which of those questions to vocalize and present outwardly and which to keep to ourselves. As you begin to challenge and question everything you will come to understand those parts of your testing that are done for a good reason and those that are not founded on a solid basis. When you discover those that are not founded on a solid basis, you have an opportunity to add expertise and improve the situation. Get in the habit of questioning everything and you will learn which questions lead to productive outcomes. You will also learn that you have to question a lot of things silently in your own mind or you will drive people crazy.

James Whittaker taught that he will approach students while they are testing and ask, "What are you doing?" [5] If the student can't explain what they are doing in terms of what attack are they applying, or what type of bug they are looking for, or what weakness they are exploring then the student fails that question for that day. We should pose that same question to ourselves through out our test activities. If we are going about our test activities without real purpose we are not likely focusing on the questions that count most.

- What are you doing right now? Why?
- Why do you automate what you automate?
- Why do you test the way you do?

- Are you sure this is the best way to test?
- Who will use the information you report? What will they do with it?
- What type of errors do you expect to find?
- Are you confident that this is finding the type of errors you want to find?

Generally, when we are given a problem we can solve it. When we are posed a question we can answer it. Our challenge is to ensure that our focus is on answering the right questions and not losing time dwelling on less productive questions. Thus, in many cases having the right questions is more important than having the right answers.

6. Better Testing Requires More Than Technical Testing Skills

It is not enough to just know how to test. We work with people, organizations, and deadlines. We spend a lot of time breaking things and pointing out that they are broken. We need to work hard to be viewed as a contributor rather than an obstacle. Besides test skills, we need additional working skills to be effective team members, such as metrics, reporting, communicating and so forth. Better testing is context driven and management usually wants testing to be consistent and the same as last time. We need to learn how to make the two work together effectively. Remember, ultimately almost everything is a people issue. Learn to work well with people and it will smooth the way for everything else you need or want to do.

It is outside the scope of this paper to delve into all the related skills in this area. However, it is essential to note that technical testing skills alone are not sufficient in our quest to become better testers. We must develop our people skills and organizational skills. We must generate energy and exercise discipline or our technical skills will be devalued.

7. Better Testing Results From Knowing Our Customers Better

The more you know about the domain in which your customers operate the better you will be able to make sure your software is fit for their use.

To the extent possible you should seek real expertise. Real expertise comes from actually working in the domain. The true domain experts are those who have really worked in the domain. The next best thing is to interact with the customer and observe domain expertise.

Hiring testers that have domain expertise has shown to be a very effective means of upgrading the domain expertise of the whole test team.

There may be more opportunities to interact with the customer than you realize. Do you know how many times your organization comes into contact with the customer? Consider the following ways to learn more: Customer visits, customer briefings, support calls, executive visits with customers. Perhaps you could arrange for someone in your organization to sit in on some of these.

Learn from people who talk to customers. Take a customer support person to lunch and find out their perception of what the customers are doing and saying. Ask for debriefings from visits to resolve crit sits. Talk to marketing about what was learned at customer briefings. At round tables and skip levels ask your executives what they are learning from customers.

Take advantage of the information that you do have available. Use Orthogonal Defect Classification, ODC, to understand field reported defects. Use Orthogonal Problem Classification, OPC to understand field reported problems that don't result in defects. Study the web, trade journals, and articles related to the domain. Take classes that teach about the fundamentals of the customers business.

Customer information is available in various forms to the resourceful tester that is willing to go get it.

Summary

As I observe testers who I respect for their expertise I notice several characteristics that serve as separators for distinguishing them as better testers. As a result, I have compiled this list of suggestions for the rest of us who desire to become better testers.

- The goodness of a test technique or practice depends on its context. Becoming a better tester requires that you learn to ask context setting questions and fit the test to the context.
- Heuristic models remind testers of what to consider when formulating a test and help to ensure proper coverage of all essential elements of the test. A good model is key to quickly and effectively determining what and how to test.
- Using the right test technique in the right context can save a lot of time and improve test effectiveness. Becoming a better tester requires that you become familiar with techniques that are demonstrating usefulness in the industry. You can learn about the techniques through conferences, web sites, observing peers, and through reading books and articles.
- Your skill in applying a technique will directly impact its effectiveness and efficiency. If you want to improve your skill you must practice. Ongoing and frequent practices are a must if you want to get better.
- A significant factor in becoming a better tester is the ability to put focus where it does the most good. Asking the right questions is the key to getting proper focus. Many testers spend too much time correctly answering less important questions. A key to proper focus is to constantly challenge how and why things are being tested and to do it in a constructive way.
- You cannot survive on technical test skills alone. Your success as a tester is dependent on your ability to communicate, organize, and work with people.
- The more you know about your customers and how they operate the more effective you can be in ensuring the software will meet their expectations. You can learn about the customer by actual experience in the customer domain, by talking to people that talk to the customer, and by utilizing available data such as defects, ODC data, crit sit reports, and so forth. In most cases, you have more access to knowledge about the customer than you realize.

Glossary

Active Reading: Use a breakdown of categories of information as a framework for dissecting the content of a document. For example, extract all the information from the document relating to inputs, processing, outputs, reliability, serviceability, etc.

Ambiguity Review: A test technique to identify the ambiguities in the logic and structure of wording in a document. For example: dangling else, ambiguous pronoun references and so forth.

Automated Test: A test that, to one degree or another, is submitted/performed, controlled, monitored and/or verified by a computer program instead of being manually submitted/performed, controlled, monitored and/or verified.

Boundary Value Analysis: Identifying data values that corresponds to a minimum or maximum input, internal, or output value specified for a system or component and identifying values which lies at, or just inside or just outside a specified range of valid input and output values

Crit Sits: A critical situation relating to the product encountered by the customer

Domain Expertise: Knowledge and abilities relating to the customer usage of the software to be tested.

Equivalence Class Partitioning: Partitioning the input domain of a program into a finite number of sets such that any value chosen from a set would be expected to produce the same results as any other value chosen from the set.

FVT - Phase of Testing: Functional Verification Test. A black-box test of the software's functionality.

Heuristic Models: Guidelines, rules-of-thumb, checklists, etc. that trigger the discovery or identification of ideas for testing

James Whittaker's Software Attacks: A set of test techniques for finding software errors.[3]

Manual Exploratory Testing: The opposite of scripted testing. The tester designs and executes tests while in the process of exploring the product.

Operational Profile (Software Reliability Engineering): The set of operations to be performed by the customer and their probability of occurrence. Describes the usage of the product by the customer.

Pre-Unit - Phase of Testing: Testing prior to unit testing which would include test techniques Like design and code reviews.

Post-SVT - Phase of Testing: Any testing that follows SVT, such as beta test.

Regression Test: A test to validate modified parts of the software and ensuring that no new errors are introduced into previously tested code

Satisfice Heuristic Test Strategy Model: A set of patterns for designing a test strategy. The immediate purpose of this model is to remind testers of what to think about when they are creating tests. Ultimately, it is intended to be customized and used to facilitate dialog, self-directed learning, and more fully conscious testing among professional testers.[2]

Skip Level: A meeting with your manager's manager. Sometimes used to increase communication between management and employees.

Software Reliability Engineering Model: The guidelines for the Software Reliability Engineering Practice defined by John Musa. [6]

SVT - Phase of Testing: System Verification Test. Testing at a system level to determine how a software product fairs under workload, stress and interactions with external entities.

Unit - Phase of Testing: Testing an individual unit of code. This is typically a white box test that precedes a function test.

Use Case:	A collection of possible sequences of interactions between the system under discussion and its external actors, related to a particular goal.
Use Case Analysis:	The process of identifying use cases.

1. James Bach, **Becoming an Expert Tester**, StarEast May 13-17, 2002
2. James Bach, **Satisfice Context Model**, http://www.satsifice.com/tools/satisfice-cm.pdf
3. James Whittaker, **How to Break Software: A Practical Guide to Testing**, Pierson Education, Inc., 2003
4. James Whittaker, **A General Software Fault Model for Exploratory Testing**, http://test.ibm.com/s_dir/swtest/swtest.nsf/alias/whitfault
5. James Whittaker, How to Break Software, IBM Test Symposium, Toronto Canada, 2003
6. John Musa, **Software Reliability Engineering**, http://members.aol.com/JohnDMusa
7. James Bach, **Satisfice Heuristic Test Strategy Model**, http://www.satisfice.com/tools/satisfice-tsm.pdf
8. Becky Winant, **Visual Requirements**, STQE May/June 2003, pp 34-42.
9. ODC, http://www.research.ibm.com/softeng/ODC/ODC.HTM
10. Jerry Cobry, **A Better System Test Model**, IBM Test Symposium, Tucson, Arizona, May 17-21, 2004
11. GJ Myers, **The Art of Software Testing**, Wiley, New York,USA, 1979
12. Brian Marick, **Testing in Changing Times**, StarWest October 27-31, 2003, San Jose, CA

An Automation and Analysis Framework for Testing Multi-tiered Applications

Kingsum Chow, Zhidong Yu, Lixin Su, Michael LQ Jones and Huijun Yan

Software and Solutions Group, Intel Corporation

kingsum.chow@intel.com

Key Words/Phrases

Automation Framework, Performance Testing, Multi-tiered Applications

Biography

Kingsum Chow is a senior performance architect with the Managed Runtime Environments (MRTE) within Intel's Software and Solutions Group (SSG). He has been involved in performance modeling and optimization of middleware application server stacks, with emphasis on J2EE and Java Virtual Machines for 3 years. Last year, he increased SPECjAppServer2002 performance four fold by a combination of software and hardware improvements. In 1996, he received his Ph.D. degree in Computer Science and Engineering from the University of Washington and joined Intel's microcomputer research labs. He spent 2 years with Intel Online Services before he joined MRTE. He has published 20 technical papers.

Zhidong Yu is a software engineer working with the MRTE within SSG. Zhidong has been working on performance analysis of enterprise Java applications. He received his MS degree from Shanghai Jiao Tong University in 2001.

Lixin Su is a Ph.D. student and a member of the Predictive High-Performance Architecture Research Mavens (PHARM) group in the Electrical and Computer Engineering department, University of Wisconsin – Madison. He received his M.S from the University of Wisconsin – Madison in 2004.

Michael Jones is a performance engineer working with the MRTE within SSG. Mike has been working on performance analysis of enterprise Java applications. He graduated from Portland State University in 2000.

Huijun Yan is a software engineer working with the MRTE within SSG. Huijun has been working on performance testing of Java programs. She received her MS degree in Computer Science from Brigham Young University in 1991.

Abstract

In the performance testing of multi-tiered applications, many experiments need to be run with various configurations of parameters where measurements, data collections, analysis and debugging need to be performed. The complexity is further increased by the need to handle multiple layers of software components, including operating systems, virtual machines, databases, and application servers, as well as different hardware platforms. The combinations of different software and hardware platforms under testing and the intense tuning work for each combination unavoidably leads to a very demanding number of quality assurance (QA) performance testing tasks. If daily and weekly software changes are introduced to performance testing, only the most sophisticated QA lab may have enough resources to handle the load – by having sufficient resources and machines to setup for all combinations of configurations that need to be tested. To make it even worse, performance testing often requires many runs with the same set of run-time parameters to assess the performance of the software system.

To address the constraints caused by the limited amount of resources, we developed a light-weight approach to automate the performance testing experiments, which can change run-time parameters and schedule runs. This dramatically increases the productivity of the QA engineers. The time saving is about 2 hours per day per engineer after the approach has been implemented versus 8 hours per day per engineer before the change. In addition, the possibility of running wrong experiments (e.g., due to run-time parameter changes) has been greatly reduced. Therefore, the actual productivity gain is bigger than

what is measured by savings in time only. Thus, a more than four fold productivity improvement was observed.

In this paper we share our experience in automation and analysis framework for performance testing. Our approach to automation and analysis is mostly based on composing components from free tools. We will highlight how we build the tools, what problems we face and what lessons we learned.

1 Introduction

Functional testing and performance testing are two complementary testing activities and both have important objectives. While the focus of functional testing is program correctness, the focus of performance testing is program performance. There may not be a clear line between these two activities, however. A performance requirement can be tested within the scope of functional testing, for example. Performance testing may tend to stress more on the performance impact of software components, statically compiled and run time components, run time configurations and hardware platforms. We focus on the response time and throughput metrics of performance testing but do not include graphical user interface (GUI) testing within the scope of this paper.

Earlier work on test automation focuses on functional aspects of software testing. For example, Hoffman [1] described using oracles for automated test verification. In this paper, we describe automation of performance testing as a complementary approach. In our example, the performance criteria are well defined while the automation of test environment and the analysis of the results are just too daunting if without automation.

This paper summarizes our experiences in a case study overcoming performance testing problems involving a multi-tiered application where multiple driver machines are used to increase the load on the system; multiple mid-tier application servers are handling the requests with the help of multiple backend database servers. A single configuration instance is depicted in Figure 1 while many more configurations need to be tested.

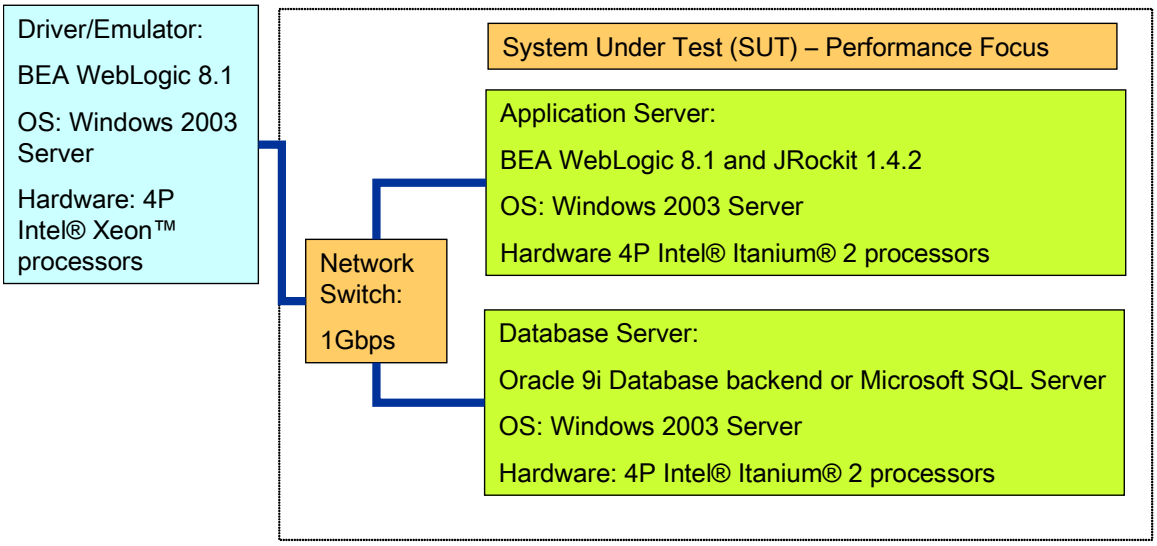


Figure 1: Configuration of an instance of a simple multi-tier application. In this test case, there are only 3 machines connected by an Ethernet switch. The driver drives the workload. The application server uses the database server to help provide services to the driver.

In the beginning of performance testing, each test environment needs to be manually setup and each experiment needs to be manually made. The test engineer has to spend full time adjusting the test environment for different test cases, running the experiments and recording the data. Throughout the course of improvement in test and analysis automation, the test engineer is able to spend only 2 hours a

day in scheduling the test runs and recording the data and spend 6 hours in analyzing and root causing the performance problems. We observed a four fold increase in productivity (by reducing 8 hours of work to 2 hours) due to test and analysis automation.

The analysis automation described in this paper is complementary to our earlier work on reducing test cases [29], performance prediction [30], characterization [31] and capacity planning [32].

1.1 A 4-step approach to performance testing

We have a simple approach to performance testing including workload setup, data collection, performance characteristics and performance analysis.

The first step is to setup the workload that mimics a real usage scenario. This step is often one of the most difficult tasks as customer usage of the software product varies and it is hard to quantify which setup actually mimics a real usage scenario. In the case that a public benchmark exists, a workload based on a benchmark would be a good candidate as it represents combinations of activities from many organizations participating in the development of the benchmark. Two examples of multi-tiered benchmarks are SPECjAppServer2002 [5] and TPCW [6].

The second step is to increase the load on the system while measuring performance behavior. Most systems perform fairly well when the load is not high. When the load is increased, performance may first degrade gradually and then degrade rapidly after a certain point. Examples to assess the load of the system are the number of users accessing the system, the rate of requests and the kinds of requests made to the system, and also the resource utilizations such as CPU, memory, disk and network on the system. Examples to assess the performance of the system are the throughput of transactions and the response time for each kind of transactions. The architect of the software system should define load and performance requirements in the design documents.

The third step is to collect performance characteristics data for multiple software and hardware configuration combinations. Today more software products need to run on a variety of software and hardware platforms. Maintaining competitive performance on all common platforms is important. In the case of an application that runs on top of a application server (e.g. BEA WebLogic Platform [2], IBM WebSphere Server [3] and Oracle Application Server [4]), there are a total of 4 dimensions of layers that affect performance, from top to bottom:(1) Application Servers, (2) Virtual Machines (3) Operating Systems (OS) and (4) Hardware Platforms. Even if there are only 3 choices at each layer, the total combinations of performance testing would be 3x3x3x3 = 81! If front end and back end systems are also variables, which are not uncommon, the combinations can easily explode beyond an unmanageable size. Thus it would be nice to collect performance characteristics and generalize the performance behavior within clusters of configurations. Examples of clusters are similar virtual machine configurations, similar OS and similar hardware platforms. Clustering of configurations can reduce the number of test cases by trading off some level of data accuracy and completeness.

The fourth step is to analyze collected data, summarize performance characteristics and suggest areas of improvements within hardware or software components. Considering the amount of data for load and performance measurements, and the combinations of software and hardware components, this could be a daunting task by itself. It is not productive to ask the QA engineer to examine the data manually. An automation analysis tool can be created to summarize the data and suggest performance problems. The QA engineer can thus examine the automated report and focus on performance issues that are already reported and examine the data more closely to validate the problem, and to suggest solutions by discussing with designers, developers and vendors.

The four steps we laid out in this section form the core of the architecture of our approach to performance testing. In next section, we describe the design of our test environment and automation following this architecture.

2 Designing the Test Environment for Automation,

We design our test environment and automation with the following goals:

- Increase QA productivity:
    - The test environment must be easy to operate remotely without physically going to the lab.
    - The test environment and automation must be easy to maintain.
- Increase performance analysis efficiency:
    - Performance characteristics data must be collected with <1% overhead, e.g., CPU utilizations.
    - Performance analysis must be automated for quick turn around reporting.

We will describe our design philosophy in three areas: network, hardware and software platforms and test and analysis automation.

2.1 Network

The two goals in the network design for performance testing are:

- Sufficient network speed and capacity in the test environment.
- Sufficient isolation among different test environments and the office environment.

Not unexpectedly, we introduce a gateway machine between the office environment and the lab environment. The gateway and the office machines can be managed by standard office information technology (IT) department, including the regulation of software products that can be installed as well as the security rules such as installation of anti-virus programs.

Anti-virus programs introduce an overhead on performance measurement and make analysis difficult due to their intrusion and the arbitrary timing of their activities. They are removed within our test environments. The gateway is used as a shield to protect the test environments from virus attack. We disable all network communications between lab machines and the office environment with the only exception being the gateway. Two gateway machines can be used to increase the availability of the lab environments in case one goes down for repair or maintenance. The gateway machines also have excessive storage to store all the data from the test environments. As there is not a lot of data movement between the lab and the office environments, a 100 Mbps switch is often sufficient.

Each setup in the lab environments has access to a 1Gbps switch as any network performance issues should be avoided as much as possible, since the focus of our performance testing is on the software system. Furthermore, 1Gbps switches and network interface cards (NIC) are relatively cheap enough so budgeting for them is no longer an issue.

Figure 2 shows the network configuration between the office local area network (LAN) and the lab environment through a gateway machine and an Ethernet switch (100 Mbps). In this example, two separate setups (Setup A and Setup B) were supported and each setup has access to its own Ethernet switch (1Gpbs), application and database servers.

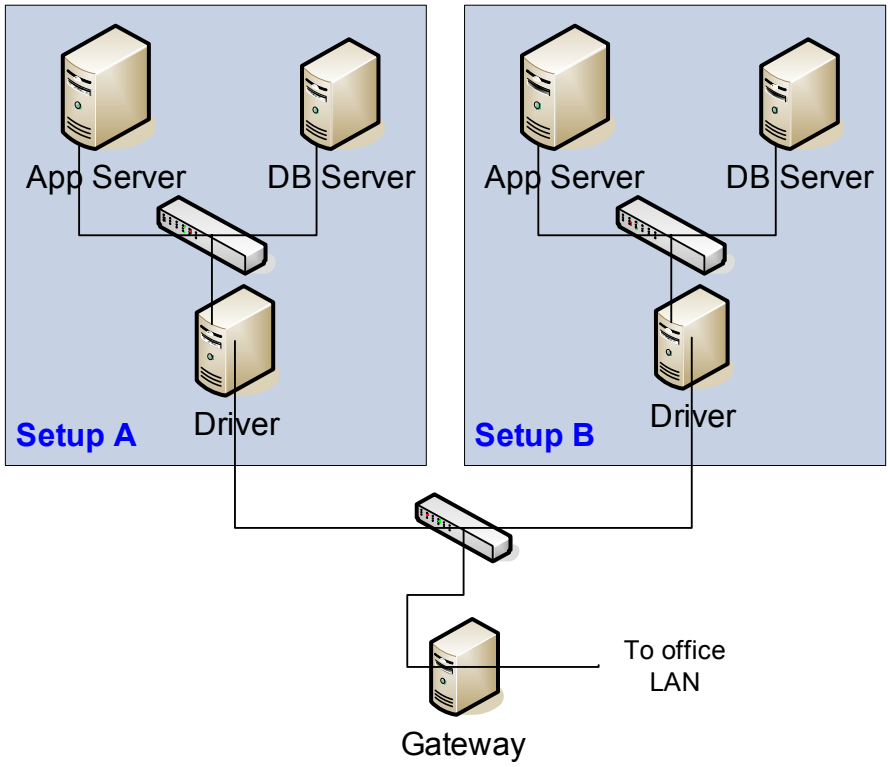


Figure 2: Network configurations to access multiple setups in the performance testing laboratory. Setup A and Setup B are two different setups that can be accessed through the same gateway from the office environment to the lab environment.

The QA engineer from the office LAN environment needs to go through 2 hops to reach a specific test environment – once through the gateway machine and another through the driver. In our case study, the driver starts the test automation so there is no need for the QA engineer to access machines beyond the driver. To support the dual purposes of allowing access and running the performance test, the driver is equipped with 2 NIC's – 1 100Mbps for the switch and 1 1Gbps for the test setup.

2.2 Hardware Platforms

While some design documents may specify the hardware requirements for the workload under test, it is sometimes not complete for reasons that hardware platforms may go through changes while the software is being developed. When the software applications are finally available, the latest hardware might be more advanced than their original design goals. The performance test lab may opt to test with the latest hardware components, including the selection of processors, chipsets, motherboards, memory, disk drives and NIC cards. Many of such selections come in standard from hardware vendors. The only major upgrade that needs attention is perhaps the size of the physical memory.

The performance test lab may choose to use the latest model and the latest processors, chipsets and motherboards available from the vendor. The frequency, cache sizes, the front side bus, memory speed and sizes would have significant impact on the overall system performance. However, the exact relationship between those factors and the performance of an application is largely unknown until the application is actually tested.

We chose to use the latest models to realize the potential maximum performance of the applications. In addition, we want the performance test plan to find performance issues with the latest models, not any old ones. This would aid a more productive software development lifecycle.

2.3 Software Platforms

In addition to the selection latest hardware platforms, the performance test lab should also select latest and stable software platforms. At the OS level, the test plan may select specific released versions of

Windows [7] or Linux [8] builds. In the case of Java [9] applications, there are many Java virtual machines (JVM) [10] available in the market, which may yield different performance with the test application [11]. Enterprise Java applications are often run on top of Java application servers [1-4]. To make the matter even more complicated, each software layer – OS, JVM and App Server – introduce a set of parameters that may need to be adjusted for optimal performance.

One may argue why we have to spend so much effort in performance testing. The two main reasons are:

- To get the application to use and adapt the latest and most advanced features from the software vendors.
- Getting better performance on a single machine means potentially reducing the number of machines that is required to run the application under load, i.e., potentially reducing the cost of managing the software systems in the data center.

Understanding the network, hardware and software selections are required prior to test automation and analysis design.

3 Automating the Test and Data Collection/Reporting Process.

For the case study in this paper, the workload under performance test is SPECjAppServer2002. Thus, the generation of basic performance data files and performance requirements are already defined within the scope of the benchmark. While automating running of the workload is challenging, it does not help us to pin point of source of errors if they exist, and to analyze performance issues. Along with the automating running of the workload, we introduced collecting run logs on all machines for error checking and debugging.

The detection of errors in the log files is not completely automatic, as there are just many possible errors. Instead, we automated the search of errors based on key words such as “error” or “warning” which indicate some kind of problems and captured them in a single line of tab-separated output for each experiment.

The detection of performance problems is easily extracted from the basic performance data files as required by the benchmark. We automate the analysis by not only capturing the performance data files (e.g. Perfmon and VTune data) but also extract them into the same tab-separated output for each experiment.

In short, the QA engineer uses the same output of runs to look for errors and performance issues. Since the output is captured in tab-separated format, one can use Excel to open the output and navigate multiple runs of data for detailed analysis.

3.1 Test Automation Design

The philosophy of our test automation design is to be able to select different hardware and software components to run with the workload and obtain performance data, along with whether it misses or meets the performance criteria.

The consideration of network, hardware and software configurations at multiple layers of the system leads to a complex test environment. This complex test environment requires an adequate test automation design to run the workload under test with different hardware and software layers. It is not practical to change hardware in real time during automation, so most hardware systems are fixed in their own configurations of components such as processors, chipsets and memory. The need to select OS, application servers and JVM also varies depending on the design of the application, e.g. whether it is tied into a specific software stack or not or whether clustering is required

We need to make the automation design flexible enough to cope with the varieties of combinations of setup while it is productive for both QA and analysis work.

The automation should be written in scripts as much as possible so that the source code and comments provide an adequate level of documentation. We would not use binaries as a different document needs to be provided and the document is not necessarily in sync with the binaries after several revisions. Furthermore, binaries are not portable across OS platforms and several versions may need to be built

and maintained. We are heavily in favor of scripts that can run across multiple OS, so far as the total overhead of performance intrusion of the automation scripts and the data collection tools is less than 1%.

All test configuration parameters can be specified on the driver, so a batch of runs can be configured and designed there. This would help the productivity and accuracy of test data as direct access to the mid-tier and backend systems are not needed – thus less likely to make unintentional mistakes. The batch runs should be able to run for multiple days without intervention – thus increasing QA productivity.

The automation design should generate the same sets of files baring small differences due to software and hardware configurations. The same sets of files can be analyzed more effectively by the QA engineers.

As far as possible, the same scripts that are run by automation can be run manually also. This would aid debugging automation scripts and reproducing test results that are in doubt.

3.2 Analysis Automation Design

Successful implementation of test automation increases the amount of data that can be collected every day. The traditional approach to examine performance data would not scale without test automation. Scripts should be created to parse and process the performance data as they are collected to summarize the data, highlight performance and run time problems.

We designed a script called “rollup”, to rollup the performance data in tables that can be imported into Excel [12]. Rollup generates a row of data for each experiment, capturing the run time configurations as well as performance data. It is a high level data summarization tool that enables the QA engineer to locate specific experiments to dive into the detailed data for analysis.

3.3 Tools

In this section, we exclude commercial automation tools and GUI automation, which are important but are not directly relevant to the scope of this paper. We actively look for tools that are small and can be composed easily into a larger automation tool set for our application. We are particularly interested in inexpensive tools that have low performance overhead (<1%). The two classes of tools we examine are: automation and performance tools.

3.3.1 Automation Tools

We use the following tools, that are also commonly available, to compose our automation suite. These tools allow us to launch programs on different machines that need to work together for performance testing. While we found the following set of tools sufficient for our needs, there are certainly many others available for other needs.

- xCmd [13] developed by Zoltan Csizmadia is a free tool allow user to execute applications on remote systems without installing any client software. The tool contains only one executable file, xCmd.exe, which can run under Windows systems. To run the program, you must be an administrator of the remote system.
- PsExec is an alternative remote invocation tool from Sysinternals [14]. The single executable psexec.exe is a part of a growing kit, named PsTools, of Sysinternals. The latest version may have richer function. While xCmd is good, we encountered some stability issues across Win32 and Win64 environments and in that case, PsExec offers a more stable remote execution environment.
- Rsh [15] can be use to invoke commands remotely when the target machine where you want to execute program is a Linux/Unix based system. There is also a RSH client, named rsh.exe, shipped with Windows. We use it to launch remote Linux programs from Windows systems.
- Perl [16] is a very versatile language has been implemented in many platforms, and we use a free Perl distribution called ActivePerl [17] on Windows. As an example, the “rollup” tool mentioned above is written in Perl because of its powerful capability of text processing. We also used Perl scripts to generate run time configurations to setup the experiments for performance testing.

- Shell [18] and Windows Command [19] scripts are used to fulfill the automated testing and data analysis. With some kind of Linux-like environment for Windows, like Cygwin [18], shell scripts can also be run on Windows. The main reason we use Shell scripts is to maintain portable programs across platforms. Shell scripts are more convenient in launching and executing other scripts though Perl can also do the job quite well.

3.3.2 Performance Tools

We collect performance data from our tests. While there are many commercial tools that can generate a variety of user load simulations and obtain end user performance statistics, we have primarily focused on the efficiency of the applications that are tested. We found the following set of tools meet our basic needs without incurring high cost.

- Perfmon [21] is the general name of "System Monitor" and "Performance Logs and Alerts", which are the built-in parts of Windows. With Performance Logs and Alerts, performance data, like CPU utilization, disk I/O counters and network I/O counters, etc., can be collected from local and remote computers. The ability to collect data from both local and remote computers is critical to our performance analysis, as it can show the fluctuation of data across machines in the same sample. Furthermore, Perfmon also enables logging of application specific counters, e.g., many performance counters for SQL server are available when SQL server is used as the database backend. The logged counter data can be viewed by using System Monitor or be exported to Excel. While "perfmon.exe" is used to launch the Perfmon GUI, "logman.exe" can be used to start/stop a configured counter log collection in command line. Thus, "logman.exe" comes in handy in automation of Perfmon data collection.
- Sar [22] reports system activity and we used it on Linux systems. The statistics reported by sar include I/O transfer rates, paging activity, process-related activities, interrupts, network activity, memory and swap space utilization, CPU utilization, etc. The data collected by sar can be saved into a binary format, which can be extracted to standard output later. We developed a script to merge Perfmon and sar data together so we can compare performance across machines with different OS.
- Vmstat [23] reports virtual memory statistics. It's useful for obtaining an overall picture of CPU, paging, and memory usage. The reported statistics include kernel threads in the run and wait queue, memory, paging, disks, interrupts, system calls, context switches, and CPU activity. Here the CPU activity is a percentage breakdown of user mode, system mode, idle time, and waits for disk I/O.
- Iostat [24] reports I/O statistics, as well as CPU utilization. The reported I/O statistics include transfers per second, block reads/writes per second, bytes read/written per second, and read/write requests per second, etc.
- Top [25] monitors system's performance in real time. It displays a listing of the most CPU-intensive processes on the system. It can provide each process's PID, priority, memory size, CPU percentage, and status, etc. Top can run in either an ongoing mode or batch mode.
- Intel® VTune™ Performance Analyzer [26] can help locate and remove software performance bottlenecks by collecting, analyzing, and displaying hot spots or performance data from the system-wide level down to the source level. The "VTL" command line interface within VTune suite can be used for automation of performance data collection.

4 Implementation for test automation and performance analysis

We now turn to one complex instance of a multi-tiered application (see Figure 3). In this instance, two drivers are used to deliver the load to a cluster of 8 application servers and 2 database servers are used to provide data services for those 8 application servers. 8 application servers and 2 database servers are used to test the scalability of the application. The application itself is ready to run manually but automation is required to collect performance data on all the machines.

There is no hard OS requirement for the drivers. We chose to run Windows on the drivers for the ease of automation and the access to Excel to analyze data. We used "logman.exe" on the first driver to collect

system performance counters from all Windows machines on the network. A mix of Windows command and cygwin shell scripts are used for automation.

The 8 application servers are a cluster of Linux machines. We used “sar” to collect system performance counters on these machines locally. Shell scripts are used within these machines for automation.

The 2 database servers in this particular instance are Windows machines running SQL server. Windows command scripts are used for automation. Performance counters are collected by the Perfmon initiated by “logman.exe” from Driver 1.

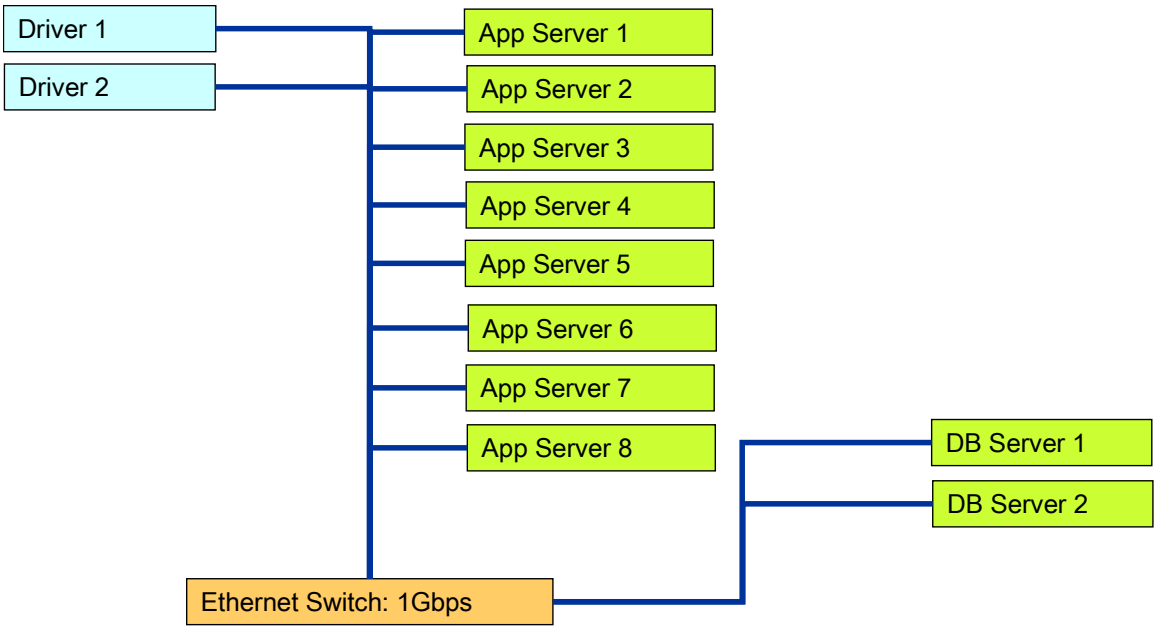


Figure 3: An instance of a complex multi-tiered application that needs performance test automation. Two drivers are used to drive the load while eight application servers provide the service with the use of 2 database servers.

Data files, automation scripts and analysis automation are three key components to our implementation. We will first describe what data files we need to collect. Then we will describe how we collect all the data we want. Since automation can generate tons of data, we will also talk about how we automate the analysis of the data we collect.

4.1 Automation Scripts

The Automation scripts have the following functions: configure all the tiers based on a set of predefined configuration files; drive a run; collect the performance data at run time; collect the results data and save the configurations; generate a set of configuration files by modifying some frequently changed fields in a set of template files; and schedule a batch of runs. The first three functions are executed in all single run automation and the last two functions are only needed for multiple runs. This section starts by describing the automation of a single run followed by the automation of multiple continuous runs. In the end we highlight the problems solved and lessons learned.

4.1.1 Automate a single run

Most multi-tier workloads include a script on the driver boxes to launch the workload. Here we assume such a script is called driver.cmd. For any single run, our automation script will eventually call driver.cmd and start the workload. We assume our single run automation script is called automation.cmd. However,

a lot of automation work needs to be done before driver.cmd can be called and post-run processing is also necessary after driver.cmd is called.

Our automation scripts start by configuring all the machines, including driver machines, middle tiers and back-end databases. The configurations of the application servers and the database software are performed by remotely copying configuration files from drivers to middle tiers and back-ends. The configuration files must be prepared before the single run automation scripts start.

In addition to remote configuration file replication, other techniques such as global synchronization and self-logging, should also be adopted for valid single run automation. To achieve global synchronization, system time should be synchronized across machines. Furthermore, database and application server should be started before the launch of the driver and the driver also needs to execute data loading and cache warm-up after it is launched. Self-logging is an important technique that logs all the information for all the processes/tasks executed remotely on middle tiers or back-ends. It can greatly reduce the debugging time if anything goes wrong. The following is an example for self-logging. The *startAppServer.sh* file is used to launch application server on middle tiers. We add following code into its beginning:

```
if [ "Manual$1" != "Manual" ]
then
  echo "Starting app server: logging stdout/stderr to $1.appserver.log"
  ./startAppServer.sh >& $1.appserver.log
else
#…… the normal scripts
fi
```

The script can log the output to a file with an identifier that is the run ID generated by the automation script.

The automation script is also responsible to schedule the performance data collection tasks. The start times of such tasks should be calculated accurately to ensure that the data collections are started and stopped at the right moments, e.g. at the beginning and the end the steady state of the workload. To modularize the automation scripts, the data collection tasks are written in separate files called by the main automation script. The data collection scripts should be aware of the underlying operating systems where data is collected as performance data collection is different across OS.

After the run is finished, the automation script needs to copy all the results, including the output of the workload, log files due to self-logging, configuration files and performance data to a local directory and the locally directory should follow a certain naming convention for easy lookup. The generated local directory contains all the files needed for data analysis.

4.1.2 Automate multiple runs

The success of multiple run automation script lies in the efficient handling of the large number of configuration parameters for the database, the application server, and the driver. It can reuse the single run automation scripts to execute a single run. Our approach is the combination of the template configuration files and the command line. The template configuration files provide default values for all configuration parameters. The command line provides values for certain parameters that can be used to dynamically replace the default values in the template configuration files. The real configuration files are generated after the replacements are finished. This approach takes advantage of the fact that many configuration parameters are fixed for a particular study and it can greatly reduce the overhead caused by dynamic parameter generation. For example, the most frequently changed parameters in middle tiers are the JVM versions, Java parameters, and thread pool sizes, etc. Our script only takes these several parameters as the configuration options. The script passes such parameters into the template configuration file and generate the real configuration file for middle tiers. We assume the key script of our multiple run automation script is called run.cmd. To fire a customized run, users just need to type following line of command. N lines of such commands can fire N runs. The multiple run automation script includes multiple such command lines.

```
run.cmd 100 15 20 3 pess gds bsql specdb perfmon jdk1.4.1 parallel 1024m
```

The first parameter, 100, is the Injection Rate, which controls the load pressure on the middle tier. The second and the third numbers are two thread pool sizes for application server. The fourth number is a similar parameter on the driver side. The last parameter is the heap size of the JVM. All the 5 parameters can be directly passed into the template configuration files.

The rest of the parameters in the command line control more sophisticated configurations. Based on them, *run.cmd* can determine which configuration file should be used, what kind of configuration combination should be used, and what kind of performance data should be collected. They are specific to our application.

4.1.3 Lessons and Experiences

The automation scripts release engineers from the pressure of manual submissions. Before the automation scripts were created, engineers had to manually submit a run every hour and they had to type in multiple commands on different machines for each submission. Furthermore, no runs could be submitted at night or on weekends. With the automation scripts, engineers only need to type in one simple command to submit multiple runs and the scripts can submit multiple runs for engineers. The automation scripts have greatly improved our productivity.

In the process of the creation, the maintenance and the application of these scripts, we have learned valuable lessons and experiences. First, simplicity is very important for automation scripts. We should always try to adopt the light-weight methodology. The scripting languages are simple and easy to maintain compared with binaries. We should avoid writing C/C++ or Java programs to take charge in automation. Second, source code should be accessible by every engineer. It is very important for every engineer to understand the scripts even if he/she doesn't have to modify it. It can help engineers avoid making mistakes. In addition, the availability of source makes training easier. Engineers can also be more independent when they move automation among setups. Third, preparation and verification are vital to efficient usage of such scripts. An engineer should try to know the system as much as possible and make sure the automation scripts won't do anything surprising. It's very easy to just type in a command and leave. Sometimes no runs will be finished due to some unexpected errors. An engineer should always try to verify the first run is successfully launched to avoid wasting the whole night or weekend.

4.2 Data Files

The automation we implemented need to capture three kinds of data. They are the configuration data, the result data and the performance data. The automation creates a result folder for each test run.

The configuration data are specific to the servers used in the test. It includes the driver data such as what the load is applied to the system, how long the load is applied, how many processes and threads are used to load the system on the driver. It includes the application server configurations such as how many threads are used for various transactions. It also includes the database server configurations. Configuration data files are copied over from the application and database servers at the end of each run. They are stored in the result folder at the end of each test run.

The result data are specific to the nature of the application under test. It includes the event logs on all three groups of machines. It also includes application specific performance data such as the number of transactions that are processed and the response times of the transactions. Driver result data files are generated in the result folder directly while log files of key events are copied over from all the machines to the same result folder for the ease of analysis in a central location.

The performance data are specific to the tools that are used to collect such data. They are mostly system wide performance data that indicate the overall health of the system. They are used to identify resource utilizations in many levels. Perfmon data files are stored in the result folder directly while "sar" data files are copied over to the driver. A script is used to merge "Perfmon" and "sar" data together into a single tab separated format file for ease of analysis using Excel to compare data across machines, which each row corresponds to the same sample collected at the same time.

4.3 Analysis Automation

The availability of automation scripts enable us to have multiple continuous runs, which leads to enormous amount of data to be analyzed. Manual data analysis is not only time consuming but also error-prone. Therefore, analysis automation is a necessity to prevent data analysis from being the bottleneck of the experiments. This section first describes our data analysis automation and then we share our lessons and experiences.

4.3.1 Data Analysis Automation Methodology

Our analysis automation is performed in an efficient way and it also has an expandable framework that can accommodate any changes of the workloads. First, our analysis automation uses Perl to collect useful information from each run. The collected information includes metrics, error messages, and performance analysis data. Metrics are specific to each benchmark and the Perl scripts should be able to alter metrics according to the benchmark. An alternative approach is to include a comprehensive set of metrics for all benchmarks. The scripts should be able to capture all error messages to avoid reporting wrong results. The scripts should also be able to differentiate error messages from warnings. The performance analysis data may include benchmark specific data, OS performance monitoring data, database specific tuning data, and VTune hot spots. The benchmark specific data can help you identify inherent workload trend and set up correct workload parameters. The OS performance monitoring data is OS dependent. Windows uses Perfmon while LINUX uses iostat, vmstat, sar, etc. The database specific tuning data is important to identify the database constraint. The VTune data facilitates further study of the workloads. They are not used when only the workload performance is concerned. The scripts finally generate all useful data in a tabular form for the second stage of data analysis. Second, Excel is used to open the outputs of the first stage. Excel is a very handy tool for data comparison and analysis. Last, conclusions need to be reached and feedbacks need to be sent back to re-run the workload. This step is still error-prone compared with the previous two steps. It involves subjective judgments. Details of performance analysis and tuning methodology are covered in earlier work [11,33-34].

4.3.2 Lessons and Experiences

Our analysis automation dramatically increases the overall data analysis throughput and assists in the improvement of our workload performance. It releases engineers from the burden on the sea of raw data and can help engineers make quick decisions.

Since the application of analysis automation, we have acquired precious lessons and experiences. First, the general approach of data analysis phase should remain stable. The development of workloads, the depth of the data analysis performed, and the introduction of new performance analysis methodology may require the collection of different data for analysis. However, the general approach of analysis automation and the interface between human beings and analysis automation software should be maintained. Second, the analysis automation framework needs to be adapted to the changes of the workloads. Transitions among different workloads may also require certain changes in the framework. The framework should be designed with these requirements in mind. The framework should be divided into two parts. The first part is the interface that abstracts the user from the second part. The second part performs the real work and can be modified for the emerging needs. Finally, a novice engineer should always consult an experienced engineer to avoid drawing wrong conclusions while performing data analysis. A wrong conclusion may result in the submission of useless runs and the waste of valuable computing time.

5 Summary

In this paper, we highlighted many challenges to performance testing of multi-tiered applications. The combinations of machine types, OS, virtual machines and application servers made it a daunting task just to run the tests. The parameters introduced in each layer make the test process time consuming and error prone for the QA engineer. Until the test process is automated, the QA engineer has to spend full time just to configure test, run the test, and collect performance and log data for analysis. Since each experiment requires about 1 hour running, the QA engineer can only schedule 8 runs in a day and perhaps only 1 hour for test data analysis. During the run, the QA engineer has to trigger different

activities such as launching of services on different machines and launching the performance data collections.

There are indeed many challenges to automation, in the areas of maintenance, usability and reliable data collection. These challenges can be overcome by a top down design and implementation of a light weight automation system as described in the paper. Once the automation is completed, the QA engineer only has to spend 1 hour to configure the automation to run for the night and 1 hour to examine the results in the morning. 15 runs can be scheduled during the night for routine QA testing. The day time can be used to schedule runs for more urgent test cases when they arise. The QA engineer can also spend the day time analyzing performance issues and help find the sources of software errors from the performance and log data.

We estimated the productivity gain with our light weight automation system to be four fold (improved from 8 hours to 2 hours in a day) as measured by the time savings in getting test runs done by the engineer. The productivity is three fold (improved from 8 runs to 24 runs in a day) as measured by the number of runs that can actually be made. The productivity is 6 fold (improved form 1 hour to 6 hours in a day) as measured by the increase in time the QA engineer can spend on data analysis. The quality of the test runs made was also much better as fewer human errors were introduced in the test runs.

We developed the light weight automation system using mostly off-the-shelf and even free products. Scripts were exclusively used for portability and maintainability reasons. In addition, the same scripts can be used for manual as well as automation testing. This has the advantage that a mix of automation and manual tests can continue while the automation scripts are being developed. Only tools that exhibit minimal overhead (<1%) to the software systems were used to collect performance data in order not to disturb the system too much, which may change the behavior of the system under test.

There are several limitations to our approach. We did not use a generic load testing system so the test driver is custom made for the test cases. We did not capture detailed response time profiles as available by some commercial tools that can do that from the end user's point of view.

We have only scratched the surface of the complexity of automating performance test cases for multi-tiered applications. During the course of implementation, we discovered more tools on the Internet that can aid automation development. We also discovered that timely launching services on multiple boxes are critical for good performance data collection. Indeed several portions of our automation scripts are just to get the dependency of timings right. Trade-offs between simplicity of automation and reproducibility performance data were regularly made, as not all performance tools work the same way.

Acknowledgments

Elizabeth Ness, Ricardo Morin and Ron Talwalkar provided feedback to early versions of the paper. Gim Deisher, Jason Davidson, Ashish Jha, Aravind Pavuluri and Xiang Fang contributed in the tools.

References

1. Douglas Hoffman, "Using Oracles in Test Automation", Proceedings of Pacific Northwest Software Quality Conference, October, 2001, Portland, Oregon, USA, pp 90-102.
2. BEA WebLogic Server http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/server/
3. IBM WebSphere Application Server, http://www-306.ibm.com/software/info1/websphere/index.jsp
4. Oracle Application Server, http://www.oracle.com/appserver/
5. Standard Performance Evaluation Corporation (SPEC), The SPECjAppServer2002 benchmark http://www.spec.org/jAppServer2002/
6. Transaction Processing Performance Council, TPC-W http://www.tpc.org/tpcw/
7. Microsoft Windows, http://www.microsoft.com/windows/default.mspx
8. Linux Online Inc., "What is Linux" http://www.linux.org/info/
9. Java Technology, http://java.sun.com/
10. Sun Microsystems, Inc., "The Java Virtual Machine Specification" http://java.sun.com/docs/books/vmspec/

11. Kingsum Chow, Ricardo Morin, Kumar Shiv, “Enterprise Java Performance: Best Practices”, Intel Technical Journal, 2003 Q1. http://www.intel.com/technology/itj/
12. Microsoft Corporation, “Microsoft Office Online Home Page” http://office.microsoft.com/home/default.aspx
13. Zoltan Csizmadia, “”Execute Applications on Remote Systems http://www.codeguru.com/Cpp/I-N/network/remoteinvocation/article.php/c5433
14. Mark Russinovich and Bryce Cogswell, “Sysinternals” http://www.sysinternals.com/ntw2k/utilities.shtml
15. Machtelt Garrels, “Remote execution of applications” http://tille.soti.org/training/tldp/ch10s03.html
16. Jarkko Hietaniemi, “Comprehensive Perl Archive Network” http://www.cpan.org/ports/index.html
17. ActiveState, “ActiveState ActivePerl” http://www.activestate.com/Products/ActivePerl/
18. Vivek G. Gite, “What is Linux Shell” http://www.freeos.com/guides/lsst/ch01sec07.html
19. Microsoft Corporation, “Command-line reference A-Z” http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/ntcmds.mspx
20. Cygwin, http://cygwin.com/
21. Microsoft Corporation, “Performance Logs and Alerts overview“ http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/sag_mpmonperf_02a.mspx
22. Sebastien Godard, ”sar(1) - Linux man page” http://www.die.net/doc/linux/man/man1/sar.1.html
23. Henry Ware, ”vmstat(8) - Linux man page” http://www.die.net/doc/linux/man/man8/vmstat.8.html
24. Sebastien Godard, ”iostat(1) - Linux man page” http://www.die.net/doc/linux/man/man1/iostat.1.html
25. Michael K. Johnson, “top(1) - Linux man page” http://www.die.net/doc/linux/man/man1/top.1.html
26. Intel Corporation, “VTune™ Performance Analyzer,” http://www.intel.com/software/products/vtune
27. BEA Systems, Inc., “BEA WebLogic JRockit 8.1” http://e-docs.bea.com/wljrockit/docs81/index.html
28. Kingsum Chow and Gim Deisher, “SPECjAppServer2002 Performance Tuning”, WebLogic Developer’s Journal, September 2003. (http://sys-con.com/weblogic/)
29. Kingsum Chow, Mahesh Bhat and Jason A. Davidson, “Minimizing Performance Test Cases for Multi-Tiered Software Systems”, Proceedings of Pacific Northwest Software Quality Conference, October 14-16, 2002, Portland, Oregon, USA.
30. Kingsum Chow and Mahesh Bhat, “Methods for Web Performance Prediction And Capacity Planning”, BEA eWorld 2002.
31. Kingsum Chow, Mahesh Bhat, Ashish Jha, Colin Cunningham, “Characterization of Java Application Server Workloads”, Proceedings of the Workshop on Workload Characterization, held in conjunction with Micro-34, The 34th Annual ACM/IEEE International Symposium on Microarchitecture, Dec 1-5, 2001, Austin, Texas, USA.
32. Kingsum Chow, “Methods for e-Commerce Web Performance Prediction And Capacity Planning”, Proceedings of the Intel Quality and Reliability Conference, Oct 16-18, 2000, San Diego, California, USA.
33. Kingsum Chow, “Optimizing J2EE* Performance for the SPECjAppServer Family of Benchmarks”, Intel Developer Services, http://www.intel.com/ids, 2004.
34. Kingsum Chow, “Laying the Foundation for J2EE* Performance Optimization”, Intel Developer Services, http://www.intel.com/ids, 2004.

Pairwise Testing: A Best Practice That Isn't

James Bach
Principal, Satisfice, Inc.
james@satisfice.com

Patrick J. Schroeder
Professor, Department of EE/CS
Milwaukee School of Engineering
patrick.schroeder@msoe.edu

James Bach (http://www.satisfice.com) is a pioneer in the discipline of exploratory software testing and a founding member of the Context-Driven School of Test Methodology. He is the author (with Kaner and Pettichord) of Lessons Learned in Software Testing: A Context-Driven Approach. Starting as a programmer in 1983, James turned to testing in 1987 at Apple Computer, going on to work at several market-driven software companies and testing companies that follow the Silicon Valley tradition of high innovation and agility. James founded Satisfice, Inc. in 1999, a test training and outsourcing company based in Front Royal, Virginia.

Patrick J. Schroeder is a professor in the Electrical Engineering and Computer Science Department at the Milwaukee School of Engineering in Milwaukee, Wisconsin. Dr. Schroeder teaches courses in software testing, software engineering, and software project management. Dr. Schroeder is the current President of the Association of Software Testing, and newly formed organization dedicated to improving the practice of software testing by advancing the science of testing and its application. Prior to joining academia, Dr. Schroeder spent 14 years in the software industry, including seven years of at AT&T Bell Laboratories.

Abstract

Pairwise testing is a wildly popular approach to combinatorial testing problems. The number of articles and textbooks covering the topic continues to grow, as do the number of commercial and academic courses that teach the technique. Despite the technique's popularity and its reputation as a best practice, we find the technique to be over promoted and poorly understood. In this paper, we define pairwise testing and review many of the studies conducted using pairwise testing. Based on these studies and our experience with pairwise testing, we discuss weaknesses we perceive in pairwise testing. Knowledge of the weaknesses of the pairwise testing technique, or of any testing technique, is essential if we are to apply the technique wisely. We conclude by re-stating the story of pairwise testing and by warning testers against blindly accepting best practices.

No Simple Formulas for Good Testing

A commonly cited rule of thumb among test managers is that testing accounts for half the budget of a typical complex commercial software project. But testing isn't just expensive, it's arbitrarily expensive. That's because there are more distinct imaginable test cases for even a simple software product then can be performed in the natural lifetime of any tester [1]. Pragmatic software testing, therefore, requires that we take shortcuts that keep costs down. Each shortcut has its pitfalls.

We absolutely need shortcuts. But we also need to choose them and manage them wisely. Contrary to the fondest wishes of management, there are no pat formulas that dictate the best course of testing. In testing software, there are no "best practices" that we simply must "follow" in order to achieve success.

Take *domain partitioning* as an example. In this technique, instead of testing with every possible value of every possible variable, the tester divides test or test conditions into different sets wherein each member of each set is more or less equivalent to any other member of the same set for the purposes of discovering defects. These are called equivalence classes. For example, instead of testing with every single model of printer, the tester might treat all Hewlett-Packard inkjet printers as roughly equivalent. He might therefore test with only one of them as a representative of that entire set. This method can save us a vast amount of time and energy without risking too much of our confidence in our test coverage— as long as we can tell what is equivalent to what. This turns out to be quite difficult, in many cases. And if we get it wrong, our test coverage won't be as good as we think it is [2].

Despite the problem that it's not obvious what tests or test data are actually equivalent among the many possibilities, domain partitioning is touted as a technique we should be using [3]. But the instruction "do domain testing" is almost useless, unless the tester is well versed in the technology to be tested and proficient in the analysis required for domain partitioning. It trivializes the testing craft to promote a practice without promoting the skill and knowledge needed to do it well.

This article is about another apparent best practice that turns out to be less than it seems: pairwise testing. Pairwise testing can be helpful, or it can create false confidence. On the whole, we believe that this technique is over promoted and poorly understood. To apply it wisely, we think it's important to put pairwise testing into a sensible perspective.

Combinations are Hard to Test

Combinatorial testing is a problem that faces us whenever we have a product that processes multiple variables that may interact. The variables may come from a variety of sources, such the user interface, the operating system, peripherals, a database, or from across a network. The task in combinatorial testing goes beyond testing individual variables (although that must also be

done, as well). In combinatorial testing the task is to verify that different combinations of variables are handled correctly by the system.

An example of a combinatorial testing problem is the options dialog of Microsoft Word. Consider just one sub-section of one panel (fig. 1). Maybe when the status bar is turned off, and the vertical scroll bar is also turned off, the product will crash. Or maybe it will crash only when the status bar is on and the vertical bar is off. If you consider any of those conditions to represent interesting risks, you will want to test them.

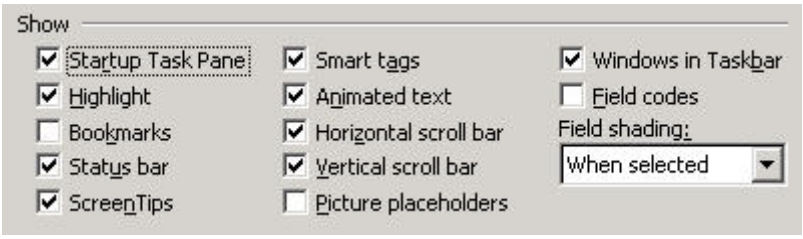


Figure 1. Portion of Options Dialog in MS Word

Combinatorial testing is difficult because of the large number of possible test cases (a result of the "combinatorial explosion" of selected test data values for the system's input variables). For example, in the Word options dialog box example, there are 12,288 combinations (2^{12} times 3 for the Field shading menu, which contains three items). Running all possible combinatorial test cases is generally not possible due to the large amount of time and resources required.

Pairwise Testing is a Combinatorial Testing Shortcut

Pairwise testing is an economical alternative to testing all possible combinations of a set of variables. In pairwise testing a set of test cases is generated that covers all combinations of the selected test data values for each *pair* of variables. Pairwise testing is also referred to as *all-pairs* testing and *2-way* testing. It is also possible to do all triples (3-way) or all quadruples (4-way) testing, of course, but the size of the higher order test sets grows very rapidly.

Pairwise testing normally begins by selecting values for the system's input variables. These individual values are often selected using domain partitioning. The values are then permuted to achieve coverage of all the pairings. This is very tedious to do by hand. Practical techniques used to create pairwise test sets include Orthogonal Arrays, a technique borrowed from the design of statistical experiments [4-6], and algorithmic approaches such as the *greedy* algorithm, presented by Cohen, et al. [7]. A free, open source tool to produce pairwise test cases, written by one of the authors (Bach), is available from Satisfice, Inc. [http://www.satisfice.com/tools/pairs.zip].

For a simple example of pairwise testing, consider the system in Figure 2. System S has three input variables X, Y, and Z. Assume that set *D*, a set of test data values, has been selected for each of the input variables such that $D(X) = \{1, 2\}$; $D(Y) = \{Q, R\}$; and $D(Z) = \{5, 6\}$.

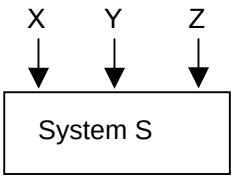


Figure 2. System S with 3 input variables

The total number of possible test cases is 2×2×2 = 8 test cases. The pairwise test set has a size of only 4 test cases and is shown in Table 1.

Table 1: Pairwise test cases for System S

Test ID	Input X	Input Y	Input Z
TC_1	1	Q	5
TC_2	1	R	6
TC_3	2	Q	6
TC_4	2	R	5

In this example, the pairwise test set of size 4 is a 50% reduction from the full combinatorial test set of size 8. You can see from the table that every pair of values is represented in at least one of the rows. If the number of variables and values per variable were to grow, the reduction in size of the test set would be more pronounced. Cohen, et al. [8] present examples where a full combinatorial test set of size 2^{120} is reduced to 10 pairwise test cases, and a full combinatorial test set of size 10^{29} is reduced to 28 test cases. In the case of the MS Word options dialog box, 12,288 potential tests are reduced to 10, when using the Satisfice Allpairs tool.

Pairwise testing is a wildly popular topic. It is often used in configuration testing [9, 10] where combinatorial explosion is a clear and present issue. It is the subject of over 40 conference and journal papers, and topic in almost all new books on software testing (e.g., [10-14]). The topic is so popular as to be taught in commercial and university testing courses (such as those of the authors), and testing conferences continue to solicit for more papers on the topic (www.sqe.com/starwest/speak.asp). Pairwise testing recently became much more accessible to testing practitioners through a number of open source and free software tools that generate sets of pairwise test cases.

In discussing a table representation of a pairwise test set created from an orthogonal array, Dalal and Mallows comment that the set "protects against any incorrect implementation of the code involving any pairwise interaction, and also whatever other higher order interactions happen to be represented in the tables [15]."

The pairwise approach is the popular because it generates small test sets that are relatively easy to manage and execute. But it is not just the significant reduction in test set size that makes pairwise testing appealing. It is also thought to focus testing on a richer source of more important bugs. It is believed by many, for instance, that most faults are not due to complex interactions, but rather, "most defects arise from simple pairwise interactions [6]." This is idea is supported by an internal memorandum at Bellcore indicating that "most field faults are caused by interactions of 1 or 2 fields [16]." Furthermore, it is argued that the probability of more than two particular values coinciding cannot be greater than and is probably a lot less than the probability

of only two coinciding values. Perhaps a problem that only occurs when, say, seven variable values line up in a perfect conjunction may not *ever* occur except in a test lab under artificial conditions.

This then is the pairwise testing story. See how nicely it fits together:

1) Pairwise testing protects against pairwise bugs
2) *while* dramatically reducing the number tests to perform,
3) *which is especially cool because* pairwise bugs represent the majority of combinatoric bugs,
4) *and* such bugs are a lot more likely to happen than ones that only happen with more variables.
5) *Plus,* the availability of tools means you no longer need to create these tests by hand.

The story is commonly believed to be grounded in practitioner experience and empirical studies. In referring to pairwise testing, Copeland states: "The success of this technique on many projects, both documented and undocumented, is a great motivation for its use [11]." Lei states that: "Empirical results show that pairwise testing is practical and effective for various types of software systems [17]," and refers the readers to several papers, including [7, 8, 16].

Well, let's look at the empirical evidence for pairwise testing.

What Do Empirical Studies Say About Pairwise Testing?

Early work by Mandl [4] described the use of orthogonal Latin Squares in the validation of an Ada compiler. The test objective was to verify that "ordering operators on enumeration values are evaluated correctly even when these enumeration values are ASCII characters." For the test data values selected, 256 test cases were possible; Mandl employed a Greco-Latin square to achieve pairwise testing resulting in 16 test cases. Mandl has great confidence that these 16 test cases provide as much useful information all 256 test cases. He has this confidence because he analyzed the software being tested, considered how it could fail, and determined that pairwise testing was appropriate in his situation. This analysis is enormously important! It is this analysis that we find missing in almost all other pairwise testing case studies. Instead of blindly applying pairwise testing as a "best practice" in all combinatorial testing situations, we should be analyzing the software being tested, determining how inputs are combined to create outputs and how these operations could fail, and applying an appropriate combinatorial testing strategy. Unfortunately, this lesson is lost early in the history of pairwise testing.

As commercial tools became available to generate pairwise and n-way test suites, case studies were executed using these tools. These studies are frequently conducted by employees of the pairwise tool vendor or by consultants that provide pairwise testing services.

In a study presented by Brownlie, Prowse, and Phadke [18], a technique referred to as Robust Testing™ (Robust Testing is a registered trademark of Phadke Associates, Inc.) is used. The Robust Testing strategy uses the OATS (Orthogonal Array Testing System) to produce pairwise

test sets in the system level testing of AT&T's PMX/StarMAIL product. In the study, a "hypothetical" conventional test plan is evaluated. It is determined that 1500 test cases are need, although time and resources are available for only 1000 test cases. Instead of executing the conventional test plan, the Robust Testing technique is applied to generate pairwise test set that combines both hardware configuration parameters and the software's input variables. The results of applying the technique are: 1) the number of test cases is reduced from 1000 to 422; 2) the testing effort is halved; and 3) the defect detection rate improves, with the conventional test plan finding 22% fewer faults than the Robust Testing technique. The problem with the study is that the conventional test plan is never executed. Important information, such as the number of faults detected by executing the conventional test plan, is estimated, not directly measured. The use of such estimates threatens the validity of the results and the conclusions that can be drawn from them.

Another commercial tool that can generate pairwise and n-way test suites is the Automatic Efficient Test Generator or AETG™ (AETG is a registered trademark of Telcordia Technologies, Inc., formerly Bellcore, the research arm of the Bell Operating Companies). Many industrial case studies have been conducted using the AETG. Cohen, et al. present three studies that apply pairwise testing to a Bellcore inventory database systems [7, 8, 16]. Software faults found during these studies lead the authors to conclude that the AETG coverage seems to be better than the *previous* test coverage. Dalal, et al. [19] presents two case studies using the ATEG on Bellcore's Integrated Services Control Point (ISCP). Similar studies by Dalal, et al. are presented in [20]. In all of these studies the pairwise testing technique appears to be effective in finding software faults. Some studies compare the fault detection of the pairwise testing with a less effective concurrently executed testing process. (Note that the AETG™ is a sophisticated combinatorial testing tool that provides complete n-way testing capabilities that can accommodate any combinatorial testing strategy; however, in most reported case studies it is used to generate pairwise test sets.)

The results of these case studies are difficult to interpret. In most of the studies, pairwise testing performs better than *conventional*, *concurrent*, or *traditional* testing practices; however, these practices are never described in enough detail to understand the significance of these results (some studies seem to be comparing pairwise testing to 1-way or *all values* testing). Additionally, little is known about the characteristics of software systems used in the studies (e.g., how much combinatorial processing do the systems perform?), and no information is supplied about the skill and motivation of the testers, nor about the kinds of bugs that are considered important. Clearly such factors have a bearing on the comparability of test projects.

A great deal has been written about achieving coverage using pairwise testing [7, 8, 15, 20-24]. "Coverage" in the majority of the studies refers to code coverage, or the measurement of the number of lines, branches, decisions or paths of code executed by a particular test suite. High code coverage has been correlated with high fault detection in some studies [25]. In general, the conclusion of these studies is that testing *efficiency* (i.e., amount time and resources required to conduct testing) is improved because the much smaller pairwise test suite achieves the same level of coverage as larger combinatorial test suites. While testing efficiency improves, we do not know the exact impact on the defect removal efficiency of the testing process. When executing combinatorial testing, one believes that executing different combinations of test data

values may expose faults. Therefore, one is not satisfied with a single execution of a hypothetical path through the code (as one is in code coverage testing), but rather may execute the same path many times with different data value combinations in order to expose combinatorial faults. Consequently, we find the use of code coverage to measure the efficiency of pairwise testing to be confusing. If one believes that code coverage is adequate for testing a product, one would not execute combinatorial testing. One would manually construct a white box test set that is almost certainly guaranteed to be smaller than the pairwise test set.

What do we know about the defect removal efficiency of pairwise testing? Not a great deal. Jones states that in the U.S., on average, the defect removal efficiency of our software processes is 85% [26]. This means that the combinations of all fault detection techniques, including reviews, inspections, walkthroughs, and various forms of testing remove 85% of the faults in software before it is released.

In a study performed by Wallace and Kuhn [27], 15 years of failure data from recalled medical devices is analyzed. They conclude that 98% of the failures could have been detected in testing *if* all pairs of parameters had been tested (they didn't execute pairwise testing, they analyzed failure data and speculate about the type of testing that would have detected the defects). In this case, it appears as if adding pairwise testing to the current medical device testing processes could improve its defect removal efficiency to a "best in class" status, as determined by Jones [26].

On the other hand, Smith, et al. [28] present their experience with pairwise testing of the Remote Agent Experiment (RAX) software used to control NASA spacecraft. Their analysis indicates that pairwise testing detected 88% of the faults classified as correctness and convergence faults, but only 50% of the interface and engine faults. In this study, pairwise testing apparently needs to be augmented with other types of testing to improve the defect removal efficiency, especially in the project context of a NASA spacecraft. Detecting only 50% of the interface and engine faults is well below the 85% U.S. average and presumably intolerable under NASA standards. The lesson here seems to be that one cannot blindly apply pairwise testing and expect high defect removal efficiency. Defect removal efficiency depends not only on the testing technique, but also on the characteristics of the software under test. As Mandl [4] has shown us, analyzing the software under test is an important step in determining if pairwise testing is appropriate; it is also an important step in determining what additional testing technique should be used in a specific testing situation.

Given that we know little about the defect removal efficiency of pairwise testing when applied in real software projects, we would at least like to know that it performs better than random testing. That is, our well planned, well reasoned best practice will remove more faults from software than just randomly selecting test cases. Unfortunately, we do not even know this for certain. In a controlled study in an academic environment, Schroeder, et al. [29] (referred to in this paper as Schroeder's study) found *no significant difference* in the detection removal efficiency of pairwise test sets and same-size randomly selected test sets. There was also no difference in the defect removal efficiency of 3-way and 4-way test sets when compared with random test sets (random test sets in this study where generated by randomly selecting from the full combinatorial test set without replacement; the same test data values were used to create the n-way and random test sets).

Schroeder's study is only a single study and the study was conducted using fault injected from a fault model (rather than real faults made by developers), but the results are distressing. Once again our best testing techniques appear to work no better than random selection [30]. The mystery disappears when we look closely at the random and pairwise test set. It turns out that random test sets are very similar to pairwise test sets! A pairwise test set covers 100% of the combinations of the selected values for each pair of input variables. For the two systems used in the study, random combinatorial test sets, on average, covered 94.5% and 99.6% of test data combination pairs [29].

Dalal and Cohen [15] present similar results in the analysis of pairwise and random test data sets generated for several hypothetical cases in which they varied both the number of input parameters and the number of selected test data values. For small test sets, they found that random sets covered a substantial proportion of pairs. For a test set of size 4, the random set covered on 68.4% of the pairs of data combinations; for a test set of size 9 random covered 65.4% of the pairs of data combinations. For 18 other test sets ranging in size from 10 to 40 test cases, the coverage of pairs by random sets ranged from 82.9% to 99.9% with an average coverage of 95.1%. White [23] provides a counter example that suggests that for very large test data sets, pairwise sets of size 121, 127, and 163, randomly selected test sets would not cover a high percentage of pairs of data combinations.

When one of the authors of this paper (Schroeder) confronted the other (Bach) with this data, Bach didn't initially believe it. "I had, after all, written a program to do pairwise testing, and promoted the technique in my testing classes," says Bach. Bach continues:

> *It just stands to reason that it would outperform mere random testing. So, with Schroeder driving me to back the airport after giving me a tour of his lab in Milwaukee, and the guy going on and on about pairwise testing being less useful than it appears, I pulled out my computer and modified my pairwise tool to choose tests randomly. To my surprise, the random selections covered the pairs quite well. As I expected, many more random tests were needed to cover all the pairs, but the random tests very quickly covered most of them. Schroeder seemed enormously pleased. I was shocked.*

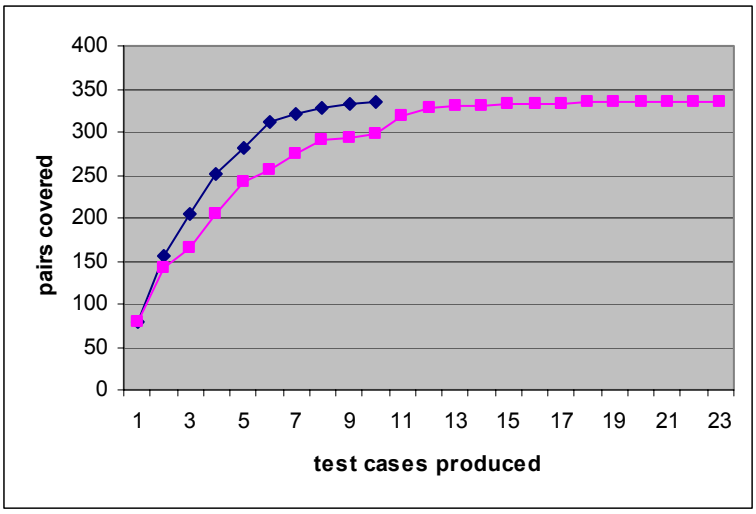


Figure 3. Pairwise vs. Random for the MS Word Options example

Figure 3 shows a comparison of pairwise and random testing in terms of covering pairs over a set of test cases. This is the MS Word Options dialog example that requires 12,288 cases to cover all combinations. The shorter curve is pairwise. In both cases we see a cumulative exponential distribution. The pairwise example is a steeper, shorter curve, but notice how even though the random selection requires double the test cases to complete, that's mainly due to the difficulty of finding the last few pairings. At the time the pairwise test set is complete, the random set has already covered 88% of the pairs. When you look at the coverage growth curve instead of merely the number of test cases to cover 100% of the pairs, it becomes clear why pairwise coverage algorithms aren't necessarily much better than simple random testing.

In additional research conducted by Bach, the same basic pattern in eight different and widely varying scenarios for test sets ranging up to 1200 test cases.

How Pairwise Testing Fails

Based on our review the literature and additional work in combinatorial testing, we feel the pairwise "story" needs to be re-written.

In Schroeder's study, two software systems (the Loan Arranger System (LAS) and the Data Management and Analysis System (DMAS)) were injected with faults. Each system was then tested with n-way test sets where n = 2, 3, and 4. Ten n-way test set of each type were generated using a greedy algorithm similar to that presented by Cohn, et al. [7].

Table 2. Fault classification for injected faults

Fault Type	LAS	DMAS
2-way	30	29
3-way	4	12
4-way	7	1
> 4-way	7	3
Not Found	34	43

Table 2 categorizes the faults in Schroeder's study, based on the n-way test set that exposed the fault. Faults in the "> 4-way" category were found by some 4-way test suites, indicating that data combinations of 5 or more values may be required to ensure the fault is detected.

Pairwise testing fails when you don't select the right values to test with.

The faults in the "not found" category in Table 2 are particularly interesting. Additional analysis of these faults shows that the majority of them are one-way faults. Since pairwise testing subsumes one-way testing, why weren't they found? Not only were they not detected by the pairwise tests, they weren't detected by any of the n-way test sets and would not be detected even if the full combinatorial test set was executed on the particular values we chose for those variables. This is because the test data value required to expose the fault was not selected. Similarly, a number of "not found" faults were 2-way faults that were not detected because a particular combination of data values had not been selected. Ironically, the statement that "most field faults are caused by the interaction of 1 or 2 fields [16]" is true; however, in this study it is true before *and after* pairwise testing is executed.

It is not true that a pairwise test set "protects against any incorrect implementation of the code involving any pairwise interaction … [15];" pairwise testing can only protect against pairwise interactions of *the input values you happen to select for testing*. The selected values are normally an extremely small subset of the actual number of possible values. And if you select values individually using domain analysis, you might miss productive test cases that contain special-case data combinations and error-prone combinations. The weaknesses of domain partitioning are carried over into pairwise testing and magnified.

Pairwise testing fails when you don't have a good enough oracle.

The selection problem is not an issue for the MS Word Options example. In that case we select all the values for testing, leaving out none. Does that mean pairwise testing will discover the important combinatorial bugs? The answer is still no, because of the oracle problem. The "oracle problem" is the challenge of discovering whether a particular test has or has not revealed a bug. In the Options case, we set the options a certain way, but we then have to use Word for a while to see not only whether all the options are behaving correctly, but also whether any other aspect of Word has failed. There's no sure way to know that some subtle problem, say a progressive corruption of the document, is not occurring. We simply do a little tour of Word and

hope for the best. Maybe among the pillars of some testing Olympus it is possible to specify coverage conditions and assume that any failures that happen will be immediately obvious, but on Earth it doesn't work that way. The pairwise test sets produced by our tools or orthogonal arrays can't help us solve that particular problem.

Pairwise testing fails when highly probable combinations get too little attention.

Another issue that is evident in the Options example is the non-uniformity of input distribution. Each set of options is not equally likely to be selected by users. Some are more popular. While we can't know the exact probability, perhaps there is one particular set of options, out of the 12,288 possibilities, that is far more likely to occur than any of the others: the default set. If the percentage of people who never change their options isn't 95%, it must be something close to that, considering that most people are not professional desktop publishers. Therefore, it might be more profitable to cluster our tests near that default. If we tested the default option set, then changed only one variable at a time from the default, that would be 14 tests. If we changed two variables at a time, that would be something on the order of 100 tests. But those combinations would be more realistic than ones that ignored the existence of default input. Of course, we can do both pairwise testing and this default input mutation testing. It's not one or the other, but the point is that pairwise testing might *systematically* cause us to ignore highly popular combinations.

Pairwise testing fails when you don't know how the variables interact.

A key factor in the success or failure of pairwise testing is the degree of interaction among the input variables in creating outputs. A testing practitioner may apply the pairwise technique to two different programs and have wildly different outcomes because of the combinatorial characteristics of the software being testing. Some software programs combine only a small number of input variables in creating outputs; some software programs combine a large number of input variables (e.g., 8 to 12) in creating outputs. For example, Figure 4 displays a program that has three input variables (X, Y, Z) and three outputs numbered 1 through 3. The arrows in the diagram indicate how input variables are combined to create program outputs. For example, inputs X and Y are combined in creating output #1 and inputs X and Z are combined in creating output #2. For this program, pairwise testing seems to be a highly appropriate choice. One would expect that executing pairwise testing on this program to be effective, and that use of this program in a case study of the effectiveness of pairwise testing would yield a very positive result.

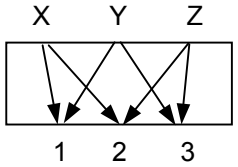


Figure 4. A program with 3 inputs and 3 outputs

What happens when outputs are created by the interaction of more the 2 input variables? Is pairwise testing still the best strategy? Figure 5 presents the interaction of input variables in creating output values for the DMAS system used in Schroeder's study. For the combinatorial operation being tested, DMAS had 42 output fields (numbered along the Y axis in Figure 5). The X axis in Figure 5 indicates how many input variables influence, or interact to create, each output value. The figure indicates that output #1, for example, is created using the value of only 1 input variable, and that output #42 is created by combining 10 different input variables.

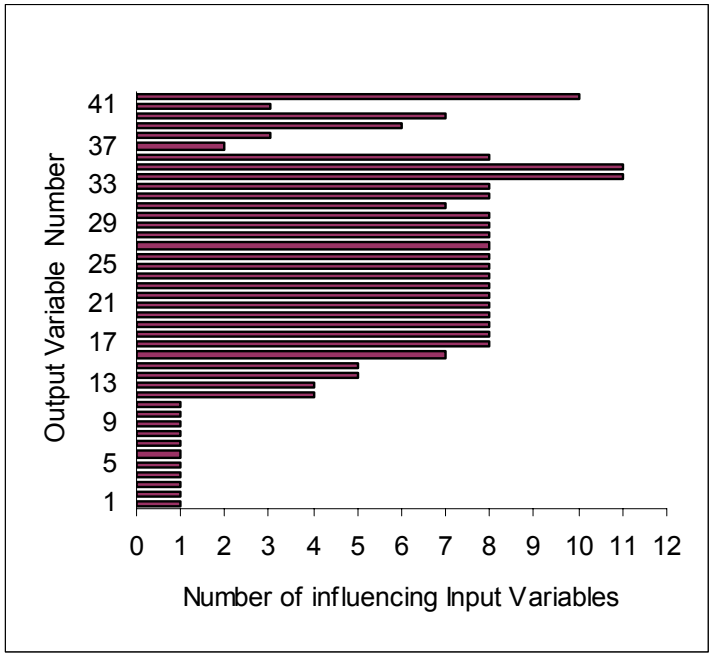


Figure 5. Number of inputs influencing an output for DMAS

Figure 5 indicates that DMAS has many outputs that are created by combining more than 2 input variables. Twenty of the program's output values are created by combining eight or more of the programs input variables. When an output is created from more than 2 input variables, it becomes possible for developers to create faults that may not be exposed in pairwise testing, but may be exposed only by higher-order test data combinations. For example, when testing DMAS in Schroeder's study, twelve 3-way faults, one 4-way fault, and three faults requiring combinations of 5 or above were detected. Given this understanding of how DMAS' input variables interact, some testers may feel that pairwise testing does not adequately address that risk inherent in the software and may focus on high-order interactions of the program's input variables.

The problem, as we see it, is that the key concept of how program inputs variables interact to create outputs is missing from the pairwise testing discussion. The pairwise testing technique does not even consider outputs of the program (i.e., the definition and application of the pairwise technique is accomplished without any mention of the program's outputs). In some instances, blindly applying pairwise testing to combinatorial testing problems may increase the risk of delivering faulty software. While the statement "most field faults are caused by the interaction of 1 or 2 fields [16]" may be true, testing practitioners must be cautioned that in the case of

pairwise testing, detecting "most" faults may result in a defect removal efficiency closer to the 50% experienced by Smith, et al. [28] rather than to the 98% projected by Wallace and Kuhn [27].

If pairwise testing is anointed as a best practice, it seems to us to become easier for novice testers, and experienced testers alike, to blindly apply it to every combinatorial testing problem, rather than doing the hard work of figuring out how the software's inputs interact in creating outputs. Gaining a black-box understanding of a program's interactions for a given set of test data values is not an easy task. Testers currently do this analysis in the process of determining the expected results for their test cases. That is, to correctly determine an expected output of a test case one must know which inputs are used in creating the output. This is an error prone process and automation can sometimes be used to determine how a program's inputs interact. The display in Figure 5, for example, was created using a technique called input-output analysis [31], which is an automated approach to determining how a program's input variables influence the creation of the program's outputs.

Conclusion

The typical pairwise testing story goes like this:

1) Pairwise testing protects against pairwise bugs
2) *while* dramatically reducing the number tests to perform,
3) *which is especially cool because* pairwise bugs represent the majority of combinatoric bugs,
4) *and* such bugs are a lot more likely to happen than ones that only happen with more variables.
5) *Plus,* you no longer need to create these tests by hand.

Critical thinking and empirical analysis requires us to change the story:

1) Pairwise testing **might find some** pairwise bugs
2) *while* dramatically reducing the number tests to perform, **compared to testing all combinations, but not necessarily compared to testing just the combinations that matter.**
3) *which is especially cool because* pairwise bugs **might** represent the majority of combinatoric bugs, **or might not, depending on the actual dependencies among variables in the product.**
4) *and* **some** such bugs are more likely to happen than ones that only happen with more variables, **or less likely to happen, because user inputs are not randomly distributed.**
5) *Plus,* you no longer need to create these tests by hand, **except for the work of analyzing the product, selecting variables and values, actually configuring and performing the test, and analyzing the results.**

The new story may not be as marketable as the original, but it considerably more accurate. Sorting it out, there are at least seven factors that strongly influence the outcome of your pairwise testing:

1) The actual interdependencies among variables in the product under test.
2) The probability of any given combination of variables occurring in the field.
3) The severity of any given problem that may be triggered by a particular combination of variables.
4) The particular variables you decide to combine.
5) The particular values of each variable you decide to test with.
6) The combinations of values you actually test.
7) Your ability to detect a problem if it occurs.

Items 1 through 3 are often beyond your control. You may or may not have enough technical knowledge to evaluate them in detail. If you don't understand enough about how variables might interact, or you test with combinations that are extremely unlikely to occur in the field, or if the problems that occur are unimportant, then your testing will not be effective.

Items 4 and 5 make a huge difference in the outcome of the testing. Item 4 is directly dependent on item 1. Item 5 is also dependent on item 1, but it also relates to your ability to find relevant equivalence classes. Similarly, item 7 depends on your ability to determine if the software has indeed failed.

The pairwise testing process, as described in the literature, addresses only item 6. That is not much help! Item 6 is solved by the pairwise testing tool you are using. Because of the availability of tools, this is the only one of the seven factors that is no longer much of a problem.

Pairwise testing permeates the testing world. It has invaded our conferences, journals, and testing courses. The pairwise story is one that all testers would like to believe: a small cleverly constructed set of test cases can do the work of thousands or millions of test cases. As appealing as the story is, it is not going to be true in many situations. Practitioners must realize that pairwise testing does not protect them from all faults caused by the interaction of 1 or 2 fields. Practitioners must realize that blindly applying pairwise testing may increase the risk profile of the project.

Should you adopt pairwise testing? If you can reduce your expectations of the technique from their currently lofty level the answer is *yes*. Pairwise testing should be one tool in a toolbox of combinatorial testing techniques. In the best of all possible worlds, a tester selects a testing strategy based on the testing problem at hand, rather than fitting the problem at hand to a pre-existing test strategy [10].

Develop Skill, Take Ownership, but Do Not Trust "Best Practice"

All useful and practical test techniques are heuristic: they are shortcuts that might help, but they do not guarantee success. Because test techniques are heuristic, there can be no pat formulas for great testing. No single test technique will serve; we are obliged to use a diversified strategy. And to be consistently successful requires skill and judgment in the selection and application of those techniques.

We, the authors, both began with the assumption, based on a good-sounding story, that pairwise testing is obviously a good practice. By paying attention to our experiences, we discovered that our first impression of pairwise testing was naïve.

Because test techniques are heuristic, focus not on the technique, but your skills as a tester. Develop your skills.

We recommend that you refuse to follow any technique that you don't yet understand, except as an experiment to further your education. You cannot be a responsible tester and at the same time do a technique just because some perceived authority says you should, unless that authority is personally taking responsibility for the quality of your work. Instead, relentlessly question the justification for using each and any test technique, and try to imagine how that technique can fail. This is what we have done with pairwise testing.

Don't follow techniques, own them.

Achieving excellence in testing is therefore a long-term learning process. It's learning for each of us, personally, and it's also learning on the scale of the entire craft of testing. We the authors consider pairwise testing to be a valuable tool, but the important part of our story is how we came to be wiser in our understanding of the pairs technique, and came to see its popularity in a new light.

We believe testers have a professional responsibility to subject every technique they use to this sort of critical thinking. Only in that way can our craft take its place among other respectable engineering disciplines.

References

[1] C. Kaner, J. L. Falk, and H. Q. Nguyen, *Testing Computer Software*, 2nd ed. New York: Van Nostrand Reinhold, 1993.

[2] C. Kaner, "Teaching Domain Testing: A Status Report," presented at Conference on Software Engineering Education & Training, Norfolk, Virginia, 2004.

[3] British Computer Society, "Information Systems Examinations Board Practitioner Certificate in Software Testing Guidelines and Syllabus, Ver. 1.1," http://www.bcs.org/BCS/Products/Qualifications/ISEB/Areas/SoftTest/Syllabus.htm.

[4] R. Mandl, "Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing," *Communication of the ACM*, vol. 28, no. 10, pp. 1054-1058, 1985.

[5] A. S. Hedayat, N. J. A. Sloane, and J. Stufken, *Orthogonal Arrays: Theory and Applications*. New York: Springer-Verlag, 1999.

[6] J. M. Harrel, "Orthogonal Array Testing Strategy (OATS) Technique," http://www.seilevel.com/OATS.html, 05/28/2004.

[7] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437-444, 1997.

[8] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The Combinatorial Design Approach to Automatic Test Generation," *IEEE Software*, vol. 13, no. 5, pp. 83-88, 1996.

[9] A. L. Williams and R. L. Probert, "A practical strategy for testing pairwise coverage at network interfaces," in *Proc. IEEE ISSRE: Int'l Symp. Softw. Reliability Eng*. White Plains, NY, 1996.

[10] R. D. Craig and S. P. Jaskiel, *Systematic Software Testing*. Boston: Artech House, 2002.

[11] L. Copeland, *A Practitioner's Guide to Software Test Design*. Boston, MA: Artech House Publishers, 2003.

[12] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing: A Context Driven Approach*. New York: John Wiley & Sons, Inc., 2002.

[13] S. Splaine and S. P. Jaskiel, *The Web Testing Handbook*. Orange Park, FL: STQE Publishing, 2001.

[14] J. D. McGregor and D. A. Sykes, *A Practical Guide to Testing Object-Oriented Software*. Boston, MA: Addison-Wesley, 2001.

[15] S. Dalal and C. L. Mallows, "Factor-Covering Designs for Testing Software," *Technometrics*, vol. 50, no. 3, pp. 234-243, 1998.

[16] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton, "The Automatic Efficient Test Generator (AETG) System," in *Proc. of the 5th Int'l Symposium on Software Reliability Engineering*: IEEE Computer Society Press, 1994, pp. 303-309.

[17] Y. Lei and K. C. Tai, "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing," in *Proc. of the 3rd IEEE High-Assurance Systems Engineering Symposium*, 1998, pp. 254-261.

[18] R. Brownlie, J. Prowse, and M. Phadke, "Robust Testing of AT&T PMX/StarMail Using OATS," *AT&T Technical Journal*, vol. 71, no. 3, pp. 41-47, 1992.

[19] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, and C. Lott, "Model-Based Testing of a Highly Programmable System," in *Proceedings of the Nineth International Symposium on Software Reliability Engineering (ISSRE 98)*. Paderborn, Germany, 1998, pp. 174-178.

[20] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. Lott, G. C. Patton, and B. M. Horowitz, "Model-Based Testing in Practice," in *Proceedings of the International Conference on Software Engineering*. Los Angeles, CA, 1999, pp. 285-294.

[21] K. Burroughs, A. Jain, and R. L. Erickson, "Improved Quality of Protocol Testing Through Techniques of Experimental Design," in *Proc. Supercomm./IEEE International Conference on Communications*, 1994, pp. 745-752.

[22] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino, "Applying Design of Experiments to Software Testing," in *Proceedings of the Nineteenth International Conference on Software Engineering*. Boston, MA, 1997, pp. 205-215.

[23] L. J. White, "Regression Testing of GUI Event Interactions," in *Proceedings of the International Conference on Software Maintenance*. Washington, 1996, pp. 350-358.

[24] H. Yin, Z. Lebne-Dengel, and Y. K. Malaiya, "Automatic Test Generation using Checkpoint Encoding and Antirandom Testing," in *Proceedings of the Eighth International Symposium on Software Reliability Engineering (ISSRE '97)*. Albuquerque, NM, 1997, pp. 84-95.

[25] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of Test Set Size and Block Coverage on Fault Detection Effectiveness," in *Proceedings of the Fifth IEEE International Symposium on Software Reliability Engineering*. Monterey, CA, 1994, pp. 230-238.

[26] C. Jones, *Sofware Assessments, Benchmarks, and Best Practices*. Boston, MA: Addison Wesley Longman, 2000.

[27] D. R. Wallace and D. R. Kuhn, "Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data," *Int'l Jour. of Reliability, Quality and Safety Engineering*, vol. 8, no. 4, pp. 351-371, 2001.

[28] B. Smith, M. S. Feather, and N. Muscettola, "Challenges and Methods in Testing the Remote Agent Planner," in *Proc. 5th Int'l Conf. on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, 2000, pp. 254-263.

[29] P. J. Schroeder, P. Bolaki, and V. Gopu, "Comparing the Fault Detection Effectiveness of N-way and Random Test Suites," in *Proc. of the Int'l Symposium on Empirical Software Engineering*. Redondo Beach, CA, 2004.

[30] D. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 202-213, 1990.

[31] P. J. Schroeder, B. Korel, and P. Faherty, "Generating Expected Results for Automated Black-Box Testing," in *Proc. of the Int'l Conf. on Automated Software Engineering (ASE2002)*. Edinburgh, UK, 2002, pp. 139-48.

Design Patterns for Customer Testing

Misha Rybalov, Quality Centered Developer
autotest@gomisha.com

Abstract

This paper presents design patterns for automated customer tests. As applications continue to evolve, these patterns help lower maintenance costs and facilitate the creation of new customer tests. Each design pattern includes a definition, a discussion of the pattern details highlighting its suitability for customer test automation, and a code sample illustrating its use. These design patterns can be used regardless of programming language and test tool. As such, specific test tool code is omitted in this paper and for consistency examples are given in Java, using the JUnit testing framework. This paper aims to illustrate that test code should be treated with the same importance as application code. This assertion has the following consequences:

- tests should have a clean design that facilitates code reuse without duplication
- tests should be easily adaptable to modifications in the application
- tests should be easy to create
- tests should have a low maintenance cost

Using application development design principles, automated test development can become a well factored art form that is adaptable to application changes. This adaptability results in lowered maintenance costs and easier test creation.

About the Author

Misha Rybalov is a quality centered developer that works with Development and Quality Assurance departments to create automated testing frameworks and build quality into the product from the start of the development cycle. He has automated the testing of a wide array of projects including web, .NET, and Java customer tests, as well as standard unit tests, and a multi-threaded security system. Misha's passion is designing and building automated testing frameworks using design patterns. He is the author of the AbcUnit web testing tool (www.abcunit.org) and can be reached at autotest@gomisha.com.

Design Patterns for Customer Testing, Copyright © 2004 by Misha Rybalov

Introduction

Automated testing offers numerous benefits to any software organization, including finding defects cost-effectively early in the development cycle, providing rapid feedback, and giving developers the courage to refactor their code. With the recent popularization of Test Driven Development,[1] automated testing has also become a design tool for application development.

Ironically, with all the attention paid to the benefits of automated testing, the actual art of designing and writing automated tests is often assumed to be taken care of by the test tool or the test writer. The principles of simple design, refactoring, and design patterns that are so common in software development are under utilized when it comes to test development.

While unit testing involves very simple method calls and checking for expected results, customer tests are significantly more complicated. It is quite common to have a single customer test scenario involving dozens of navigations and checks which span multiple screens and modules. The consequences of a haphazard approach to automated customer testing can result in massive code duplication, tests that are cumbersome to create, costly to maintain, and buggy.

This paper presents a catalog of design patterns that facilitate the creation and maintenance of automated customer tests. Each pattern consists of 1) a definition, 2) a discussion of the pattern details highlighting its suitability for customer test automation, and 3) a code sample illustrating its use.

This paper uses a fictional web-based shopping cart and catalog application to demonstrate how each pattern can be implemented. We will test the application's capabilities of searching for, adding, and removing items from the cart, as well as checking out and processing coupons.

Test Tools and Code

The design patterns introduced in this paper are not test tool specific in any way. One of the goals of applying these patterns is to decouple test code from the test tool code. For our purposes, we will use the JUnit testing framework[2] to demonstrate the application of these patterns. These tests could just as easily have been written with another object-oriented testing framework (see Extreme Programming Software for a list of object-oriented testing frameworks[3]).

Customer Tests

This paper uses the term "customer tests" to refer to tests that exercise the application from an end-user's perspective. These types of tests are also referred to as functional or acceptance tests. Customer testing can be performed through a web browser (e.g. Internet Explorer, Netscape), an operating system's graphical user interface (e.g. Windows, Unix, MacOS), or any other user interface (e.g. Java, .NET). The patterns presented in this

paper need not be limited to customer tests - they can be successfully applied to unit tests as well. This paper, however, focuses on customer tests because they are inherently more complex than unit tests and thus will derive the greatest benefit from design patterns.

Why Use Design Patterns to Automate Customer Tests?

Design patterns are used "to build flexible, reusable software."[4] An automated test is, in essence, a program that checks another program. Consequently, it is vulnerable to the same design problems and entropy as application code. These vulnerabilities, often referred to as "code smells,"[5] include test code duplication, tight coupling between the application and tests, and long test methods.

Traditionally, design patterns were not used for test code because test code was considered more simplistic and subordinate to application code. Also, most test tools generate customer test code through a record-and-playback mechanism (i.e. they record all user actions as the tester navigates through the application, create the corresponding code, and then play it back as the test), which encourages test developers to use the tool generated code without modifications. The challenge with both of these approaches is that tests become more difficult to maintain as the application changes over time. Design patterns help us minimize maintenance costs by making tests easier to update and more adaptable to application changes.

The patterns presented in this paper have been successfully used on real projects. These patterns can be applied in any object-oriented language such as Java, C++ or C#. They can also be used to refactor auto generated code from a commercial testing tool, but the mostly procedural nature of tool languages yields less benefit than an object-oriented language.

Patterns Catalog

I. Basic Patterns

This group of patterns applies object-oriented programming principles to reduce test code maintenance costs and facilitate new test creation. The patterns included are Template, Object Genie, Domain Test Object, and Transporter.

1. Template

Definition

Group a common sequence of steps for testing a module into a class and have subclasses specify unique test case data without changing the main algorithm's structure.

Discussion

Customer tests involve navigating to a particular part of the application and trying different scenarios. Even though each scenario is unique, there are still many

commonalities between test cases. For instance, the way that you navigate to the module and the assertions that are made along the way can be grouped into one place - the Template class, which is a test-specific application of the Template Method design pattern.[6]

When the application inevitably changes, the majority of updates will occur in the Template class instead of within the test cases. This is because the Template class encapsulates navigation and assertion logic, which is more vulnerable to application changes than the data contained in test cases.

After each update to the Template class, the modifications will cascade automatically to all subclasses. This has the added benefit of making each test case (subclass) simpler since it is only required to specify the unique aspects of the test instead of the entire sequence of test steps and assertions. This reduces test code duplication and standardizes common navigations and assertions of related test cases.

If not using this design pattern, then every time the application changed or a new test case was added, the test writer would have to copy or re-record all the navigations and assertions that are part of each test. In that case, the automation effort would incur serious maintenance costs.

Since the Template pattern is adaptable to application changes, it can be used early in the development cycle without having to wait for application completion. Templates can be created and then continually updated as features are developed. This lends itself to agile software development practices such as Test Driven Development[7] where tests are written early in the software lifecycle and are an integral part of the development effort.

Example

Figure 1 presents two traditional tests which are then contrasted with Figures 2 and 3 which use the Template pattern. These tests check the functionality of adding and removing items from a shopping cart.

Figure 1: Non-template test cases.

```
import junit.framework.*;

public class ShoppingCartTest extends TestCase {
        public ShoppingCartTest(String pName) { super(pName); }
        public static Test suite() { return new TestSuite(ShoppingCartTest.class); }
        public void testAddItem() {  //1st test case
                login();
                searchForItem1();
                addItem1ToCart();
                searchForItem2();
                addItem2ToCart();
                logout();
        }
```

```
        public void testRemoveItem() {  //2nd test case
                login();
                searchForItem1();
                addItem1ToCart();
                addItem1ToCart();
                removeItem1FromCart();
                searchForItem2();
                addItem2ToCart();
                removeItem2FromCart();
                logout();
        }
        //test tool specific implementations of methods
        //(e.g. login(), searchForItem1(), etc.)
}
```

The first test logs in, searches for item one, adds it to the cart, searches for item two, adds it to the cart, and logs out. The second test logs in, searches for item one, adds it twice to the cart, removes one quantity of item one from the cart, searches for item two, adds it to the cart, removes item two from the cart, and logs out. These two traditional tests have the following common elements: logging in, searching for item one, searching for item two, and logging out. These common steps can be moved into a new Template class.

Figure 2: Template class.

```
import junit.framework.*;

public abstract class ShoppingCartTemplate extends TestCase {
        public ShoppingCartTemplate(String pName) { super(pName); }

        protected void setUp() {
                login();
                searchForItem1();
                cartAction1();  // <------ test hook 1
                searchForItem2();
                cartAction2(); //<-------- test hook 2
                logout();
        }

        //Subclasses are supposed to implement these test hooks anyway they like.
        abstract protected void cartAction1();
        abstract protected void cartAction2();

        //test tool specific implementations of methods
        //(e.g. login(), searchForItem1(), etc.)
}
```

The Template defines the main sequence of steps as 1) login, 2) search for item one, 3) do something (to be defined by subclasses), 4) search for item two, 5) do something else (to be defined by subclasses), and 6) log out. Note that JUnit's setUp() method is used to define the sequence of steps, which enables this sequence of steps to occur for every test case subclass.

Each test case can now be moved to a separate subclass and implement the abstract methods of the Template.

Figure 3: Two test cases using the Template.

```
public class ShoppingCartTest_AddItem extends ShoppingCartTemplate {
        public ShoppingCartTest_AddItem(String pName) { super(pName); }
        protected void cartAction1() {
                addItem1ToCart();
        }
        protected void cartAction2() {
                addItem2ToCart();
        }
}

public class ShoppingCartTest_RemoveItem extends ShoppingCartTemplate {
        public ShoppingCartTest_RemoveItem(String pName) { super(pName); }
        protected void cartAction1() {
                addItem1ToCart();
                addItem1ToCart();
                removeItem1FromCart();
        }
        protected void cartAction2() {
                addItem2ToCart();
                removeItem2FromCart();
        }
}
```

The Add test case (ShoppingCartTest_AddItem) implements the two abstract methods by adding items one and two to the cart, respectively. The Remove test case (ShoppingCartTest_RemoveItem) implements the first abstract method by adding item one twice and then removing it. It implements the second abstract method by adding item two to the cart and then removing it.

Even though it appears that there is not a great reduction in code size in this example, code consolidation becomes more evident in real life scenarios which contain dozens of test steps and hundreds of test cases.

Adding more tests is relatively easy with the Template pattern since one only needs to implement the two abstract methods of the Template (i.e. cartAction1() and cartAction2()) rather than having to write all the setup and navigation code. This

consolidation of redundant test steps into one place can drastically reduce maintenance costs as the application continues to evolve. For instance, if the application were to change so that customers now had to go to a specials screen before being able to add any items to the their cart, our Template can be updated to account for this and have the change cascade to every test without having to modify each test individually.

Figure 4: Modification of Template.

```
import junit.framework.*;

public abstract class ShoppingCartTemplate extends TestCase {
        public ShoppingCartTemplate(String pName) { super(pName); }

        protected void setUp() {
                login();
                gotoSpecials();    // <------ new test step added
                searchForItem1();
                cartAction1();     // <------ test hook 1
                searchForItem2();
                cartAction2();     // <------ test hook 2
                logout();
        }
        abstract protected void cartAction1();
        abstract protected void cartAction2();
}
```

2. Object Genie

Definition

Relieve tests from having to instantiate and initialize common non-trivial objects by having them ask for those objects from a central authority that grants objects in a pre-defined state. This pattern is also known as ObjectMother.[8]

Discussion

There are many occasions during testing when multiple tests will require the same object and then each proceed to interact with it in different ways. If it is a simple object, each test can just instantiate it normally (SomeObject object = new SomeObject();) However, there are many occasions where the object that is needed by tests is cumbersome to instantiate or initialize.

The Object Genie consolidates the construction and initialization of that object to just one place, thereby reducing future maintenance costs and facilitating the creation of new tests. If the way an object is instantiated or initialized ever changes, the update would only be done in the Object Genie and the test cases that use that Object Genie would be unaffected.

Example

In our shopping cart application, multiple tests (e.g. test deleting an item from a cart, test the checkout process, test processing a coupon) require a shopping cart with a few items in it. Without the Object Genie pattern, each test would have to re-create and re-populate the shopping cart, increasing code duplication and maintenance costs. Using an Object Genie, a test can request a non-empty shopping cart and then proceed with testing. The Object Genie handles the instantiation of the shopping cart and populates it. In Figure 5, the Object Genie populates the shopping cart with two items (orange juice and cereal).

Figure 5: Defining a shopping cart Object Genie.

```
public class CartGenie {
        public static ShoppingCart getNonEmptyCart() {
                ShoppingCart shoppingCart = new ShoppingCart();
                Item ojItem = searchForItem("orange juice");
                shoppingCart.add(ojItem);
                Item cerealItem = searchForItem("cereal");
                shoppingCart.add(cerealItem);
                return shoppingCart;
        }
        private static Item searchForItem(String pItemId) {
                //search for the item based on the item id
        }
}
```

Each test would call CartGenie.getNonEmptyCart() to receive an initialized shopping cart and then proceed with the details of the test. The tests would be decoupled from having to know how a shopping cart is instantiated and populated, thereby reducing future maintenance costs (see Dumb Test in Section II: General Design Principles).

3. Domain Test Object (DTO)

Definition

Encapsulates an application's visual components into objects that can be reused by many tests.

Discussion

We use object-oriented programming (OOP) to model real world objects inside our applications. We can also use OOP to model visual representations of the application (e.g. a shopping cart, a coupon, and a catalog) in our tests. In other words, the application uses objects to model the real world, while the tests use objects to model the application's world.

The DTO pattern groups visual representations of the application into objects that can be used by tests. Each test communicates to the DTO what data is expected but not how the data should appear (i.e. order and name of input fields to which the data applies or

placement of the data on the page). The DTO then checks the test provided data against the actual data in the application. Thus, the DTO allows us to decouple the expected data from its visual representation. When the visual representation of the application changes (e.g. new table columns, new fields, and new labels), the update is done to the DTO instead of to each test. All the tests that use that DTO will be insulated from the change and will not need updating, thereby reducing maintenance costs. In essence, the DTO is a test-specific application of the Model/View/Controller (MVC) design pattern,[9] where the tests are the Model and the DTO is the View.

Example

Customer tests need to check the contents of the shopping cart after certain actions are performed (e.g. adding, removing, updating quantity of cart items). A simple table can represent the shopping cart by displaying the list of items, quantity, unit price and total price.

Table 1: Visual representation of shopping cart contents.

Item	Quantity	Unit Price	Price
Milk	1	$2.99	$2.99
Bread	2	$1.50	$3.00
Total			$5.99

This table is the application's representation of a shopping cart. Every time a test wants to check the cart's contents it must check this table. Thus, any changes to this table (e.g. a new column being added) could have severe test maintenance consequences. To minimize this maintenance cost, we can model the shopping cart table as a DTO and have tests interact with the DTO instead of with the table directly. The following code sample demonstrates checking the shopping cart page without using a DTO. This is then contrasted with doing the same check using a DTO.

Figure 6: Checking shopping cart contents without DTO.

```
public void checkCart_noDto() {
        String [][] expectedCartTableCells = {
                {"Item",  "Quantity", "Unit Price", "Price"},
                {"Milk",  "1",        "$2.99",      "$2.99"},
                {"Break", "2",        "$1.50",      "$3.00"},
                {"",          "",         "",           "",    },
                {"Total",  "",          "",           "$5.99"}
        };
        //tool-specific way to retrieve table cell contents
        String [][] actualCartTableCells = getCartTableCells();

        //check every web table cell of the cart page
        assertEquals(actualCartTableCells.length, expectedCartTableCells.length);
        for(int i=0; i<actualCartTableCells.length; i++) {
```

```
            assertEquals(expectedCartTableCells[i].length, actualCartTableCells[i].length);
            for(int j=0; j<actualCartTableCells[i].length; j++) {
                assertEquals(expectedCartTableCells[i][j], actualCartTableCells[i][j]);
            }
        }
}
```

Without using the DTO pattern, we are forced to specify the expected content in a rigid order that is vulnerable to application changes. Every time we need to check the contents of the shopping cart page, we would have to repeat the same checking code, but with different values. For example, we would need to replicate the same checking code after adding to the cart, removing from a cart, updating the quantity of items in the cart, etc. The maintenance problem occurs when the layout of the cart changes and we would have to update all the checking code for each cart action. For instance, the application might change so that the shopping cart page has a new column called "Discount" which contains percentage values of the discount for an item. The application might also change by having the quantity and unit price columns switched. In both of these cases, all our test code would break because now the web table wouldn't match our expected table values. By using a DTO, we can avoid this problem and be more adaptable to application changes. The following code demonstrates how a DTO object can be used:

Figure 7: Checking shopping cart contents with DTO.

```
public void checkCart_withDto() {
        ShoppingCart shoppingCart = new ShoppingCart();

        //...add/remove/edit cart using other patterns

        ShoppingCartDto shoppingCartDto = new ShoppingCartDto();

        //setup expected values
        shoppingCartDto.setItem("1", "2.99", "2.99", "Milk");
        shoppingCartDto.setItem("2", "1.50", "3.00", "Bread");
        shoppingCartDto.setTotal("5.99");
        shoppingCartDto.check(shoppingCart);
}

public class ShoppingCartDto {
        public void setItem(String pQuantity, String pUnitPrice, String pTotalPrice,
        String pName) {
                //code to store expected item internally would follow
        }

        public void setTotal(String pTotal) {
                //code to store expected total internally would follow
        }
```

```
        public void check(ShoppingCart pShoppingCart) {
                //code to check expected versus actual shopping cart contents would follow
        }
}
```

From Figure 7, you can see that the DTO is a separate class that is instantiated and configured by test cases. Note that the DTO knows how to check itself so tests never have to explicitly specify the column order or any other details about the shopping cart table. The tests simply tell the DTO the items that are expected to be in the cart and then ask the DTO to check the cart contents. Now even if the entire shopping cart page changed so that the cart is no longer represented by a table, the update would be done to the DTO while leaving the tests insulated from such a drastic change.

4. Transporter

Definition

Create a central authority that navigates through the application on behalf of tests.

Discussion

Similar to the Object Genie is the concept of a Transporter. Whereas the Object Genie grants tests whatever objects they need, a Transporter encapsulates navigation code so that tests don't have to know the details of how to move around the application. Instead, tests use the Transporter to navigate throughout the application on their behalf.

If the application changes so that users must now alter the way they navigate to a specific module, the modifications would be localized to the Transporter and changes would automatically cascade to all tests that use that Transporter. This reduces maintenance costs since none of the tests would be affected by this change and facilitates creating new tests since there is less test code to write.

Example

Each test needs a way to navigate to pages and then perform specific actions (e.g. log in, enter a coupon, buy an item on special, etc.). To use a coupon, one must first do the following navigations: login and navigate to the checkout page. There must also be items in the cart to which the coupon applies - this is taken care of by the Object Genie. The following code samples demonstrate the creation and use of a Transporter that navigates to the login and checkout pages.

Figure 8: Defining a Transporter.

```
public class Transporter {
        public static void login() {
                //Tool specific code to navigate to login screen and login to the application
        }
```

```
        public static void gotoCheckout() {
                //Tool specific code to navigate to the checkout page
        }
}
```

Figure 9: Using a Transporter.

```
import junit.framework.*;

public class CouponTest extends TestCase {
        public CouponTest(String pName) { super(pName); }

        public static Test suite() { return new TestSuite(CouponTest.class); }

        protected void setUp() {
                Transporter.login();                // <---- loose coupling
                ShoppingCart shoppingCart = CartGenie.getNonEmptyCart();
                Transporter.gotoCheckout();   //<----- loose coupling
        }
        public void testCheckoutWithValidCoupon() {   //tests for a valid coupon
                applyValidCoupon();
                checkCouponAccepted();
        }
        public void testCheckoutWithExpiredCoupon() { //test for an expired coupon
                applyExpiredCoupon();
                checkCouponRejected();
        }
        public void testCheckoutWithInvalidCoupon() {  //test for an invalid coupon
                applyInvalidCoupon();
                checkCouponRejected();
        }
        //… specific implementations of test methods (e.g. applyValidCoupon())
}
```

By moving navigation to the login and checkout pages to a central location, we allow other tests to reuse that functionality. The CouponTest class is now only responsible for testing coupon functionality without worrying about the details of how to get to the coupon page. This loose coupling makes the test more maintainable.

II. General Design Principles

These patterns introduce general test design principles that can be used in conjunction with the basic patterns introduced previously. This group includes Dumb Test, Independent Test, Don't Repeat Yourself, and Multiple Failures.

1. Dumb Test

Definition

Minimize a test's knowledge about navigation, checking logic and user interface details.

Discussion

This principle is commonly used in application programming where decoupling objects from each other allows each object to change without affecting the other. Applying this principle to test development, the Dumb Test pattern focuses on making an individual test know as little about its surroundings as possible. This makes Dumb Tests less vulnerable to changes in the application, which in turn lead to lower test maintenance costs. Each Dumb Test doesn't know the sequence of steps that were taken to get to the component being tested nor about common checking that has been done by other test modules. Ideally, the only things a Dumb Test is aware of are its inputs and its expected outputs.

Dumb tests work well with the Template, Object Genie, DTO, and Transporter patterns since common test steps can be consolidated into these classes. When the application changes, the bulk of the updates will need to be made in those classes, while individual Dumb Tests will be mostly insulated from the changes. This creates very compact tests that are quick to create and cost less to maintain than tests that provide all the details about navigation and assertions. In a nutshell, a Dumb Test doesn't see the big picture.

Example

Both test cases in the Template pattern example are Dumb Tests (see Figure 3). They have no knowledge of how to login, logout, go to the specials page or search for items. If these parts of the application were to ever change, the Dumb Tests would be unaffected by it. The tests only know that they need to add and remove items from the cart.

2. Independent Test

Definition

Each test leaves the application in the same pre-defined state, irrespective of whether the test passed or failed.

Discussion

In order to ensure test integrity, it is important that tests be unaffected by the outcome of other tests. Each test expects to start from a constant application state. This requires each test to de-initialize to this constant state upon completion or failure. If this does not occur then subsequent tests will fail to execute. Unit testing frameworks such as JUnit recognize the importance of this principle and have mechanisms to initialize and de-initialize each test in order to keep the application in a consistent state (i.e. using setUp() and tearDown() methods).

Example

For our sample application, it is important that each test start on the login page, with an empty shopping cart. This necessitates a mechanism for emptying the shopping cart and logging out at the end of each test or upon test failure. If we use the Template pattern, we can insert these common steps of initialization and de-initialization, right into the Template, thereby relieving test writers from having to remember to include these steps in every test and ensuring tests stay independent.

Figure 10: Common initialization and de-initialization steps for each test case.

```
protected void setUp() {
        login();          // <------ common setup code for all test cases
        searchForItem1();
        cartAction1();
        searchForItem2();
        cartAction2();
        emptyCart();  // <------- common tear down code for all test cases
        logout();
}
```

3. Don't Repeat Yourself (DRY)

Definition

Minimize test code duplication to lower test maintenance costs.

Discussion

The evils of code duplication have been documented extensively in software development literature.[10] The same principles that apply to application development also apply to test development. Namely, just as it becomes a maintenance problem to modify application code in multiple places, the same problem occurs when tests need modification in multiple places. Many of the patterns mentioned in this paper (e.g. Template, Object Genie, DTO, Transporter) minimize code duplication by consolidating common test code into one place instead of duplicating it. The advantage of this approach is that it becomes easier to update tests as the application changes, which in turn lowers the maintenance costs of tests.

Example

To test the coupon feature of the application, we need tests for submitting a valid coupon, an expired coupon, and an invalid coupon. In order to test these scenarios, we need to load our cart with items to which the coupon would apply. This would be the setup part of the test (see Independent Test pattern), since before we could test the checkout process, we'd have to have a cart with items in it already. If we don't adhere to DRY, we would simply copy and paste previous test code that searched for an item(s) and inserted it into the cart. Figures 11 and 12 illustrate non-DRY and DRY approaches, respectively.

Figure 11: Repeating test code for coupon test cases.

```
import junit.framework.*;

public class CouponTest extends TestCase {
        public CouponTest(String pName) { super(pName); }

        public static Test suite() { return new TestSuite(CouponTest.class); }

        public void testValidCoupon() {
                login();
                searchForItem1();
                addItem1ToCart();
                searchForItem2();
                addItem2ToCart();
                gotoCheckout();
                applyValidCoupon();
                checkCouponAccepted();
        }

        public void testExpiredCoupon() {
                login();
                searchForItem1();
                addItem1ToCart();
                searchForItem2();
                addItem2ToCart();
                gotoCheckout();
                applyExpiredCoupon();
                checkCouponRejected();
        }

        public void testInvalidCoupon() {
                login();
                searchForItem1();
                addItem1ToCart();
                searchForItem2();
                addItem2ToCart();
                gotoCheckout();
                applyInvalidCoupon();
                checkCouponRejected();
        }
        //tool-specific implementations of methods (e.g. login(), addItem2ToCart(), etc.)
}
```

This non-DRY approach is recommended by some due to its inherent simplicity and explicitness.[11] The argument is that if a test developer tries to get too fancy with their test code, they will have to start writing tests for the test code, which can quickly get out of

hand. However, writing more maintainable code does not mean that the code has to get more complex.

Using DRY, we have several options: 1) use the Template pattern to extract the act of filling up your cart to a super class, 2) use an Object Genie to grant us a cart with items already in it, 3) extract the setup code into a set up method so that each test case uses the same setup code, and 4) combine the above methods.

Template Pattern

Using a Template pattern, we can extract all the setup code into a super class and have each subclass implement a few small methods. All the common code is encapsulated in the super class and the subclasses don't have to concern themselves of dealing with it. The disadvantage, though, is that a separate subclass has to be created for each test case.

Object Genie

An Object Genie can grant us a cart with items in it anytime we want it. The advantage of this technique is that we are not forced to use subclasses. We just request the cart with items in it at the appropriate time.

Setup Method

JUnit supports Setup Methods that allow a test writer to place all common setup code into one place. The setup code gets executed before each test case is run. This is a good start to avoiding duplication, but often the code that gets factored out into the Setup Method could be duplicated in other Setup Methods for other tests. In our example, the setup code of filling a cart with items is needed by other tests, as well. For instance, to test cart manipulation functions (adding, removing, modifying items in the cart) we would need to have a cart with items in it as setup code. This implies that the setup code should be located in a more central location, as outlined in the previous two examples.

Combining Patterns

Filling up a cart is a common test step that is useful to many customer tests. The three pattern options to address this step can be used in combination to yield a greater benefit. Figure 12 combines the use of an Object Genie and a Setup Method. Other combinations are possible as well, including using all three patterns. We use the Setup Method to login, call the Object Genie, and go to the checkout page. The Object Genie returns a shopping cart with items in it. The result is that each test case only has to worry about entering a specific coupon and checking whether the coupon was accepted or rejected.

Figure 12: Consolidating test code using Object Genie and Setup Method.

```
import junit.framework.*;

public class CouponTest extends TestCase {
        public CouponTest(String pName) { super(pName); }
        public static Test suite() { return new TestSuite(CouponTest.class); }
        protected void setUp() {
                login();
```

```
            ShoppingCart shoppingCart = CartGenie.getNonEmptyCart();
            gotoCheckout();
        }
        public void testCheckoutWithValidCoupon() {
            applyValidCoupon();
            checkCouponAccepted();
        }
        public void testCheckoutWithExpiredCoupon() {
            applyExpiredCoupon();
            checkCouponRejected();
        }
        public void testCheckoutWithInvalidCoupon() {
            applyInvalidCoupon();
            checkCouponRejected();
        }
        //implementations of methods (e.g. login(), addItem2ToCart(), etc.)
}
```

We have now significantly reduced the size of each test, eliminated test code duplication and reduced future test code maintenance costs.

4. Multiple Failures

Definition

Create a mechanism to allow a customer test to continue executing after a non-critical failure.

Discussion

Traditionally, unit testing tools such as JUnit fail a test automatically after encountering the first failure. This does not impede unit testing, since most unit tests execute only a few steps where each is dependent on the previous one. Customer tests, however, often execute dozens of steps and can have steps that are independent of each other.

When testing a module that is buried deep within an application, there are many steps involved to reach the module of interest. If one of those initial steps fails, all subsequent steps would not be executed. This includes all test steps related to the module of interest. Bugs beyond this first failure can be masked within the application. This also introduces inefficiency since each bug must be found and fixed before the next one can be found. Finding multiple bugs at a time increases efficiency and allows bugs to be addressed in a priority sequence.

If the initial failure was due to incorrect information but that information is not required by the module of interest, then the test could potentially continue validly executing and performing additional steps. Thus, when it comes to customer tests, an approach is needed to allow multiple failures.

The Multiple Failures pattern, however, is more difficult to implement than the other patterns, since JUnit does not support multiple failures. In collaboration, the author has modified the JUnit testing framework to add this functionality.

Example

In this example, there is a problem with the specials page such that incorrect items are displayed. This page is mandatory in the navigation path of searching for and adding an item to your cart. Therefore, without a multiple failures mechanism, the adding function would not be tested because the test would fail and stop executing on the specials page. However, the specials page is not integral to adding an item to the cart. By allowing multiple failures, the specials page failure can be logged, but the test can proceed and test the function of adding an item to the cart. Figure 13 illustrates how the multiple failure mechanism is applied.

Figure 13: Setting up multiple failures mechanism.

```
import junit.framework.*;

public abstract class ShoppingCartTemplate extends TestCase {
        public ShoppingCartTemplate(String pName) { super(pName); }

        protected void setUp() {
                login();
                gotoSpecialsPage();    //navigate to the specials page
                setStopOnFail(false);  //JUnit extension - test will continue if check fails
                checkSpecialsPage();  //if this check fails, the test will not stop
                setStopOnFail(true);   //JUnit extension - test failure will now stop the test
                searchForItem1();
                cartAction1();           // <------ test hook 1
                searchForItem2();
                cartAction2();          //<-------- test hook 2
                logout();
        }
        abstract protected void cartAction1();
        abstract protected void cartAction2();
}
```

The setStopOnFail() method demarcates when to start or stop the multiple failure mechanism. Notice that we demarcated only the checking of the specials pages and not the navigation. This is because a problem with the application's navigation cannot be overcome by our test. On the other hand, a checking failure does not prevent us from navigating to other parts of the application and performing our add to cart test. By demarcating which parts of a test allow multiple failures, we can exercise more parts of the application.

III. Architectural Patterns

Architectural patterns dictate the organization of test code into a framework. This group consists of the Three-Tier Testing Architecture.

1. Three-Tier Testing Architecture

Definition

Create three layers of test code: Tool Layer, Application Testing Interface Layer and Test Case Layer.

Discussion

It is a well established idea in distributed object architecture to divide an application into separate tiers. The first tier encapsulates the presentation logic, the second tier the business logic, and the third tier the data storage logic.[12] Using this paradigm, application maintenance costs are reduced since components inside each tier can change without impacting other tiers. The same principle can be applied to customer testing architecture.

Test code can be separated into three layers: the Tool Layer, the Application Testing Interface (ATI) Layer and the Test Case Layer. Each layer has a specific responsibility, with the overall goal of reducing the maintenance costs of tests and facilitating new test creation.

Tool Layer

The Tool Layer's responsibility is to provide an extensive Application Programming Interface (API) to facilitate test writing. For instance, the tool can allow tests to find and interact with objects such as buttons, forms, and tables. Each tool is generic such that it can be used to write tests for various applications of a specific type (e.g. web, .NET or Java). Some tools have capabilities to test applications of several types.

Test tools usually fall into two types: record-playback (e.g. QuickTest[13]) and programmer-oriented (e.g. AbcUnit[14]). Using a record-playback tool, test developers can quickly start recording tests and playing them back. However, the code that is generated by the tool is usually not object-oriented and uses a proprietary language. Tools that are programmer-oriented are meant to be used by someone with programming experience. There is usually no record and playback mechanism because test developers are expected to write the tests themselves. The advantage of these tools is that they use a standard object-oriented language, which makes them very suitable to using design patterns.

Application Testing Interface (ATI) Layer

The ATI Layer[15] is responsible for consolidating all functions common to multiple tests. Every application will have a specific ATI Layer. For instance, each application is sufficiently unique as to require different navigations and assertions. Thus, the common ones contained in this layer would vary between applications. The ATI Layer is in effect a utility service provider for individual tests. It offloads the majority of the work that each test would otherwise be required to do.

The Template, Domain Test Object, Object Genie, and Transporter patterns are all examples of ATI layer code. The ATI Layer uses the tool (from the Tool Layer) to implement each pattern's details. If the tool or the application changes, then the corresponding ATI code would need to be updated to accommodate the changes. The ATI layer is the primary factor in lowering maintenance costs, since a well designed ATI will encapsulate many common navigations and checks that are used by test cases in the Test Case Layer.

Test Layer

The purpose of the Test Layer is to outline specific inputs and expected outputs for each test. The Test Layer relies heavily on the ATI Layer and as a result, is quite brief. Ideally, the Test Layer contains only data unique to each test. For this reason, the tests are insulated from changes in the application. Examples of Test Layer code are the subclasses of a Template, and the tests that use an Object Genie, Transporter, or DTO.

Conclusion

Applications are constantly evolving, which poses serious maintenance problems for automated customer tests. Maintenance problems can be addressed by using design patterns and thus, treating test code with the same importance as application code. This paper presented a catalog of design patterns that demonstrate how to facilitate: test code reuse without duplication, test adaptability to application changes, creation of new tests, and maintenance of existing tests. To this end, the following nine design patterns were presented:

- Group 1: Basic Patterns
  - Template - consolidate sequence of steps to perform a test
  - Object Genie - consolidate object creation and initialization logic
  - Domain Test Object - consolidate knowledge of application's UI components
  - Transporter - consolidate navigation logic
- Group 2: General Design Principles
  - Dumb Test - minimize tests' knowledge of the application
  - Independent Test - maximize test independence from other tests
  - Don't Repeat Yourself - minimize duplication of code between tests
  - Multiple Failures - maximize the information obtained from each test
- Group 3: Architectural Patterns
  - Three-Tier Testing Architecture - separate test code into layers

The design patterns were categorized into three groups: Basic Patterns, General Design Principles, and Architectural Patterns. The basic patterns form the building blocks that the general design principles use to establish best practices. The architectural pattern dictates the organization of all test code into a reusable framework.

This paper has demonstrated how design patterns can be applied to increase the effectiveness of automated customer testing. This results in tests that consistently provide valuable and reliable feedback about an application. It is my hope that this information will be of benefit to others in their automated testing endeavors.

Acknowledgments

I would like to thank Kit Bradley and Susan Wolfman for their valuable feedback and guidance throughout this process. I would also like to thank Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, the authors of *Design Patterns* and Martin Fowler, the author of *Refactoring*, for showing me that software development can be elevated to an art form.

References

[1] Beck, Kent *Test Driven Development: By Example.* Addison-Wesley, 2002.
[2] JUnit Website, online at http://www.junit.org/
[3] Extreme Programming Software, online at http://www.xprogramming.com/software.htm
[4] Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
[5] Fowler, Martin *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
[6] Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
[7] Beck, Kent *Test Driven Development: By Example.* Addison-Wesley, 2002.
[8] Schuh, Peter, Stephanie Punke *ObjectMother: Easing Test Object Creation in XP.* XP Universe 2001, online at http://www.xpuniverse.com/2001/pdfs/Testing03.pdf
[9] Krasner, Glenn E., Stephen T. Pope *A cookbook for using the model-view controller user interface paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, 1(3):26-49, August/September 1988.
[10] Fowler, Martin *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
[11] Kaner, Cem, James Bach and Bret Pettichord *Lessons Learned in Software Testing: A Context Driven Approach.* Wiley, 2001.
[12] Monson-Haefel, Richard *Enterprise JavaBeans, Second Edition.* O'Reilly, 2000.
[13] Mercury Interactive QuickTest Professional, online at http://www.mercury.com/us/products/quality-center/functional-testing/quicktest-professional/
[14] AbcUnit Website, online at http://www.abcunit.org/
[15] Crispin, Lisa, Tip House *Testing Extreme Programming.* Addison-Wesley, 2002.

Using a Database to Automate and Manage Interdependent Functional Tests Across Multiple Platforms

Jeff Pearson
Pearsonality LLC
Jeff@pearsonality.com

Jennifer Brock
ClearLake Solutions LLC
Jenniferbrock@clearlakesolutions.com

Test Automation can be a complex and intimidating endeavor to attempt. This paper is about how a small team approached the problem and the solutions discovered along the way. The objective is to automate testing of financial like transactions across multiple platforms, leveraging the business users expertise about the application; using the tools available to us. In addition, test maintainability became a constraint we as a team put on ourselves as we had either experienced or heard horror stories about maintaining automated scripts once they were complete.

This assignment became an adventure for the team as we encountered hidden requirements, constraints and issues along the way. The applications under test are complex due to many factors like multiple platforms and exponential number of combinations etc. We also discovered constraints relating to the compatibility of the test tools with the applications under test. As we learned more about the applications and the test tools themselves we began to realize additional requirements such as time to process the automated scripts.

"An Opportunity"

Management presented us with an opportunity to automate the tests. In this situation opportunity translates to challenge or a problem. We decided to view this as a puzzle to solve rather than a problem or challenge.

Now we embark on our adventure into test automation and we begin with the limitations.

Limited resources- the test automation team consisted of three individuals, not necessarily full time either. There were projects going on that required the expertise of the team members to assist with those projects at the same time undertaking this project. Also, not all the team members had the same level of automation expertise, one knew Rational's XDE Tester and Robot J, another knew Rational Robot and another had broad experience with both Mercury and Rational Products.

Schedule- the test automation schedule was driven by the release or project schedule, as the automation was needed in order to meet these projects schedule and quality objectives.

Maintainability- this is a constraint we put on ourselves as a team as we had either experienced or heard horror stories pertaining to maintaining the scripts once they are developed. Our objective was to create scripts that can easily adapt as the applications under test change.

Tool Availability- the tools available to us were Mercury Quick Test Pro and the Rational test tools. We evaluated the two vendors tools to determine the best choice for our efforts. After several weeks of evaluation we chose the Rational tool suite. This decision was based on a familiarity with the tool set, which meant we had already encountered challenges and overcame them and we knew we would encounter different challenges (as we did during the evaluation) with the Mercury tools, but we hadn't

worked through the solutions with our applications.

Skills-fortunately we had adequate skill coverage across the test tools. Unfortunately as a three-member team we didn't have the depth in skills; as we had one member proficient with XDE tester and Robot J, another was proficient with Rational Robot. The third member of the team had a broad knowledge of both the Rational and Mercury tool sets. It limited our ability to work on a wide range of the applications simultaneously, causing the workload to be assigned in more of a silo rather than collaboration.

We understand the limitations we are up against now we begin our discovery phase of our adventure

What tools should we use?

The first challenge was to choose the tool(s) we would use to solve this puzzle. As there were already two tool sets available to us, we evaluated them for ease of use, maintainability, adaptability and how easy it is to resolve issues as they arise. Because issues will arise and the key is to how easily and quickly they can be rectified, rather than pretending they won't happen at all. We chose the Rational tool suite, mainly due to the teams familiarity with the tool. This familiarity meant the team had already encountered issues with the applications, resolved them and continued on. During the evaluation we realized that it doesn't matter what tool we use we will encounter unique issues when using these tools with our applications. We believe this to be true in most organizations as well.

How do we create scripts with maintainability?

We knew from either past experience or hearing others horror stories; for this effort to truly be successful we must develop easily maintainable scripts. The data driven approach is commonly known to minimize the maintainability, but somehow many organizations don't embrace this technique. This could be due to the fact it does take a little longer up front to set up the scripts, but the payback is on the backend when modifying one script versus all the tests is exponential. We decided to use the data driven approach, but rather than using an Excel spreadsheet, CSV file or the data pool built into the Rational tools we chose to use a database. By using a database to input our data into the scripts, we have one script running many tests. The script acts as a shell and the data is fed from the database. When the application GUI changes only one script needs to be modified. If these scripts were hard coded with the data input to each script we would have to modify all the scripts when the application changes.

How do we leverage the Business users expertise with automation?

With limited resources it was imperative we use the business users expertise to provide the data, which in turn provides the test conditions for the items being automated. In order to provide the users with a user-friendly manner of inputting this data, the actual input screens were mimicked to look like the application, including the drop down

menus. The input from these screens then updated the appropriate fields in the database for the automated scripts to read for the input values when executing.

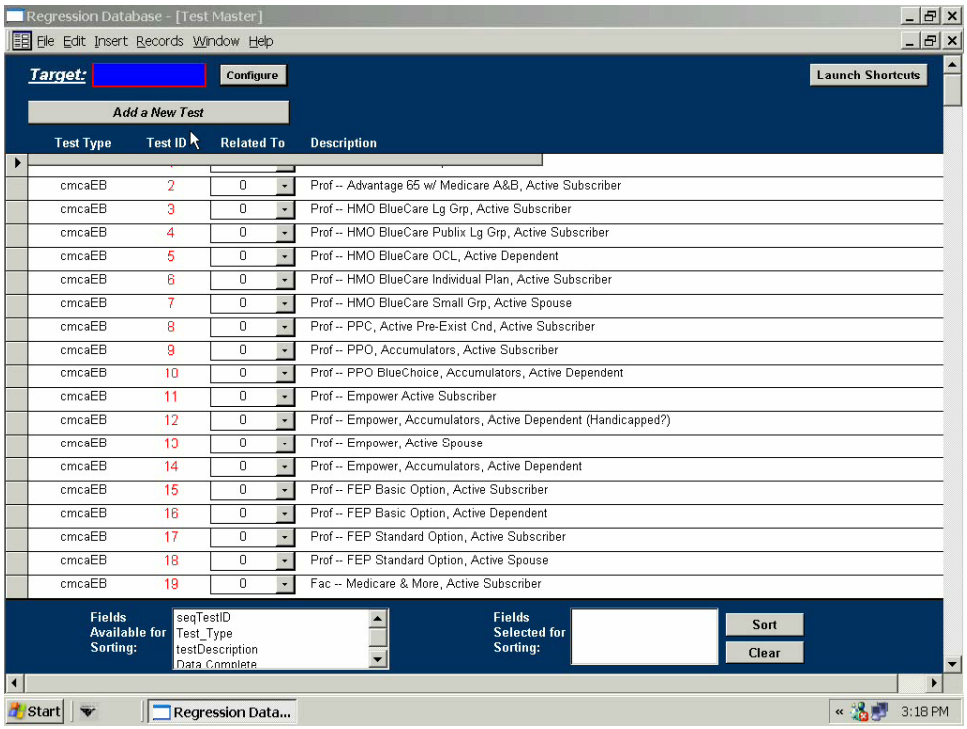
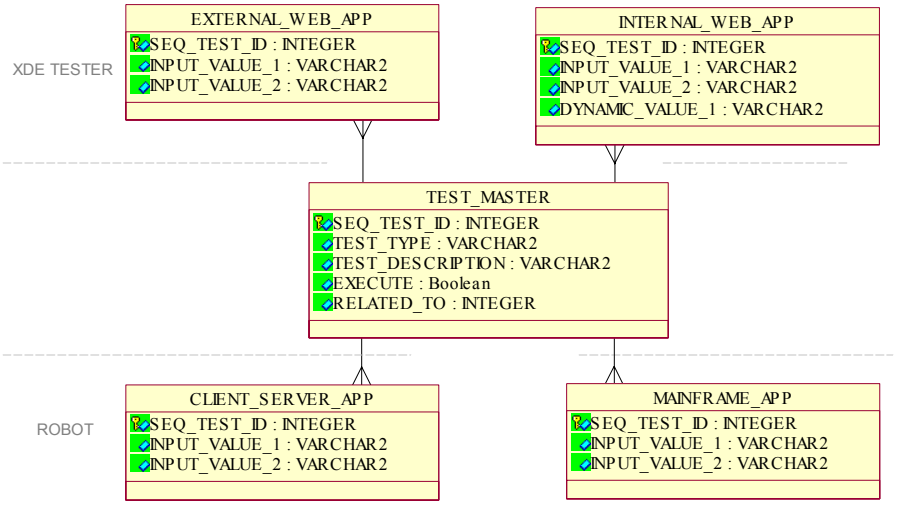


Figure 1

This seemed like a brilliant idea at the time and does have a lot of advantages particularly for business users not accustomed to automation, although the downside is a tremendous amount of maintenance for these input screens as the application changes the screens need to as well.

Building the database



These tables correspond to the input screens the users are accustomed to. The Test Master is the center of the schema establishing the Test ID, which is used in the scripts, and the test manager as well. Using a standard naming convention is critical to the success of this process. Otherwise when you get hundreds of tests it would be difficult to manage and keep straight.

The adventure continues as we uncover additional questions that were unknown when we first started this journey

The transactions themselves equate to one test and we needed the automation pass or fail to reflect that. We will describe how we accomplished this.

These transactions originate at an external web front end which is outside the firewall, they then do some verifications against an internal web, mainframe and client server databases, do some updates, calculations and also generate an internal dynamic variable so the transaction can be treated as one when processing through these platforms.

In order to automate these it is critical to capture this dynamic variable, we used XDE Tester and wrote some code to capture and update the database, and Robot then reads this variable and continues processing. Because we have web applications, mainframe and client server applications we used XDE tester for the web applications and Rational Robot for the mainframe and client server applications. These two tools don't talk to each other so we needed to develop a method for these scripts to run seamlessly. This leads us to our next challenge

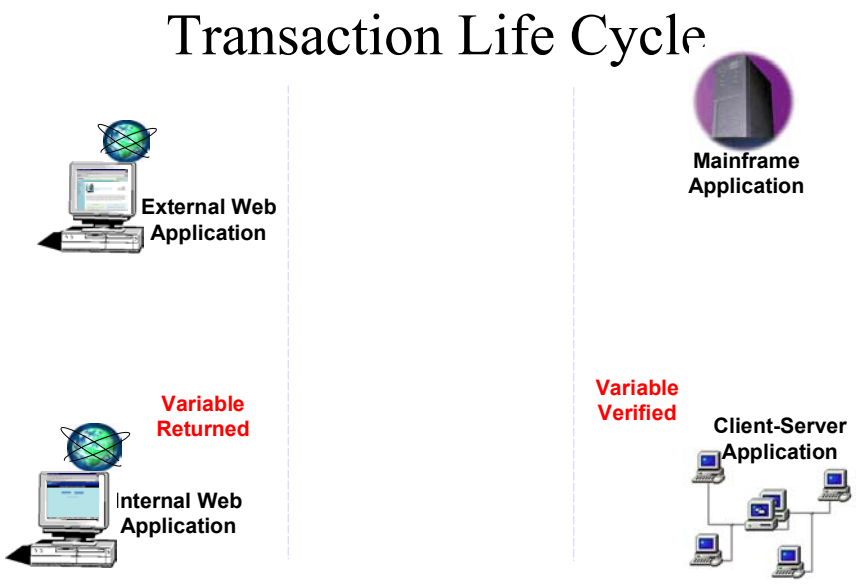


Figure 2

We need 4 scripts to run "seamlessly" as one test, how do we accomplish this?

Robot is used for the Mainframe and Client Server application scripts, these are running successfully, XDE Tester is used for the Web applications and the scripts are running successfully. These scripts are reading the database for the input data, the verification points are set up and working as designed. So you might ask what is the problem? It appears that everything is working and it can be considered successful.

Four scripts exist for one test, we need the 4 scripts to run seamlessly and create either a pass or fail at the test level not the script level. When we attempted to associate multiple scripts with a test case in Test Manager; it doesn't allow it. This caused considerable frustration on our part, but being the stubborn group of people that we are we continued on and found a solution. We used the suite in Test manager; creating a suite with all four scripts included in it. We then associate the suite to the test case. Now we are onto our next dilemma.

We now have a suite with the 4 scripts running but need to run several of these suites otherwise it will be a cumbersome task to run the automation so we investigated running a suite within a suite.

Fortunately we can run a suite within a suite. This took some ingenuity to make it happen but it works and it works well. As mentioned earlier maintainability is a self-imposed limitation. The scripts were designed to be modular, meaning there are a logon script, a navigation script and a logoff script for each application. The execution suite runs the logon and navigation scripts, then the test cases which have suites associated to them, one for each of the applications, then there is a logoff script for each application. In one execution; the logon, navigation, test cases and the logoffs are all run. This is what we wanted, but now our next dilemma is the time it takes to run these automated tests and it ties up a computer during the time it takes as well. The number of tests was getting up to be over 150 now and if each one of them takes 2 minutes to execute that is 300 minutes or five hours. While the accuracy, consistency and, yes, the speed is much quicker than manual testing, this is still not adequate.

How can we distribute the workload of these scripts running?

We became aware the volume of tests we were planning to run would cause each execution suite to take several hours. We investigated solutions to run the execution suite on several computers simultaneously. This load can be distributed across many computers and the load can be balanced as well. This solution saves a tremendous amount of time as the number of computers we run these tests on increases. Let's take a look at sample timesavings. Theoretically we have 100 tests which take 4 minutes each to execute in their entirety, on one computer this take 400 minutes, now lets run the same 100 tests across 10 computers, it takes 40 minutes. That is a tremendous timesaving, increasing the use of the automated tests.

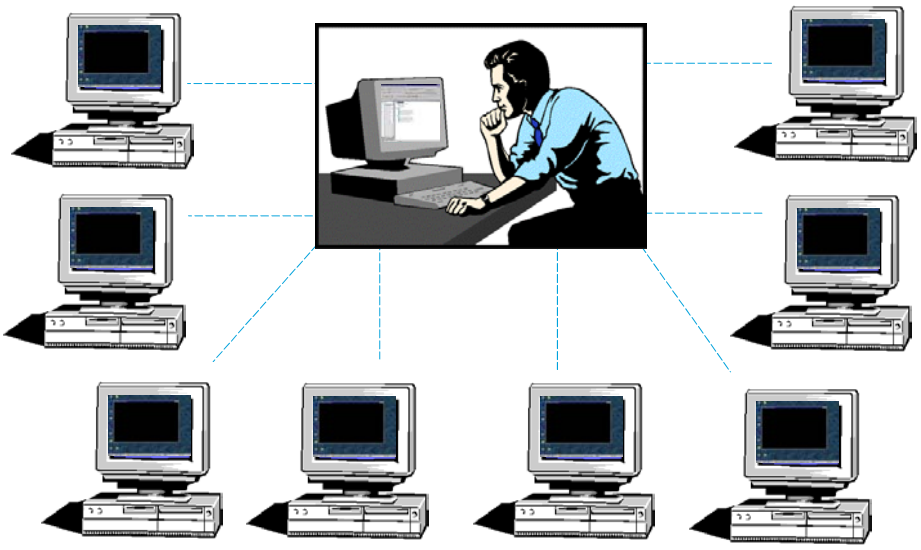
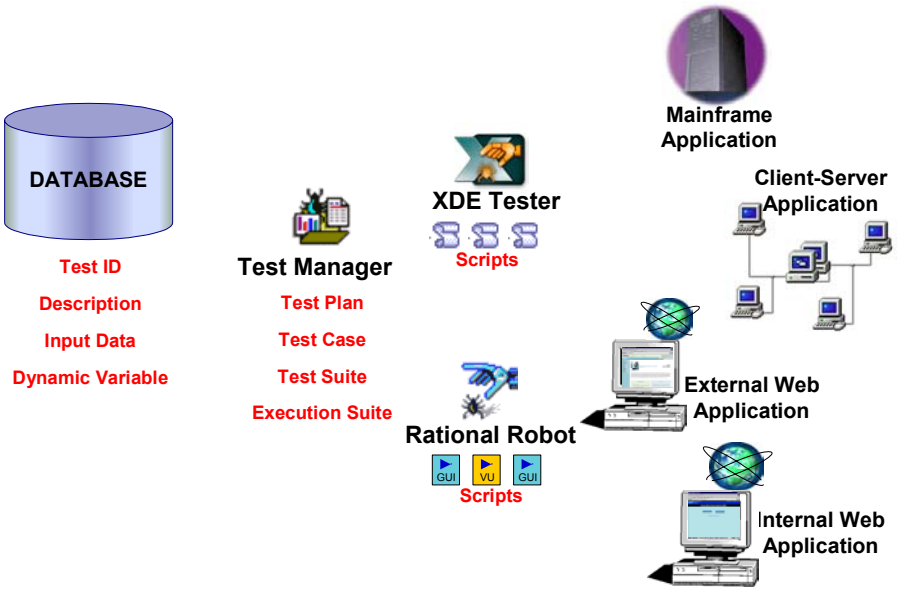


Figure 3

Let's take a look at where we are at this point

We have the database with the test id, input data and the dynamic variable stored, the test cases, test suite with the four scripts are in the test manager, the scripts using the standard naming convention read the data in the database and update and retrieve data from the database, the execution suite successfully runs the logon, navigate, logoff and test case scripts and these can be run in a distributed environment, incorporating load balancing principles.



Evolution

The puzzle appears to be solved and it is for the moment. This is software development and things change. Test automation is no different and it is necessary to adapt to the changes. This next section explores different ways to take advantage of the tools available to us. We hope these will inspire others to discover new ways of enhancing the testing.

Open Source Testing Framework for Handling A Multiple Product Stack

Judith Lebzelter

Open Source Development Lab
12725 SW Milikan Way, Suite 400
Beaverton, OR 97005
judith@osdl.org

Abstract:
For mission-critical and safety-critical systems, Open Source will not work without some concept of “open trust”, defined as a kind of open record on how the Open Source component has been tested for performance and reliability.

We describe here one approach to developing such open trust. The Scalable Test Platform (STP) Project at Open Source Development Lab (OSDL). The STP is a fully automated system for building, integrating, and testing software products on a variety of computer environments. Automated performance and regression tests can be run against fully integrated product stacks.

The project consists of two major components:
1. Patch Lifecycle Manager(PLM)-Handles the access to the product source and build information, manages relationships between source trees and discrete patches, compiles source and provides source or a compiled binary.
2. Scalable Test Platform(STP)-Manages a set of hosts which execute tests using PLM to build product stacks. Its web interface allows user access to submit and browse test requests.

The STP is an open source project published under the GNU General Public License[1] and available through http://www.sourceforge.net/projects/stp. It is designed primarily for testing the Linux Kernel, but has been extended to other open source software and could be used for commercial software as well.

OSDL makes an instance available to developers and others interested in testing open source software.

Biography:
Judith Lebzelter is a Test and Performance Engineer at Open Source Development Lab. She is experienced in Open Source and commercial database development.

Introduction:

Our company, OSDL, exists to promote Linux adoption in the commercial and enterprise market. Among other efforts, we provide hardware to test scalability and performance on the kernel as it develops. Through an automated test framework called the Scalable Test Platform, a user can submit tests requests on that hardware. A major goal of the STP project has been to expand beyond Linux kernel testing to include testing of components of the operating system and applications. This paper describes the history of STP, then discusses how it has evolved to meet the needs of the Open Source community with the introduction of multiple product stacks. These additional requirements needed by the open source community are described as are the design details of the expanded STP.

Background on the Linux Kernel:

GNU/Linux is an Open Source operating system that displays behaviors similar to UNIX.[2] It is being adopted more every day, even for mission critical and safety critical systems. For example, the docking controller for the European Space Agency's Automatic Transfer Vehicle will be operated from a Linux laptop by astronauts on-board the International Space Station via a system called 'Remote ATV control at ISS', abbreviated RACSI.[3] Daimler Chrysler intends to build up to a dozen ATV's between 2003 and 2013. Still, in general, Open Source components come without statements or guarantees of quality. Developers and users will not adopt Linux for more uses unless they trust the open source components for the whole system completely. How are we to generate such trust?

One roadblock to the widespread adoption of Linux is the issue of quality. How good are Open Source components such as Linux? Consider, for example, the community development process for the Linux kernel. Code is released by a maintainer to public repositories and mailing lists. A large and varied population of users and developers report and fix issues via public mailing lists and other open forums. Interest in testing varies widely. The process is heavily weighted toward developers, who typically test according to their own immediate interests and have little inclination for comprehensive testing. Their submissions must pass through the scrutiny of kernel maintainers and the mailing list, but even so, errors can get through.

An exception is a group called the Linux Technology Center(LTC) at IBM, which publishes some excellent work in Linux Testing.[4] Their focus is, however, specifically on Linux, and on publishing their own results. We believe a broader approach is necessary.

Our work focuses on a particular moment in the release of any open source component. When such a component, e.g. the Linux kernel is released certain issues are generally addressed quickly. Compilation bugs are reported immediately at least for the more popular architectures. More complex issues, like performance under different workloads, scalability, performance on larger systems, may not be examined at all. This gap led to the creation of the STP.

Project History:

The Scalable Test Platform Project was started in June of 2001.[5] It began as a project to fill what we at OSDL saw as a need for more, diverse and reproducible testing in Open Source, particularly Linux kernel development. It is an Open Source test framework and a library of tests that exercise new kernel builds on a variety of platforms with diverse configurations.

Patch Lifecycle Manager was created to provide further automation for the storage and management of patches (systematic change sets) to base source. These changes need to be

tested prior to inclusion in a production tree, and the PLM needs to allow developers the ability to easily manage discrete patches for testing. The complete patch sets and source tree are then pulled into the test environment (STP) and built with a few calls to the PLM.

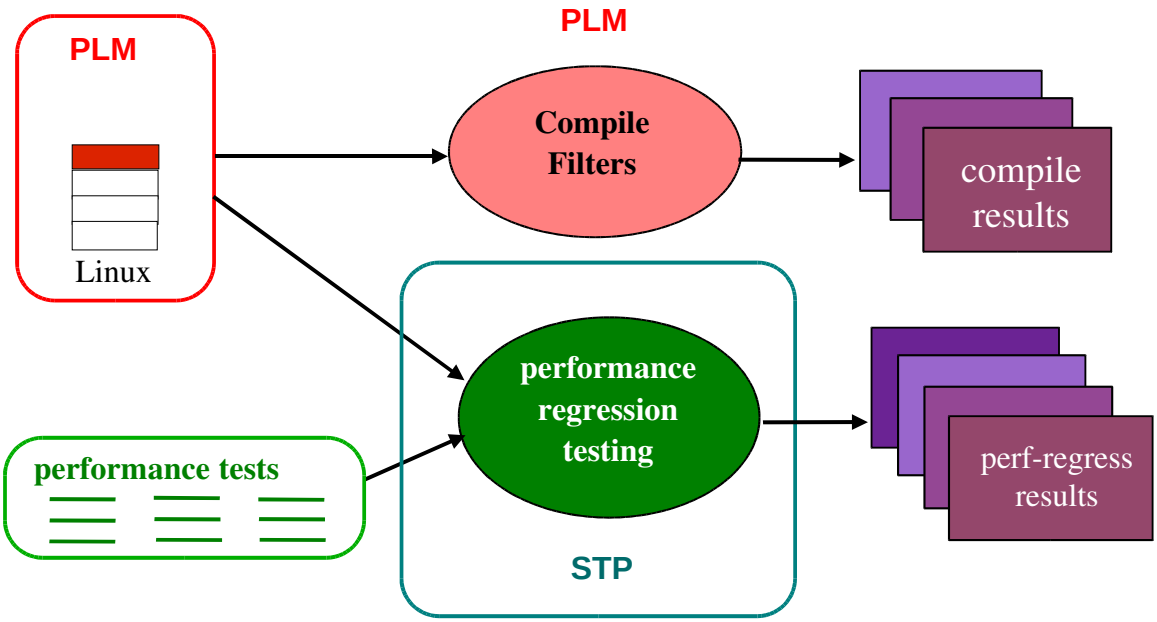
PLM has an additional functionality which seperately allows some preliminary compile time testing within PLM. It has a system for adding compile scripts (called 'filters') according to software type, which are executed as the patch is submitted. These filters can be extremely simple, such as testing whether the patch applies, to more complicated, such as testing whether it compiles with various configurations. The results of these filters should give an indication whether the patch set is suitable for further testing.

PLM patch information and STP test information are stored in a database. Access to detailed test results is presented through a web interface with the results currently being stored in flat files. Since the flat file storage method has made it somewhat difficult to analyze trends and compare multiple tests, we are adding an Advanced Search Tool to give people outside the OSDL direct access to the results database.

This has been a project about integration: integration of all the parts needed to totally automate Linux kernel testing and results reporting. The code itself is not especially complicated, but bringing all the parts together to make a completely automated system for allowing users to run tests has been the challenge. For example, the system under test needs to have the correct operating system distribution installed automatically. We use an open source tool called 'SystemImager' on the STP server to install one of several choices, Redhat 9.0, RedHat 7.3 or Suse 9.0 onto the system under test.

Figure 1 shows the flow of the Linux centered application. PLM tracks the Linux kernels sources and produces compile filter results. Test requests are submitted to STP. STP retrieves the Linux kernel from PLM to run tests on the STP test hardware and produces test results. In the future, we plan to store the binaries generated from the filters and pull those into STP.

Figure 1: Flow of Linux Centered STP/PLM



Multiple Product Stack Uses

The STP user population includes Linux kernel engineers, who want to test source before releasing code, database developers (PostgreSQL/MySQL), who are testing the performance and reliability of their database on Linux, tool developers (sysstat) who would like to see the accuracy of their tools for kernel measurements and test developers (Tiobench) who are working to verify the accuracy of and add features to their tests for the Linux kernel. These users (and perhaps other future users) would benefit by being allowed to directly patch sub-systems of interest, like postgreSQL or sysstat. In addition, we would like to extend system usefulness to include testing system and application integration, to check the interoperability of a product suite or specification. Being able to add components would make that possible.

This approach will allow users to customize software stack, and choose tools, libraries, compilers, applications and tests as appropriate. This will make the system more useful to many different classes of users well beyond the Linux kernel development community.

For example, a PostgreSQL developer may need a patched version of PostgreSQL, Iostat version 2.5.6, and a patched version of the Linux kernel that provides the latest AIO fixes. Or an application tester for the Data Center Linux Specification may need separate versions of glibc, gcc, file system utilities etc. for both Linux version 2.4 and 2.6. The combinations are only limited by interest and how quickly OSDL can add the initial base components for developers to build on.

Multiple Product Stack Requirements

The requirement for this phase of development was to expand from management and testing of a kernel source tree, to the management and testing of a 'stack' of any number and any type of software components. It would allow a machine under test to be set up with a specific, controlled, and reproducible software configuration.

Any of the software to be included in a product stack should be able to be patched independently.

Sources or the binaries generated from them can be configured to be acquired from various repositories and types of repositories, like web files, ftp, Concurrent Versions System (CVS) or BitKeeper (another source control system).

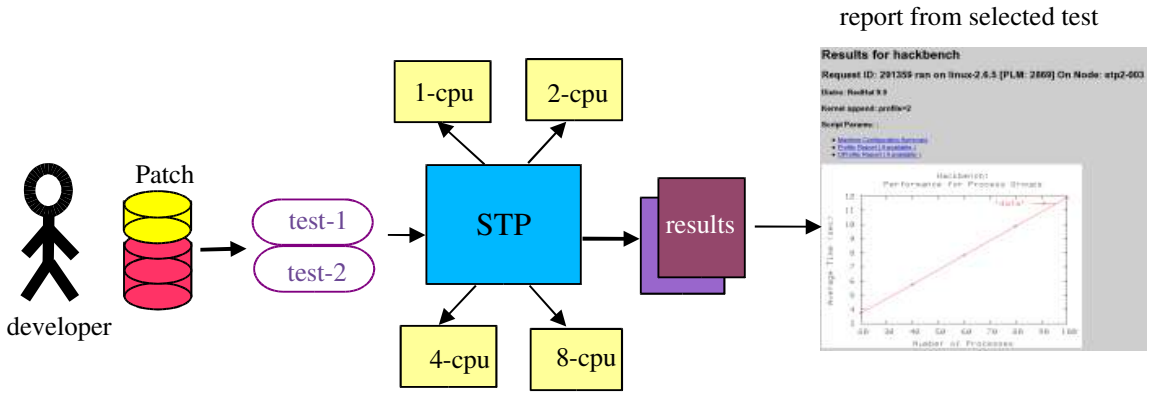
Project Architecture

All information is stored in a MySQL Database. One database instance set up for STP contains the information for the test requests, and the configuration and state of the test system. The STP system under test (SUT) runs autonomously, also querying the STP database to check and update state. A separate database instance for PLM holds the information about all the components including source background and build scripts.

PLM clients access the PLM database through SOAP/RPC. The STP test hosts are call PLM code and only need to run two commands, 'plm_build_tree.pl' for source access and 'plm_build_app.pl' for build and installation.

The STP server manages the set of hosts which execute the tests (see figure 2). It checks the state in the database, fixes problems, assigns tests and initiates the hosts which will run them.

Figure 2: Overview of STP Function

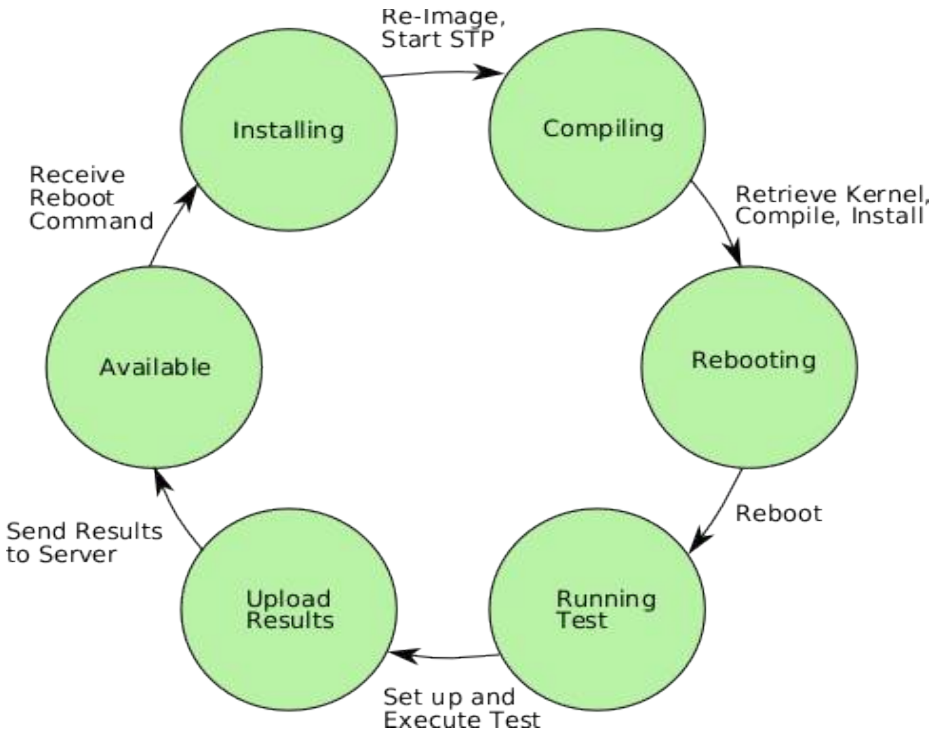


The STP server has two important programs, the 'stp_cron.pl' and the 'stp_server.pl'. The stp_cron.pl is the 'engine' that drives STP. It checks states of all the hosts in its pool. If they have exceeded time limits, it terminates the current test. Under certain problem situations, it will place a host out of service. It re-queues tests when failures occur due to system issues. It does general database consistency checks between the tables which store the system states. It assigns test requests to hosts that are in an available state by priority. Afterwards, it sets the host up for reimaging and then remotely power cycles it. The script stp_server.pl is a SOAP/RPC server which handles information access between the database and the SUT. The SUT can access its state information and query for test information.

The SUT's need to verify and update their states in the database and also get information about the test request to run. The test request contains all the information needed to run the test. This includes the complete product stack, the system environmental parameters, the system parameters, the kernel boot options, and the test and its options. The 'stp_server.pl' is a SOAP server which has access to functions that the test hosts call for database access.

The STP SUT testing cycle (see figure 3) starts with the host receiving a reboot command from the STP server. The test host hardware is configured in the BIOS to reimage via Preboot Execution Environment (PXE) if this has been requested by the server. Otherwise, it boots locally. Therefore, when the STP server power cycles it, it reimages with the correct operating system, and start up the 'stp_client.pl' script. The script queries the database via RPC for test request information, installs the requested kernel, sets the system parameters, and reboots. After reboot the 'wrap.sh' script is executed again. It checks the state, then sets up the test and run it. When the test is complete, the script uploads the results files to the server and sets the host's state to 'available' and the test state to 'complete'.

Figure 3: State Diagram in the System Under Test (SUT)



Tests must be appropriately custom packaged in order to run in STP. They are stored and installed self-contained in a compressed tape archive (tar) file. The STP specific packaging starts with the script called 'wrap.sh' which is executed by the STP client engine. STP provides an API of script calls which must be called to access specific environmental parameters settings and adds functionality such as profiling. The STP client engine initiates the rest via a 'wrap.sh' script included in the package. After running the test, this script gathers all the pertinent logs and result files and prepares them for uploading.

Test runs are requested through a multi-page set of web forms that walk the user through the valid options for each test.

'Advanced Search' is the most recent addition to the project. This tool is a PHP based web interface which allows users read access to the database and lets them freely generate their own SQL queries. It also allows users to create data sets and do plot generation through GnuPlot to compare the data sets. Plot generation is still in progress because each test needs to be addressed individually, deciding what data is essential, and then parsing it from the raw data flat files into the database. This task is not yet completed.

Expanding the Design

The idea of creating a system where any number of software packages could be built, installed, configured and executed was a bit daunting, especially considering how complicated the system already was for the kernel alone. With due thought we realized that much of the framework was already present. The basic architecture of the system could remain the same, as could the database. Changes would be built into the existing structures. It was familiar, had known issues that could easily be handled and would take a minimum of data rearranging. The state storage/control through the database remained, and could be built upon by the addition of extra needed states. The division of labor where PLM handles component information access and STP

handles system integration test execution seemed even more appropriate. This would allow delivery of the required features with a minimum of effort and the scalability desired.

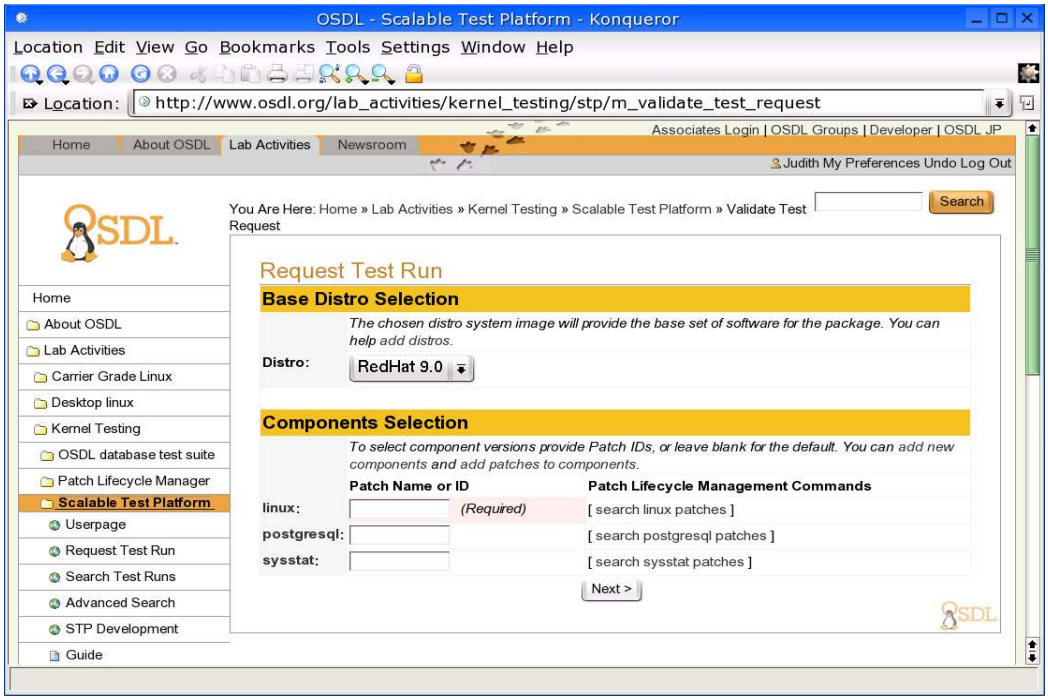
The design makes it easy to add components, start with just the Linux kernel and a few others, and grow the list as needed. Specific components will be associated only with appropriate tests on the test request input form.

Predefined stacks are managed through STP. Once the product stack grows to more than about three, it becomes tedious for test submission. Predefined product stacks will speed up the process of submitting tests and minimize errors. The user can choose a predefined stack and edit it, or still enter in all the components manually. Standard stacks may be defined for users who merely want a current, working system in which to put their application.

Build and install information will be configured in PLM. Therefore, the STP does not need to store that information. It will request the component from PLM then request PLM to 'build' or 'install' the component. For tests with very specialized configuration issues (currently the database tests), the test wrapper will need to be designed to attend to those issues.

The web interface test request form also increased in complexity, going from two pages to three pages plus a confirmation page. However, since it was carefully laid out, it flows easily through the additional information. Also added was the new logic for saving product stacks for reuse.

Figure 4: Component selection for test dbt3-pgsql.



On the test host the STP test installation also needed to be able to handle varying numbers of products to be installed in correct order. This was a matter of adding the additional PLM identifiers, software type and installation orders to the test request data structure. When processing, the script loops through the list with PLM calls until all are installed.

A few changes outside the requirements also were added to the plan, simply because they were 'feature requests' and they seemed to fit in with the other changes. In order to add more control

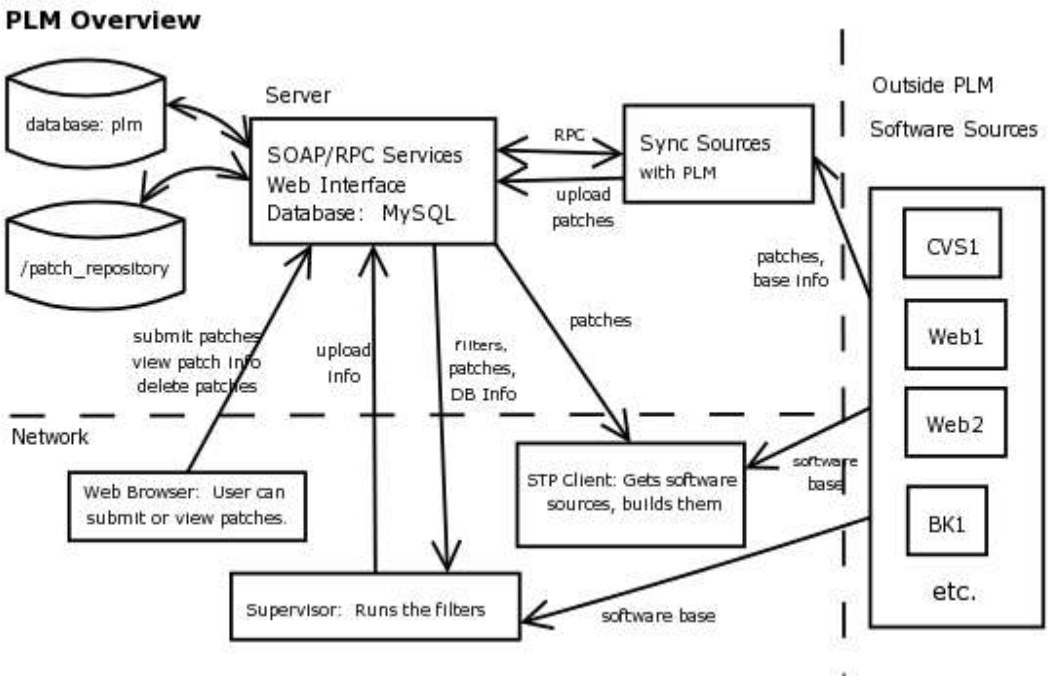
over the system, more types of input parameters were added to allow the users to set system environmental variables and /proc parameters.

Some structural changes were also judged necessary. After reassessing the code, it was decided for maintenance reasons to eliminate a rather complicated object model in STP in favor of simplified database queries and simplification of object access via SOAP. For PLM, a proprietary RPC protocol made it difficult to access information from outside the application, except through direct database queries. This was removed and replaced with standard SOAP/RPC, which we already used in STP.

The PLM functionality is illustrated in the figure below (figure 4). The server synchronizes PLM with source trees of interest, which are defined in the database. These source trees can be from CVS, Web accessible files or FTP accessible files. The server also facilitates access to the base sources for the PLM clients and tracks additional patches to the source for the users.

The 'supervisors' are hosts that are set up to run the filters, which are generally source code compile tests. The STP Client (SUT) runs PLM scripts which handle retrieving the base source and applying any number of PLM patches in order, and then handle building the source into an application. The web browser allows users to submit or delete patches, view filter results, view patches.

Figure 4: Overview of PLM



Implementation of Multiple Product Stack

Although we tried to divide up tasks so that we could do multiple smaller releases, in reality that was not practical because the web changes could not be made incrementally. Instead we did a single major release.

Inclusion of additional components beyond Linux will be gradual, and as needed. Immediate

additions included PostgreSQL and the system utilities sysstat and oprofile. These were all components we had previously needed to put custom solutions into each test for. Adding additional components such as procps, e2fsprogs, reiserfsprogs, reiser4progs, jfsutils, and xfsprogs will take relatively little effort and testing .

PLM was modified to retrieve from an FTP sight, a web sight, or a CVS repository in order to be able to pull from project repositories in the manner that most suits the developers. For production OSDL chose that these be local mirrors of the actual source repositories in order to avoid server access issues and in order to control the file maintenance defaults. However for less critical situations (in testing), we pull directly from remote sights.

Future Goals

The following is a list of goals for the project which should be implemented within the next six months.

- PLM adding more sorts of repositories, especially BitKeeper which is the Linux kernel hacker's preference. By making it as easy as possible for users to get their changes into this system for test, they will be encouraged to use the system. They should not have to change habits.

- Make PLM a gating factor for queueing STP tests. STP tests are queued automatically regardless of filter results. A check for 'PASS' on certain compile filters would leaad to fewer tests being queued on kernels with basic compile problems.

- Have PLM make available compiled binaries for recent sources in order decrease set up time for tests.

- Enable the ability to pull in custom RPMs without source, in order to be able to extend use to products where the source is unavailable or private.

- Expand the Advance Search Tool. This is the key tool in enabling users outside the system easy access to their data. Otherwise, the convenience of automation and access to hardware is overshadowed by the difficulty of reporting.

- Automated publication of compiled STP results sets. Data across runs

Participating in STP/PLM Development

STP is 'Open Source', so we strive for transparency in all phases of development. This means planning and working where anyone who cares to can see what is going on, when it is going on. 'Release early, release often' is the mantra. This is accomplished though typical open source means which follow.

As stated previously, STP is available to run tests via the web interface at http://www.osdl.org/lab_activities/kernel_testing/stp/ . The complete source code is available at http://www.sourceforge.net/projects/stp and http://www.sourceforge.net/projects/plm or via anonymous checkout from our BitKeeper repository bk://developer.osdl.org:/stp and bk://developer.osdl.org:/plm. The STP project can be reached at stp-devel@lists.sourceforge.net and PLM at plm-devel@lists.sourceforge.net. This is a forum for project members and outsiders to ask questions, submit patches, follow progress, make requests send compliments or complaints. Our development web site is at http://developer.osdl.org/dev/stp. It is where we place planning and design documents, links to the other STP resources and try to maintain all STP information.

All the STP test results are available on our web site. For some of the tests we automatically post comparisons for the most recent Linux kernel results:

- http://developer.osdl.org/judith/tiobench_index.html
- http://developer.osdl.org/judith/iozone_index.html
- http://developer.osdl.org/cliffw/reaim/index.html

The project has come up with other useful tools. PLM uses cross compilers for many architectures and they can be downloaded off our web site.

OSDL's Test and Performance Group has contributed much interaction with Open Source Developers, reporting bugs, providing patches, working with Linux kernel developers to test their work, running requested tests for developers. Participation in peripheral projects like Perl CVS module, LTP, gcc, has also naturally resulted because in using these tools we become members of their user community.

Conclusion

OSDL has improved its automated Scalable Test Platform to be able to build customized, configurable product stacks. We hope this will increase the usefulness to many classes of developers and testers and also lead to an easier sharing of test results which are well-defined, reproducible and meaningful. These three qualities will lead to a greater acceptance of Open Source software for applications where performance and reliability are essential.

Some Open Source Resources Used By STP

- Comprehensive Perl Archive Network(CPAN): www.cpan.org
- Concurrent Versions System: http://www.cvshome.org
- CrossTool: http://www.kegel.com/crosstool/
- Free Software Foundation, http://www.gnu.org
- Linux kernel: http://www.kernel.org
- Linux Kernel Mailing List: Majordomo@vger.kernel.org
- MySQL Database Home: http://dev.mysql.com/
- Reaim Benchmark: bk://developer.osdl.org/reaim
- SourceForge web site: http://www.sourceforge.net
- SystemImager: http://www.systemimager.org/
- Tiobench: http://sourceforge.net/projects/tiobench

References

1 Free Software Foundation, *GNU General Public License, http://www.gnu.org/licenses/licenses.html.*

2 'What is Linux', http://www.linux.org/info/index.html.

3 Ortega, Guillermo 'Linux for the International Space Station Program', *Linux Journal*, Issue 59, March 01, 1999.

4 Ge, Li; Scott, Linda; and VanderWeile, Mark, 'Putting Linux reliability to the test: The Linux Technology Center evaluates the long-term reliability of Linux'*, DeveloperWorks,* 17 December 2003, http://www.gnu.org.

5 Dabney, Nathan, 'The Scalable Test Platform' , *Linux Journal,* Issue 91, November 01, 2001.

6 Thomas, Craig; '*A Survey of Quality Practices in Open Source Software and the Linux Kernel*', Pacific Northwest Software Quality Conference, 2003 Proceedings.

7 Thomas, Craig; 'Configuration Management Practices and Tools Used for Linux Kernel Development', 14th International Conference on Software Quality, 2003.

Tools May Come, and Tools May Go

…But Some Things Never Change

Scott A. Whitmire
ODS Software, LLC
swhitmire@odssoft.com
Whitmire@ieee.org

Biographical Sketch: Scott Whitmire has been developing software professionally for over 24. He has designed and built a number of large applications and many smaller applications, primarily in the business software domain. He has researched and written extensively on software measurement and software design. In 1990, he developed 3D Function Points, an extension of the Function Point measure of software size. His work since has focused on measurement and design techniques as they apply to object-oriented software. In 1996, he developed a theory of objects based on category theory from mathematics and used that model to develop a number of measures for many characteristics, including several never before measured. Dynamic Design Analysis resulted from that same research.

Mr. Whitmire has written many papers and articles for both conferences and magazines. He is a contributing author to *The Software Engineering Productivity Handbook*, by Jessica Keyes, and *The Encyclopedia of Software Engineering*, edited by John J. Marciniak. Mr. Whitmire has spoken at several conferences, including the International Function Point User Group (IFPUG) Conference, the Application of Software Measurement Conference, and the Pacific Northwest Software Quality Conference. In 1997, he published, with John Wiley & Sons, the book *Object-Oriented Design Measurement*.

Mr. Whitmire is currently the application support manager for ODS Software and is one of the principal engineers for their current application. He has worked on software measurement implementation teams and was one of the founding members of the Software Engineering Process Group for his previous employer. He holds a degree in accounting from the University of Washington, and a Master of Software Engineering from Seattle University.

Abstract: Tools change nearly as fast as requirements in our business. Rather than focus on a set of tools, we are better served by learning software engineering techniques and principles that never change. Tools then become secondary concerns, with each new tool being easier to learn than the last. This paper presents 11 lessons learned over the years, both from developing software and from other disciplines. Starting with people over tools, the lessons include the critical nature of design, the need to analyze designs before code exists, the wisdom of learning from your mistakes, the need for performance and experience, the efficacy of peer reviews and inspections, the notion that complexity can't be designed but must evolve from simplicity, and the need to engineer everything associated with the product and project.

Copyright © 2004 by Scott A. Whitmire. All rights reserved.

Software development today is like Calvinball: We make the rules up as we go along and we never play the game the same way twice. It is no wonder that we software people are so enamored with technology; it's the only tangible thing in this business.

- Object-Oriented Design Measurement, (Whitmire, 1997)

In today's environment, tools change nearly as fast as requirements. If you stick with one set of tools over the course of a large project, you find yourself one or even two generations behind when it ends. If you use a tool long enough to develop some expertise with it, you are so far behind, you can't catch up, or so it seems.

If you haven't worked on a variety of projects, or worked in a variety of problem domains, or used a variety of tools and techniques, you may not have noticed there are a number of constants in software development. There are core principles and practices of software engineering that do not change as we move from tool to tool. If we learn these, we can excel at developing software and tools will become secondary concerns. Learning a new programming language is easier when you realize that all languages provide the same basic structures, but wrap them in different syntax.

Rather than worry about being made obsolete, we should learn to excel at developing software no matter what kind of tools we use. We should master one or more problem domains, and learn ways to quickly master new problem domains. The ability to learn a new business is a highly marketable skill.

The sections that follow describe many of the core principles I have learned over the last 24 years. While there are many others, I have found these 11 to be key to the success of many projects, and the causes of failure for many, many more. Further, these are not new; they come out of lessons learned a long time ago by some, and not learned at all by many. The principles are listed roughly in order of importance, but don't attach too much significance to the order; they are all important.

1. Tools Don't Build Software, People Do

Good people can build good software using bad tools. In fact, good people can build good software using any tool. Good tools will not result in good software unless you have good people. We see this time and time again, yet we still behave as if all we need to do to save the next project is buy the latest tool and use the cheapest labor.

I've participated in a number of projects where management, and most of the developers, believed that using a particular tool was all that was needed to produce quality. Concepts such as coupling and cohesion no longer mattered. And design? It was a thing of the past. In every case, just a short time into the project, problems started to pop up. We quickly found ourselves changing the same function for the fourth or fifth time, adding yet another new parameter to a function. Of course, this forced changes to all of the code that uses the function. We quickly sank into an abyss of rework; rework not caused by defects, but rework due entirely to the fact that we failed to design the software, rushing too quickly to code. Slowly, the developers, at least, came to realize that the tool did not solve the problems (and that it brought a few new ones), that coupling and cohesion still mattered, and that designing before you build can save countless hours in wasted effort.

At the same time, as I move from project to project, I find that learning a new tool gets easier each time. I've observed this pattern in others, too. Some people just know how to build good software, and tools just don't matter.

The main difference between engineering a product and merely building one is that the engineer looks for ways the product can fail and designs around them before the product is built. Software people don't think this way: We tend to build the first design that comes to mind that looks like it might work. We no longer have time to build a design then rebuild it one or two more times after we discover that it doesn't fulfill even the basic requirements.

This change in emphasis requires that we refocus our efforts on how our design might fail, then adjust it to remove the potential for failure. We continue this process until we can think of no more ways for the design to fail. This doesn't take as long as it sounds, and certainly takes less time than building something twice.

Many people claim that the tools will drive the shape of the product. A current project is transitioning to the .Net framework and I've been told a number of times that .Net is different, that everything about our application will have to be different than anything I've ever done before (mostly by people who've never used anything else). The truth of the matter is that .Net is object-oriented and makes use of many of design patterns that allow a high degree of flexibility, including run-time decision-making. The solutions look like some of the designs I've implemented or have seen implemented. .Net is not a new technology; it's a new set of tools packaged for an old technology. Of course, that is no reflection on .Net as a product. Good or bad, smart engineers will be able to make it work, like any other tool.

Furthermore, the architecture of our application has not changed as we have changed tools. It has always been a three-tier application partitioned vertically into user interface, business logic, and database layers. The only difference has been the tools we use to implement the various layers and the interfaces between them. The point is that tools do not define the architecture of a system; the problem defines it.

In bridge design, the length of the span is the primary factor in selecting the type of bridge to build. The nature of the ground underneath the foundations at the abutments and piers, the climate in the area, and even local politics may alter that selection, or they may render the location unsuitable for any kind of bridge.

The architecture of a business system is determined by how many users there are and where they are located, by where the data is generated and where it is used, and by how much of the time the system must be available for use. These issues drive the selection of tools, not the other way around, though many of us find that the tools are a given. The architecture of a scientific or numeric processing application is driven by the nature and volume of the data, the nature of the processing to be performed, and the required form of the output. The architecture of a real-time process control system is driven by the nature of the process, the number of simultaneous steps, the speed with which the system must react to anomalies, and the degree of required or desired human intervention.

In all of these cases, the architecture can be largely determined in a short conversation with the problem domain experts. Tools never enter into the discussion, at least not initially. In most cases, budget and time constraints have more influence over the architecture than do the tools used. Tools and technologies can and do have a major influence on the physical design and packaging of the logic into modules, however. We have a tendency to let our tools drive the logical design; this is usually a serious mistake.

2. Analysis and Design are Activities, not Phases

One characteristic of most of the early analysis techniques, such as DeMarco's Structured Analysis (DeMarco, 1978), is that they were more design than analysis. This doesn't mean they didn't work. I still use the notation and some of the processes in Structured Analysis, but when I do, I'm designing, not analyzing. Given some real or conceptual thing in the problem domain, there are always two or more ways to represent it in software and choosing which representation to use is a design decision. Determining the properties of the thing is analysis, deciding which properties to include and how to represent them in a model or in code is design.

To analyze something means to take it apart. Analysis activities work to discover properties of things that already exist. They are exploratory in nature, working from the whole to the various parts. Design, on the other hand, means to build something new. Design activities are creational in nature. They work to

build a thing that does not yet exist, working from the various parts to the whole. We analyze a problem to understand it; we design a solution to solve it.

All phases of a software project have both analysis and design activities. At the start of a project, we analyze the business and its practices to understand which things we need to include in our models, and which practices we need to support. We are designing when we determine how to do that. As we move towards construction, we analyze potential solutions in terms of how well they will support the overall design, how they will behave under the expected loads, and how easily they can be modified. We also constantly analyze and design while writing code. We evaluate various code structures to see which one will best meet our needs, and we analyze our source code to detect errors.

This also applies to testing. When we first receive code to test, we must analyze it to understand how it works and how we might determine if it works correctly. Knowing what "correct" even means may require significant analysis. Once we understand the testing problem, we design our test cases and our testing procedures in order to develop an experiment that will support or refute the hypothesis built into the design.

3. Software Design is a Critical Activity, Not a Luxury

When we design and build a bridge or other structure, the end product is a physical thing: We can see it, touch it, and use it. We know if our design is correct when the structure does not fail during the design life of the structure. Building software, however, is more difficult. It has been described as one of the most difficult mental activities attempted by humans. The biggest difficulty is that all software work is in done in the abstract. We conceive of designs, we write code, we execute that code, but we can actually see only the indirect results of that execution, never the execution itself. Working in the abstract is always more difficult that working in the concrete. We should use all of the tools at our disposal to make our work easier, and design is the neglected stepchild in our toolkit.

I troubleshoot production software for a living, and the vast majority of the problems I find are design mistakes. The most common is to not find all of the failure modes, followed closely by not finding all of the unintended side effects. Little or no thought was put into changes to the code, no questions were raised about how a change might affect other parts of the system, and no thinking was done as to how a change might fail.

Managers and developers complain they spend too much time writing documentation and not enough time writing code. Their response is to shorten the design time even further on the next project. They fail to notice, however, that the real result is almost always lengthened development and system test cycles. The mistake is not realizing that the design product isn't the same as the design process (as opposed to the definition of the process) and is far less important. The thinking and discussions that take place during design are critical to the success of the project. Even though the thinking is more important than the product, once you've made a design decision, you may as well write it down. My own design documents usually consist of hand-written notes and diagrams and I tend to record large portions of my notes in my code. I learned to do this after looking at some old code and wondering what I was thinking when I wrote it.

Eliminating the need to design software has long been a stated goal of new tools and methods. We just need to build it and the tool will take care of everything. Trouble is, no tool or technique can eliminate the need to think about what we're going to do before we do it. Materials on agile methods describe techniques for constructing software and do not mention specific design activities. These techniques can be very effective, once a feature's place in and interaction with the rest of the application has been determined. Small teams focused on a single module, function, or feature should not concern themselves with how they might affect the rest of the system. Their design contract needs to be defined before they start as it provides their requirements, their expectations, and gives hints as to how they might test their product.

Regardless of the tools or techniques you use to build your software, at some point before any software is built, someone with a lot of mental horsepower has to conceive of a product that can never physically exist. We've all been involved in projects where this didn't happen, and we all know the results.

In any design consisting of more than one module, the interfaces between the modules must be defined as early as possible. In fact, any point at which two or more modules interface, including the database, needs to be fixed as soon as possible, and most certainly before the modules are built. Bitter experience shows that if we fail to deliberately establish these interface points, they will change over the course of the project, with dramatic consequences to the code inside individual modules. Changing a design is easy, but once the design is written into code, it may as well be steel and concrete. Changing code is very expensive, and should be avoided if at all possible; it is easier and cheaper to build something right the first time than it is to build it over.

The whole purpose of engineering as an activity is to obviate failure (Petroski, 1985). One of the primary tasks of any engineer is to imagine how a product might fail and then adjust the design to eliminate or reduce that possibility. Design is one of the three critical activities in any software project. We can't escape it or deny it: All software is designed; most is designed poorly. Of course, understanding how a design might fail first requires understanding how a design will behave under various conditions before it is built.

4. "Analysis" Includes Figuring Out if a Design Will Work Before it is Built

The classic V-structure, in Figure 1, shows the delay between doing something and knowing whether you did it right. The width of the V represents the length of the delay. This V also highlights another interesting thing about software development: The activities on the left are listed in order of importance from top to bottom. Requirements are more important than design because you must first solve the correct problem, then solve it correctly. The fact that the delay is greater for the more important activities tells us that we should do something to shorten it. There is a lot of good material about how to gather and test your requirements early in the process. Nothing comparable exists for design.

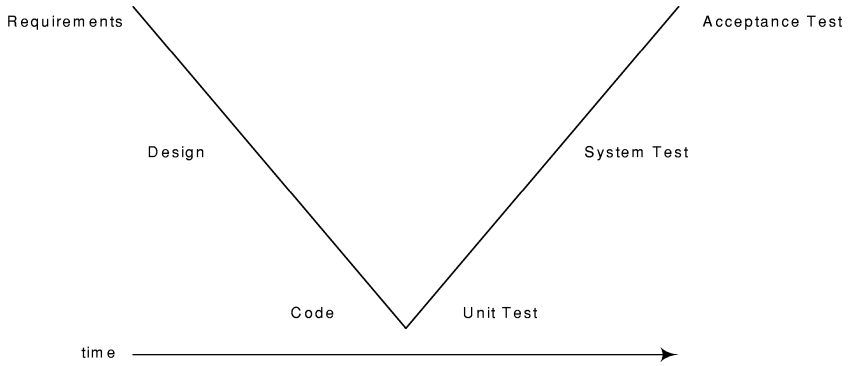


Figure 1. Delay between doing an activity and the ability to validate the results.

With respect to a design, you need to know that it will work, that it will scale well, and that it can adapt, all before it is built. A design works when it fulfills its intended function correctly and does not fail under any foreseeable circumstances. Graceful refusal to perform some operation is one of the goals of careful design and constitutes correct behavior. If some event must be avoided at all costs, the application must ensure that the event either never occurs, or is dealt with in a safe and graceful manner.

A design scales well when it can handle a much smaller or much larger volume of activity than first anticipated. In a data sampling application, the sampling interval suitable for a high volume of data might be too coarse to detect a small sample. On the other end, a design can be overwhelmed by a large volume of data because of performance problems in one or more areas. One of my recent projects required a complete re-engineering of a major part of the database design because of unforeseen performance problems built into the initial design (on the advice of the database vendor). These kinds of changes can easily derail a project, and nearly derailed ours.

A design is adaptable when you can change what needs to be changed without needlessly changing other parts of the system. We know that requirements will change over time, and we probably have a good feel for where those changes will likely occur. A smart engineer will isolate these potential hot spots from the rest of the design so they can be adapted without causing changes to ripple through the rest of the system. This design strategy also aids in testing an application.

To determine whether a design will work, you need to be able to analyze its behavior before you have code to test. Taken as a noun, a design is a thing: It has properties and contains parts. We can analyze a design to discover these properties and identify it parts, including its dynamic behavior in the presence of external stimuli. Dynamic Design Analysis is a technique to represent a design's static structure and dynamic behavior in a way that we can manipulate, seed the design with an event, and trace the messages and state changes in response to that event (see Whitmire, 1997). Dynamic Design Analysis also allows us to test the adaptability of a design by modeling changes and seeding the modified design with an event so we can see where new failures might arise.

5. Learn From Your Mistakes, and Those of Others

Pilots in training are advised to learn from the mistakes of others, as they won't live long enough to make them all themselves. The entire infrastructure built around aviation regulation exists to prevent accidents. When one occurs, this infrastructure is mobilized to find out what went wrong and what can be done to prevent it from happening again. We would do well to adopt a similar view. It's a lot easier to learn from the mistakes of others when we don't have to suffer the consequences. To learn from others, read everything you can get your hands on.

To learn from your own mistakes, you have to first know what you're doing wrong. Every project has a rich source of data about its mistakes. Analyze your defects, both reported defects and failed test cases, to determine the root cause or source of the error. Once you find it, you need to figure out a way to prevent it from happening in the future.

On a large manufacturing application, our group had quadrupled in size in an attempt to cut down a two-year backlog. While we quadrupled the output of changes, we didn't cut the backlog. More importantly, we quadrupled the number of defects that caused the factory to shut down. We analyzed our defects and traced over half of them, including 80% of those that interrupted production, to a single cause. We fixed that cause, using both procedural and technical means, and the defect production immediately dropped by the expected 50%.

On my current project, the cause of our biggest set of production problems is the failure to consider existing data. When we install a new release of the software, there are always transactions created prior to the release that must be closed by the new software. Invariably, we miss some rule change that renders these old transactions unusable or a conversion script is run too late, and we get production failures the first few days after the release.

In terms of process improvement, if an organization measures nothing else, it should track and analyze its defects by cause. Doing so will teach them more than any other measurement about how they really work and will lead to more beneficial improvements than any other method.

6. Performance Matters

Your customers will complain once if you deliver your product late. They'll complain every time they use it if it doesn't perform well and may even quit using it. After incorrect behavior, poor performance is the most visible problem you will face.

We've been conditioned over the last 20 years to believe that the easiest way to solve a performance issue is to install the application on a bigger, faster machine. The trouble is, they don't make enough hardware to fix software that isn't designed with performance in mind. We need only to look at our experience with desktop applications to see this.

Performance doesn't matter in every part of an application, but every application has some parts that must perform well. One of the big challenges for the system's designers is to identify these parts up front and design them to perform.

It is not always easy to tell which parts of the system have critical performance requirements. One thing we've learned about user interface design is that people find a system easier to use if the system waits for them more than they have to wait for the system. Every class of user has one or two critical tasks they perform repeatedly, some part of the system where they spend most of their time. If that part performs, they'll be happy, even if they have to wait for some task they don't do very often. The easiest way to find these critical tasks is to watch the users work. They don't always know which tasks are the most critical if you just ask them.

In other types of systems, the critical performance requirements come from sources other than users. If a real-time process control system has to respond to an emergency in less than $1/100^{th}$ of a second, it has to be designed so that no matter what else happens, it can detect an emergency and start the response within that $1/100^{th}$ of a second. If the data coming into a function has to be sampled every 100^{th} interval, then the system must perform fast enough to keep up with the data stream regardless of the speed at which data comes, otherwise, it's not a useful system, and may even be harmful.

7. Experience Matters

It is easier to teach an accountant how to write good code than it is to teach a programmer good accounting. In other words, experience in your application domain is more important than experience with a particular set of tools or even software development in general. When building a project team, start with people who have successfully solved similar problems in the same or similar domains. If you're building a wholesale distribution system, start with people who've built one before, and at least three is even better. Make them your senior developers and one of them your chief engineer.

Next, you want to load the team with good, solid software development experience. Find people who have a history of developing quality product, including people who write code that can be read by someone else. Smart people can pick up a new tool in no time. It's much more difficult to learn a new application domain and, sadly, sound software development practices.

In general, experience matters far more than certification. Certification programs qualify you for an entry-level position in a shop that uses a particular tool. If that certification doesn't come with lots of experience, a team will flounder the first time it's confronted with something it hasn't seen before. While it is possible to grow experience, and we all have to start somewhere, it's not a good idea to load your team with certificates without experience.

Make use of tool experts, but don't expect everyone to be one. A single, top-notch tool expert can support a surprisingly large project. Include expertise in all of your tools, including the operating system itself. Further, don't let your tool experts drive the project. Let the domain experts drive the project, assisted by your experienced software developers.

One job of the project manager is to allocate the budget so that the project has the optimum mix of skills and experience. Money spent on domain expertise will go further than money spent on tool expertise. A team with experience in the problem domain can deliver a quality product without serious expertise in their tools. A team with little experience in the problem domain will almost never deliver a quality product. We see this in other disciplines, too.

On every project to build a large structure, be it a bridge or software application, one person needs to serve as chief engineer ("architect" isn't the right name for the job). The chief engineer is responsible for the end-to-end technical aspects of the system. As teams work on detailed parts of the system, the chief engineer ensures that teams don't interfere with each other. All decisions that affect more than one team need to be reviewed by the chief engineer. Large projects that are not organized this way almost always implode from their own weight. Teams will build components that don't work with other components. Worse, they may change components upon which others depend, with the results not discovered until system or acceptance testing (this is the main reason for regression testing).

A wag once said that judgment comes from experience, and experience comes from lack of judgment. This is cliché that happens to be true. Experience brings with it lessons learned, mostly from mistakes, since we tend to learn more from our failures than from our successes. The history of bridge design is full of stories about how a successful design (a bridge that didn't fail) failed when used to cross a slightly longer span, sometimes with disastrous results. When this happens, experience learns that designs don't always scale well and will exercise more care the next time a longer span is attempted.

A similar notion applies to software developers and their tools. After we've built a product using a set of tools or technologies, whether successful or not, we have learned things about the tools and ourselves that get recorded as part of our experience. One of the drawbacks to introducing a new set of tools for every new project is that we have to relearn much of that prior experience. This is like the bridge designer who always uses the latest materials or designs on the next project. He never has the experience with either the materials or the designs to understand their inherent strengths and weaknesses. If he happens to overlook a critical weakness, a bridge fails. The same is true for software projects.

This does not mean that new tools and technologies are to be avoided. Even here, we can learn a lesson from civil engineering. When confronted with a new problem—a longer span, swifter water—or when first using a new material or technique, a competent civil engineer will exercise more caution than usual. They might perform load calculations three and four times rather than twice. They might conduct experiments to determine the true characteristics of the material or design. New materials are used with more than the usual caution, and new techniques are subjected to more than the usual tests and analyses.

Thus, when we decide to bring in a new set of tools, we need to exercise more care than usual. This may mean building a throwaway prototype to get a feel for the tools. We may conduct a side experiment to build some test harness for a particularly nasty part of the problem domain. We may build a test environment or simulator to provide experience with how the tools, and products built using the tools, will stand up to various load levels. This naturally means that for a project of a given size, it will take longer to build with a new tool than with one we've used before. This is to be expected and should be accepted as one of the costs of incorporating the new tool. When we don't have experience, we have to get it somehow, and it's safer to get it from experiments and extra caution than from charging forward and building the final product.

8. Coupling and Cohesion do Matter

Coupling describes the nature and extent of the connections between elements of a design. These connections can be either logical or physical. Dependencies follow lines of coupling. As the scope of the coupling increases, so does the portion of the system that must be changed when any part within that scope is changed. Tight coupling limits the degree that we can reuse components without dragging along

other, unwanted components. It also leads to a particularly nasty class of defects when a tightly coupled component is not changed along with its partners.

Coupling does not depend on the language you use or the paradigm in which you build your system. It originates in the problem domain and can only be made worse by the designer. Berard (1993) defines two classes of coupling: necessary and unnecessary. Necessary coupling is that which is inherent in the problem domain and includes the coupling required to support interactions among components of the design. This necessary coupling can be mitigated, but not eliminated.

Unnecessary coupling includes everything else and is nearly always the result of bad design. Functions that are coded without much design work become tightly coupled to other components as work progresses. It is easier to add a control flag and modify a working function slightly than it is to build two new functions, even though this may be the best choice in the long run.

The design goal is to minimize undesirable forms of coupling. We do this by examining the interfaces between components. Myers (1976) defined six types of coupling between components. They are, from better to worse: no coupling, data coupling, stamp coupling, control coupling, common coupling, and content coupling.

One desirable form of coupling that came along after Myers's list is abstract coupling (Gamma, et al, 1995). A class participates in abstract coupling if it is coupled to an abstract class. In this form of coupling, the class is dependent only on the type or interface of the abstract class, not on the implementation of its properties. Making use of this form of coupling helps avoid compile-time dependencies between classes in a design.

Another form of coupling is called value coupling and was not addressed 1997 (Whitmire, 1997). It exists when one object's value, or the value of one of its attributes, depends on the value of some other object, and there is no other form of dependence. As an example, consider the case were the domain-level identifier for an object is composed of two or more attributes and the combination must be unique. If another object comes along with the same combination, the identifiers for both the new object and the original must be changed using some sort of tiebreaker scheme. This form of coupling can affect entire chains of objects, and you have to change the problem domain to get rid of it.

Value coupling is a variant of Myers's common coupling, but is not the same in that the dependence exists only in the problem domain and is not a physical property of the design. This form of coupling is insidious because you have to change the problem domain to get rid of it, and keeping it can cause changes to echo back and forth between components.

Cohesion is the degree to which parts of a software component are related to each other. In the ideal, a function or method does one thing and only one thing. An ideal class would represent one and only one domain object, or one and only one design concept. Or, as Booch puts it: "…The class Dog is functionally cohesive if its semantics embrace the behavior of a dog, the whole dog, and nothing but the dog" (Booch, 1994, p. 137). On a technical level, cohesion deals with the strength of the logical connections between elements of a component, or between the component and a single, well-defined purpose. Unlike coupling, which originates in the problem domain and can only be made worse by the designer, cohesion is entirely in the control of the designer.

Highly cohesive components are easier to maintain and reuse, mainly because their purpose is very clear. A component that is both cohesive and complete (see Whitmire, 1997, ch. 11) can be used in any application that requires the abstraction the component implements. A library of complete and cohesive domain components would save somewhere around 80 percent of the effort required to build an application in that domain.

Coupling and cohesion often behave as the inverse of each other, with tighter coupling leading to lower cohesion. When dealing with the two, you can work the coupling and the cohesion will usually take care of itself.

9. Complexity Evolves from Simplicity

You just cannot build a complex system the first time you try. As Gall put it: "A complex system that works is invariably found to have evolved from a simple system that worked…A complex system designed from scratch never works and cannot be patched up to make it work" (Gall, 1986). We see this time and again in nature. Many of us have proven this in our software experiences. In my own case, the most successful software products started out as simple programs that did one thing. I built upon these programs, adding a new feature here, or expanding a feature there, until I ended up with some incredibly complex yet robust software. This is the basic principle behind the concept of evolutionary prototyping.

Managing complexity is the hidden purpose underlying many of the techniques we use to analyze a problem, design and build software, and manage software projects. Brooks says that complexity is an essential property of software, not an accidental one (Brooks, 1987). That is, we may be able to manage it, but we can never get rid of it. The first chapter of Booch's *Object-Oriented Analysis and Design With Applications* (Booch, 1994) provides a wonderful essay on the nature and sources of complexity in software.

There are many techniques for partitioning a problem into manageable chunks. I find that working in terms of objects is one of the best. When faced with a complex problem domain, I first identify the relevant objects in the problem domain, both real and conceptual, and then concentrate on each one in turn in order to understand its properties and behaviors. Since we rarely have time to start simple with working code, we have to move from the simple to complex while still doing analysis and design. This might mean that our physical design closely follows the problem domain the first time. Then, as we look at each part in turn, we apply our catalog of design patterns, making the design more complex, but easier to build (your design can never be less complex than the problem domain). Analysis and design methods help with this process to a large extent, but don't always go far enough. Newer techniques such as Dynamic Design Analysis can help take us to the next level as we try to master complexity innate to our problem before we try to build the code.

We can apply this principle to the build sequence, the order in which we build the modules. In most systems, the critical interface, and the place where most of the errors and changes will happen, is the point at which the application meets the outside world. Building this interface first can identify some critical problems early, when they are easy to fix, and also provides working code you can show off. The principle is to build from the outside in, and it applies to all types of applications. It also applies to papers written for conferences.

As it is built, this interface point needs to be supported by scaffolding code in order to function. This scaffolding code can be built with the correct signatures, but might at first hard-code any expected results. As you move down through the application's design, you replace this scaffolding with fully functional software, in turn supported by its own scaffolding.

The genius in software design is to take a complex problem and be able to find just the right way to break it into it component parts so that people can understand each part more fully. Some people excel at this task; if you find one, hang onto them, I've met very few in the last 24 years.

10. Peer Reviews and Inspections Work

By now, this should be a no-brainer. All of the published data support the conclusion that peer reviews and inspections help prevent defects. Further, they are among the most cost-effective tools to

come along. I have seen no data to support the opposite conclusion. Why, then, are we so reluctant to accept them?

My experience over several projects in several different organizations is that the effort saved by detecting defects with reviews is roughly ten times what it would have cost to fix the defect had it been found in system test or production. My current project team once found a design defect that would have taken several hundred hours to fix had it made it into the code. It took us only a dozen person hours to find it in the design, which is quite a payoff. Other organizations have similar experiences.

Karl Wiegers describes many benefits from peer reviews and inspections, both financial and less tangible (Wiegers, 2002). One benefit is the knowledge passed around the project team in regards to the product, the project, and even basic software development skills. Peer reviews are an excellent way to pass experience from one team member to another. In addition, they help ensure that everyone is on the same page, so that individuals and small sub teams don't stray too far from the project vision. They also allow members of one sub team to hear about and potentially use solutions developed by members of another sub team.

There are very few downsides to using peer reviews and inspections. The reasons for not using them usually have their basis in misperception or fear. There are, however, some organizations that have stopped using inspections, but not for the reasons many would believe. I know of one major company that was inspecting requirements, designs, and code. After a time, they discovered that their code inspections weren't finding as many defects per inspection hour as they would expect, and finally decided to quit doing them. The reason? Their requirements and design inspections were so effective that very few defects managed to escape into the code.

11. Engineer Everything

No system consists solely of source code; there are always database scripts, batch jobs, job schedules, stored procedures, shell scripts, and processes that work with the core source code to get the job done. In order to work together smoothly, each component as well as the total package must be carefully engineered. Processes include the procedures used by the people that work as part of the system's support structure.

While under construction, and during modification, the code is supported by scaffolding, test harnesses, simulators, and test cases. Each of these need to be designed with the same care and rigor as the actual source code. If the scaffolding is engineered carefully, the construction process proceeds more smoothly. Higher levels do not have to change as the scaffolding is replaced with finished code. This is true, however, only if you have first verified the design. Scaffolding is not free, and you have to balance a number of issues in order to get the most benefit for the money.

Much has been written about the need to engineer test cases. However, test cases are not all that is required to test a system. Often, extra code must be written as a surrogate for other parts of the system. This surrogate code needs to be designed and engineered, if only to ensure the validity of the tests.

Even the sequence in which we build the parts of the system needs to be engineered. To partition a project into manageable chunks of work, we need to look at the design, the nature of the problem at hand, end even the politics surrounding the project. There are always at least two sequences in which we can build these chunks, as we've seen: from the inside out, and from the outside in. The choice of build sequence involves a number of factors and tradeoffs. Whichever sequence is used, someone on the project must know exactly what needs to be built eventually, and where all those pieces will fit when they finally get built.

Two of the most important processes in any development organization are configuration management and the processes used to release and distribute the product, including the conversion of existing data. These processes must be correct and stable. A correct process prevents errors from getting

into the final product. A stable process doesn't change constantly, causing problems with every change. However, a process is not an end in and of itself. A process allows a competent engineer to come to reasonably correct solutions to a given problem. It cannot guarantee that several engineers will come to the same conclusion, nor can it guarantee that a single engineer will come to the same conclusion every time the process is used.

A process can be thought of as a software program executed by people. This means that we can use all of the tools we have for designing and analyzing software on our processes. We are masters at working in the abstract, so working process changes should be second nature to us.

Most of the effective process changes that we adopt come about because defects are getting into the final product. Like aviation regulations, process changes are the result of analyzing a group of defects to try and stop them. A good process catches more defects than it lets through. A perfect process lets no defects through. Since perfection is hard to obtain, we'll settle for very, very good. This is the reasoning behind many of the recent developments in management such as Total Quality Management, Six Sigma, and the Software Engineering Institute's Capability Maturity Model. In all cases, the intentions are in the right place, but some adopters go overboard and lose sight of the primary intent. When that happens, "process" becomes a dirty word, and as you well know, that happens a lot.

Despite admonishments that processes should be improved constantly, too much change too fast can be dangerous. Like releases made midway through a transaction, process changes can allow defects to sneak into the product, especially when they are instituted in mid-activity. In addition, an unstable process frustrates not only those who use it, but everyone merely connected to it.

One last point is that a good process, like good tools, is not a substitute for good people. Software is primarily a mental activity, so start with good people first, especially people who've solved problems similar to yours.

Summary

In an environment where tools change as fast as requirements, we sometimes fail to notice that some core principles never change. While software development is a very complex undertaking, we have learned many ways to manage that complexity. We can learn to excel as software developers independently of the tools we use. Learning the core principles of software engineering can help prevent us from being made obsolete by the continuous change in the tools market.

None of the principles and observations in this paper is new. Even so, few practitioners act as if they've learned the lessons. Time and again I watch and participate in projects that fail because they don't follow these few, simple principles.

If you find yourself becoming obsolete, learn from my mistakes as well as your own, and become a better software developer. Excelling at a skill that is broadly required is always better than continuously working as a beginner on the latest tool.

References

Berard, 1993
: Berard, E. V., *Essays on Object-Oriented Software Engineering, Vol 1*, Prentice-Hall, 1993.

Booch, 1994
: Booch, G., *Object-Oriented Analysis and Design with Applications*, 2e, Benjamin Cummings, 1994.

Brooks, 1987
Brooks, F., No Silver Bullet: Essence and Accidents of Software Engineering, *IEEE Computer*, vol 20(4), 1987, p12.

DeMarco, 1978
DeMarco, T., *Structured Analysis and System Specification*, Yourdon Press, 1978.

Gamma, et al, 1995
Gamma, E., R. Helm, R, Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

Gall, 1986
Gall, J., *Systematics: How Systems Really Work and How They Fail*, 2e, The General Systematics Press, 1986, p65.

Myers, 1976
Myers, G., *Software Reliability—Principles and Practices*, John Wiley & Sons, 1976.

Petroski, 1985
Petroski, H., *To Engineer is Human*, St. Martins Press, 1985.

Whitmire, 1997
Whitmire, S., *Object-Oriented Design Measurement*, John Wiley & Sons, 1997.

Wiegers, 2002
Wiegers, K., *Peer Reviews in Software: A Practical Guide*, Addison-Wesley, 2002.

How Quality is Assured by Evolutionary Methods

Niels Malotaux
N R Malotaux - Consultancy
Bongerdlaan 53
3723 VB Bilthoven
The Netherlands
+31-30-2288868
niels@malotaux.nl
www.malotaux.nl/nrm/English

Author bio

Niels Malotaux is an independent Project Coach specializing in optimizing project performance. He has over 30 years experience in designing hardware and software systems. Since 1998 he devotes his expertise to teaching projects how to deliver Quality On Time: delivering what the customer needs, when he needs it, to enable customer success. Since 2001 he coached more than 25 projects, which led to a wealth of experience in which approaches work better and which work less.

Abstract

After several years of experience as a Project Coach, introducing Evolutionary Project Management Methods (Evo) in software and other development projects, I think I can claim that Quality is Assured if projects apply these methods. This paper describes an evolutionary approach to project management that emphasizes that the purpose of software development projects is to deliver what the customer needs, at the time he needs it, to create substantially greater value than the cost of development and to enable customer success.
We describe the basic Evo approach, followed by several practical details, which the reader can implement tomorrow in development projects, to increase the quality of the products being developed, as well as decrease the time needed to implement the products.

Working the Evo way means organizing the work in weekly (or even shorter) Task-cycles. In these Task-cycles we optimize estimation, planning, and tracking. Task-cycles feed bi-weekly (or shorter) Delivery-cycles by which we optimize the requirements and our assumptions. We use a practice known as TimeLine to create and maintain the total project scope and to connect the Project Result, through the Deliveries, with the actual work organized in Tasks. Evo combines Estimation, Planning, Tracking, Requirements Engineering, Requirements Management, and Risk Management into Result Management. Result is defined as the combined value we provide to all the Stakeholders of our product, ultimately leading to customer success. Evo relies on a fanatical dedication to ROI: Whatever we do should contribute to the Result and we avoid whatever does not contribute.

Deming said that quality cannot be tested into a product; it has to be designed in from the beginning. That's exactly what we are doing in Evo projects. We define what Quality is and then we pursue the defined Quality, constantly optimizing based on what we learn along the way. All the Quality Assurance people need to do is guide and coach us, watching over our shoulder to ensure we stay disciplined. Not because we like discipline, but because we like success.

Copyright © 2004, N R Malotaux - Consultancy
20040709

1 Introduction

After several years of experience as a Project Coach introducing Evolutionary Project Management Methods (Evo) in software and other development projects, I think I can claim that Quality is Assured if projects apply these methods. Does this mean that the Quality Assurance function is not needed any more? No. QA is still needed, because one of the main factors jeopardizing the Assured Quality is lack of discipline - discipline to keep applying the methods in order to meet our commitments.
In many cases, people know the best way to do their work. However, if nobody is watching, people tend to take shortcuts. If somebody is watching over their shoulder, people tend to take fewer shortcuts. The Project Manager can watch over the shoulders of the team. The team can watch over each other's shoulders. But who's watching over the Project Managers' shoulder? This task is the responsibility of management, but the Quality Assurance function can help.

Even with an assurance function in place, team members still have to know what is the best way to do their work in the first place. Since there is no absolute "best way", while the "best way" is even dynamically changing, we must also provide the people with methods to actively find out the best way while working in the project.
Evo is actually rapidly and frequently applying the Plan-Do-Check-Act (or Deming) cycle, not just for the development of the *product*, but at the same time for the organization of the *project* and even for assessing and improving the *methods* used on the project. We need to continuously ask ourselves: "What should we do *now*, in which *order*, to which *level of detail for this moment*".

Working the Evo way means organizing the work in weekly (or even shorter) Task-cycles. In these Task-cycles we optimize estimation, planning, and tracking. Task-cycles feed bi-weekly (or shorter) Delivery-cycles by which we optimize the requirements and our assumptions. We use a practice known as TimeLine to create and maintain the total project scope and to connect the Project Result, through the Deliveries, with the actual work organized in Tasks. Evo combines Estimation, Planning, Tracking, Requirements Engineering, Requirements Management, and Risk Management into Result Management. Result is defined as the combined value we provide to all the Stakeholders of our product, ultimately leading to customer success. Evo has a fanatical view of ROI: Whatever we do should contribute to the Result and we try to avoid whatever does not contribute.

In this paper I will explain the basics of this Evolutionary approach and practical details people can start applying immediately.

2 The Goal

Let's assume that the purpose of software development projects is to deliver what the customer needs, at the time he needs it, to create substantially greater value than the cost of development and to enable customer success. In short, we call this Quality On Time.

It is important to note that the *functionality* we are working on in most development projects *already exists*. Usually, all we are supposed to do is enhance the performance of specified functionality to create more value for the customer. The set of functions we are enhancing defines the scope of the project. The scope should be chosen such that it provides more value for cost than any another scope.

> *Banks have banked for thousands of years. First using clay tablets, then using card-trays and now using computers. Banks are, however, still doing what they did before. The function is still the same, while the performance (ease, speed, accuracy of transactions) is enhanced. If a new system does not deliver sufficiently more value than the old system, there will be no funds to pay for the new system and the developers.*

It would be nice if we could in one project develop the ultimate solution, creating the ultimate value. Apart from the risk that, when done, we could be out of work, this is not possible because of limited resources such as:

- The available time (time to market may strongly influence time to profit)
- The available money
- The available people and the capabilities of these people (it would be nice if we could hire the best people. Normally, however, the challenge is to succeed with *average* people)

- The available experience on the subject
- The available technology
- The capability of the users to adopt the new system

In development projects we can only strive to optimize the compromise between value creation and the available limited resources. If the results we can achieve, given these limited resources, are insufficient to provide significant value for customer success, we shouldn't even start the project. Given these limited resources we are not even satisfied with *good* results, we actively want to *maximize* the Result created. Looking back at the end of a project, not only should our customer have a big smile of satisfaction, we should ourselves also be confident that we couldn't have done better.

This implies that we should feel a Sense of Urgency to constantly optimize the results we are working on, to constantly optimize our success. Without this Sense of Urgency, Evo doesn't work.

3 Plan-Do-Check-Act

Since childhood we learn intuitively through experience. Besides learning from our own experience, we also learn from accepting the experience of others: at school, in workshops and at conferences. This learning process is rather slow. We can, however, stimulate the learning process by actively using the Plan-Do-Check-Act cycle, as presented by Deming:

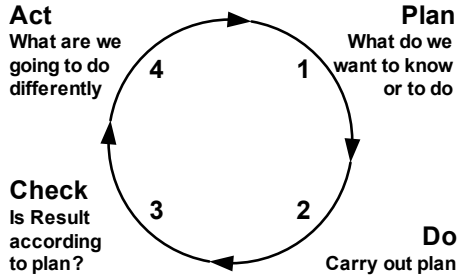


Figure 1: PDCA or Deming cycle

- **Plan**
  What are we supposed to accomplish and how are we going to accomplish it?
- **Do**
  Carry out the Plan
- **Check**
  Is the result according to the Plan?
- **Act**
  - If the result was not according to the Plan, what are we going to differently the next time to achieve a better result?
  - If the result was according to the Plan, was it accidental? How do we make sure next time the result is equally according to Plan?

Do is never a problem: we "do" all the time. *Plan* we do more or less, usually less. For *Check* and *Act*, however, we have no time because we think we want to go to the next Do. Well, that's what I believed until recently. Taking a closer look at what really is happening we can see that Check *is* often done: people seem to be quite aware what is going wrong and often even know what should be done about it. The real problem is that we don't Act: taking what we know and doing something about it.

Sometimes I hear people in a project week after week complaining about the same problem, usually that somebody else is doing something wrong. My advice is: either deal with it or stop complaining. Don't keep wasting energy complaining about the same problems over and over. Do something: Act! Find a solution, plan the time needed and solve the problem.

4 Evo

4.1 Evo

Evo is short for Evolutionary Development, Evolutionary Delivery, Evolutionary Project-Management, deliberately going through the Plan-Do-Check-Act learning cycle rapidly and frequently, for product, project and process, continuously thinking "*what* to do, in which *order*, to which level of *detail* for now". It's a label for a set of methods that allow us to effectively and efficiently run projects, delivering Quality On Time. Evo integrates Planning, Requirements and Risk Management into *Result Management*. It's *actively induced e*volution because we don't wait for evolution to happen, we make it happen.

Many organizations mandate a Project Evaluation at the end of every project. Even so, few projects do the actual evaluation because they feel that these evaluations do not contribute to better results. Why is this? Consider one-year projects. People have to evaluate what went wrong and what went accidentally right (and why) as long as a year ago. In addition, they may not be able to use the learning from an event until as long as a year after the fact. The idea of evaluation is valuable. The time constants of this process as described above are, however, beyond the capabilities of the human mind. In Evo, we do evaluations (PDCA) every week. This tunes the time dimension to the human mind's abilities and enables us to rapidly implement what we learn.

4.2 Evo and the Product

We don't know the real requirements. *They* don't know the real requirements either. So, stop pretending we know, and accept that we have to find out what the real requirements are, together. This includes finding out who *they* are. We can make the nicest systems, given unlimited time and money. However, our customer doesn't have unlimited time and money. If the customer cannot afford all what is possible, we must find out the best Result we can achieve within the limited resources. If that's less than the customer needs for success, we shouldn't even start.

Result is the value gained by the use of what we developed. Result ultimately is customer success. If no value is gained, there is nothing to pay our salaries from. Because not all customers are aware of this, we have to work with the customer to find out what the optimum Result is to make sure that we are generating significant value. In Evo we work with a no-cure no-pay attitude. Whatever does *not* contribute to customer success, we don't do.

4.3 Evo and the Project

The optimum Result is the best product for the least cost. At the start of the project we don't know what the optimum Result is, so we must organize the project in such a way that we discover and implement the optimum Result at the lowest cost. This implies optimizing the effectiveness and efficiency of discovery and implementation. It also means that we have to change our estimation practice from optimistic to realistic, so that we can predict the future more accurately. We have to accept the realistic estimates and plan accordingly. We have to dynamically keep our plans up to date in order to keep control over the Result. We must learn better time management and better priority management. These are among many issues we can improve. In Evo we are constantly, dynamically improving on these issues because our success is at stake. We do not only design the product, we also *design* the *project.*

We take time and money budgets very seriously. This means that we don't ask for more when we were supposed to deliver. If the budgets really were insufficient we could have predicted this way before the budgets ran out and, together with the customer, we could have acted accordingly.

4.4 Evo and the Process

Because it is continually being improved as a process, Evo is made up of the best set of methods we know at a given time. If we find a better way, we change to the better way. Not only do we employ PDCA on the product and project activities, we also constantly and dynamically apply the PDCA cycle to the methods we use. If another method seems better, we try it. We may experiment. But we deliberately Check and Act: if the new method is better, we change to the better method. If the new method is not better, we revert to the last known best method. Methods, processes and procedures are there to help us. If they don't, we discard them. A side effect is that Evo processes may be different between projects and between organizations because of differences in culture or differences in experience. The common property is always the urge for success in defined goals.

4.5 Does Evo cost more time?

Some people fear that all these evaluations, intensive planning and constant improvements will cost a lot of extra time. It does not: experience based on many projects proves that it *saves* time. The "extra" things we do in Evo projects are the things that should be done anyway on any project to make it successful. So, we don't really do "extra" things. We only do those things that contribute to Quality On Time.

4.6 When do you not need Evo?

There are circumstances where you may consider not using Evo, such as:

- The requirements are completely clear and nothing will change. *This is production, not development.*
- The requirements can be easily met with the available resources in the available time. *Still, Evo can make you achieve better results in shorter time.*
- The customer can wait until you are ready. *Still, Evo can make you achieve better results in shorter time. Why waste your time while you can do more interesting things?*
- The customer doesn't care about the result. *Should we contemplate this project? Is he going to pay?*
- You don't care about the cost or time. *Could be a hobby or a vacation.*
- Your boss doesn't care about the cost or time. *He probably doesn't know what to do with his money.*
- Management doesn't know what to do with the time saved. *Be careful, they may frustrate your project.*
- There is no Sense of Urgency.

Sense of Urgency is an important issue to watch for. Most people, including management, will immediately affirm the urgency of the best Result at the lowest cost. That's trivial. However, there are cases where their actions tell a different story. The remedy is either to educate them by coaching, or not to bother them with Evo. There are plenty of places where you can be successful with Evo, so why bother if they don't want to be more successful. Besides, Evo is never a goal in itself. Result is all that counts. *If* they get optimum results *their* way, you shouldn't complain, but rather learn from how they do it.

5 Evo basics

We organize Evo projects on several levels. We use the TaskCycle to organize the work, the DeliveryCycle to organize the Results and TimeLine for making sure we'll be on time.

5.1 TaskCycles

In the TaskCycle we organize the work. We are checking whether we are doing the right *things*, in the right *order*, to the right *level of detail*. We are optimizing our estimation, planning and tracking abilities to better predict the future. We select the highest priority Tasks, never do lower priority Tasks and never do undefined Tasks. As a practical rule, we plan 2/3 of the available time and in the remaining 1/3 of the time we do all those things we also have to do in the project, like small interrupts, helping each other, project meetings and many other things. If we plan 100% of our available time, we will still do all those other things, and we will never succeed in what we planned.

TaskCycles take at most one week, in some cases even less. Every Cycle we decide what is most important to do, how much time it takes to do it completely (we define what completely means) and then what we can do in the available time. We also decide what we will *not* do in this Cycle, because there is no time to do it. Now we can focus all our energy on what we *can* do, making us more relaxed and more productive. Some managers fear that planning only 2/3 of the available time makes people do too little. In practice we see people do more.

5.2 Task Selection Criteria

The following set of Task Selection Criteria proved useful for deciding the priority of Tasks:

- Most important requirements
- Highest risks
- Most educational or supporting things
- Active Synchronization with others outside your project

Remember: Every Cycle delivers a useful, completed Result.

5.3 DeliveryCycles

In the DeliveryCycle we organize Results to be delivered to selected Stakeholders. We are checking whether we are *delivering* the right things, in the right order, to the right level of detail. We are optimizing the requirements and checking our assumptions.

A DeliveryCycle normally takes not more than two weeks. Novice Evo practitioners, almost without exception, have trouble with the short DeliveryCycle. They think it cannot be done. In practice we see that, without exception, it *always* can be done. It just takes practice. One of the important reasons for the short length of the cycle is that we want to check our (and their) assumptions before we have done a lot of work that later may prove unnecessary, losing valuable time. Short DeliveryCycles help us do this with minimum risk and cost.

A common misconception of Deliveries is that people think they always have to deliver to users or customers. On the contrary, we can deliver to any Stakeholder: the user or customer, ourselves or any Stakeholder in between. This makes it easier to define Deliveries. However, we must always optimize Deliveries for optimum feedback: we must know what we are doing right and what we are still doing wrong.

5.4 Delivery Selection Criteria

The following set of Delivery Selection Criteria proved useful for deciding the contents of Deliveries:

1. **What will generate optimum feedback**
2. **What will make Stakeholders more productive *now***
3. **Delivering the juiciest, most important Stakeholder values that can be made at the least cost, to raise the Stakeholder's interest to provide optimum feedback**

Also remember that:

- Every Delivery must have a useful set of values, otherwise the Stakeholders get stuck (for example, if there is a *Copy* function, there should also be a *Paste* function)
- Every Delivery must offer clear incremental value, otherwise the Stakeholders stop producing feedback
- Every Delivery delivers the smallest clear increment, to get the most rapid and frequent feedback
- If the contents of a Delivery takes more than two weeks, it can be shortened: *try harder*

5.5 TimeLine

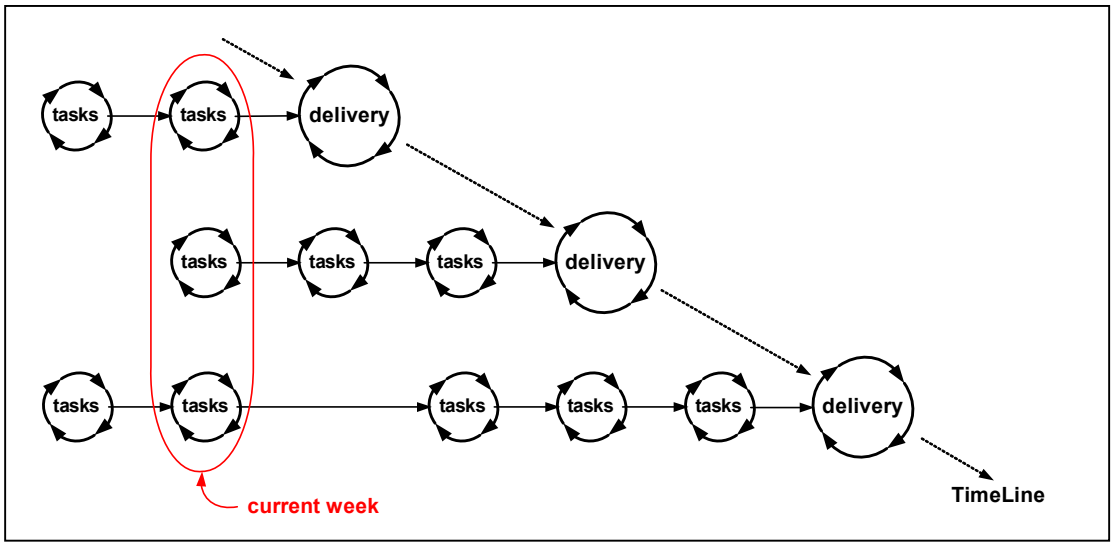
We use the TimeLine technique to make sure that we will be on time (or even early). A TimeLine is a line between now and then. *Then* is any deadline (we also call it FatalDate): End of Task, End of Delivery, End of sub-project or milestone, or End of Project. A FatalDate is a commitment to deliver successfully, no excuses. We took the responsibility, so the Result will simply be there. If it is not, we failed to deliver on time. At the FatalDate, any excuse is pointless, because you *could have known before*. The moment you can foresee that, for whatever reason, you are not going to meet the FatalDate, you could have told the appropriate Stakeholders and we could have adapted our plans accordingly. Any day later you realize that you cannot meet the FatalDate, you have a day less to cope with it. If the time is up, there is no time left. You cannot change history.

During a project we constantly monitor where we are now, what the FatalDate is, and constantly optimize what we should and what we can do in between.

5.6 Tasks, Deliveries and TimeLine

Tasks feed Deliveries (see figure 2). Deliveries create focus for what to do in Tasks. In any TaskCycle we are working on the current Delivery. Because some Deliveries need more than two weeks to prepare we may also work on Tasks for future Deliveries. That said, we shouldn't start working on future Deliveries too soon, because the longer we work on a Delivery, the more the world may have changed, so that what we already did has become irrelevant. It really is a challenge to define Deliveries and to start working on the right Delivery, Just-in-Time.

On the TimeLine we are scheduling Deliveries in the best order to achieve the best Result in the least time. This is a dynamic process, because we may have to redefine Deliveries based on experience of the developers, feedback from Stakeholders, and market changes. We are constantly challenging the order of Deliveries to get the best route to the Result, with the fewest iterations. We may also have to change the order of Deliveries if somebody crucial for a Delivery is ill, or is needed temporarily on another project.



6 Evo practice

By collecting the experience of more than twenty-five projects between 2001 and 2004, we have arrived at several best practices that you can use to start new Evo projects.

These practices do not describe theoretical processes or how someone *thinks* we should work. They rather describe what works in day-to-day reality, where we have to cope with human psychological behavior that is not always as logical as we intuitively assume or might wish were true. In fact, Evo thrives on reality. Because of this, you can start using these practices tomorrow and immediately benefit. You don't have to call it Evo. Result is all that counts. That is never just "following process". Result is always measured as customer success at the least cost.

6.1 TaskCycle planning

At the start of the weekly TaskCycle, this is what we do:

1. **Determine the number of hours you have available for this project this TaskCycle**
   People may work less than the full week. For example, they may take a vacation, follow a course, visit a dentist or work for more than one project. So we determine the number of available hours for this project first, because then we know when we can stop adding Tasks.
2. **Divide this gross number of available hours into:**
   - **Available Plannable Hours (default 2/3 of gross available hours)**
   - **Available Unplannable Hours (default 1/3 of gross available hours)**

   We only plan those Tasks that don't get done unless planned. If you plan, you *have* time, and after that time, the Task will be done.
   We do *not* plan Tasks that will get done anyway, even without planning. As a default ratio we start with 2/3 plannable and 1/3 unplannable time. In many projects this proves to be realistic. In a 40 hour work week, this means 27 hours plannable time, 13 hours unplannable time.
3. **Define Tasks for this cycle, using the Task Selection Criteria**
   Focus on finding Tasks that are most important now and don't waste time on less urgent tasks for the moment. Based on what we learn from current tasks, the definition of later Tasks could change, so don't plan too far ahead. Use the Delivery definition to focus on what to work in the Tasks.
4. **Estimate the number of effort hours needed to completely accomplish each Task**
   We always estimate *effort* hours. Ask people to estimate in days, and they come up with lead time (the time between starting and finishing the Task). Ask people to estimate in hours, and you'll find that they usually come up with effort (the *net* time needed for completing the Task). The reason for keeping effort and lead time separate is that the causes of variation are different: If effort is incorrectly

estimated, it's a *complexity* assessment issue. If there is less time than planned, it's a *time-management* issue. Keeping these separate enables us to learn.
Only the person who is going to do the Task is allowed to define the duration of the Task. Others may not even hint, because this influences the estimator psychologically. If others do not agree with the estimation, they may only challenge the (perceived) contents of the Task, never the estimated time itself. Ultimately, when we agree on the requirements of the Task, the implementer decides how much time he is going to need; otherwise there will be no commitment to succeed.

5. **Split Tasks of more than about 6 hours into smaller Tasks**
   We split the work into manageable portions. Estimation is not an exact science, so there will be some variation in the estimates. We are not bound by the exact estimated effort hours. We are only bound by the Result: at the end of the week, all committed work is done. If one task takes a bit more and the other a bit less, who cares? If you have several tasks to do, the variations can cancel out. If you have a massive task of 27 hours, it is more difficult to estimate and the averaging trick cannot save you any more.
6. **Fill the available plannable hours with the most important Tasks**
   Never select less important Tasks. *Always* fill the available plannable hours completely.
7. **Ascertain that indeed these are the most important Tasks to do and that you are confident that the work can be done in the estimated time**
   - Any doubt undermines your commitment, so make sure you can deliver.
   - Acknowledge that by accepting the list of tasks for this cycle means accepting the responsibility towards yourself and your team, and that these tasks will be done, completely done, at the end of the cycle.

At this point, you will have a list of Tasks that will get done. If you cannot accept the consequence that some other Tasks will not be done, do something! You could:

- Reconsider the priorities.
- Get additional help to do some of the Tasks. Beware, however, that it may cost some time to transfer the Task to somebody else. If you don't plan this time, you won't have time.
- If no alternative is possible, accept reality. Hoping that the impossible will happen will only postpone the inevitable. The later you choose to do something about it, the less time you have left to do it. Don't be an ostrich: in Evo we take our head out of the sand and actively confront the challenges.

6.2 Evo Task Administrator tool

In all the projects coached since 2002, we introduced the Evo Task Administrator, or ETA tool, which is used to administer the Tasks. This MS-Access application can be downloaded free from www.malotaux.nl/nrm/Evo/ETAF.htm, together with an explanatory text. A screen shot is shown in figure 3 on the next page.

6.3 TaskSheet

We use the TaskSheet to define what "completely accomplished" means. It helps us to check whether we are going to do exactly what is needed at this moment, not less and not more.
On the TaskSheet we can document:

- The requirements of the Task (Functional: *what*, Quality: *how well*, Constraints: *what not*)
- Task validation: how we are going to establish that the Task's requirements are met
- The strategy to succeed this Task (planning within the Task, design approach)
- Whatever is still unclear

Before starting with the "real" Task, we ask the Project Manager, the Architect, or a colleague to review the TaskSheet. This may take only a few minutes, but it can also take more time. The longer it takes, the more important the review. Most reviews lead to changes in the TaskSheet. That's nice, because we will be working more on the right things than we would have otherwise. After the definition of the Task has been changed, or better defined, the Task time estimate should be reconsidered by the person who is going to execute the Task.

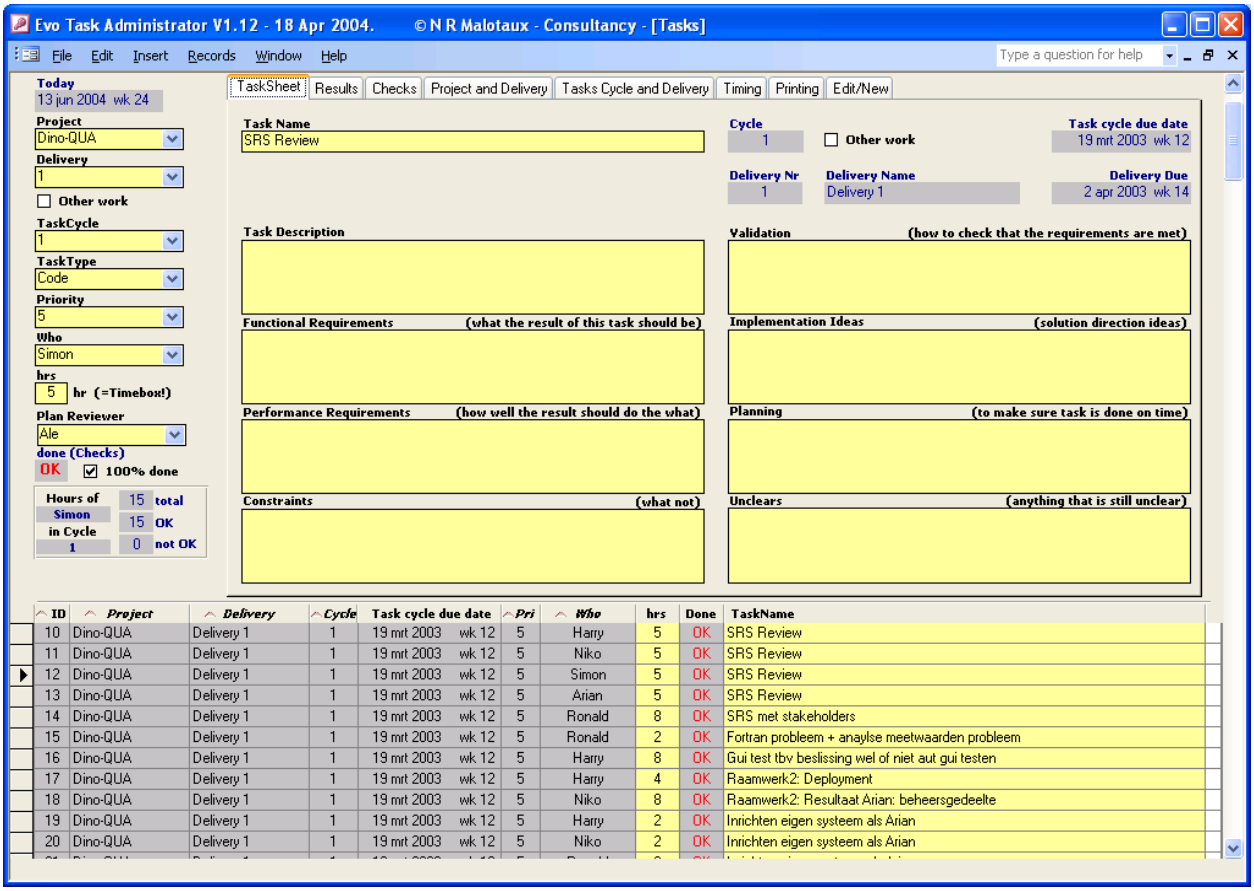


Figure 3: Evo Task Administrator tool screen shot

You might be concerned that the TaskSheet takes extra time. Using the TaskSheet doesn't cost time. It *saves* time. Try before you decide. If it ever proves to cost you time, find out why and act accordingly.
Note: If "completely accomplished" is defined as "first half of larger task finished", the TaskSheet should indicate how "first half finished" can be established. Don't settle for weak, un-measurable outcomes.
In the Evo Task Administrator (ETA) tool we have incorporated the TaskSheet for each Task, as shown in figure 3.

6.4 TimeBox

The number of effort hours planned for a Task is a TimeBox: this is the time available for finishing the Task *completely*, *no need to think about it any more*. If a Task proves to need more time than anticipated, don't just use more time:

- People tend to do more than necessary, so we may be able to do less without doing too little. The better the requirements of the Task are defined, the more focused you can go straight for the goal. That's why we use the TaskSheet.
- If you really cannot finish your task within the TimeBox, first complete the other Tasks. These were also chosen to have the highest priority: others may be waiting for their results.
- If you have time left after all other Tasks are done, you may still try to complete the Task.
- If the Task really cannot be finished, check:
  - What did you do
  - What did you not yet do
  - What do you still have to do

Then define new Tasks with estimations. These new Tasks may be considered in subsequent cycles.
If the immediate continuation of the Task really seems to be more important than anything else: use the InterruptProcedure (see below).

Never decide *alone* that you can use more time than the TimeBox. As soon as you find out that the Task is going to need more time than you have available, discuss with the Project Manager: We decided to do this Task, based on the expected outcome (Result) against the expected estimation (cost). If the Task turns out to cost much more than expected, will the investment still be worth it? We might not even have started the Task, so the moment you find out, reconsider the priority: *don't just go on*.

6.5 At the end of the Cycle we Check, Act and Plan:

1. **Was all planned work really done? If a Task was not completed, we have to learn:**
   - **Was the time spent but the work not done?**
     This is an effort estimation problem. Discuss what the causes may be and decide how to change your estimation habits.
   - **Was the time not spent?**
     This is a time management problem:
     - Too much distraction
     - Too much time spent on other (poorly-estimated) Tasks
     - Too much time spent on unplanned Tasks.

     Discuss what the causes may be and decide how to change your time management habits.
2. **Conclude unfinished Tasks after having dealt with the consequences:**
   - Feed the disappointment of "failure" into your intuition mechanism for next time. This is why commitment is so important: only with commitment we can feel disappointment. We must use the right psychology to feed our intuition properly.
   - Define new Tasks, with estimates, and put them on the Candidate Task List. They will surface in due time. If they do not surface immediately, we apparently stopped at the right time. This ensures that we first work on the most important things.
   - Declare the Task finished after having taken the consequences: remember that you cannot work on this Task any more, as it is impossible to do anything in the past.
3. **Now continue with planning the Tasks for the next cycle**

6.6 Analysis Tasks

If it will take significant time to define or estimate Tasks, we define an Analysis Task. In such a Task we don't *do* anything, we just analyze what we may have to do. At the end of the Analysis Task we check:

- What we know now
- What we still do not know
- What we still have to know

Then we define new Tasks or Analysis Tasks with estimations

Analysis Tasks get a deliberately small TimeBox. After, say, 2 hours we probably know a lot more than before starting. So after the short timebox we can much better define new Tasks or even new Analysis Tasks. By using a deliberately short timebox, we avoid spending more time than necessary. Analysis Tasks allow us to explore Requirements or to explore new techniques: we don't just start, we rather first analyze.

6.7 Interrupt

We know that requirements may change at any time, but we try to keep them stable during the TaskCycle. Sometimes, however, there are interruptions during the TaskCycle. For example: what do you do when the boss comes in and asks you to paint his fence? Or what do you do when a customer of your previous project reports a bug? In Evo, we don't immediately do such things because it's the boss or a customer. We also don't immediately reject the request, because it could be more important than anything else we are doing. However, because interrupts usually *seem* more important than they may be, we must never decide to change the plan and execute the interrupt on our own. *Always* consult the Project Manager.

If a new task suddenly appears in the middle of a TaskCycle (we call this an Interrupt) we follow this procedure, based on the principle “We shall work only on planned Tasks”:

1. Define the expected Result of the new Task properly
2. Estimate the time needed to perform the new Task, to the level of detail needed
3. Consult the Project Manager, or if unavailable, a colleague. You *must* seek a second opinion.
4. Check the Task planning
5. Decide which of the planned Tasks are going to be sacrificed (up to the number of hours needed for the new Task)
6. Weigh the priorities of the new Task against the Tasks to be sacrificed
7. Decide which is more important
8. If the new Task is more important: replan accordingly
9. If the new Task is *not* more important, then do *not* replan and do *not* work on the new Task. Of course the new Task may be added to the Candidate Task List
10. Now we are still working on planned Tasks

Small interrupts don’t need the InterruptProcedure, as long as they don’t jeopardize the completion of all the planned Tasks. Because our life is full of small Interrupts (drinking coffee, going to the bathroom, telephone calls, helping each other, and much more), we reserve the unplannable time for these unplannable Interrupts. The InterruptProcedure itself may be handled as a small Interrupt. If it needs more time, define an Interrupt Analysis Task first.

I know this may seem rather formal and bureaucratic. The only reason why we accept the bureaucratic rule in this case is because Interrupts are a big risk for the project and must be handled as such.

6.8 TimeLine

TimeLine is simply a line from Now to Then. We all can apply TimeLine quite well if we have to catch a plane: We know when the plane leaves and count back the time for checking in, the time to go to the airport, the time to get dressed and eat. This leads us to how we have to set the alarm clock the night before to make sure we will catch the plane. We also know that as soon as we can predict that we are going to miss the plane, we can abort the process even before going to the airport: we know we will be late, so it’s no use trying any more.

In projects it is not very different, other than that what happens between now and then is a bit more complicated and a bit less predictable.

We call this technique of making sure we will be on time “TimeLine”. It can be used on any scale: on a project, on deliveries, on tasks, the technique is always same:

1. Define a deadline or FatalDate. It is better to start with the end: planning beyond the available time/money budget is useless, so we can stop quicker if we find out that what we have to do takes way more time than we have.
2. Write down whatever you have to accomplish
3. List in order of priority
4. Write the same down in logical groups of Results
5. List these groups in order of priority
6. Translate the groups into Tasks: what you have to do
7. Estimate the Tasks in hours of effort (estimate less urgent tasks in less detail: they will be done later and hence will probably different from what you think now. Don’t waste time on irrelevant detail)
8. Cut the most urgent Tasks into work-Tasks of ~6 hrs effort or less
9. Review the order of the list
10. Ask the team to add forgotten tasks and add effort estimates
11. Get consensus on large variations of estimates (use a Delphi process)
12. Add up the number of effort hours
13. Divide by the number of available effort hours: This is the first estimate of the duration

What the customer wants, he cannot afford
The estimate of the duration is usually way beyond the required duration. At least we know now:

- What, at the FatalDate surely *will* be done
- What will *not* be done
- What may be done (estimation is not an exact science)

We also made sure that we plan to work on the most important issues first and the bells and whistles last.

Now you can discuss this with your customer. If what is surely done is not sufficient for success, you better stop now, to avoid wasting time and money and to spend it on more profitable activities.

In the beginning, customers can follow your reasoning, but still want it all. Remember that they don't even exactly know what they really want, so "wanting it all" usually is a fallacy, although you'd better not say that.
What you can say is: "OK, we have two options: In a conventional project, at the fatal day, I would come to you and tell that we didn't make it. In *this* project, however, we have another option. We already *know*, so I can tell you *now* that we will not be able to make it and then we can discuss what we are going to do about it. Which option shall we choose?"
If you explain it carefully, the customer will, eventually, choose the latter option. He will grumble a bit the first few weeks. Soon, however, he will forget the whole issue, because what you deliver is what you promise. This enforces trust. Remember that many customers ask *more*, because they *expect* to get less. He also will get confident: He is getting deliveries way before he ever expected it. And he will recognize soon that what he asked was not what he needed, so why bother to getting "it all".
The very first encounter with a new customer you cannot use this method, telling the customer that he will not get it all. You competitor will promise to deliver it all (which he won't, assuming that you are not less capable than your competitor), so you lose if you don't tell the same, just as you did yourself before using Evo. If, after you won the contract, you start working the Evo way, you will soon get the confidence of your customer and on the next project he will understand and only want to work with you.

6.9 Weekly 3-step procedure

Based on the experience gained, starting with the weekly team meetings we found in most projects, we arrived at a weekly 3-step process, which proves instrumental for the success of Evo implementation. In this process we minimize and optimize the time used for organizing the Evo planning.
The steps are:

1. **Individual preparation**
   In this step the individual team members do what they can do alone:
   - Conclude current tasks
   - Determine what they think the most important Tasks are for the next week
   - Estimate the time needed for these Tasks
   - Determine how much time they will have available for the project the coming week

   The Project Manager also prepares for his team what *he* thinks are the most important Tasks, what he thinks these Tasks may take (based on his own perception of the contents of each Task and the capabilities of the Individual) and how much time he needs from every person in the Team.
2. **1-to-1's: Modulation with and coaching by Project Management**
   In this step the individual team members meet individually (1-to-1) with Project Management (Project Manager and/or Architect). In this meeting we modulate on the results of the Individual preparations:
   - We check the status and coach where people did not yet succeed in their intentions
   - We compare what the Individual and the Project Management thought to be the most important Tasks. In case of differences, we discuss until we agree
   - We check the feasibility of getting all these Tasks done, based on the estimations
   - We iterate until we are satisfied with the set of Tasks for the next cycle, checking for real commitment. Now we have the work package for the coming cycle.

   We use an LCD projector on every meeting, even on the 1-to-1's. Preferably we use a computer connected directly to the Intranet, so that we are using the actual files. This is to ensure that we all are looking at and talking about the same things. If people scribble on their own paper, they all

scribble something different. The others don't see what you scribble and cannot correct you if you misunderstand something.

If there is no projector readily available for your project: buy one! The cost of these projectors nowadays should never be an obstacle: you will recover the cost in a very short time.

There is not just one scribe. People change place behind the computer depending on the subject or the document. If the Project Manager writes down the Task descriptions in the Task database (like the ETA tool), people watch more or less and easily accept what the Project Manager writes. As soon as people write down *their own* Task descriptions, you can see how they tune the words, really thinking of what the words mean. This enhances the commitment a lot. And the Project Manager can watch and discuss if what is typed is not the same as what's in his mind. And when we are connected to the Intranet, the Task database is immediately up to date and people can even immediately print their individual Task lists.

3. **Team meeting: Synchronization with the group**
   In this step, usually at the end of the day, after all the 1-to-1's are concluded, we meet with the whole team. In this meeting we do those things we really need all the people for:
   - While the Tasks are listed on the projection screen (as in figure 3), people read aloud their planned Tasks for the week. This leads to the synergy effect: People say: "If you are going to do that, we must discuss …", or "You can't do that, because …" Apparently we overlooked something. Now we can discuss what to do about it and change the plans accordingly. The gain is that we don't together *generate* the plans, we only have to *modulate*. This saves time.
   - If something came up at a 1-to1 which is important for the group to know, it can be discussed now. In conventional team meetings we regularly see that we discuss a lot over the first subject that pops up, leaving no time for the real important subject that happened to be mentioned later. In the Evo team meetings we select which subject is most important to discuss together.
   - To learn and to socialize.

At every step of the process we try to minimize the number of people involved. First we added the 1-to-1's to the process. The aim was to relieve the team meeting from individual status reporting and from too detailed 1-to-1 discussions. We found, however, that these 1-to-1's easily took about one hour each. One Project Manager said: "Niels, with 6 people in my team, I can just manage in one day. But what would you do if there are 15 people in the team? I want these meetings to take not more than 30 minutes". Watching closely what was happening in the 1-to-1's, we saw that there was a lot of thinking and waiting: "What are you going to do the next cycle?" Pause for thinking. "What effort do you estimate for this Task?" Pause for thinking. "How much time do you have for the project this week?" "I don't know. I have to discuss with the Project Manager of the other project". Sigh. Why didn't you check *before* the meeting? Now we cannot decide!
This led to the Individual Preparation step, where people prepare these issues before the meeting. The result was that the 1-to-1's went from one hour to 20 minutes. That was much better than we expected. The reason is probably that now people come to the meeting much more prepared, needing even less time to get to the point.
Now having optimized the 1-to-1's, Project Managers invariably say that these 1-to-1's are one of the most powerful elements of the Evo approach.
Team meetings usually take not more than 20 minutes. Do we discuss less than before? No, we just discuss the right things effectively and efficiently.

7 Conclusion: How Quality is Assured by Evo

Deming said that quality cannot be tested into a product; it has to be designed in from the beginning. Aren't we doing just that? In Evo projects we define what Quality is and then we pursue the defined Quality, constantly optimizing based on what we learn along the way. All the Quality Assurance people need to do is guide and coach us, watching over our shoulder to ensure we stay disciplined. Not because we like discipline, but because we like success.

Demystifying Microsoft's Quality Assurance Process

How Microsoft Incorporates Testing into the Product Cycle

Ambrosio Blanco
Microsoft Corporation

About the Author:

Ambrosio Blanco joined Microsoft's Languages Division in 1992 as an API tester on Microsoft COBOL. Since then he has worked on Microsoft Fortran, Integrated Mathematical and Statistical Libraries (IMSL), Office 97 & 2000, Internet Explorer 4.0, Netdocs (later called InfoPath), and Customer Relationship Management (CRM), as both a Test Lead and Test Manager. He is currently a Test Architect in the Knowledge Interchange incubation team in the Information Worker division.

Prior to joining Microsoft Mr. Blanco was a programming consultant in New York for two years, working with clients such as Unicef and McGraw Hill. He received his degree in Computer Science from the University of the Philippines in 1990.

Abstract:

This paper will describe how Microsoft meets the challenges of maintaining high code quality while balancing the business need to ship quickly and ship often. This paper will include an overview of a typical multi-milestone product cycle, with emphasis on the role testing plays throughout the process. I will then discuss in-depth 'standard' QA processes such as daily builds, rolling builds, check-in verification tests, full test passes, and code coverage. The trade-offs involved in defining each of these practices (e.g. what level of coverage is good enough?) will also be described.

The primary goal of the talk is to show test team managers how Microsoft distributes its QA resources and which QA practices are widely adopted, vs. those which – although theoretically useful -- have not been adopted due to practical constraints such as maintenance overhead, team expertise, and even 'political' opposition.

Legal Disclaimer:
The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2004 Microsoft Corporation. All rights reserved.

Introduction

As one of the largest companies in the software industry Microsoft and its products routinely undergo intense scrutiny. Consequently negative quality issues garner significant media attention and such visibility has contributed to a general attitude of equating Microsoft products with poor quality, and therefore poor Quality Assurance practices.

This paper aims to demystify the Quality Assurance process within Microsoft, by describing the nuts and bolts of how a typical testing organization is run, and how the difficult decision to ship or slip is made. Like any company there are minor variances in practices between teams, but the overall strategy remains consistent.

This will be a straightforward presentation of current practices, not an evangelization for more widespread adoption. Comparative analyses of this model with other Testing Models are therefore outside the scope of this paper.

The Product Unit

Microsoft has 3 distinct organizations in each product team: Program Management (PM), Development, and QA. Each is a separate organization, and is dedicated to one role throughout the product cycle. At the top of each is a Group Program Manager (GPM), Dev Manager, or Test Manager respectively. They report to a Product Unit Manager (PUM) or General Manager (GM). Therefore the most senior tester in a Product Unit is usually the Test Manager.

Members are not transferred between Dev, PM, and QA to load balance tasks, or to catch up with the schedule. Each discipline is specialized enough that shifting between them is difficult, and occur mostly at lower levels of responsibility. Most employees stay with one discipline for most of their career.

QA is a *peer* with Dev or PM. QA has a strong voice in all key processes such as scheduling and feature planning. This forms a system of checks and balances that prevent dominance of a product group by any one discipline.

In addition to the separation between disciplines a Product Unit is also divided into multiple ‘virtual’ Feature Teams. Feature Teams are comprised of at least one Developer, one tester, and one PM who are responsible for building a feature area. They are considered virtual because they do not all report to the same manager. Testers could belong to more than one Feature Team and frequently do, especially on complex products with broad Feature sets.

Unlike Dev and PM, the Testing discipline has two distinct sub-disciplines: the Software Test Engineer (STE), and the Software Development Engineer commonly referred to as “SDE/T” or “SDE in Test” to distinguish them from regular Devs or SDEs. SDE/Ts have the additional responsibility of developing and deploying test automation tools, and white box practices such as static code analysis, code coverage, performance profiling, etc. As such, the requirements for hiring an SDE/T is a superset of that for SDEs.

Ideally the ratio of Testers to Developers should be at or close to 1:1. Traditionally there has been no set goal for the STE to SDE/T ratio, as this can vary greatly based on the product (e.g. a Game vs Compiler).

The Product Cycle Model

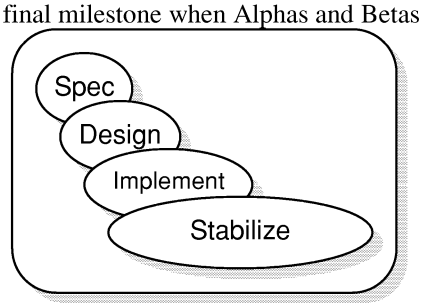
Phases

The Lifecycle Model used by most groups combines aspects of the Staged Delivery and Sashimi approaches.

At the beginning of a new product just enough Requirements Analysis and Design is performed to indicate how many Stages (internally referred to as Milestones) will be in the ship cycle, and what set of functionality will be in each. This initial phase is typically called Milestone 0, or “M0,” and is driven primarily by Program Management.

Dev and Test's role at this point is to provide input on cost of implementation and testing, and while it is possible for either discipline to propose new features, they generally defer final judgment to PM.

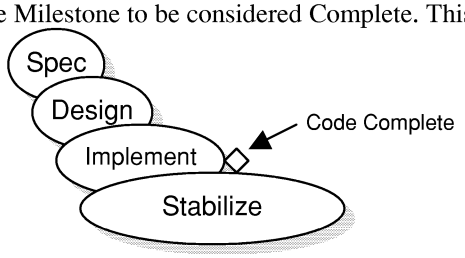
As in classic Staged Delivery at the end of each Milestone the product is installable and fairly stable, but will only have the functionality planned for that Milestone, which at the early stages could be very limited. Where this approach differs from classic Staged Delivery is that the product is not delivered to customers, except during the final milestone when Alphas and Betas are conducted.



Typical Milestone

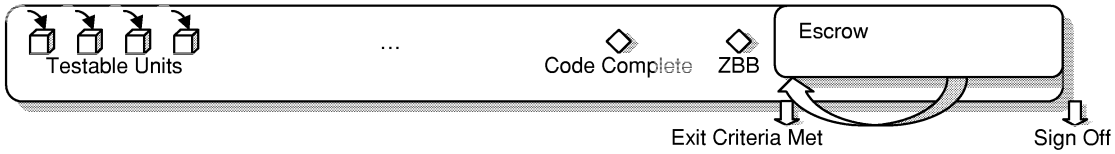
Each Milestone adopts aspects of the Sashimi[1] model, where the Designing, Coding, and Testing phases overlap.

During the spec and design phases QA is fully involved e.g. by providing estimates for feature testing, and giving feedback on proposed design approaches from the testability and usability perspective. Additionally QA is the primary driver of the Milestone Exit Criteria, which is a document listing the conditions which must be satisfied for the Milestone to be considered Complete. This is defined early in the Milestone, with the approval of PM and Dev.



Once all features are checked-in by Dev to the source tree then the Milestone is considered "Code Complete." This is simply a transition point that indicates Devs have shifted from working on new features to working on bug-fixes; it is in no way used as an indicator of quality.

As you can see the Stabilization phase begins long before Code Complete. This is possible because features are implemented in increments – sometimes called 'Testable Units (TUs)' – which can undergo Testing as soon as it is delivered.



Stabilization Phase

After Code Complete and during the Stabilization phase, the next goal is to reach ZBB, or "Zero Bug Bounce," which is the state of having zero active bugs in the bug database. This is usually a temporary state (hence the "bounce") but is nevertheless an important milestone since it indicates that the team is nearing Release. Once ZBB is achieved, then the bar for bug-fixes is raised significantly in order to reduce code churn: only fixes to showstopper issues are allowed to be checked-in.

---

[1] The Sashimi model originated from Fuji-Xerox's experiences with the Waterfall model, and refers to a way of presenting overlapping slices of raw fish. Unlike the classic waterfall, phases can greatly overlap, and is particularly useful in cases where feedback from a subsequent phase is required for an earlier phase. See "Rapid Development" for more information.

After ZBB and after all other Exit Criteria are satisfied then the product goes into Escrow (sometimes referred to as "Baking"). This is the team's final window of opportunity to find any show-stoppers before the product is shipped, so QA generally goes into overdrive to find as many bugs as they can. There is no hard and fast rule around the length of Escrow; depending on the complexity of the product it could be anywhere from a couple of weeks to several months.

If a showstopper issue is found at this point then the Escrow timer is on hold until the issue is resolved. If the resolution is a code fix then the Escrow 'counter' is reset, to account for destabilizations introduced by the code churn. It is possible for a team to reset the timer several times, though if this happens more than twice then this generally indicates a deeper issue.

The 'End Game'

As previously mentioned after ZBB the *Triage* bar (see the next section) is raised to its highest level: true showstoppers are fixed; fit-and-finish bugs are automatically disqualified, and everything else in-between must be approved by a group called the 'War Team.' Bug-fixes also undergo more rigorous inspection e.g. through code reviews, buddy-builds, private testing, and Focused Test Passes (although some teams may do this throughout the cycle). Triage Teams enter real-time mode to reduce turn-around time on new bugs. Additionally, War Team meets at least daily, but could also go into real-time mode. As automation runs wind down, Testers spend increasingly more time performing ad-hoc tests to fill-in missing areas.

During Escrow the Milestone Exit Criteria are applied. If all criteria are met and it is the final Milestone of the release then the Release Manager or Test Manager initiates the Sign Off process by petitioning all area owners and Testers to give a final Go/No Go call, to allow for any last minute comments

The Sign Off is immediately followed by a Release to Manufacturing or RTM.

Release Criteria & Sign Off

The Sign Off is the culmination of all QA efforts since it is the final roll-up of all test results. Compliance with all Release Criteria triggers Sign Off, so having comprehensive and well-defined Release Criteria can have a direct impact on the quality of the Release. Some examples of typical Release Criteria are:

- Meet pass rate requirements, broken down by functional area
- Meet performance requirements
- Must pass Escrow, e.g. 'no showstoppers found during 3 weeks of active testing'

All Release Criteria are crafted to be measurable and objective; however the human element is not discounted. The additional Sign Off step provides individual testers a final opportunity to evaluate their objective and subjective impressions.

Servicing

Post RTM support is provided by Product Support Services (PSS) and Sustained Engineering (SE). PSS is the initial line of support that customers interact with, and is distributed to call centers around the globe. SE on the other hand is considered part of each Product Group, and is therefore physically located in close proximity to the Product Group. PSS collates and triages feedback for escalation then passes it to SE, who then translates it into Collaboration Requests (CRs) to the core team. If further investigation concludes a Hotfix is necessary then SE will engineer and test the Hotfix *in collaboration* with the core team; i.e. SE does not solely own the work. As a result it is common for a Product Group to be so inundated with CRs and Hotfixes that they may for a time make very little progress on the next version.

Why this Product Cycle Model?

The phases and processes described in this section were devised in the early days of the company as a way of balancing a) shipping on time, b) quality, and c) simplicity (not necessarily in that order). At this point the PCM has

tremendous historical inertia, with a long track record of successfully shipping products (albeit the quality of these products may vary).

Conversion to other development models is invariably resisted because they *require* additional processes such as (in the case of the classic Waterfall there must be a formal validation at the end of every phase), whereas the current model is flexible enough to allow additional process, but not impose it. Moreover, under the current model each of the three disciplines have the freedom of tailoring the PCM within each of their organizations. For example, Extreme Programming practices may be favored by many Dev organizations, but not as much by PM or Test.

The trade-off however is that this flexibility results in varying degrees of enforcement of best practices, resulting in varying quality between teams. This variability is reflected in the fact that the current Defects/KLOC between teams could be anywhere from 12 to 25.

There is no effort underway to switch the company to a different model anytime soon. The current culture would make this a difficult proposition. Even if a better one exists on paper, until a Product Group implements it through an entire Product Cycle and successfully ships, such a change will not be taken seriously by other groups. As with any proposal to change, the need must be perceived to be great enough to outweigh the risk, which is currently not the case.

When are we done testing?

Teams generally include the following metrics as primary indicators towards test completion (these may also be included in the Exit Criteria):

- **Bug count**. Typically must be at zero at time of RTM. Does not mean that all bugs are fixed, just Resolved. A resolution could be a fix, but it could also be a decision to punt the bug to the next version, or not to fix it at all (e.g. if the relevant use case is deemed too unlikely).
- **Bug slope**. The descent path of the bug count near the end of the cycle must not exhibit wide variances due to high incoming bug rates.
- **Bug types**. There must be few High Priority bugs found near the end of the cycle.
- **Test Case Count**. The count of automated and manual cases listed in the Test Case database must be reasonably high (subjective).
- **Code Coverage**. If automation is a big part of the test strategy then the Code Coverage must be high.
- **Customer Feedback**. Feedback from Alpha and Beta programs must indicate a certain level of satisfaction with the product's quality.
- **Personal Impressions**. From testers in the team, through the Sign Off process.

These are only some of the metrics that a Test Manager may use to determine the final judgment call at the Sign Off.

QA Practices

During the previous section several processes were mentioned that will now be described in detail here, as well as a couple of additional ones that are considered common practice.

Triage

Triage is a term taken from the medical field, and refers to the process of prioritizing incoming issues. This ensures that high-impact issues are fixed first, therefore the resulting churn – which is generally greater – can be dealt with earlier.

Triage is essentially a recurring meeting conducted by each Feature Team to inspect newly-reported bugs; there is usually one Tester, PM, and Dev present. In addition to setting bug priorities Triage may also Postpone a bug, or choose the nature of the fix, e.g. by documenting the issue instead of changing code. Frequently Devs will spin a fix as just a 'one-line change,' the implication being that the risk is very low. It is the Tester's responsibility to always challenge these claims, especially toward the end of the release when the cost of destabilizing the code is high.

War Team

If Triage cannot reach a consensus then the issue is escalated to a second Triage committee, usually called the "War Team." War Team is comprised of senior technical leads and managers, and will make the final call on these issues. Once a product reaches ZBB War Team may take over all Triage responsibilities, to ensure enforcement of the higher check-in bar.

A team may also establish a "Dev War Team" made up of senior Developers to review the design and implementation of fixes approved by the War Team. As such, they can only approve or disapprove a particular implementation of a change, but not the change itself.

Builds

In order to ensure consistency product code is not compiled and built by individual testers. Instead there is an automated process that performs a snapshot of the source code tree regularly (daily for most groups), and builds the product from that snapshot. Each build is then stamped with a unique version, so bugs can be reported against specific builds.

Taking a snapshot of the source tree may be a lengthy process, so it is important to restrict check-ins in the meantime to guarantee that the code is at 'steady state.' One option is to take the snapshot in the evening; another is to establish a 'check-in window' outside of which check-ins are disallowed.

The daily builds provide QA with frequent updates and allow for reproducibility and consistency of test results. It also catches build breaks early, so QA avoids wasting time on a bad build. Some groups enhance this by building even more frequently, e.g. on each check-in or batch of check-ins, though such builds are purely for finding build breaks since they are not uniquely versioned and therefore cannot be used in formal Testing.

The disadvantage of having a frequent build is that it may result in an increasingly narrow check-in window, resulting in a backlog of pending check-ins. It also encourages the habit of 'batching' check-ins, which increases likelihood that it will contain a breaking change. Finally a complex and frequent build process requires a dedicated staff; in some small teams the headcount for this may come out of the Testing organization, further straining test efforts. However the benefits to having a formal build are so compelling almost every group has one, although with varying amounts of recurrence (usually directly proportional to the size of the group).

Once the product reaches ZBB the regular builds may be replaced by 'on-demand' builds since there are fewer check-ins, and since any check-ins during this phase need to be incorporated as quickly as possible to avoid continued testing time on the now 'older' build.

All builds that will be sent externally (e.g. as part of an Alpha or Beta program, or as the RTM 'Golden build') are called Release Candidate or RC builds. All builds done during escrow are RC builds.

Test Passes

Test Passes are usually planned well in advance because of their cost. At a minimum a group will do one Full Test Pass (where all tests are included) at the end of each Milestone. Most groups also opt to do small Test Passes (Basic Verification Tests or BVTs) on the daily builds to validate prerequisite basic functionality e.g. Setup, Application launch, and UI/API visibility. If a build fails BVTs then large areas that may not be testable; consequently a High Priority or "drop everything" bug is reported to Dev for a quick turn-around.

Once Code Complete is reached then a Test Pass is usually performed – sometimes referred to as an "Acceptance Pass". The Acceptance Pass will include only Happy Path tests (i.e. no testing for error conditions) for Pri 1 features. Once the Test Team completes implementing all Test Cases then a Full Test Pass is performed. Since this is very costly it does not begin until all major bug-fixes have been checked-in. Once it is complete it is not repeated unless more major changes are checked-in; and even then a group may decide to risk a smaller Test Pass in order to prevent slipping the Milestone.

Once again the disadvantages to Test passes become clear only if they are overused – frequent Test Passes can rapidly drain your Test Team's morale, since much of the work involved is repetitive. Additionally, some teams freeze builds during a Test Pass, so if this happens too frequently pending changes are backlogged for longer periods.

Check-in Tests and Unit Tests

Unit Tests are a fairly common practice at this point, although enforcement of this practice varies widely.

Check-in tests can be the same as the Unit Tests, or just a subset. Check-in tests must pass for every check-in, even if it is seemingly unrelated to what is being modified. This is to ensure that new code does not accidentally break other core features. Larger teams enforce check-in tests using a regular build process, while smaller teams use the Honor System. Check-in tests are not as widely adopted as unit tests.

The downside: Unit Testing UI-based components can be non-trivial; the dev team may need to construct a set of helper libraries to manipulate UI.

Code Coverage Analysis

Code Coverage is a given in large groups, but spottily used in smaller ones. Teams that do deploy this also use the metric as part of their Release Criteria.

If team has a prior record of achieving a certain block coverage in past versions of the product then those goals are generally carried forward. If it is a new codebase then a goal is estimated, then adjusted up or down based on the test effort to reach it. Once we've hit the point of diminishing returns then it is generally agreed that the coverage is 'good enough.' Components that have a high degree of complexity or interdependence may have higher goals. Typical code coverage goals are around 70-80% block coverage.

Advantage: Easy to measure, can be a reasonable metric for pointing out large areas for concern. Disadvantage: requires effort and code-reading skills to drill-down the metrics into actionable steps. Also: can be difficult to define coverage goals; is generally eschewed (or shallowly used) if a team does not already have other white box efforts.

Static Code Analysis

Static Code Analysis tools like Lint perform static inspections of code (i.e. no execution). They are easy to deploy, and can therefore be leveraged as a check-in test. However the results need code-reading skills to make sense, the signal to noise ratio on results can be low, and only a certain class of code defects can be discovered (e.g. limited in effectiveness for defects arising from complex component dependencies). Consequently deployment of these tools has been slow, but is growing faster as the technical skills of Test teams increase.

Other QA Practices

Following are some QA practices that has begun to be adopted, but are not yet standard practices.

Model Based Testing

MBT has been around for a while, and several tools have been built to implement it. It has not yet received widespread adoption due to a) steep skill set requirements from testers, b) cost of deployment, and c) general skepticism since it is an unfamiliar and radical departure from classic test methods.

Pairwise Testing

Pairwise Testing has also begun to gain ground in some teams. Primary reason for the slow deployment is insufficient visibility (many teams don't even know the technique exists), and a lack of integration with existing test frameworks. Both of these issues are now being addressed however, so I expect this to be a standard practice within the next few years.

Fault Injection

Fault injection can be viewed as hijacking an API call to induce a faulty condition in order to test the code's error handling, e.g. replace calls to malloc so it always returns NULL. Although the tools have existed for several years some teams resist it because a) it requires knowledge of the product code, b) the automation framework is costlier to deploy since it does not fit into a traditional test harness, and c) Developers can argue bugs found with this method are corner case issues that could never happen. In the malloc case for example a Developer can claim that in a memory-starved environment other parts of the Application will fail long before the bug is hit.

Root Cause Analysis

Root Cause Analysis can require a significant amount of effort to investigate the circumstances around a defect, especially if it was found months or even years before. It may also require the support of Senior Managers since recommendations stemming from RCA can span all three disciplines. It is therefore perceived as a heavyweight process that only large teams can afford to do regularly. Smaller teams may still opt to perform a one-off RCA against particularly onerous issues, e.g. ones that cause a Product Recall.

Common Mistakes

Following are some possible causes for a sever bug to 'slip through the cracks:'

Incorrect Triage Call

A bug may be resolved by Triage as 'Won't Fix' because the consensus is that the user scenario is 'unlikely.' This may be a bad call if interaction with other features is not considered, if a use case is missing from the spec, or if an incorrect assumption is made regarding how the corresponding feature is implemented.

Also awareness that Sustained Engineering can produce Hotfixes may influence Triage. It should not: if it was (or wasn't) worth fixing before then it still is (or isn't). Unfortunately some triage decisions in the past were made on this exact basis.

It is frequently the case that a critical issue will be reported in the Press, a team will research it, find that it is a known issue, but it was not fixed because (or inadequately fixed) due to a bad call in Triage.

Inadequate Testing of Late Changes

If a showstopper bug is found late in the cycle then pressure is high to get it resolved quickly in order to meet the target ship date, so QA may decide to forego repeating a Full Test Pass to save time. If all dependencies of the modified code is not identified then the abbreviated pass may miss a bug that was introduced by the change.

Incomplete Environment Testing

Most testers use identical or very similar hardware in their offices and in labs. This vanilla hardware is further standardized through constant re-imaging. This means that there could be insufficient coverage on systems with different hardware components, or with different third-party applications installed.

Current Challenges

Recruiting and Retention

Recruiting for SDE/T positions is difficult, as there is a perception among many that all QA positions involve manual testing and very little coding. In addition there has been some attrition from QA to Development.

Changing Roles and Skill Sets

As groups mature their test methodology there is an increasing need for technical skills in larger portions of the Testing Team. Besides the training challenge, automation may go through a period of instability as testers learn how to code.

Limited standardization

Though some tools are standardized company-wide – e.g. code coverage, static analysis, and defect reporting tools – there is still a large amount of homegrown tools covering basic areas like test case management, test case deployment, UI Automation, and results reporting. If you move to a different group then chances are you will have to learn a different set of automation tools. This tendency to reinvent the wheel is due to a perception that maintenance will be cheaper if it originated in-house; sometimes this results in the creation of dedicated (but frequently short-lived) tools teams.

Slow pace of innovation

Innovation in the context of a product group is a difficult proposition due to the immediate demands associated with shipping a product on time. While a QA team may have enough downtime between ship cycles to make some improvements it is usually not enough for major efforts, especially when a substantial portion of existing tools need to be rewritten to accommodate for changes in the next version of the product. Oftentimes a group would go through several product cycles before its basic automation infrastructure and process are mature and self-sustaining enough to the point where sufficient bandwidth is now available to move on to true innovations. In the worst case the automation infrastructure itself drains any spare bandwidth as it grows obsolete and demands more maintenance.

This challenge has led to a culture of conservatism; in some groups major QA innovation is derisively referred to as 'science projects,' the implication being that resources can be better spent on achieving 'real results.' Hence those precious few Testers who are passionate about innovation develop solutions and prototype tools on their own time over several years, and potentially with little or no recognition. Some attempts have been successful, but they can only go so far without additional support.

The situation is further exacerbated by the attrition of the few 'veteran' testers that are capable of such innovations, though ironically such attrition is frequently caused by this slow pace.

Lastly some of the most compelling QA techniques available are difficult to deploy on Product code, either due to an incompatibility with the code (e.g. the technique may be language-specific), or of the tools. Though not insoluble these issues raise the adoption cost, thereby lowering the likelihood of deployment.

Increasing Product Complexity

In the past Microsoft's stable of products have primarily been stand-alone applications that can be tested on a single machine. Now however we have vastly increased the number of server products in development, and many traditionally stand-alone applications have evolved server-based features such as online Help, dynamic component installation, and downloadable plug-ins. Chances are if you are a Microsoft Tester, you also own testing features that require a network and a server.

The resulting client-server environment can be difficult to model because the 'state' of the product is distributed to different machines. Hardware costs also force many teams to share servers between testers, which increases the likelihood of cross-pollution between tests, thereby making bugs less reproducible and difficult to debug.

Also, the increasing use of multithreading results in more issues that are difficult to reproduce due to timing-sensitive code.

Furthermore, some products may use bleeding-edge platforms that are in parallel development and therefore could still be in flux and is usually poorly documented. If it is a fundamental change such as a new language then this could break existing test tools. For example when many teams migrated from C++ to C# our internal static code analyzer and code coverage tools were not useable for months.

Lastly, the need to be backward compatible with previous versions means old product features continue to exist in future versions, potentially long after they become obsolete. This along with the skyrocketing number of SKUs greatly increases the test matrix.

QA Initiatives

To increase sensitivity towards Quality issues we have begun two major company-wide initiatives that are enforced though team 'score cards' and individual employee performance reviews that hold everyone in groups and divisions accountable for the quality of their product.

Engineering Excellence (EE)

The Engineering Excellence is an initiative to formalize software production by treating it like a classic engineering discipline. This impacts all phases of software development through the establishment of common best practices, and the adoption of standard metrics such as code coverage for inspection and oversight. EE's main goal is to structure and standardize most aspects of the software development process in order to minimize variance in how different teams ship products.

EE has been in place for a little over a year, but progress has been slower than expected. The broadness of the messages coming out of EE has contributed to some confusion; however the problems that EE is attempting to solve are complicated enough that this will probably be a long-term effort.

Trustworthy Computing (TwC)

Trustworthy Computing is a more focused initiative that aims to address the growing concern around the security vulnerabilities of Microsoft products. As such TwC practices are more specific and stringent than most EE practices. Furthermore there is a dedicated team (the Secure Windows Initiative team, or SWI) that was established to serve as a training and support resource for all teams engaged in deploying TwC. Some examples of TwC practices include group-wide training of Secure Coding practices; the deployment of security inspection tools; a team-wide Security push including a lengthy threat modeling exercise; and an audit by the SWI team that must be passed before a product can ship.

Final Comments

It is important to note that the Quality Assurance processes within Microsoft undergo a continuous state of evolution. However as product complexity increases, it has been challenging for Test Organizations simply to keep up, let alone make drastic improvements. The larger groups such as Windows and Office have achieved significant improvements over time, but smaller teams may not have yet reached a similar level of sophistication.

References

1. McConnell, S. (1996) Rapid Development. Microsoft Press.

Bibliography

1. McConnell, S. (1993) Code Complete. Microsoft Press.
2. Maguire, S. (1993) Writing Solid Code. Microsoft Press.

Dynamic Design Analysis:

Testing Without Code

Scott A. Whitmire

> *What distinguishes the engineer from the technician is largely the ability to formulate and carry out the detailed calculations for forces and deflections, concentrations and flows, voltages and current, that are required to test a proposed design on paper with regard to failure criteria.*
> - Henry Petroski, *Invention by Design*, p. 89 (Petroski, 1996)

The early history of using iron to build railroad bridges is filled with bridges collapsing without warning on a regular basis, sometimes with spectacular and catastrophic results. This was due largely to the poorly understood relationship between metal fatigue and the dynamic forces of a heavy, constantly moving load. By examining failed bridges and experimentation, we came, over time, to understand metal fatigue and predict the nature of these dynamic forces. Today, a competent engineer can predict the dynamic behavior of a bridge design before the bridge is ever built. This provides the first important lesson for those of us who build software: We learn more from our failures than from our successes.

The spectacular collapse of the partially completed Quebec Bridge, still the longest cantilever span in the world, highlighted another set of issues. The ultimate cause of the bridge's collapse was the failure on the part of its engineers to recalculate the dead weight of the bridge after its span was lengthened by some 200 feet and how this affected the design while under construction. The first issue is that designs don't always scale well; what worked before will not always work when enlarged, expanded, or otherwise extended. The second is that a design has a unique set of behaviors while it is still under construction. A successful design must account for these two sets of static and dynamic forces.

The design and construction of bridges and other major structures offer strong parallels with the design and construction of software. While the forces to which our designs are subjected are different, they can and do have the same results if we don't account for them in our designs. Most software developers don't know if their product will work until after they've built and run it the first time. Even then, we don't know whether we've solved the right problem until our customers get their hands on it. Very few of us design our software; almost nobody analyzes their designs. Inspecting a design is not the same thing as analyzing it. Looking at a design, most developers couldn't say, with any degree of confidence, how the software will respond to a given external event, or how it will hold up under stress, or how it will handle modifications over time.

The whole purpose of design is to obviate failure (Petroski, 1985). We create and then analyze a design in order to determine how it can fail. When we find a way, we design around it. We repeat this process until we can think of no more ways for it to fail. This doesn't mean, however, that we have found all of the possible ways for a design to fail. This leads to the notion of design as hypothesis (Petroski, 1985), but that's a topic for another paper.

There is a perception among many, our users and managers in particular, that software is easy—easy to build and easy to modify. We know from bitter experience that this viewpoint simply isn't true, yet we continue to behave as if we believe it ourselves. The truth is, software is among the most complex things ever built by humans. The rate at which software projects fail, for whatever reason, is evidence that we don't fully understand just what we're up against. Normally, when we do something we know to be difficult, we take extra care to ensure that our product will turn out as we intend. Yet, we more often than not fail do this when building software.

Copyright © 2004 by Scott A. Whitmire. All rights reserved.

As we build larger and more complex software systems, as we build systems of systems, and as we use techniques such as object-oriented development and frameworks such as .Net, the complexity moves out of individual source code modules and into the design. It is more important than ever that we understand how our designs are going to behave, both while under construction and when complete, before we build them.

Whether the design process is formal or informal, defined or undefined, we analyze and select basic architectures and from among alternative designs while building software. We might do this deliberately and explicitly, or not, but we always do it. There is a significant time lag between the design coming together in our heads and being able to test that design with working code. We need to verify and validate our designs before we spend the time and effort crafting code that might not work, and might not be able to be made to work.

When modifying a design or an existing system, we need to understand how the modification will affect the behavior of the system as a whole. Are we fixing the problem as we intend, or are we creating a new, unanticipated problem? We may need a way to analyze a potential change to a design in terms of its effects on the design as a whole. We need to do this before the change is made in order to avoid having to undo expensive modifications to code.

Understanding how a design can fail requires first understanding how a design behaves under various loads and stresses. In structural and mechanical engineering, to analyze a design means to compute the static and dynamic forces that transmit through each detail part in order to tell whether that part will withstand those forces. The point is to find those parts likely to fail under the expected load in order to design them to withstand that load. What complicates matters is that a change in one detail part changes the loads on other parts in the design, and those parts need to be revisited. Sometimes we find that the whole design is inadequate.

In electrical engineering, design analysis tests for radio frequency interference between components, voltage and current potentials, physical size, and even the amount of energy released as heat so these issues can be dealt with in the design. Electrical engineers tend to specialize in understanding these very different kinds of loads and stresses on an electronic circuit. Circuit designers tend to focus on the first two issues, while packaging designers tend to focus on the second two.

The premise in all of these cases is to find out before a design is built whether it is likely to fail. Hard and bitter experience has taught civil, mechanical, and electrical engineers that it is easier and cheaper to find and fix any problems before spending lots of time and money to build the design. We in software have the experience necessary to reach the same conclusions. We have lots of data that should prove beyond a reasonable doubt that the earlier in the process we catch an error, the cheaper it is to fix.

In software, analyzing a design translates into understanding how a design will respond to various events given certain sets of initial conditions.

In summary, three great lessons that software engineers can learn from other engineering disciplines are:

1. We learn more from our failures than from our successes;
2. It is easier and cheaper to revise a design before it is built; and
3. The earlier in the process we catch an error, the cheaper it is to fix.

Defect analysis, one of the most effective techniques available to any software organization, is the best way to apply the first lesson. This paper concentrates on the second lesson, providing a technique to analyze the static properties and dynamic behavior of a software design, or part of one, before time and money are spent writing code.

Overview of Dynamic Design Analysis

Dynamic design analysis is a technique for seeding a design in a steady state with an external event and propagating messages until a new steady state is reached. A steady state is one in which the design is essentially at rest: no objects change state without external influence. This doesn't mean that no messages are passed, as some systems require near constant message passing to detect external events, such as a system that polls a series of sensors looking for changes in value.

A software design is "stressed" when it is asked to respond to an event that it did not generate. In this sense, the load which a software system must bear deals with both the types and quantity of events to which it must respond, including gracefully refusing to do anything. Here, quantity has to with the frequency with which events occur. A design's performance under this kind of load can be tested only when we do our volume tests with the working code. We can, however, estimate how our design might perform under load using computations such as the "Big-O" technique (see Whitmire, 1997, for a description and further references) and other algorithm analysis methods.

We can more easily analyze a design's response to one instance of a type of event and this is the emphasis of Dynamic Design Analysis. In order to analyze a design, we first need to know a few things about it, or at least about the part of the design we wish to analyze. We need to know the static properties of the design—classes, objects, their attributes, and the relationships between them—and the relevant methods in terms of their parameters, preconditions, side effects, and postconditions. In addition, we must define a starting state for the design. This involves naming the set of objects assumed to be in scope and defining a current state for each of the objects relevant to our analysis.

A message is passed when a method of one object invokes a method of another (or the same) object. To keep things simple, we define an event as a message to an object that originates outside the system. To send a message, the sender needs to know the class and identity of the receiver. For a message to be passed successfully, three conditions must be met:

1. The receiving object must be in scope (or the target function must be visible);
2. The receiving object must have a method that matches the syntax and content of the message (its signature); and
3. The method's preconditions must be met.

When all three conditions are met, we can assume the target method's postconditions, including the sending of any messages and the updates or modifications to any persistent data specified for the method (these are both side effects of the method's execution). Because a method often (usually) sends more than one message as part of its execution, the number of message paths we must follow can explode rapidly. In order to catch problems as close to the source as possible, it is best to follow these paths in a breadth-first order.

The technique is adapted from Object-Z (Rose, 1992) and OPUS (Mens et. al., 1994) and is based on the formal model of objects and software designs I first presented to the Object-Oriented Systems, Languages, and Applications (OOPSLA) conference in 1996 and later published in Object-Oriented Design Measurement (Whitmire, 1997). The model is built on category theory from mathematics and defines operations and relationships using set theory and relational algebra. Object-Z adds object-oriented extensions to the formal specification language Z (pronounced "Zed") and forms the basis for the notation used to describe states and define methods. OPUS provides a lambda-calculus to define and describe message passing and binding, and side effects.

The technique defines ways to specify a state for an object as seed states for the entire design, and to define the parameters, preconditions, and postconditions for a method. Inference or substitution rules make analysis easier by defining ways to manipulate strings of symbols based on their syntactic shape, without having to assign them meaning, forming an algebra of software behavior.

To ease communication, dynamic design analysis adopts a specific and formal notation. However, notation isn't important when you are analyzing your own designs. When more than one person is involved, however, the notations needs to be agreed upon in order to avoid misunderstandings. Despite its technical origins, the technique is deceptively easy to use.

Analyzing any non-trivial software design is a huge and time-consuming task. Any analysis technique, including Dynamic Domain Analysis, is ill-suited for analyzing an entire system at one sitting. When an engineer analyzes a physical structure, he or she focuses on a single structural member and those members that join to it. Analyzing a software design is a similar problem. We are never interested in the behavior of the entire design, since the entire design very rarely responds to any given event. When we analyze a software design, we focus on a single event and a small subset of the full system. We start with the component of the design that is tasked with recognizing the event, and we trace messages only through those components that become involved in the response to that event. This significantly limits the scope of the analysis and makes design analysis techniques useable.

The Nuts and Bolts

Dynamic design analysis consists of several tools developed out of the underlying mathematics. These tools assist the designer in manipulating the objects and their states and for formally following the flow of messages through the active objects.

Describing States, Methods, and Messages

The Object-Z notation for the specification of a class is given by

Class Name [generic parameters]

visibility list
inherited classes
type definitions
constant definitions
state schema
initial state schema
operation schemas
history invariant

A full description of each section is given in Rose (1992). Rather than describe all of the extensions here, we will concentrate on those features that are most useful to our purposes: The constant definitions, the state schema, the initial state schema, and the operation schemas.

Describing Attributes and States

The constant definitions section describes the constants for the class, and may name ranges or allowable values for these constants. The value of a constant is set at object creation and is never changed during the life of that object. It is possible for different objects of a class to have different values for a given constant. A constant description might look like

$$max : \mathbb{N}$$

$$max \leqslant 100.$$

The upper line names the type or domain of the constant *max*, in this case the natural numbers (zero plus the positive integers). The lower line defines any restrictions we place on the values drawn from the domain.

The state schema describes the state variables of the class and perhaps gives a state predicate. The constants and state variables together comprise the attributes of the class. The domain and restrictions for the constants in conjunction with the state predicate form the class invariant. The attributes and class invariant are included in the initial state schema and the all of the operation schemas implicitly. This means that all attributes are available to these schemas without further definition, and all of the conditions in the class invariant are taken in conjunction with any predicates specified in a schema. The state schema for a doubly linked list class with an arbitrary maximum number of entries might look like

$$
\begin{array}{|l}
\hline
count : \mathbb{N} \\
head, tail, current : ListEntry\ Reference \\
\hline
count \leqslant max \\
\hline
\end{array}
$$

Again, the upper line names the domains of the state variables. The lower line places a restriction on the value of any state variable named in the predicate. Here, the state variable *count* is drawn from the natural numbers, and is limited to not to exceed the value of *max*. The domain *ListEntry Reference* means that the state variables *head*, *tail*, and *current* point to objects from a *ListEntry* class defined elsewhere. At least, we hope it is defined elsewhere; this is one of the more common types of design errors.

Describing an Initial State

The initial state schema describes the state an object of the class will have immediately after creation. It is possible in Object-Z to describe more than one initial state, although it is not clear how we can determine which state to apply to any given object. An initial state for the *List* class, in which we allow only one possible initial state, might look like

$$
\begin{array}{|l}
\hline
INIT \\
count = 0 \\
head, tail, current = NULL \\
\hline
\end{array}
$$

We can also use this form to define any intermediate state. We simply declare the values of the relevant attributes. We can give the state a name by replacing “INIT” with whatever name we choose.

Describing a Method or Operation

The operation schemas describe the state of an object both before and after the execution of a method, and thus define the method in terms of its preconditions and postconditions. An operation schema includes a Δ-list that names the state variables of the class that *may* be modified by the operation (the value of constants cannot be changed by a class operation). Operation parameters are defined following the Δ-list in the upper section. The preconditions and postconditions are a list of predicates taken in conjunction with each other and the class invariant. The postconditions define the final values of state variables, which are indicated by the variable name in primed form, as in $count'$. Messages sent by this method are shown using the λ-notation that names the receiving object, gives the name of the method to be invoked, and provides an ordered list of parameters. The shape of this notation is part of the message binding process defined in OPUS.

The operation schema for the *AddEntry* operation for our *List* class might look like

AddEntry

$\Delta(count, head, tail, current)$
object?: *Object Reference*

$count < max$
$object? \neq NULL$
$count' = count + 1$
$entry! = ListEntry.Create(object?)$
$\lambda entry!.SetNext(head)$
$\lambda entry!.SetPrevious(NULL)$
$head \neq NULL \Rightarrow \lambda head.SetPrevious(entry!)$
$head' = entry!$
$current' = entry!$
$count = 0 \Rightarrow tail' = entry!$

In this operation, *object*? represents a new object to be added to the list (in Z, an input variable is represented by *name*? and an output variable is represented by *name*! and the final values of state variables are indicated by the variable name in primed form, as in *count*′). The preconditions for this operation require that the value of *count* is initially less than the value of *max* and the input variable *object*? is nonnull. Note that this design takes the objects to be stored in the list and creates the *ListEntry* objects that reference them, hence the call to *ListEntry.Create*(*object*?). As a result of the operation, the value of *count* is incremented by one, *SetNext* and *SetPrevious* messages are sent to the newly created entry object and the current *head* entry, if one exists, and the value of *head* and *current* are set to reference the new entry. If the value of *count* is initially zero, the final value *tail* is also set to reference the new entry. The conditional message sent to the *head* entry and the final value of *tail*′ are examples of conditional side effects. This is a complete specification of the *AddEntry* operation, yet it reveals no details about the data structure used to implement the list in software, nor does it describe the implementation of the operation itself.

The state of a design, in preparation for dynamic analysis, includes a list of those objects in scope and the descriptions of the current states for each object at the time the analysis begins. This list can range from empty to a nearly infinite variety and number of objects. The mechanism for specifying this list is deceptively simple: You simply name the members of a set called *ActiveObjects*. An object is identified by its class and a unique identifier, as in *List*:1. Of course, objects can also be referred to by variables, as we saw in the operation schema above.

Propagating Messages

OPUS is an object calculus developed to support the development of object-oriented programming languages. Part of this calculus is a well-developed scheme for representing the passing of messages and the binding of those messages to methods in a class. We borrow the ideas of this scheme, though not the notation, to develop our mechanisms for method selection and binding. Part of this mechanism includes checking the set *ActiveObjects* using the function *scope*() to determine if the object to which a message is addressed is in scope.

When an object is created, its identifier is added to the set *ActiveObjects*. When the object is deleted, its identifier is removed from the set. Thus, the function to determine whether an object is in scope is a

predicate which states that the object's identifier is indeed a member of *ActiveObjects*. Thus, the function *scope*() is defined as

$$scope(x) = \begin{cases} true & \text{iff } x \in ActiveObjects \\ false & \text{otherwise} \end{cases}. \quad (1)$$

We define the form of a message as $\lambda x.y(a_i)_{i \in I}$, where x is the object identifier of the receiving object, y is the name of the message, and $(a_i)_{i \in I}$ is an ordered i-tuple which represents the message contents. For binding to occur, the class of x must contain a method m for which the name matches y and the list of parameters $(p_j)_{j \in J}$ matches the message contents of (a_i) on an item-for-item basis. Formally,

$$\lambda x.y(a_i)_{\,i \in I} \bullet \exists m(p_j)_{j \in J} \in Class(x) \mid$$
$$y = m \wedge (\forall i \in I, j \in J \bullet i = j \Rightarrow \text{dom } a_i = \text{dom } p_j \wedge I = J). \quad (2)$$

In OPUS, the method selection mechanism is a simple lookup which compares a message to each method in turn moving from right to left in the object specification. The syntax for OPUS provides a set of reduction rules that govern the lookup and matching process. The reduction rules in OPUS are required since OPUS allows both "ordinary" methods and "constant" methods. A constant method is one that simply returns a constant value. OPUS uses constant methods to represent attributes in an object. They correspond to the state functions in our class model, which we reference by simply naming the attribute.

In our case, we don't need the reduction methods, since we match tuples directly. More specifically, we require the name of the first item in each tuple to match, and the domains of the remaining items to match when compared on an item-for-item basis. Because of uniqueness requirements, we are not allowed to have two methods with the same signature in any class. This gives us the advantage that our lookup will either find one and only one method that maps to a given message, or it will find no such method. Given this simple lookup mechanism, we can perform the lookup by matching categories, by matching tuples, or by using equation (2). If we find a match, we can always be assured that we have found *the* method for which we were searching.

We describe an external event as a message originating outside the design to some object in our design. We can represent this message categorically as an arrow from an object that is not part of the design (such as a user) to the object we wish to receive the message. Algebraically, we can simply describe the message, as in λ*List*:1.*AddEntry*(*object*), which denotes sending the *AddEntry* message, containing the atom *object*, to the first instance of our *List* class. Binding takes place if *scope*(*List*:1) returns true, and if the class *List* contains an *AddEntry* operation that requires a single parameter of the same type as *object*.

If binding fails, we go no further. However, if binding succeeds, we need to check to see if the preconditions for the *AddEntry* operation in the class *List* are satisfied. If they are not, we go no further. If they are satisfied, we can assume the postconditions for the operation without further proof. If these postconditions include messages to other objects, we repeat this process for each message.

The messages propagating through our design will form a many-branched tree. We need to evaluate every path on this tree using our analysis techniques. Our path through this tree is a basic tree search problem. We can employ any of three search patterns to this problem: breadth-first, depth-first, and in-order. In this case, due to the possibility of a branch being cut short by binding failure, it is best to apply the breadth-first pattern. This would mean evaluating all of the messages generated by a method before following the links created by those messages.

Inference Rules

A deductive apparatus allows us to manipulate strings of symbols based on their syntactic shape, without having to assign them any meaning (Woodcock and Loomes, 1989). A deductive apparatus consists of axioms, which are primitive statements of a formal language, and inference rules, which allow us to deduce statements in a formal language as an immediate consequence of other statements. The axioms are provided by the object theory underlying dynamic design analysis. We define the inference rules in this section. An inference rule has two parts: the set S_1 of statements you need in order to apply the rule, and a set S_2 of one or more statements that result from applying the rule. An inference rule is denoted by

$$\frac{S_1}{S_2}. \tag{3}$$

You can think of an inference rule as a form of allowable substitution: given the set of statements S_1, you can substitute the statements in S_2 without changing the meaning of the expression. Inference rules are most often used for simplifying a set of statements.

The underlying object model provides for a number of inference rules, in addition to those provided implicitly by propositional and predicate calculus and relational algebra. These rules allow the substitution of classes for objects, objects for classes, methods for messages, messages for methods, and states for sequences of one or more methods.

Object Elimination

Given an object x of class A, denoted $A{:}x$, we can substitute the class A:

$$\frac{A{:}x}{A}. \tag{4}$$

Object Introduction

Given a class A, we can substitute any object x of the class that is currently in scope:

$$\frac{A,\ x \in ActiveObjects}{A{:}\ x}. \tag{5}$$

Message Elimination

Given a message and a content tuple $\lambda x.y(a_i)$, we can substitute the method $m(p_i)$ of the object's class, provided that the message can be bound to the method:

$$\frac{\lambda x.y(a_i)\ ,\ y = m \wedge m \in Class(x),\ x \in ActiveObjects}{m(p_i)}. \tag{6}$$

Message Introduction

Given a method $m(p_i)$ of a class A, we can substitute a message to any object x of the class that is currently in scope:

$$\frac{m(p_i),\ m \in A,\ y = m \wedge x \in A,\ x \in ActiveObjects}{\lambda x.y(a_i)}. \quad (7)$$

Subtype Substitution

Given an object x of a class A which is a generalization of a class B, we can substitute an object y of class B (this is the Liskov substitution principle):

$$\frac{x \in A,\ (A, B) \in R_g,\ y \in B}{y}. \quad (8)$$

Method/State Substitution

Given a state S_1 of an object x of a class A, and a method m of the class A, and that the state S_1 satisfies the preconditions of m, we can substitute the resulting state S_2 of m applied to an object in state S_1:

$$\frac{S_1,\ m,\ m\colon S_1 \rightarrow S_2}{S_2, \text{ with side effects of } m}. \quad (9)$$

The method/state substitution rule can be applied to a state S_1 and a sequence of methods m_1; m_2;...; m_n applied in the order given:

$$\frac{S_1,\ m_1;\ m_2;...;\ m_n,\ \cup m_i\colon S_1 \rightarrow S_2}{S_2, \text{ with side effects of each } m_i}. \quad (10)$$

This allows the substitution of a terminal state for a starting state and a sequence of methods, making the use of predefined method sequences easier to model. The method/state substitution rule can also be applied to design changes and design states:

$$\frac{DS_1,\ o_1;\ o_2;...;\ o_n,\ \cup o_i\colon DS_1 \rightarrow DS_2}{DS_2}. \quad (11)$$

An Example

As an example, we look at the design of a simple doubly linked list which constists of two classes: *List* which represents the list itself, and *ListEntry* which represents the nodes in the list and contain references to arbitrary objects. The class *List* has, as attributes:

- *head* -- A reference to the first entry in the list;
- *tail* -- A reference to the last entry in the list;
- *current* -- A reference to the most recently accessed entry; and
- *count* -- A count of the number of entries currently in the list.

As we noted previously, the domain for *count* is the set of positive integers and zero, otherwise known as the natural numbers, but in this design, it does not have a maximum. The modified schema is

count: ℕ *head*, *tail*, *current*: *ListEntry Reference*

The domain for *head*, *tail*, and *current* is the set of valid references to objects of the class *ListEntry*.

The class *List* has, as methods:

- *AddEntry* -- Attaches a given object to the list preceding the *head* entry, creating a new instance of *ListEntry* adjusting the attributes *tail*, *current*, and *count* as appropriate;
- *DropEntry* -- Removes a given entry from the list, adjusting the attributes *head*, *tail*, *current*, and *count*, as appropriate;
- *GetCurrent* -- Returns the entry referenced by the attribute *current*;
- *GetHead* -- Returns the entry referenced by the attribute *head*, and sets *current* to reference the same entry as *head*;
- *GetTail* -- Returns the entry referenced by the attribute *tail*, and sets *current* to reference the same entry as *tail*; and
- *IsMember* -- Returns true if the provided entry exists in the list, or false if not. This method looks for an entry in the list that equals the entry that we pass in. For our purposes, two entries are equal if they have the same values for *previous*, *next*, and *contents* (see the definition of the *ListEntry* class below).

The class *List* has three states:

- *Empty* -- The list contains no entries: *head*, *tail*, and *current* are null references, and *count* = 0;
- *One* -- The list contains one entry: *head*, *tail*, and *current* reference the same entry, and *count* = 1; and
- *Two* -- The list contains two or more entries: *head* and *tail* refer to different entries, *current* may refer to any entry in the list, and *count* equals an integer representing the number of entries in the list.

The initial state for the class *List* is *Empty*.

INIT

count = 0 *head*, *tail*, *current* = *NULL*

The diagram for the class *List* is shown in Figure 1. As you can see, a diagram for even a simple class can get ugly in a hurry. The main reason is that both static and dynamic elements are shown. Most object-oriented methods create separate models for different elements of the design for this very reason. Our design analysis technique builds upon these category diagrams for classes and designs, but does not require us to define them.

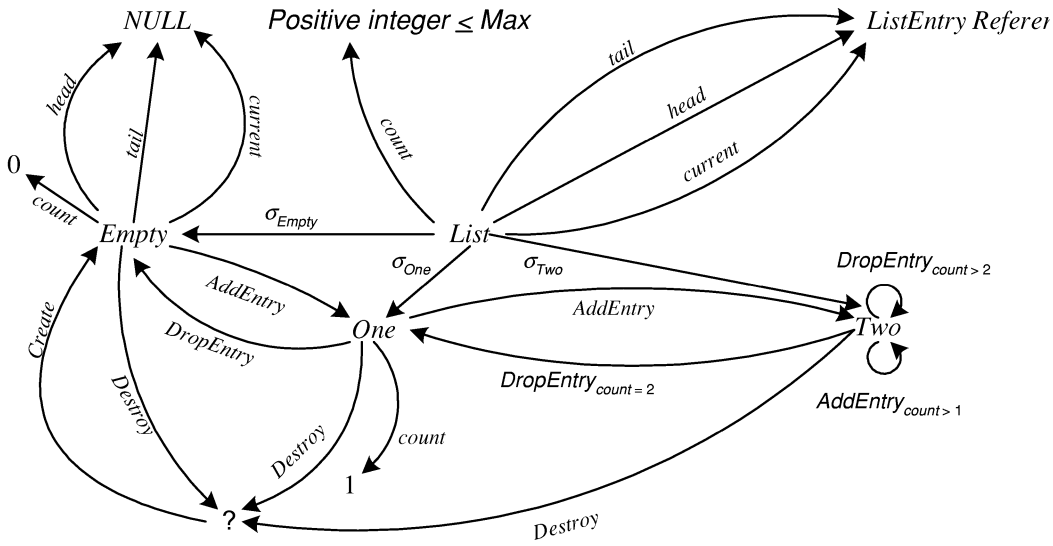


Figure 1. Class category for *List* class.

We're going to define the operations for our *List* class, then analyze what happens when we invoke various methods when we start in each of several states. Some of these invocations will result in errors, and we'll see how easy it is to determine whether these errors are expected or we need to modify our design to deal with them.

AddEntry

$\Delta(count, head, tail, current)$
object?: *Object Reference*

$object? \neq NULL$
$count' = count + 1$
$entry! = ListEntry.Create(object?)$
$entry! \neq NULL$
$\lambda entry!.SetNext(head)$
$\lambda entry!.SetPrevious(NULL)$
$head \neq NULL \Rightarrow \lambda head.SetPrevious(entry!)$
$head' = entry!$
$current' = entry!$
$count = 0 \Rightarrow tail' = entry!$

AddEntry takes an arbitrary object *object*? and adds it to the list, creating the *ListEntry* object that will hold the reference to *object*?. The preconditions for this operation require that the value of the input variable *object*? is nonnull. As a result of the operation, the value of *count* is incremented by one, *SetNext* and *SetPrevious* messages are sent to the new entry, a *SetPrevious* message is sent to the current *head* entry if one exists, and the value of *head* and *current* are set to reference the new entry. If the value of *count* is initially zero, the final value *tail* is also set to reference the new entry.

Note that this is only one of many possible designs for a doubly linked list. Note also that we have not begun to discuss the underlying data structures (the data types for *List* and *ListEntry*).

DropEntry

$\Delta(count, head, tail, current)$
entry?: *List Entry Reference*

$0 < count$
$entry? \neq NULL$
$\exists\ entry \bullet entry = entry?$
$count' = count - 1$
$cuurent' = entry?.GetPrevious()$
$entry?.GetNext() \neq NULL \Rightarrow \lambda entry?.GetNext().SetPrevious(entry?.GetPrevious())$
$entry?.GetPrevious() \neq NULL \Rightarrow \lambda entry?.GetPrevious().SetNext(entry?.GetNext())$
$head = entry? \Rightarrow head' = entry?.GetNext()$
$tail = entry? \Rightarrow tail' = entry?.GetPrevious()$
$\lambda entry?.Destroy$
$count = 2 \Rightarrow (current' = head'; tail' = head')$
$count = 1 \Rightarrow (current' = NULL; head' = NULL; tail' = NULL)$

DropEntry takes a *ListEntry* object *entry*? and removes it from the list. We remove a particular *ListEntry* object rather than any entry that references some *object*? because we allow more than one entry to reference the same object. *DropEntry* also repairs the hole left in the links by the removal. The preconditions for this operation require that the value of the input variable *entry*? is nonnull and that an entry exists in the list that is equal to *entry*?. As a result of the operation, the value of *count* is decremented by one. If *entry*?.*next* is nonnull, the entry it references is sent the *SetPreivous* object with the value *entry*?.*GetPrevious*(). If *entry*?.*previous* is nonnull, the entry it references is sent the *SetNext* message with the value *entry*?.*GetNext*(). The *entry*? object is sent the *Destroy* messasge. If the current *head* references *entry*?, it is set to the next entry in the list. If the current *tail* references *entry*?, it is set to the previous entry on the list. If the value of *count* is initially 2, the final values of *head*, *tail,* and *current* all now reference the same object. If count is initially 1, the values of *head*, *tail*, and *current* are null, and we're back to our initial state.

GetCurrent

entry!: *ListEntry Reference*

---

entry! = *current*

GetCurrent simply returns the entry referenced by *current*.

GetHead

$\Delta(current)$
entry!: *ListEntry Reference*

---

entry! = *head*
$current' = head$

GetHead returns the entry referenced by *head* and sets *current* to reference that entry.

GetTail

$\Delta(current)$
entry!: *ListEntry Reference*

---

entry! = *tail*
current′ = *tail*

GetTail returns the entry referenced by *tail* and sets *current* to reference that entry.

IsMember

entry?: *ListEntry Reference*
q!: *Boolean*

$\exists$ *entry* • *entry* = *entry*? $\Rightarrow$ *q*! = *True*
$\neg\exists$ *entry* • *entry* = *entry*? $\Rightarrow$ *q*! = *False*

IsMember returns *True* if there is a list entry that is equal to *entry*?, or *False* if there is no such entry. We define equality in this case to mean that *next*, *previous*, and *contents* in *entry*? must all equal the respective values of some entry in the list. *IsMember* probably requires an iterator function of some kind, but there may be an implementation that can avoid an iterator.

The state schema for the class *ListEntry* is

contensts: *Object Reference*
next, *previous*: *ListEntry Reference*

---

contents ≠ NULL

The restriction that *contents* cannot be null reflects a notion that we don't want empty entries in our list. This design doesn't deal with the fact that an object referenced by a list entry can be destroyed, rendering the value of *contents* invalid. This list and its *ListEntry* objects don't really care. Users of *GetContents* would care, and we'd have to include that in the design of any method that used the list.

The class *ListEntry* has, as methods:

- *GetPrevious* -- Returns the value of previous;
- *GetNext* -- Returns the value of next;
- *GetContents* -- Returns the object referenced by the list entry;
- *SetPrevious* -- Sets the value of previous to the value passed in the message; and
- *SetNext* -- Sets the value of next to the value passed in the message.

The operation schemas for *ListEntry* are

GetPrevious

entry!: *ListEntry Reference*

---

entry! = *previous*

GetNext

entry!: *ListEntry Reference*

---

entry! = *next*

GetContents

object!: *Object Reference*

---

object! = *contents*

SetPrevious

Δ(*previous*)
entry?: *ListEntry Reference*

---

previous′ = *entry*?

SetNext

Δ(*next*)
entry?: *ListEntry Reference*

---

next′ = *entry*?

The class *ListEntry* has only one state. We enforce this by requiring an object reference as an input parameter to the *Create* method, to be consistent with our state definition. We define the *Create* method as

Create

Δ(*next, previous, contents*)
object?: *Object Reference*

---

next′ = *NULL*
previous′ = *NULL*
contents′ = *object*?

Now that we've defined our two classes, we want to see if the design is sufficient. Our design is sufficient when it contains enough of the essential features of a doubly linked list to be useful for our purpose, and those features won't fail unexpectedly. Any list is an abstract data type with an underlying data structure and a set of operations. We can compare the operations we've defined to an abstract

definition of the operations for a list to see if we've included all of them. All list containers must be able to create an empty copy of themselves, destroy themselves (sometimes destroying their contents as well), and provide operations to add, remove, and locate an entry. A doubly linked list must be navigable in either direction, so we have to know where to find both the head and tail of the list. Clearly, our class definitions, taken together, define an adequate doubly linked list that can hold arbitrary objects by reference. In our case, destroying the list simply destroys the list; the objects referenced by each ListEntry object are not affected.

The next step is to assume a number of initial states in a design that includes these two classes and see what happens when we trigger various events.

Scenario 1 – We try to add an object to a list that doesn't exist. This is the most obvious degenerate case. The external event is a message to a List object:

$\lambda List{:}l.AddEntry(object?)$

Since the instance *l* of the list does not exist in *ActiveObjects*, by our initial assumption, the object theory provides that the message cannot be bound and we get an error, as expected.

Scenario 2 – We try to add an object to a list that we just created. The first message is to create the list:

$l = \lambda List.Create()$

Then, we send it the object we want to add:

$\lambda List{:}l.AddEntry(object?)$

This time, the messasge can be bound and we now have a list with one entry.

Scenario 3 – This time, we'll try to remove an entry from a list that exists, but which is empty. The initial state schema is

Scenario 3

$List{:}l.count = 0$

entry?: *List Entry Reference* (previously retrieved from *List*:*1)*

The message is

$\lambda List{:}l.DropEntry(entry?)$

Looking at the operation schema for DropEntry, we can see that one of the preconditions ($0 < count$) for the operation is violated and our message would generate an error.

With any non-trivial design, the scope of an analysis is huge and the process is time-consuming. However, the time required to analyze any design is two or more orders of magnitude less than the time required to code and test it. The more complex the design, the more benefit there is by analyzing it first.

Summary

This paper presents a technique for representing and analyzing the dynamic behavior of a design before time and effort is spent to build it. The technique is based on a mathematical model of objects, software, and software designs and provides operations and inference rules to manipulate not only the objects in a design, but the design itself (see Whitmire, 1997).

The technique involves defining an initial state for a scenario and seeding that state with an event that takes the form of a message that originates outside the design. We follow the messages generated as

side effects in a breadth-first order until we reach a new steady state, one in which no object changes state without another external event.

Working through various scenarios, it is fairly easy to determine which messages will succeed and which will not. The difference between testing a design in this way and testing a design that we've put into code is the time spent writing the code. The schemas above are very easy to modify, especially when compared to the effort to modify any code. For example, I added the feature to send *AddEntry* an object reference and have the operation create the *ListEntry* object in a couple of minutes while editing this paper. Modifying code to the same extent would have taken many hours.

One additional, if unobvious, benefit of dynamic design analysis is that that you've analyzed a large part of the design, you have complete design contracts for all of the methods you have to build. The specific implementation of each method still needs to be determined, but the rest of the design's view of that method has been fixed and verified.

Using this technique does not guarantee that you will find all of the errors in your design. You still have to imagine the combinations of external events and starting states that can cause your design to fail, and it is likely that you'll miss one or two. It will, however, provide you with documented results of the combinations you did analyze, giving you a good start on your set of test cases, including the expected results. This works for combinations that lead to errors, as well as those that succeed in passing a message. The technique also encourages you to state explicitly all of your assumptions. Hidden assumptions, such as which objects are actually in scope, are a leading cause of design errors.

Like those who build bridges, machines, and electronic circuits, we can be more confident in our approach to solving a problem if we can show that our approach will indeed work before we build it. Testing a design before building it should lead to more accurate estimates because it eliminates the time required to rework code due to an error in the design, something not included in any estimate. Like steel, concrete, and electronic components, software code is very expensive to modify, so expensive, in fact, that we can no longer afford to build large, complex systems several times while trying to deliver it the first time.

References

Mens et al., 1994

Mens, Tom, Kim Mens, and Patrick Steyeart, "OPUS: A Calculus for Modeling Object-Oriented Concepts", Department of Computer Science, Brussels Free University, Technical Report vub-tinf-tr-94-04, 1994.

Petroski, 1996

Petroski, Henry, *Invention by Design*, Harvard University Press, 1996.

Rose, 1992

Rose, Gordon, "Object-Z," in *Object-Orientation in Z*, Susan Stepney, Rosalind Barden and David Cooper, eds. Springer-Verlag, 1992, pp 59-77.

Whitmire, 1997

Whitmire, S., *Object-Oriented Design Measurement*, John Wiley & Sons, 1997.

Woodcock and Loomes, 1989

Woodcock, J. and M. Loomes, *Software Engineering Mathematics*, Addison-Wesley, 1989.

Retrospectives: The Impact to Process Improvement

Debra Schratz, Intel Corporation

Does your organization take the learnings from each project and use them to increase the capabilities of the next team? Development teams at Intel are using Project and Milestone Retrospectives to reduce rework, improve product quality, minimize requirements volatility, and shorten time to market.

This paper will share the impact of conducting over 30 Retrospectives in 2003 at Intel. All the results are real life stories of the power of planning, delivering, and implementing an effective Retrospective process as a standard method for continuous process improvement.

Bio:
Debra Schratz has 7 years experience in quality engineering. Debra currently works as a Platform Quality Engineer in the Platform Quality Methods group, part of the Corporate Quality Network at Intel Corporation. Since January 2003, Debra has delivered over 50 Project and Milestone Retrospectives for Intel worldwide. Prior to her work in quality, Debra spent 8 years managing an IT department responsible for a 500+ node network for ADC Telecommunications. Debra is a member of the Rose City SPIN Steering Committee. She holds a Bachelor's of Arts degree in Management with an emphasis on Industrial Relations.

Introduction

Last Fall I picked up *Good to Great: Why Some Companies Make the Leap…and Others Don't* written by Jim Collins. As I was reading the book I kept thinking "I wonder if Retrospectives will play a part in separating those companies that are good, from those that make the leap to greatness?"

Finally, on page 72 I found it! "When you turn over rocks and look at all the squiggly things underneath, you can either put the rock down, or you can say, 'My job is to turn over rocks and look at the squiggly things,' even if what you see can scare the hell out of you." This is a quote from Fred Purdue, a Pitney Bowes executive who spends two hours at their annual management meeting discussing all the "scary squiggly things" they need to talk about to continue to be a successful company. The description of the annual meeting sounds a lot like a Retrospective.

Siemens has been using Retrospectives for many years and seeing terrific results. Standard Insurance uses Retrospectives and is finding them valuable. The State of Washington is just beginning to use Retrospectives and is excited about the possibilities.

Over the past three years I have had the pleasure of incorporating Retrospectives into the work that I do at Intel. I use Retrospectives to help teams improve productivity and increase the quality of our products. The results have been surprising. One Project Manager estimated the savings attributed to an effective 4 hour Retrospective could add up to **FOUR (4) MONTHS on one project alone**! The ratio of one hour equals one month savings is pretty exciting.

The purpose of this paper is to share the impact of conducting 20 Retrospectives at Intel. All the results are real life stories of the power of planning, delivering, and implementing an effective Retrospective process as a method for continuous process improvement.

What is a Retrospective?

Many companies support a ritual to allow employees to communicate to senior management what they feel the company needs to do differently. From *Good to Great*: "shoving rocks with squiggly things in their faces, and saying, "Look! You'd better pay attention to this."

A Retrospective is a method to capture lessons learned from projects. In Norm Kerth's book *Project Retrospectives: A Handbook for Team Reviews* he describes how to facilitate a Retrospective using specialized techniques to create an environment based on safety and trust. This methodology encourages the team to spend 2% of the total project time (Norm recommends 3 days) to uncover new ways of working together more effectively. The objective is for the team to use this information to increase the capability of the team by:

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

Norm's approach is very simple: "A Retrospective is an opportunity for the participants to learn how to improve. The focus is on learning – not fault finding."

For a Retrospective to be effective and successful it has to be done in the context that no matter what is uncovered and discussed, it is in the spirit of improvement. Norm shares in his book a way for facilitators to create a safe environment where the root cause of an issue can be discussed and solved without fear of management using this as an opportunity to reprimand or punish. A key point in his book is the Prime Directive: Regardless of what we discover, we must understand and truly believe that everyone did the best job he or she could, given:

- What was known at the time
- His or her skills and abilities
- The resources available to them
- The situation at hand

The prime directive sets the tone for the Retrospective to be an opportunity to improve not finger point or blame. Norm unveils a common sense approach of using different exercises to uncover what really happened during the life of the project. The three phases are:

Phase 1= Get Ready

- Collect project data using a survey/questionnaire and/or interviews
- Set the stage

Phase 2 = Look at the Past

- Re-create what happened on the project from all perspectives
- Harvest the data

Phase 3 = Plan for the Future

- Create Action Plans
- Communicate outcomes to management

In 2001, I met Norm at the Rose City SPIN (Software Process Improvement Network) in Portland, Oregon. To my surprise Norm lives in Portland and was willing to engage in off-line conversations about his methodology and techniques. After reading his book we

had several telephone conversations and face to face meetings discussing how his concepts could be modified to fit into Intel's culture.

Retrospectives at Intel Corporation

Since 1999, I have been part of a small team within the Corporate Quality Network (CQN) responsible for driving good product development practices to reduce rework, increase productivity and ultimately improve the quality of our products. The focus of my work is in requirements engineering. Many times, as we begin working with teams (to help them with their requirements) we have an opportunity to conduct a Retrospective to discuss and learn what issues the team has been grappling with so we can be more focused in our approach with the team.

Applying Retrospectives to a Specific Division at Intel

Late in 2002, a division within the Intel Communications Group (ICG) was willing to pilot the Retrospective methodology as a way to improve their "post mortem" process and apply the findings back into subsequent projects as a method of continuous process improvement.

Review of the Current Practice Uncovered:

- Project Managers were facilitating their own post mortem sessions which didn't allow them to share their perspective of the project
- No defined process that kept the discussions constructive
- Post mortems were done ad hoc with no streamlined/consistent process
- Meeting held sometimes months after the end of the project or not at all
- Lists of issues collected with no action plans associated with solving them
- Information recorded was many times not useful to other projects. For example: not generic enough, not process relevant.
- No consistent format of how to document findings
- Meeting length was between 1-2 hours
- No central repository to share information across teams
- Key Learning's out of post mortem were not easily retrieved to address a specific deliverables
- No close loop to require Key Learning's to be considered for subsequent projects

The approach we took to improving the post mortem process was:

- Establish a consistent process for gathering, sharing, capturing and documenting lessons learned (Norm's Retrospective approach, modified for Intel's culture)
- Plan and hold the project Retrospective within 2-6 weeks after the end of the project. (While memory of the project was fresh in their minds!)
- Use a neutral, skilled facilitator (me) to ensure consistent use of the Retrospective methodology

- Create action plans for the top items the team wanted to change.
- Share Best Known Methods (BKM's) with the division Project Managers at regularly scheduled staff meetings to ensure findings are incorporated into subsequent projects
- Communicate with management findings and action plans for improvement to allow visibility of the changes and support from management

What Happened?

In January, 2004, the manager of the group asked for a briefing of the findings of all the Retrospectives conducted in 2003. This paper is a summary of the process and results obtained.

Beginning in March, 2003 and ending December, 2003 over 30 Retrospectives were conducted in various groups at Intel. 20 were delivered in ICG alone. 85% or 17 of the Retrospectives were done at the end of the project, as part of the standard process supporting the Intel Product Life Cycle. Two Retrospectives spanned a program (a program includes hardware, software, silicon and other platform ingredients) and one Retrospective solely focused on the strategic planning process within the group. The goals of the Retrospectives were:

- Understand what is working well so we can reinforce it
- Capture what is not working so we can improve
- Create action plans for top improvement areas
- Communicate the findings to management

Phase 1 = Get Ready

Using Norm's three phased approach, we would get ready for the Retrospective by collecting project data using a web-based survey. The survey allows us to collect the team's opinions and perceptions on how they did in the project. Their feedback is treated confidentially and only consolidated data is reported back to the project team and the survey participants. We use the feedback to identify key issues that the team should focus on.

The survey is sent out to the team members approximately 2 weeks prior to the face to face Retrospective meeting to get anonymous input from the team. The survey is helpful if:

- Majority of the team members can't attend the Retrospective meeting face to face
- Safety and trust level of the team is low
- Cultural and language issues prevent the team from sharing openly

Figure 1 below shows a summary of the 20 surveys conducted in ICG.

Average Response rate	# of People surveyed	# of People who responded	Average Length of Retrospective
71%	679	486	4 hours

Figure 1

The first survey we conducted we only had a response rate of 11%. We attribute the low response rate to:

- Poor communication to the team about the objective of the survey. Team members were confused because previous "post mortems" had never asked the team to do this. They thought they could attend the face to face meeting and give feedback like they always had done.

  Key learning: We now have a standard email (Appendix A) that goes out to the survey takers explaining the Retrospective process and facilitating participation.

- Not enough time to respond to the survey. We allowed only three days for the team to take the survey and we didn't "send reminders" to get them to participate.

  Key learning: Even in Intel's disciplined culture, we found if we allow at least a week for participants to answer the survey, we can consistently get a response rate of over 70%. To get this type of response rate we send MULTIPLE reminders via email and ask managers to verbally encourage the team members during weekly team meetings. Very rarely, only in extreme circumstances (illness, travel, and key stakeholder who can't log into the network) we extend the time to take the survey.

- Too many survey questions. We asked 78 questions that spanned the entire project, which took the participants too long to take.

  Key learning: We now ask no more than 30 questions and hold milestone Retrospectives (just covering a portion of the project) and found the response rate improved.

- We didn't control who was asked to participate in the survey. We sent an email to approximately 60 key stakeholders, and added (what we thought was a benign statement) to feel free to forward the email to others on the team they felt would have a perspective to share on the project. We estimate the email was sent to over 260 project stakeholders.

  Key learning: We have a strict practice to NOT forward the web survey link so we can control the response rate.

All in all, the survey provided a consistent process for gathering data about the project and provides a basis of anonymous feedback for the team to learn from.

Phase 2 = Look at the Past

After we collected and analyzed all the survey data, we would hold the project Retrospective. We planned a face to face meeting within 2-6 weeks of the project end. That way information about the project was still relatively fresh in the team's mind. In every Retrospective, we used a neutral, skilled facilitator to ensure consistent use of the Retrospective methodology. We would conduct several of Norm's Retrospective exercises and summarize the survey data to draw out all the "squiggly things" the team should be discussing. We would spend time to answer the following questions:

- What worked well that we don't want to forget?
- What did we learn?
- What do we want to do differently next time?
- What still puzzles us?
- What needs to be discussed in more detail?

Spending adequate time answering these questions allowed us to uncover a tremendous amount of data about the project. The output of every Retrospective was to identify the top three to five items the team felt were the most important to continue doing (things that went well) and identify the "squiggly things" they wanted to change. Figure 2 summarizes the top issues the 20 Retrospectives uncovered:

Top 3 Items	Trends
1. Communication	Poor Cross-Site/Cross-Divisional communication resulted in project delays
2. Schedule/Planning	Need realistic schedules to increase effective planning
3. Requirements	Incomplete requirements lead to negative impacts to schedule and increased re-work

Figure 2

Phase 3 = Plan for the Future

All teams prepared Action Plans for their top three to five improvement areas. The Action Plan Template (Appendix B) we used allowed us to:

- Create a clear, concise **problem statement** and identify the impact to the project. For example:
  - Weak problem statement: Team did not define requirements very well
  - Better problem statement: Feedback from customers is received late which results in requirements changes after requirements were baselined.
  - How did this effect work on this project? Cost, schedule, quality, productivity, morale?
- Brainstorm a recommended **solution**
  - Advantages/Disadvantages of implementing the recommended solution.
- Discuss the **impact** of solving this problem
  - Will solving this problem save time or money?
  - Will solving this problem increase effectiveness, productivity, or morale?
- Uncover the **obstacles**
  - Who or what could get in the way of implementing the proposed solution?
- Identify who needs to **support** this recommendation
- Establish an **owner and set due dates** for the next communication for solving the problem

The teams were instructed to use their action plans (and all the data from the Retrospective) to prepare a PowerPoint presentation summarizing the team's learnings or Best Known Methods to be shared with their division project managers and senior management so findings could be rolled into new projects to improve productivity and product quality.

We found an effective management presentation had the following information:

- Clear and concise **problem statement** and all supporting information and rational so management could get a complete picture of the issue identified in the Retrospective, for example:

  - **Problem Statement**: Feedback from customers is received late which results in requirements changes after requirements were baselined.
  - **Rational**: If feedback from customer testing was received earlier in this project, customer validation of new features would identify any gap(s) in time to get it addressed.

- **Impact** on program, for example:

- Customer issues were discovered too late in the project lifecycle which resulted in increased customization which adversely impacted the project schedule.
- Features are delivered in form less than expected/desired by the customer.

- **Owner, due date and next steps clearly defined,** for example:

  - Owner: <Specific person's name, not Marketing Representative>
  - Due date: <Actual date for next update with management>
  - Next steps: This is an issue that may be incorporated into Process Improvement (PI) team work. Action Required: To avoid redundant actions, will follow up with PI team and report findings

The action planning portion of the Retrospective was an easy way to take a lot of information and distill it down into concise statements with clear plans for implementing change.

Action planning is typically the last agenda item in the Retrospective. We often found 4 hours (which was the most time a team would give us to do a reflective meeting) was not enough time to sufficiently understand the issue and develop plans to change.

- Often time would run out and action planning would not happen in the Retrospective. The project manager and I would reschedule another block of time to create action plans which caused the team to miss out on doing this together.

  Key learning: I am encouraging teams to allow 6 to 8 hours for a project Retrospective instead of 4 hours so we can ensure action planning will occur. The teams are receptive to a longer Retrospective now that they have experienced a short (4 hours) meeting and found the time for action planning wasn't enough. My wish is at some point we will standardize on 2% of the total project time!

How can this process be used for continuous process improvement?

After attending several Retrospective briefings, the division project managers held Best Known Methods meetings to ensure findings from the Retrospectives were incorporated into future projects. This resulted in:

- Sharing of information between the divisional Process Improvement team and the Retrospective action plans to reduce redundant efforts
- Documenting the BKM's into the product development process so when a new project kicks off, the project manager could review the learnings and incorporate them into their project plan
- Encouraging Project Managers to attend the Retrospective of the previous product to hear the issues and action plans so they can incorporate the learnings into their next project

In preparation for the 2003 year end summary to the divisional management, I sent out an email asking all the project managers five questions:

1. What improvements have been implemented and what was the impact?
2. What improvements are still in process?
3. What improvements are still ramping up?
4. What improvements were handed off and to whom?
5. What improvements did the teams decide were not important and are not working on?

The feedback from asking the project teams what happened was very interesting. The teams reported many improvements, for example:

- **Reduce Rework**

  Action Plans from Retrospectives were reported to management shortly (2-4 weeks) after the Retrospective face to face meeting.

  Impact: Improvement issues were more visible which reduced redundant efforts and allowed management to give their full support to making the recommended changes.

- **Improve Product Quality**

  Better Change Control was implemented at project level using bug tracking tools based on discussions from the Retrospective.

  Impact: Product quality improved because project scope and risk management was targeted with action plans from Retrospective findings.

- **Minimize Requirements Volatility**

  One team decided in a Retrospective to develop Use Cases to help gather and validate requirements because their old method wasn't meeting their needs.

  Impact: Use Cases help the team to drill-down into unclear requirements, which resulted in elimination of "false" features and ultimately improved the product quality and improved productivity of subsequent teams who used the use cases.

- **Shorten Time to Market**

  One Retrospective conducted after the Planning Phase of a project uncovered that a single point of communication was needed. The team implemented a website for the project to improve communication and reduce time to market.

> **Impact**: The team determined it was much easier to find documents and understand where the project was at all times by using the website to post all documents. This saved the team time searching for information and allowed more time to actually work on the development of the product.

Unfortunately, several of the teams when asked what they implemented from the Retrospective Action Plans responded with "What action plans? Oh, those we filled out in the meeting? Well, I think they are on my desk somewhere!" I assumed the project teams would take the action plans and do something with them. I didn't think it was necessary for me to keep a copy of the outcomes.

A few months later when I queried the teams on what they actually implemented they asked me "What was it we said we would do?"

> **Key learning**: Keep a copy of all the action plans developed by the team and follow up with the team regularly (quarterly) for updates on the changes. Since I had not documented every action plan the teams created, I had no idea what had been implemented and could not form conclusions as to the percentage of items identified to changes implemented.

When I reported out the results to the divisional manager, I asked: "Based on the results I've shared with you, do you feel the Retrospective process we have implemented within your division has saved at least X hours (X=total number of hours invested in Retrospectives)? This is based on the improvements implemented as a result of the outcomes of the Retrospectives."

The divisional manager answered "I don't know, based on your report you have missed several items we implemented such as a validation task force."

> **Key learning**: Don't assume and take the team's word for what THEY think the impact of a Retrospective has been to their projects. Ask management too, they have a broader perspective.

So, what was the Impact?

The total impact of implementing a Retrospective practice in the Intel Communications Group (ICG) is unknown…..However, many teams were so happy with the intangible outcomes such as increased communication the division is still doing Retrospectives today. One Project Manager estimated the savings attributed to an effective 4 hour Retrospective could be up to: **4 months on one project alone**! The ratio of one hour equals one month savings has encouraged me to continue to explore and expand Retrospectives into other divisions and groups at Intel.

Summary

Going into this pilot my goal was to:

Establish Project Retrospectives as a Standard Method for Continuous Process Improvement for all projects in ICG by:

1. Defining and documenting a consistent Retrospective methodology.
2. Sharing BKM's with division Project Managers on a regular basis.
3. Reporting findings to management with action plans for improvements.

What "squiggly things" did I find that I would do differently?

In closing, I want to share my lessons learned:

Key learning: Establish a "Retrospective Apprentice" program early, to build expertise in the company to allow the practice to live on after the process "guru" moves on.

- Within ICG we have conducted two "Facilitating Effective Retrospectives" full day training sessions resulting in over 25 "certified" facilitators.
- The plan is for teams to reciprocate facilitating Retrospectives for each other. The project managers are NOT to facilitate their own -- as the project manager has a perspective from the project worth capturing.

Key learning: Conduct Project Milestone Retrospectives throughout in the project life cycle.

- Many of the project Retrospectives I facilitated was for projects that were 18 months to 2 years in length. When we discussed what worked well and what they wanted to do differently, the team only covered the last few months of effort, because many of the teams couldn't remember what had happened a year earlier!
- This practice allows the team to apply the learnings as the project is progressing, rather than waiting until the end where the opportunity to improve has passed.

Key learning: Develop a centralized repository to track advancement of the action plans and share results to ensure communication to management and new teams on the key areas for improvement.

- Identify at least one resource to be responsible for:
    1. Collecting the learnings from all the Retrospectives. The mistake I made is I provided a summary report to the team after the Retrospective, but I didn't have a central database developed to gather all the learnings, other than my local hard drive.
    2. Monitoring the progress on the action items. Other priorities didn't allow me to follow up with all the teams in a timely fashion. After 3-6 months I would informally ask the project managers how things were going, but I didn't have the authority or responsibility to track the individual team efforts.

3. Communicating the results/status on the action plans. After a year I found I had a lot of interesting data and no process to effectively share it.
    - Set up a process for sharing BKM's with project managers when new teams are formed
    - Establish a quarterly briefing to Senior Management on the continued progress & results. This will keep the process improvements visible. Regular communication to management on the status of improvement actions is essential to continuous process improvement.
    - Remember: "What gets measured gets improved".

My job at Intel provides me the opportunity to work with a variety of project teams to help them roll over rocks to expose "squiggly things" in a way that is fun, effective, and contributes directly to the continuous process improvement efforts. Luckily, the Retrospective process allows me the process to discuss all the "scary" things in a way that directly impacts the effectiveness of the team. This in turn, impacts the success of Intel Corporation.

References:

Collins, J. (2001). *Good to Great: Why Some Companies Make the Leap…and Others Don't*. New York: HarperCollins.

Kerth, N. (2001). *Project Retrospectives: A Handbook for Team Reviews*. Dorset House.

Appendix A – Example email for team to participate in a Retrospective survey:

<Project Name goes here>

You are invited to participate in a Retrospective of the <Project Name goes here>. The objectives for this Retrospective are:

- ***Learn what worked so we can reuse it for future projects.***
- ***Discover what needs to work better so we can improve it.***
- ***Document and debrief the findings to management.***

Prior to the Retrospective, please complete a web-based survey to allow us to collect your opinions and perceptions on how we did in our project. Your feedback will be treated confidentially and only consolidated data will be reported back to the project team and the survey participants. The feedback will be used to identify some key issues that we should focus on.

The survey can be found at the following URL: <URL Goes Here>

Please complete the survey by ***end of day <Date goes here>.*** All input will remain anonymous. Your participation and cooperation is much appreciated. We are planning to use this information for our face to face Retrospective meeting to be scheduled <Date goes here>.

If you have any questions about the survey or the Retrospective process in general, please contact me.

Thanks,

Debra Schratz
CQN

Appendix B – Example Action Planning Template

- Problem statement and impact
  - How did this effect work on this project? Cost, schedule, quality, productivity, morale?
- Recommended solution
  - Advantages/Disadvantages
- Impact
  - Will solving this problem save time or money?
  - Will solving this problem increase effectiveness, productivity, or morale?
- Obstacles
  - Who or what could get in the way of implementing?
- Who needs to support this recommendation?
- Owner, due date, and next steps?

The Requirements Dilemma: It's Hard Work

Debra Aubrey
General Dynamics C4 Systems
Scottsdale, Arizona - United States
Debra.Aubrey@gdds.com

Biographical Sketch

Debra Aubrey is a practicing systems and software engineer, with the good fortune of working across the United States and in Europe. She divides her time between process consulting and applying requirements process techniques to eliciting, writing and maintaining requirements in real projects as a Principal Systems Engineer for General Dynamics C4 Systems.

Over the past 20 years, Debra has worked in a wide variety of industries, including scientific analysis instruments, printers, consumer electronics, semiconductors, avionics and radio products. She presented this paper at the 7th Philips Software Conference in Eindhoven, The Netherlands, where it was voted "Best Paper" by the conference attendees.

After meeting Suzanne Robertson during a Mastering the Requirement Process course in London, Debra worked with James Robertson to become a qualified instructor. Debra brings a pragmatic "just do it" approach and a wealth of experience, to the field of requirements engineering and process.

Debra received her B.S. in Engineering with concentration in Computer Engineering from the University of Connecticut in 1983 and received her M.S. in Engineering Management in 1988. She is the founding member of Solvera, LLC, and is currently living in Phoenix, Arizona.

Abstract

The ability to write "good" Product Requirements is an often-neglected skill. Perhaps because the task of creating a complete set of Product Requirements is so daunting, project teams often turn to tool vendors for a "silver-bullet" solution. The problem is that while requirement management tools do help organize the data, they do nothing to improve the quality of the requirements. This paper addresses how to develop the skill of writing requirements by introducing a set of simple practices to the project team, including terminology, notation and rules.

Mailing Address
General Dynamics C4 Systems
8220 East Roosevelt Street
MD R4238
Scottsdale, AZ 85257

© 2004 General Dynamics. All rights reserved.

I Introduction

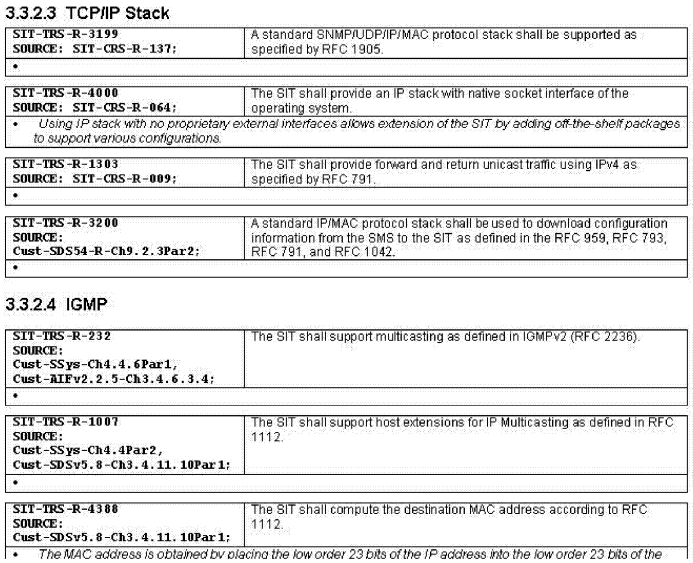
Developing a “good” set of Product Requirements is hard work. Individually, each requirement statement must be at the same time concise, implementation-free, verifiable and unambiguous. Together, the set of Product Requirements must be complete, consistent, and describe the product being built in functional and non-functional terms. Developing requirements can be a daunting task, so it is no surprise when project teams skip this step in order to "get on with" development.

Adding to the misery, as soon as a set of Baseline Product Requirements is agreed, project teams must immediately start managing requested changes to the requirements. One of the most frequently stated reasons for schedule slip, cost overrun and project cancellation is changing requirements. [7]

Change, however, is usually necessary. Customer expectations are clarified during development because the costs and risks of early decisions become clear. Sometimes the Customer’s requirements change, but often the Customer’s understanding of what the product must do simply comes into focus. "Vision" is gradually succeeded by "reality" as the product development matures and requirements change as a result. In some markets new products are constantly introduced and it is essential for product development to keep pace with these changes.

The first reaction of many project teams toward requirements is to buy a tool to help "write" and then "manage" requirements. Despite vendor claims, these tools do not improve the quality of requirements. Configuring and optimizing requirement management tools often preempts the project focus, stealing energy from the actual product development. The "tool" becomes the development target rather than the Product Requirements. If project teams do not develop the skill of writing good requirements then these tools will only be helpful in managing and categorizing long lists of useless data. In other words, when a team is not capable of writing good requirements using a simple word processor, then introducing more expensive tools will not help.

Specifying requirements depends on concentrated effort, but there is clear benefit to developing a complete set of Product Requirements. Comprehensive work breakdown structures can be created that reflect the development and productization process. Work breakdown structures based on complete



3.3.2.3 TCP/IP Stack

SIT-TRS-R-3199 **SOURCE: SIT-CRS-R-137;**	A standard SNMP/UDP/IP/MAC protocol stack shall be supported as specified by RFC 1905.
•	
SIT-TRS-R-4000 **SOURCE: SIT-CRS-R-064;**	The SIT shall provide an IP stack with native socket interface of the operating system.
• *Using IP stack with no proprietary external interfaces allows extension of the SIT by adding off-the-shelf packages to support various configurations.*	
SIT-TRS-R-1303 **SOURCE: SIT-CRS-R-009;**	The SIT shall provide forward and return unicast traffic using IPv4 as specified by RFC 791.
•	
SIT-TRS-R-3200 **SOURCE:** **Cust-SDS54-R-Ch9.2.3Par2;**	A standard IP/MAC protocol stack shall be used to download configuration information from the SMS to the SIT as defined in the RFC 959, RFC 793, RFC 791, and RFC 1042.
•	

3.3.2.4 IGMP

SIT-TRS-R-232 **SOURCE:** **Cust-SSys-Ch4.4.6Par1,** **Cust-AIFv2.2.5-Ch3.4.6.3.4;**	The SIT shall support multicasting as defined in IGMPv2 (RFC 2236).
•	
SIT-TRS-R-1007 **SOURCE:** **Cust-SSys-Ch4.4Par2,** **Cust-SDSv5.8-Ch3.4.11.10Par1;**	The SIT shall support host extensions for IP Multicasting as defined in RFC 1112.
•	
SIT-TRS-R-4388 **SOURCE:** **Cust-SDSv5.8-Ch3.4.11.10Par1;**	The SIT shall compute the destination MAC address according to RFC 1112.
• *The MAC address is obtained by placing the low order 23 bits of the IP address into the low order 23 bits of the*	

Figure 1: Sample page from requirement specification

requirements facilitate more reliable cost and resource estimates. Reliable estimates permit informed business decisions and help substantiate the business case for the product. Clear statements of product functionality facilitate architectural design and system testing. With the scope and requirements defined, the entire project is more stable and changes are less traumatic.

This paper outlines an approach to create good Product Requirements that was successfully applied across a number of products, and which is currently being applied and refined at General Dynamics C4 Systems in Scottsdale, Arizona. The approach consists of three basic steps:

1. Gather requirements
2. Document requirements according to agreed rules
3. Review and control requirements

"Beginner" criteria are suggested as guidelines for both writing and evaluating requirement statements, with examples of the notation included in an Appendix at the end of the paper. Figure 1 shows a sample page that demonstrates the format of specifications generated using this approach.

II Creating a Common Vocabulary

The first step to engaging in any productive discourse is to establish a shared understanding of terminology. In the case of requirements, there are different types of requirement statements that must be defined. Organizations often have their own definitions and might debate the definitions given below. The important point is to have a common definition of terms at the project level. This paper adopts the following terminology [2, 3]:

1. Requirement: A statement of feature or function that must be incorporated into the product. Failure to meet a Requirement may cause application restrictions or hinder operations, resulting in improper functioning of the system. A Requirement contains the word “shall”.
   Example: The product shall provide the AH protocol for authentication of IP data.
2. Constraint: A limitation imposed on the design or implementation of the product. A Constraint could be an external standard to which the product must comply or a Customer-imposed solution. A Constraint contains the words "must" or "must not".
   Example: The cryptographic library must be bundled with the IKE application software marketed by CS International.
3. Objective: A feature or function that is optional, but desirable. An Objective is not a binding requirement placed on the system; rather it is a goal. An Objective contains the words “may” or “should”.
   Example: A Novice User should be able to easily configure the software with no more than 30 minutes of Customer Support assistance.
4. Assumption: A feature or function that is not explicitly stated, but which must be present to ensure proper functioning of the final product. Assumptions often result from requirements omitted in higher-level documents. An Assumption contains the word “will”.
   Example: The target hardware will provide a real-time clock.

Throughout this paper, "requirement statement" is used to generically refer to the types defined above. Distinguishing between these types of requirement statements is important because of the

impact that each has on the final product.

A Requirement is typically the most readily understood of the different requirement types. Simply stated, if the product does not provide the required functionality or possess the required characteristic then it fails to fulfill its raison d'être. Requirements do not dictate the product implementation. *Product definition* is the subject of the requirements gathering activity. *Product design* is the subject of the analysis and design activity.

A Constraint, on the other hand, limits the implementation of the product. For example, if the product must run on Windows, the project team would waste time and money developing the product for Linux. When Constraints placed upon the product solution are not respected, the final deliverable is just as unacceptable as when the Requirements are not satisfied.

Objectives capture "Nice to Have" features or characteristics that are not essential to proper operation of the product, but that may enhance its marketability. Since Objectives are not binding requirements on the product the project team typically includes them after all Requirements and Constraints are satisfied.

Assumptions serve a different purpose. An Assumption identifies something that the product imposes on or expects from its environment. When the project team explicitly notes Assumptions made during the requirements gathering activity, the Customer and other stakeholders are given the opportunity to clarify and correct misunderstandings. Clarifying Assumptions early in the project ensures that the entire project team - not just the developers - has the same understanding regarding the conditions under which the product will operate. Knowing when Assumptions are being made is perhaps the biggest challenge related to their definition.

III Gathering the Requirements

Often the hardest part of developing Product Requirements is starting the process. Figure 2 provides

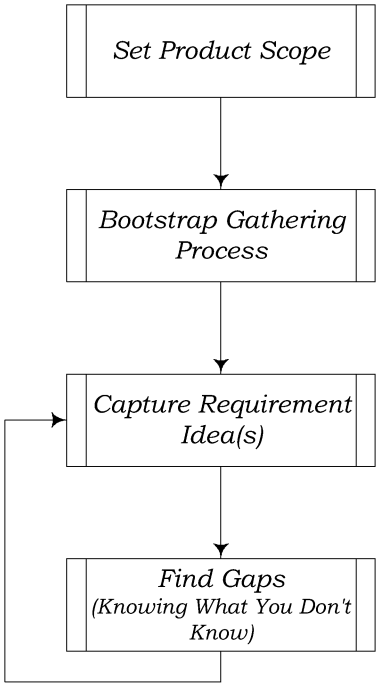


Figure 2: Gathering requirements

a simple overview of the gathering process. First, set the scope of the product development. The Product Scope [5] records the result of three activities:

- Identifying the chain of command (who is accountable for what), and
- Clarifying the product goal (what the product should do and why), and
- Defining how the product success will be measured (success criteria).

The Product Scope is essential to the ultimate success of the product. The product goal and success criteria guide the project team as it develops the Product Requirements. The Product Scope also serves as a sanity check preventing the project from straying off track. The Product Scope must not be confused with Objectives, a type of requirement statement in the previous section. The final product must meet expectations as stated in the Product Scope, whereas Objectives are optional. The product is released when the Customer verifies that success criteria have been satisfied demonstrating the product's conformance with the Product Scope.

Bootstrapping the Process

Where do requirement statements come from? There are a number of ways to bootstrap the requirements gathering activity. Some examples include evaluating competitive requirements, re-using legacy requirements, brainstorming new requirements and researching industry-standard requirements.

Competitive requirements. With the proliferation of information available over the Internet, it is easy to search for competitive product information and analyst criticism. Armed with this information, the project team is able to evaluate features and functions to determine the requirements necessary for the new product to effectively compete in the marketplace.

Legacy requirements. Reusing requirements is an effective way to kick off a new project because products are typically evolutions, rather than revolutions. In other words, the next product provides essentially the same functionality as the last product. Assuming Product Requirements were developed for the last product, many of those requirement statements can be adopted and adapted for the new product. Legacy requirements, when done well, provide valuable insight into the product domain. It is also likely that change requests were issued against the last product and deferred to later versions. These change requests may now be requirements for the new product.

New requirements. Brainstorming sessions are an excellent way to identify and capture new requirements because they use the collective imagination of a group to generate ideas. One approach is to hold theme-based sessions where the project team proposes ideas for functionality by focusing on one particular aspect of the product.

Representatives from the Customer and System Testing should also be invited by the project team to participate in brainstorming sessions. The Customer participates in generating new ideas and provides the project team quick feedback with respect to proposed functionality. System Testing is involved throughout the requirements gathering process to ensure common understanding between the development and test teams. One important contribution made by the test team is to ensure testability requirements are specified and included in the Product Requirements.

Industry-standard requirements. Given many companies' commitment to open standards, projects are often guided – at least in part – by industry standards and specifications. As a result, “document archaeology” is an effective way to identify requirements. With this technique, thick and rather academically written documents are searched for essential points to be translated into requirement statements.

During the requirements gathering phase, it is not as important to write down "good" requirement statements, as it is to capture all of the ideas being generated. For this reason, a "requirement template" is used to capture the essence of the functionality or characteristic being discussed. Figure 3 shows a sample requirement form. Use one form for each idea, and uniquely identify each idea as it is generated.

As the project proceeds, ideas captured in the requirement forms are refined to become formal requirement statements. This notion of clarifying ideas through stepwise refinement translates into "traceability". The source for every idea is captured in the form allowing the project team to return to the point of origin when questions regarding ambiguity and priority arise.

Knowing What You Don't Know

Enabling the project team to distinguish between what it does know and what it does not yet know is one of the most significant strengths of the requirements gathering activity.

The first set of Baseline Product Requirements quantifies the product in terms of what is known leaving placeholders for what needs investigation. When the development project begins, the set of Baseline Product Requirements need not be complete … merely sufficient to proceed. The project team fills these knowledge gaps during iterative development targeted towards clarifying the unknowns before starting design.

Through successive iterations, the Baseline Product Requirements converge toward completeness. The project team must remain vigilant during the requirements gathering activity to ensure that the product boundaries are not being implicitly extended. The Product Scope - its purpose and success criteria - defined at the beginning of the project sets the boundaries and guides the order in which knowledge gaps are filled.

Function ref:	SIT-TRS-R-100						
Source (principal)	Cust-SDS-R-Sec3.4.3.1.2, Cust-SDS-R-Sec3.4.3.1.5						
Function Title	SIT finds the "home" transponder						
Reviewer Initials	REI	Date	13.08.99	Version	0.0.1		
Function Description The SIT shall tune to the forward path home transponder in the network without assistance from the user.							
Assumptions/Pre-Conditions: Default transponder entered during Installation and Commissioning.							
Inputs (Signals, Events, Stimuli, Data, Etc): Electrical signal from the satellite. (Specific SAT position) The electrical signal carries an MPEG-2/DVB-compliant transport stream. *What triggers this function?*							
Actions (Dynamic Behaviour Of Function): The SIT tunes to default transponder in the network. In a first time the SIT will filter the NIT and will parses the `actual_network` (`Network_id`) to find the transponder carrying RCS-NIT. Once the `actual_network` is found the SIT will search for the relevant `linkage_descriptor` for the RCS-NIT. The SIT tunes to RCS-NIT transponder and parses RCS-NIT to find "home" transponder.							
Outputs (Result, State, Data, Etc) The SIT is tuned to the proper "home" transponder.							
Performances (Dynamic Requirements, Etc) The SIT shall tune to any forward path transponder in the network.							
Function well defined? Yes/~~No~~							
Comments *What is dynamic requirement for tuning to transponders?*							
Functional impact on modules?							
PC SW ☐	Tx ☐	Rx ☐	PC HW ☐	U/C ☐	IFL ☐	ODU ☐	Refer to Product Mgt? ☐

Figure 3: Sample requirement form

IV Documenting Requirements

Adopting a standard format to identify requirements facilitates common practice among team members in developing requirement statements. The notation suggested in Figure 4 enables traceability. A list of sources attached to each requirement statement helps ensure that the requirement can be traced to its origin.

Requirement Label	Requirement (normative) text.
• *Informative text.*	

Figure 4: Requirement statement format

Using the notation shown in Figure 4, each requirement statement is captured in tabular format. The requirement statement consists of a requirement label, normative text and informative text. The requirement label and normative text occupy the first row of the table. Informative text needed to clarify a requirement statement is highlighted using italic text, and is included as the last row of the requirement table.

Additional information or "requirement attributes" may be appended as additional rows in the table. For example, if the project management plan requires that a priority level be assigned to each requirement, a priority field can be added to the table.

Requirement Label. The Requirement Label includes a unique identifier for the requirement statement as well as a list of the sources for the requirement. The format of the requirement label is explained in detail later in this section.

Normative Text. Normative Text is the requirement statement. Requirement statements can be partitioned into two classes: functional and non-functional. Functional requirement statements answer the question “What?” and tend to be stated as verb phrases. Non-functional requirement statements describe qualities and behaviors of the product and tend to be stated as adjectives.

Informative Text. Informative Text provides additional information to help clarify the requirement statement. The Informative Text answers the question “Why?”

The strength in this notation is that it encourages good form and discourages lengthy discourse. A difficulty with natural language specifications is they tend to be narrative. While stories are nice to read, it is often hard to identify the real requirements when they are embedded in volumes of academic and descriptive verbiage. Figure 5 shows an example of a completely specified requirement.

System-FRS-R-127 **SOURCE: System-CRS-R-015;**	The source code shall comply with the ANSI C standard.
• *The product will be sold in either source or binary format. When the source code is shipped, it must comply with the ANSI C standard to help ensure that it will compile using most commercial compilers.*	

Figure 5: Requirement statement example

Traceability

Traceability provides the ability to directly link each requirement to its origin, and is used to ensure that the product is built and tested to meet the Customer’s expectations.. As a rule, each design component and test case should trace to at least one of the Product Requirements. Traceability is built into the Requirement Label by explicitly stating the source(s) of each statement.

A statement's legitimacy should be questioned when the source of a statement is missing from the requirement label. A missing source could indicate "feature creep," where non-essential (but probably fun to implement) features tacitly extend the Product Scope and consume project team resources otherwise needed to develop the product.

Ideally, each requirement statement traces to at least one statement in a higher-level source document. Product development is a stepwise refinement starting from abstract requirements regarding functionality and characteristics and ending in a detailed specification - in the form of source code or schematics – of the solution. As the project continues through the design process, more detail is

derived. As these details emerge, it is common to encounter design requirements that cannot be explicitly traced to a source document.

In these cases, a “Statement of Rationale” is used to justify implementation requirements derived through increasingly detailed specifications. The Statement of Rationale is a simple explanation of why particular functionality is required. The value of the Statement of Rationale is twofold. First, it captures the project team’s reasoning for other stakeholders' review, thereby reducing the possibility of invalid decisions being made. Second, the Statement of Rationale enables the distinction between non-essential and essential features to be made in the absence of a higher-level source statement.

Figure 6 suggests a format for a "Statement of Rationale". The **-002** index term is an unique tag assigned to each rationale statement. The index value must be unique among rationale statements within the Product Requirements.

Rationale-002	During product deployment discussions, the need for the product to provide different modes of operation was identified. The product must allow for normal operation, software upgrade, and commissioning.

Figure 6: Statement of rationale example

In Figure 6, suppose that after system analysis the project team has determined the Product Requirements are best satisfied by different modes of operation. If Marketing and the Customer do not explicitly specify that different operating modes are required, a numbered “Statement of Rationale” is formulated for use in tracing the derived requirements related to product operation.

Coverage Matrix

The coverage matrix is a simple, yet powerful tool enabled by the practice of uniquely identifying and tracing requirements. The matrix is a table cross-referencing each requirement against a downstream deliverable. The requirement notation suggested in this paper facilitates generating a coverage matrix because the source for each statement is available the requirement label.

Using the requirement statement identifiers, a coverage matrix can be built to ensure that each requirement is addressed by downstream activities, such as architecture design and test case development, and that all implementation activity is linked to higher-level requirements.

Figure 7 shows a sample coverage matrix that cross-references the commercial requirements provided by Marketing to technical requirements that answer the required functionality. Marketing is able to review the project team’s interpretation of the Product Requirements for correctness because the coverage matrix identifies exactly which technical requirement(s) maps to each commercial requirement.

The real value of a coverage matrix is in identifying "broken traces." Broken traces occur at two levels: when a Product Requirement is not implemented, and when a feature is implemented but not supported by a Product Requirement. A broken trace is not necessarily wrong, but does indicate a project risk that should be investigated.

A Word on Requirement Management Tools

The strength of Requirement Management tools is in organizing and classifying the requirement data. This paper emphasizes a very simple approach to capturing and documenting requirement statements

Appendix A CRS Coverage Matrix

CRS	TRS Coverage					
SIT-CRS-R-001	SIT-TRS-R-997					
SIT-CRS-R-002	Covered by SIT PC TRS					
SIT-CRS-R-003	Covered by SIT PC TRS					
SIT-CRS-R-004	SIT-TRS-R-1105					
SIT-CRS-R-005	SIT-TRS-R-505					
SIT-CRS-R-006	N/A					
SIT-CRS-R-007	SIT-TRS-R-507					
SIT-CRS-R-008	SIT-TRS-R-508					
SIT-CRS-R-009	SIT-TRS-R-1303					
SIT-CRS-R-010	SIT-TRS-R-1109					
SIT-CRS-R-011	N/A (NT becomes Linux)					
SIT-CRS-R-012	SIT-TRS-R-512					
SIT-CRS-R-013	SIT-TRS-R-910	SIT-TRS-R-911	SIT-TRS-R-912			
SIT-CRS-R-014	SIT-TRS-R-4239	SIT-TRS-R-4245				
SIT-CRS-R-015						
SIT-CRS-R-016	Covered by SIT PC TRS					
SIT-CRS-R-017	ADS issue					
SIT-CRS-R-018	SIT-TRS-R-4458					
SIT-CRS-R-019	Covered by SIT PC TRS					
SIT-CRS-R-020	ADS issue					
SIT-CRS-R-021	ADS issue					
SIT-CRS-R-022	SIT-TRS-R-508					
SIT-CRS-R-023	SIT-TRS-R-4001	SIT-TRS-R-4014	SIT-TRS-R-4015			
SIT-CRS-R-024	SIT-TRS-R-4455	SIT-TRS-R-4124	SIT-TRS-R-4456			
SIT-CRS-R-025	SIT-TRS-R-4014					
SIT-CRS-R-026	SIT-TRS-R-528					
SIT-CRS-R-027	SIT-TRS-R-532					
SIT-CRS-R-028	SIT-TRS-R-528					
SIT-CRS-R-029	SIT-TRS-R-1020	SIT-TRS-R-529				
SIT-CRS-R-030	SIT-TRS-R-1020	SIT-TRS-R-530				

Figure 7: Sample coverage matrix

with the goal of developing the project team's ability to write requirements. Once this skill has been mastered it is likely that the project team will begin to identify more attributes to be recorded for each statement. For example, measures of Customer importance and technical difficulty may become essential for prioritizing development. Additionally, a revision history and status for each statement may be needed to track the evolution of Product Requirements over the project.

As the sophistication of the requirements process increases, the need for automation also increases. Requirement management tools really facilitate project teams as the requirement management process matures. However, as stated earlier in this paper, these tools alone do not improve the basic quality of individual requirement statements nor can they verify completeness and consistency within a set of Product Requirements. The ability to write "good" requirements is the cornerstone upon which the process is built. Automation should be addressed only after a requirement management process is established.

More on the Requirement Label

The requirement label format is closely specified and controlled to allows automated "parsing" into a coverage matrix. The requirement label suggested in this paper adheres to the following format:

Label := [Ident] SOURCE: [Ident1], … [IdentN];

where the identifier, **[Ident]**, is a unique tag assigned to an individual requirement statement. The identifier may be as simple as a number but is most useful when it indicates the provenance of the requirement statement. The identifier format used in this paper is:

Ident := [Component]-[Document]-[Type]-[Number],

where **[Component]** uniquely identifies the part of the product being specified. **[Document]** identifies the document in which this requirement statement is located. **[Type]** indicates whether this is a Requirement, Constraint, Assumption or Objective, and **[Number]** is a unique number assigned to this

statement. Together these terms identify the component and document in which the requirement statement can be found.

For example, **SW-SRS-R-328**, where this requirement is found in the Software Requirement Specification (SRS) for the software component of the product.

In the requirement label, "**SOURCE:"** is a reserved word to indicate that the following identifier is the source for the requirement statement. A semicolon **(;)** delineates the end of the requirement label.

All requirement statements must have a source. The project team should be able to identify the reason for every function and characteristic included in the Product Requirements.

In the following example of a complete requirement label, the Requirement ("-R-") stated in the Functional Requirement Specification (FRS) is traced to one source. The source is a Constraint stated in the Commercial Requirement Specification (CRS).

Sys-FRS-R-025 SOURCE: Sys-CRS-C-002;

Requirement statements may come from a variety of places, including email messages, meeting minutes, industry standards and specifications. Regardless of the activity, document or message that inspires a requirement statement, its source must be captured and archived to ease future reference.

Requirement Labeling Rules

Each requirement is identified using a unique and permanent identifier along with the requirement source information. The following rules apply:

1. Every requirement statement is assigned an identifier.
2. Requirement identifiers are unique within a project.
3. Requirements are never re-numbered. Once assigned, an identifier is permanent.
4. Requirement identifiers vacated by deletion are never re-used or re-allocated.

The first rule is self-explanatory. Every requirement statement (Requirement, Constraint, Assumption and Objective) is assigned a fully qualified identifier.

Identifiers must be unique but need not be sequential. As Product Requirements tend to evolve over the duration of the project, one practice is to assign a new base number to requirements added through each iteration. For example, requirements in the first iteration are numbered starting from 500 and requirements in the second iteration are numbered starting from 1000. Using this approach, it is not uncommon to see large gaps between numbers assigned to adjacent requirement statements.

Once assigned to a requirement statement, the identifier must never be changed. The set of Baseline Product Requirements is released iteratively over the life of the project. Each time a new baseline is established the identifiers are used in other deliverables, such as the architectural requirements or system test specifications. Changing a requirement statement identifier midstream causes inconsistency among the project documents and introduces unnecessary risk into the project.

To avoid confusion, an identifier should never be re-used when a requirement statement is deleted from the Baseline Product Requirements. It is easy to imagine a sequence of events where a test case is developed to verify a requirement statement that is later deleted. Suppose the deleted identifier is immediately re-used. An audit of the test cases against the Product Requirements could show that the requirement is covered when, in fact, the test case is not valid for the re-used identifier.

V What Makes a “Good” Requirement?

Before writing any requirements, the project team must agree on the characteristics of a “good” requirement. These characteristics become the Quality Criteria against which each requirement statement is judged. The Quality Criteria can become quite sophisticated, but for a project team that is relatively inexperienced with developing Product Requirements the following set of "beginner" characteristics form a good starting point [2, 3]. The first four rules apply to individual requirement statements, while the last two apply to the full set of Product Requirements.

1. Requirements Must Be Concise and Singular (not Compound). The requirement statement must include one and only one requirement. It states what must be done in a simple and clear manner. The requirement statement must be easy to read and understand.
2. Requirements Must Be "Implementation Free". The requirement must state only what is required without mandating the design or implementation. A common problem is requirement statements that detail the implementation aspects while neglecting to state the actual function or non-functional attribute being required. A requirement statement should answer the question “What?” – not “How?”
3. Requirements Must Be Unambiguous. The requirement must have only one interpretation. This criterion is often difficult to satisfy and the "requirement level" (described in the next section) must be taken into consideration. Avoid ambiguous words and phrases like "support," "adequate," "fault tolerant,” "user friendly” and "as much as possible".
4. Requirements Must Be Verifiable. The requirement must be quantified in a manner that can be verified by objective analysis or test. When requirements cannot be tested, the test team can offer only a subjective opinion regarding how well the final product fits the requirements.
5. Requirements Must Be Complete. The set of requirements must completely define the functionality and characteristics of the product. When requirements are not complete, the final product will lack some functionality that will likely limit its success.
6. Requirements Must Be Consistent. The set of requirements must be consistent with each other and must not contradict requirements in the source documents.

When the Quality Criteria are agreed, each requirement statement should be reviewed using these criteria as the basis for judging its "goodness". The set of Quality Criteria enumerated above represent a beginner set. In practice, the actual rules applied will vary according to the criteria agreed by the project team.

Requirements Leveling

The appropriate “level” for each requirement should be taken into consideration when applying the Quality Criteria. Requirement statements provided by the Customer will be ambiguous to a design engineer, however, some lack of detail is appropriate at the Customer's level.

Alternatively, at the “System Requirements” level the product is viewed as a black box. Its behavioral response to external triggers is described in terms of what must happen and the properties and qualities that it must demonstrate are also described in terms of targets that must be met (such as 99% reliability).

Through step-wise refinement, the Customer's requirements are translated and expanded by downstream activities. At each step, ambiguity is progressively eliminated.

How Do You Ensure "Good" Requirements?

This paper, so far, has focused on developing and documenting good individual requirement statements assuming these will be consolidated into a document. As the requirement specification is assembled, individual requirement statements should be reviewed against the Quality Criteria. No requirement statement is included in the specification until it satisfies all of the Quality Criteria agreed by the project team.

The Quality Criteria can be viewed as a gate that each requirement statement must pass before being entered into the specification. Suzanne and James Robertson refer to this filter as the "Quality Gateway" [5]. Since requirements are written using natural language, the Gateway is best implemented by people.

Two or three people should be nominated from the project team as responsible for applying the Quality Criteria to each requirement statement. At a minimum, the owner of the specification should be tasked with this responsibility, but he would then be in the position of reviewing his own requirements.

When an independent test team exists, a representative from the test team is a necessary member of the Gateway. The test team is responsible for developing test specifications used to verify that the product delivered by the project team meets the set of Baseline Product Requirements. To develop the test specifications, the test team must have "good" requirements on which to base their work. Consequently, the test team has a stake in ensuring that the requirement statements satisfy the Quality Criteria.

Once the Quality Gateway is in place, some simple guidelines need to be established.

- The filtering process should be relatively informal.
- Requirement statements should be submitted as soon as they are developed in order to avoid "big bang" integration of the specification, where all the statements are reviewed at the same time.
- If more than one person is responsible for filtering the requirement statements, at least one of them should approve every requirement statement.
- Only individual requirement statements should be considered. Completeness and consistency are not examined until the entire set of Product Requirements is assembled.

A requirement statement is rejected when it is judged "not good enough" according to the Quality Criteria agreed by the project team. The Quality Gateway is not responsible for fixing the requirement. The original submitter is responsible for improving its quality and re-submitting the statement for inclusion in the specification. Requirement statements rejected by the Quality Gateway should be fixed - not permanently excluded.

Review and Change Control

After the Product Requirements are assembled, the project team and other stakeholders review the specification to ensure the product is completely defined and the requirements are self-consistent. The

specification review should focus on finding gaps in the product definition, duplicate requirements and contradictory requirements. The set of Product Requirements must be reviewed with the Product Scope (its purpose and success criteria) in mind. The question to be answered is: "if we build something that meets these requirements, will it satisfy the Product Scope defined for the product?"

When the project team and other stakeholders have agreed that the requirements satisfy the Product Scope and the Quality Criteria, the requirement specification is approved and the Baseline Product Requirements are established. The next challenge is to defend this baseline against uncontrolled changes; a task simplified by the existence of an approved requirement specification. By definition, if the requirement is not present in the agreed Baseline Product Requirements then it represents a change. Lacking a baseline, changes happen without notice.

As mentioned earlier, changes are necessary to ensure that the final product is viable. The Customer's understanding of what the product should be is refined as the project progresses. The project team discovers technical issues that require Customer expectations to be revised. Change should not be avoided – it should be controlled. A proven mechanism for controlling change to the Baseline Product Requirements is to organize a Change Control Board.

The Change Control Board is responsible for analyzing requested changes to the set of Baseline Product Requirements. Authorized changes are submitted to the Quality Gateway, which still guards the entry to the specification. Ensuring unauthorized changes to the Baseline Product Requirements do not occur requires discipline by the project team and vigilance by the project manager. The project team should not act upon changes to the Baseline Product Requirements until they are incorporated into the specification.

VI Conclusion

The project team's goal is to develop a product that satisfies the Customer. The Customer's expectations are communicated to the development team through an agreed set of Product Requirements. To capture these requirements, the project team must develop the basic skills of writing requirement statements. Developing requirements is still hard work but the effort can be made a little easier if project teams do three things:

1. Establish Quality Criteria for what constitutes a "good" requirement statement, and
2. Establish a requirement notation that enforces the Quality Criteria, and
3. Commit to writing requirements.

During the requirements phase of a project, the project team should spend several intensive sessions capturing, deriving and reviewing requirement statements against the Quality Criteria. Because the initial Quality Criteria are not too sophisticated, the team internalizes them quickly. The result is that, through practice during group review meetings and a common understanding of what makes a "good" requirement, the actual task of writing requirements becomes easier.

The project team limits the requirements gathering activity according to the boundaries set by the Product Scope. A coverage matrix demonstrates that all commercial requirements were taken into account, which boosts the confidence level of Marketing. Further, the productivity of the implementation team is enhanced because the Product Requirements focus their activities.

The approach outlined in this paper is simple. The ideas for bootstrapping the requirements gathering process give the project team an effective foundation on which to build the Product

Requirements. The suggested notation and rules focus the project team on developing requirement-writing skills needed before tackling the issue of automation. As long as requirements are written using natural language, a tool cannot help to develop this skill. It takes dedication by the project team to develop the ability to write requirements.

References

[1] Gause, D. C. and Weinberg, G. M., *Exploring Requirements: Quality Before Design,* Dorset House Publishing, New York, 1989.

[2] Hooks, I., "Writing Good Requirements", Proceedings of the Third International Symposium of the INCOSE, Volume 2, 1993.

[3] Kar, P. and Bailey, M., "Characteristics of Good Requirements", Presented at Sixth International Symposium of the INCOSE Symposium, 1996.

[4] McConnell, S., *Software Project Survival Guide,* Microsoft Press, Redmond, Washington, 1998.

[5] Robertson, J. and Robertson, S., *Mastering the Requirements Process,* Addison-Wesley, Harlow, England, 1999.

[6] Weigers, K E., *Software Requirements,* Microsoft Press, Redmond, Washington, 1999.

[7] Zowghi, D. and Nurmuliani, N., "A Study of the Impact of Requirements Volatility on Software Project Performance", Proceedings of the Ninth Asia-Pacific Software Engineering Conference, 2002.

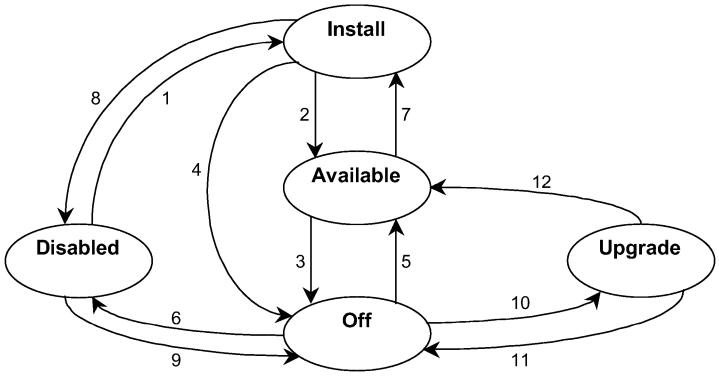
Appendix

This appendix includes a number of sample requirement statements to demonstrate how the ideas presented in this paper have been applied in real product development. The requirement statements shown in Figure 8 are traced to a Statement of Rationale because, as an example, neither Marketing nor the Customer considered how the product would be deployed when the commercial requirements were generated.

The statement of rationale used for this example is:

Rationale-002: *During product deployment discussions, the need for the product to provide different modes of operation was identified. The product must allow for normal operations, software upgrade, and commissioning.*

In Figure 8, proper coordination of the end-to-end system requires that a particular state transition

SIT-TRS-R-3001 **SOURCE: Rationale-002;**	The Operating Modes shall be related as follows: 
•	

SIT-TRS-R-3002 **SOURCE: Rationale-002,** **SIT-TRS-R-3001;**	Tr1: The SIT shall transition from "Disabled" to "Install" when the Installer/Service user logs in to the product.
•	

SIT-TRS-R-3003 **SOURCE: Rationale-002,** **SIT-TRS-R-3001;**	Tr2: The SIT shall transition from "Install" to "Available" when the Installer/Service user confirms that the I&C procedure is complete.
• *Note 1: This is relevant if the Installer/Service user logs in only for parameter reading and performs no configuration change. A configuration change will require reboot of the SIT.*	

SIT-TRS-R-4266 **SOURCE: Rationale-002;**	In INSTALL mode, the SIT shall not allow a reboot or shutdown to be performed via the Web pages.
• *The Installer must log out of the SIT. In order to reboot/shutdown the SIT, it must be in DISABLED mode.*	

SIT-TRS-R-4269 **SOURCE: Rationale-002;**	In AVAILABLE mode, the SIT shall act as a router between the Hub and the LAN.
• *This refers to normal Operations State Machine and processes described in [SDS].*	

Figure 8: State machine example

diagram be implemented. This example is included in order to demonstrate how a state transition diagram can be translated into functional requirements. The diagram is captured in the normative text of the requirement statement, **SIT-TRS-R-3001**, with each state named and each transition numbered.

The requirement statements following the state transition diagram (**R-3002** and **R-3003**) describe the conditions under which each transition occurs. These transitions are numbered according to the state transition the diagram is included as part of the informative text. Similarly, for each state (**R-4266** and **R-4269**), requirement statements are included specifying what the product must do.

Figure 9 demonstrates the tradeoff between rules and practicality when developing requirement statements. According to the Quality Criteria presented earlier, compound requirement statements are not acceptable. However, it is not always practical to divide a compound requirement into its component pieces.

SIT-TRS-R-3004 **SOURCE: Rationale-002,** **SIT-TRS-R-3001;**	Tr3: The SIT shall transition from "Available" to "Off" in either one of the following situations: a) when a shutdown or reboot is initiated by the Super User, or b) when powered off without respecting the proper shutdown procedure.
•	

Figure 9: Rules vs pragmatism

Keeping in mind that one of the goals is testability; the following requirement statements suggest how to capture compound requirements while still facilitating the development of test cases. A letter delineates each part of the requirement. Mutually exclusive parts are indicated by “or” while inclusive parts are indicated using "and". For downstream activities, these requirements are individually referenced as **SIT-TRS-R-3004a**, **R-3004b** and **R-3004c**.

Estimating and Managing Project Scope for Maintenance and Reuse Projects

Estimating project scope is considered by many to be the most difficult part of software estimation. Parametric models have been shown to give accurate estimates of cost and duration given accurate inputs of the project scope, but how do you input scope early in the lifecycle when the requirements are still vaguely understood? How can scope be estimated, quantified, and documented in a manner that is understandable to management, end users, and estimating tools? This talk focuses on scope estimates for maintenance and reuse work, including bug fixes (corrective maintenance), modifications to support changes in the operating system, database management system, compiler or other aspect of the operating environment (adaptive maintenance) and modifications of existing functionality to improve that functionality (perfective maintenance). Reuse includes any case where you are modifying an existing code base to support enhanced functionality, and includes cases where an existing application is translated to a new language.

At a high level, maintenance projects consist of three types of work:

1. Maintaining an existing, functioning application;

2. Modifying existing code to support changing requirements; and

3. Adding new functionality to an existing application.

A team doing a new build for an existing application would only be concerned with items 2 and 3. A team keeping an existing code base functioning would only do item 1. A project manager may be responsible for both areas and might need to estimate the effort required for all three. We'll deal with each individually.

Maintaining Existing Code

Maintenance as we're defining it consists of three types of activities:

- Corrective maintenance: Fixing bugs in the code and documentation. Bugs are areas where the code does not operate in accordance with the requirements used when it was built.

- Adaptive maintenance: Modifying the application to continue functioning after installation of an upgrade to the underlying virtual machine (DBMS, operating system, etc.); and

- Perfective maintenance: Correcting serious flaws in the way it achieves requirements (e.g., performance problems).

Maintenance effort is a function of the development effort spent on the original project. The larger the original project in terms of effort, the more staff must be assigned to maintain the application. A second factor is Annual Change Traffic (ACT), or the percent of the code base that will be touched each year as a result of maintenance work. Numbers for ACT between 3% and 20% are reasonable, with 12% to 15% being fairly typical. Annual maintenance effort while the application is in a maintenance steady-state will equal ACT * PM where PM is the original person months of development effort. Maintenance steady-state is typically achieved in year 4 following software delivery. During software development, staffing follows a Rayleigh curve (Figure 1), reaching a peak at the time of software completion. Between software completion and steady-state, the maintenance effort continues to follow the back slope of the Rayleigh curve starting with 1.5 * ACT * PM in year one and dropping down to the steady state value. The value of 1.5 is based on empirical analysis of maintenance project effort and defect removal rates. Software maintained beyond 9 years typically sees maintenance costs begin to climb again, due to a combination of increasingly fragile code and an increasing distance between the technology used for development and the current state-of-the-art.

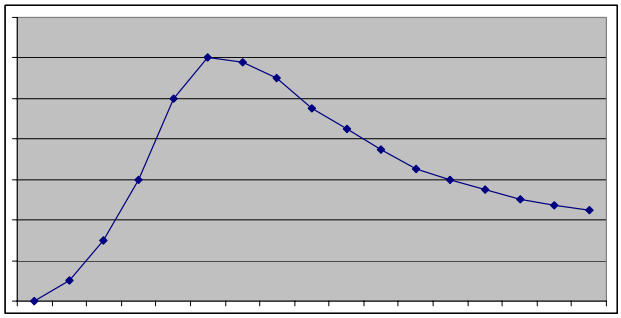


Figure 1: Staffing level during development and maintenance

If you know the effort spent on the original development, the above equations may be used as shown. If you do not know the original development effort, you must first estimate that effort, normally using a commercial estimation tool. This will require that you count the physical lines of code (or function points), and then either make educated guesses at values for environmental variables (team capability and so on) or use the tool's default values.

Modifying Existing Code

The basis of code modification is very simple: code already exists that may be utilized in any given project. You begin by actually counting the values measured in the existing application(s). For example, if using Lines-of-Code, employ a code counting utility to physically count the lines of code in the existing program modules (or use the values from your configuration management system with an adjustment to remove the impact of blank lines and comments). If using Function Points, count the existing reports, screens, tables, and so on.

Calculating the Equivalent Volume

Our goal is to convert from the known value for the volume of reusable code to an equivalent volume of new code. Think about it this way.

- If we have 100 function points worth of reusable code but the reusable code is worth nothing to us, then no effort will be saved, the equivalent amount of new code is 100 function points.
- If we have 100 function points worth of reusable code and we can reuse it without any changes, re-testing, or integration whatsoever, then using the code is a "freebie" from a developmental perspective. The equivalent amount of new code is 0 function points.
- If we have 100 function points worth of reusable code and this saves us half the effort relative to new code, then the equivalent amount of new code is 50 function points.

We convert from reused volume values to equivalent new volume values by looking at three factors: Percent Design Modification, Percent Code Modification, Percent Integration and Testing. After we have shown how this process works, we will describe how to adjust the numbers further based on three additional factors: Assessment and Assimilation, Software Understanding, and Unfamiliarity with Software.

- **Percent Design Modification** measures how much design effort the reused code will require. Basically, a low percent value indicates high code reuse, whereas a high percent value indicates low code reuse and increases the requirement to develop new code:
  - ✓ A value of 0% says that the reused code is perfectly designed for the new application and no design time will be required at all.
  - ✓ A value of 100% says that the design is totally wrong and the existing design won't save any time at all.
  - ✓ A value of 50% says that the design will require some changes and that the effort involved in making these changes is 50% of the effort of doing the design from scratch.

  For typical software reuse, the Percent Design Modification will vary from 10% to 25%.
- **Percent Code Modification** measures how much we will need to change the physical source code:
  - ✓ A value of 0% says that the reused code is perfect for the new application and the source code can be used without change.
  - ✓ If the reused code was developed in a different language and you need to port the code to your current language, the value would be 100%.
  - ✓ Numbers in between imply varying amounts of code reuse.

  The Percent Code Modification should always be at or higher than the Percent Design Modification. As a rule of thumb, we have found the Percent Code Modification is often twice the Percent Design Modification.

- **Percent Integration and Testing** measures how much integration and testing effort the reused code will require:

  ✓ A value of 0% would mean that you do not anticipate any integration or integration test effort at all.

  ✓ A value of 100% says that you plan to spend just as much time integrating and testing the code that you would if it was developed new as part of this project.

  The Percent Integration and Test should always be at or higher than the Percent Code Modification. It is recommended that you set the Percent Integration and Test to at least twice the Percent Code Modification.

  It is not unusual for this factor to be 100%, especially for mission critical systems where the risk of failure is significant. For commercial off-the-shelf components (purchased libraries) where the Percent Design Modification and Percent Code Modification are often zero, it is not unusual to see a number of 50% here to allow for the integration effort and time spent testing the application with the commercial component.

Finally, after measuring your existing code's volume and estimating your Percent Design Modification (DM), Percent Code Modification (CM), and Percent Integration and Test (I&T), calculate the equivalent volume of new code (EV) as follows:

$$EV = AAF \times ReusedVolume$$
$$= (0.4 \times DM) + (0.3 \times CM) + (0.3 \times I\&T) \times ReusedVolume$$

Equation 1: Equivalent Volume

The values .4, .3, and .3, which sum to 1.0, are based on work by Barry Boehm at USC, as documented in his books *Software Economics* and *Software Cost Estimation with COCOMO II.*

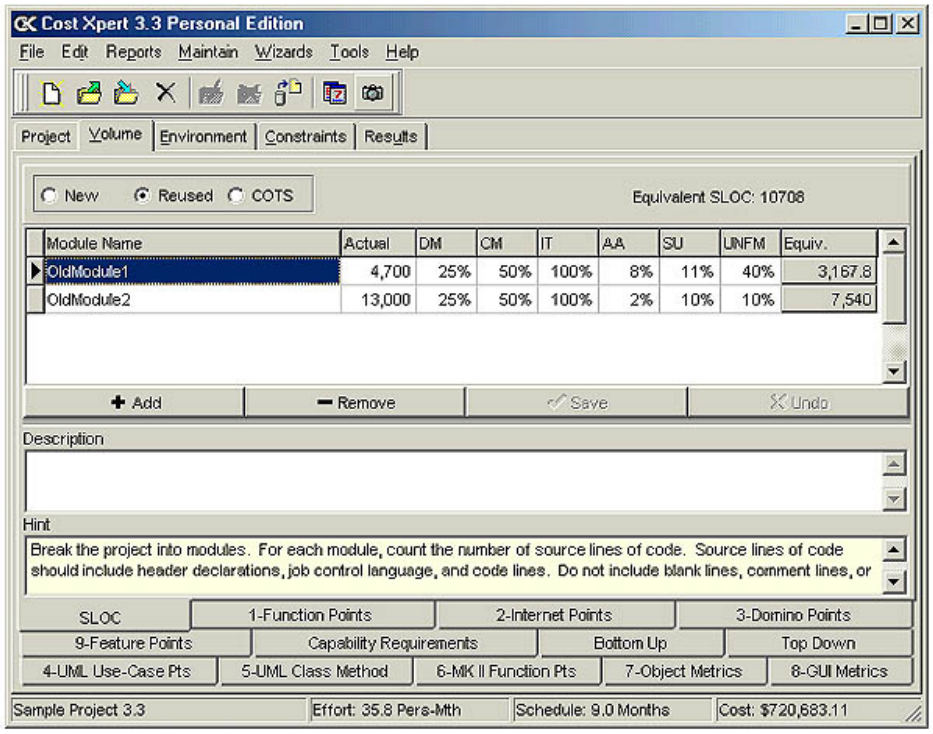


Figure 1: Example of Commercial Tool Reuse Parameter Screen

Suppose that we need to implement a new e-commerce system consisting of 15,000 source lines of code (SLOC). Let's ignore environmental adjustments for the moment. Using a standard estimation formula and plugging in a sample value of 3.08 for productivity and 1.030 for Penalty (which are beyond the scope of this article), the predicted effort will be:

$$\begin{aligned} Effort &= Productivity \times KSLOC^{Penalty} \\ &= 3.08 \times 15^{1.030} \\ &= 3.08 \times 16.27 \\ &= 50\ Person\ Months \end{aligned}$$

Now, suppose that we identify a similar application we can purchase with source code. We now have 15,000 Source Lines of Code (SLOC) that we want to reuse.
Let's assume that the correct value for design modification is 25%, the correct value for code modification is 50%, and the correct value for integration and test is 100%. What would be the new effort required?

First, we calculate the equivalent volume in source lines of code, or ESLOC. This is computed as follows:

$$\begin{aligned} ESLOC &= AAF \times 15{,}000 \\ &= [\,(0.4 \times 0.25) + (0.3 \times 0.5) + (0.3 \times 1.0)\,] \times 15{,}000 \\ &= 0.55 \times 15{,}000 \\ &= 8{,}250 \end{aligned}$$

and the new calculated effort is now:

$$\begin{aligned} Effort &= Productivity \times KSLOC^{Penalty} \\ &= 3.08 \times 8.25^{1.030} \\ &= 3.08 \times 8.79 \\ &= 27.1 \text{ Person Months} \end{aligned}$$

There are three additional factors that need to be considered to complete the reuse picture: Assessment and Assimilation (AA), Software Understanding (SU), and Unfamiliarity.

- Assessment and Assimilation indicates how much time and effort will be involved in testing, evaluating and documenting the screens and other parts of the program to see what can be reused. Values range from 0% to 8%.
- Software Understanding estimates how difficult it will be to understand the code once you are modifying it, and how conducive the software is to being understood. Is the code well-structured? Is there good correlation between the program and application? Is the code well-commented? A numeric entry between 10% and 50%, default 30%.
- Unfamiliarity with Software indicates how much your team has worked with this reusable code before. Is this their first exposure to it, or is it very familiar? The range of possible values is between 0 and 100%, default 40%.

These three factors add a form of tax to software reuse, compensating for the overhead effort associated with reusing code.

For projects where the amount of reuse is small (AAF is less than or equal to 50%), the following formula applies with adjustments per the above factors:

$$ESLOC = ASLOC \times [\, AA + AAF\,(1 + 2 \times SU \times UNFM)\,]$$

Let's take our earlier example involving 15,000 reused SLOC. Suppose we found that we could get by with 10% design changes, 20% code changes, and 40% integration and test effort. AAF would then be calculated as:

$$AAF = (0.4 \times 0.1) + (0.3 \times 0.2) + (0.3 \times 0.4) = 0.22$$

Because AAF is less than or equal to 50% we can use the formula just presented. Now, suppose that AA was 4%, SU was 30%, and UNFM was 40%.

The equivalent source lines of code, or ESLOC would now be:

$$ESLOC = 15000\,[0.04 + 0.22\,(1 + 2 \times 0.3 \times 0.4)] = 4692$$

Using our earlier assumptions, the effort required to build this software would be:

$$Effort = Productivity \times KSLOC^{Penalty}$$
$$= 3.08 \times 4.692^{1.030}$$
$$= 3.08 \times 4.915$$
$$= 15.14 \text{ Person Months}$$

The formula when reuse is low and AAF is greater than 50% changes. The formula in this situation is:

$$ESLOC = ASLOC \times [AA + AAF + (SU \times UNFM)]$$

Let's work through our same example of 15,000 lines of reused code, but let's suppose that the design modification was 50%, the code modification 100%, the integration and test was 100%, and the correct values for AA, SU, and UNFM were 8%, 50%, and 100% respectively.

AAF is now calculated as:

$$AAF = (0.4 \times 0.5) + (0.3 \times 1.0) + (0.3 \times 1.0) = 0.8$$

Because AAF is over 50%, we use the second formula as follows:

$$ESLOC = 15000 \times [0.08 + 0.8 + 0.5 \times 1.0]$$
$$= 15,000 \times 1.38$$
$$= 20700$$

Effort is now calculated as:

$$Effort = Productivity \times KSLOC^{Penalty}$$
$$= 3.08 \times 20.7^{1.030}$$
$$= 3.08 \times 22.67$$
$$= 69.8 \text{ person Months}$$

In this case, reusing those 15,000 lines of code actually takes us 23 person months more effort than writing the same code from scratch! Remember, the reused code is not initially optimized for the target application. This matches our experience on a wide range of projects where reuse was attempted but the code required too many changes for that reuse to be efficient. In fact, this phenomenon is even more pronounced than shown in the example above. If you need 15,000 lines of new functionality you will seldom fine a reusable block of code that exactly matches the functionality you are looking for. More often, the reused code will be significantly larger than the new code because it will do many functions that you are not interested in. Perhaps you will be reusing a piece of code that is 25,000 lines of code in size, all to get at those 15,000 lines of code worth of functionality that you care about. Well, the entire 25,000 lines of code will typically need to be assessed, understood, and tested to some degree. The end result is that in general, you will find that somewhere between 15% and 30% design change is the crossing point

beyond which you are typically better off rewriting the code from scratch. The correct value in this range will depend largely on how well matched the reused code is to your requirements and the quality of that code and documentation.

If you are doing an on-going series of maintenance builds with a large, relatively stable application there are some tricks to simplify your planning. Create a spreadsheet containing all of the modules and for each module, the lines of code in that module. Set percent design modification, code modification, and so on to zero for each module in the spreadsheet. It is also useful in the spreadsheet to include an area where you identify the dependent relationships between modules (or this can sometimes be done using a tool like Microsoft Project, where you treat each module as a task in the dependency diagram). Save this as your master template for planning a new build. When you are planning a build, analyze each requirement for change to identify the modules that must be modified and fill in the appropriate value for DM, CM, etc. Then, look at the modules that are dependent on these modules and put in an appropriate value for Integration and Test for those dependent modules. You can then quickly calculate the resultant equivalent scope and use this to calculate a schedule and effort required. The next build, you go back to the template you started with and repeat the process. Some commercial estimating tools support this approach as well.

Adding New Functionality

Finally, when preparing a new software build there are normally some areas where completely new functionality is added to the system. This functionality is defined and estimated as new development using the standard approaches suitable for estimating new software development.

The work in this paper is heavily dependent on work by Barry Boehm et al, as documented in his latest book, Software Cost Estimation with Cocomo II. Boehm, Barry et al. Prentice Hall, New Jersey: 2000.

SOFTWARE PROCESS IMPROVEMENT FRAMEWORKS PERCEIVED IMPACT ON TQM PRACTICES AND FINANCIAL PERFORMANCE OF SOFTWARE ORGANIZATIONS

Arthur Oster

INTRODUCTION

Software organizations in today's business environment face many challenges. Key among them are demand for higher customer satisfaction, reduced development cycle times, improved cost efficiencies, technology advances, and error-free software products. Each is essential for organizational performance and success. Meeting or exceeding these challenges creates strategic advantages and builds competitive strength. As software organizations have matured, software quality improvement has become a more prominent method to improve the organization and overcome many challenges. A significant element of software quality involves improving the software organization to promote the success of the entire organization.

Software Process Improvement Frameworks

The introduction of Total Quality Management (TQM) during the 1980s was a significant turning point for the quality movement. TQM is the most widely applied quality improvement philosophy for organizations worldwide (Tenner and DeToro, 1992). It applies an organizational focus on satisfying customer needs for a wide range of products and services (Kordupleski, Rust, and Zahorik, 1992). The TQM philosophy's strength is its wide scope, addressing most areas in an organization, including software development. A standard methodology was needed for the software industry to evaluate the level of software quality practices.

Several approaches, commonly referred to as Software Process Improvement (SPI) frameworks that apply the TQM philosophy to software organizations have been developed. Each SPI framework has defined activities to improve software organization performance. By improving the long-term development process, not just the product, SPI frameworks define specific ratings and certifications recognized in the industry. The most widely used frameworks are the International Organization for Standardization's ISO 9001 standard for quality management systems, the Software Engineering Institute's Capability Maturity Model for Software (referred to as the CMM or SW-CMM), and the ISO/IEC 15504 standard for software process assessment (Hunter & Thayer, 2001).

SPI Benefit Questions

Questions remain regarding the benefits of adopting SPI frameworks. The SEI claims organizations investing in the CMM will realize a substantial return on their investment, projects will complete on time or ahead of schedule, and total project

schedule time will decrease (Paulk, Curtis, Chrissis, & Weber, 1993). Several case studies in the literature support these claims. Since a small percentage (estimates range from under 10% to 20%) of software organizations have invested in SPI frameworks, the value of SPI frameworks has no overwhelming across-the-board acknowledgment (Paulk, 1998; Fayad & Laitenen, 1997; Saiedian & Kusara, 1995; Sheard, 1997; Yang, 2001). A major reason for not investing in a SPI framework probably is the substantial up-front financial requirement, which may be hundreds of thousands to even millions of dollars. Most organizations evaluate their return on investment as short term (less than five years). Initial results indicate there are significant long term benefits from investment in a SPI framework. Due to the initial investment cost of a SPI framework, an organization may not realize their return on investment for several years (Saiedian & Kusara, 1995).

Some software organizations that have not invested in a SPI framework do produce high quality products. This suggests most software organizations apply some quality methods and practices regardless of their formal investment in a SPI framework. Compliance with standard practices does not guarantee superior software (Strassman, 1997). Unlike manufacturing, software development requires a high degree of creativity based on human interaction categorized in procedures. Software process improvement needs to balance the creative component with process discipline for a software organization to operate in an effective and efficient manner (Conradi & Fugetta, 2002).

METHODOLOGY

An Empirical Exploratory Case Study

Oster (2004) explored perceived advantages derived from an investment in a software process improvement framework. Specific advantages measured were perceived Total Quality Management (TQM) practices and perceived financial performance. To control for the impact of dissimilar organizational cultural influences, 310 software organization managers and leaders within one parent major Fortune 500 company were invited to complete a survey via the web (see Survey in Appendix I). The parent company employs over 130,000 people worldwide. The response rate was 46.7 percent with a total of 141 usable responses.

The independent variable was “investment in a SPI framework”, measured on a discrete scale as investment/no investment during the last three years. The dependent variables were “perceived financial performance” and “perceived quality management practices”. “Perceived financial performance” was based on three items: change in revenue, change in earnings, and change in productivity compared to three years prior (based on Douglas & Judge, 2001). “Perceived quality management practices” was based on a composite survey drawing from other TQM surveys (Parzinger & Nath, 1998; Saraph, Benson, & Schroeder, 1989; Carr, Mak, & Needleham, 1997; Black & Porter, 1996; Flynn, Schroeder, & Sakakibara, 1994; Douglas & Judge, 2001). A total of 37 items were grouped into nine factors identifying the major constructs of TQM

practices within software organizations. An aggregate TQM index was computed from the nine factors of the “Perceived quality management practices”. They were:

1. Executive commitment
   The extent of top management support for the quality program. Top management provides funding for the quality program, establishes goals incorporating quality principles, and communicates the quality philosophy to the organization.

2. Employee empowerment
   The involvement and participation of employees in the quality program. Executives should establish policies that encourage employee participation in the quality program.

3. Quality philosophy
   The organization should have policies that establish a high priority for quality enhancements. The quality philosophy should place a high degree of importance in the organization’s mission statement and strategic plan for quality improvements.

4. Quality measures
   In order to evaluate the return on investment, the quality program should have measures that relate quality efficiencies to cost reduction. Programs that support reduction in defects and minimization of rework are an integral part of the TQM philosophy.

5. Role of the quality department
   The quality department within the organization must be able to perform effectively in order for the quality program to succeed.

6. Quality training
   In order to promote the organizational commitment to quality improvement, the staff needs to be trained in quality principles, tools, and techniques. Quality training should be part of the employee career development plans.

7. Closer to customers
   The customer should be involved in developing project requirements and participate in development of the software product.

8. Process evaluation
   Process evaluation incorporates methods to improve the software development processes. Evaluation of quality improvements applies the use of benchmarking programs, the investigation of quality practices at other organizations, and the application of self-assessments to measure quality improvements.

9. Process improvement
   This factor is associated with those activities that improve product efficiencies, yet still increase the quality of the software product. Reduction in defects, process efficiencies, and repeatable processes are practices that improve the quality of processes.

Demographic characteristics of organization size and number of employees in software development were collected. The instrument was determined to have face validity and content validity based on a review by 23 leaders from the parent organization's quality departments. Next, the results of a pilot survey completed by 38 leaders of software organizations within the case corporation were used to evaluate the survey for discriminate validity, convergent validity, construct validity, and reliability. Survey internal consistency reliability was determined with Cronbach alpha for each of the nine factors. Values ranged from .82 to .94.

The following summarizes the profile of survey respondents. For details refer to Table 1 in Appendix II Data Analysis Tables. A total of 141 usable surveys were returned. The majority of respondents, 55.3%, reported SPI Framework investment of 1–4%, while 9.2% reported investment of 5–8% and 5.7% reported investment of 9% or more. Less than 30% of respondents reported no SPI Framework investment. Overall, almost 70 % of respondents' companies had some level of investment in SPI Frameworks.

Of the respondents, the majority-45.4% were at CMM L1 (or equivalent), while 16.3% were CMM L2, 15.6% were CMM L3, and 2.8% were CMM L4 - L5. The majority (38.3%) of respondents were from small companies with 0-125 employees. A total of 22.7% were from companies with 125-250, 20.6% from companies with 250-500, and a total of 18.5% from companies with 500-2500 employees. Geographic representation included 58.9% for the U.S. & Canada, 25.5% from Europe with the remaining 15.6% from other areas.

More than 73 % of the respondents reported a change for each of the three indicators of Perceived Financial Performance. Individual indicators summaries were 92.2% reported Change in Relief or Revenue, 87.2% reported Change in Earnings, and 73.8% reported Change in Productivity. Positive results were reported for each of the TQM Practices. See discussion of the Hypothesis tests and Tables 2 through 7 for details.

RESULTS

Analyses

Data analyses used two-sample t-tests to compare the perceived levels of quality management practices and perceived financial performance for the group of participants reporting an investment in a SPI framework compared with those reporting little or no SPI investment. In the study, two hypotheses were developed and tested. Refer to the t-test methodology and data analysis tables in Appendix II for the detailed results.

Hypothesis 1

$H1_N$: There is no significant difference in the perceived quality management practices between software organizations that have invested in a software process framework and software organizations that have not invested in a software process framework.

$H1_A$: There is a significant difference in the perceived quality management practices between software organizations that have invested in a software process framework and software organizations that have not invested in a software process framework.

Hypothesis 1 compared the level of perceived quality management practices for software organizations that had invested in a SPI framework with software organizations that had not invested in a SPI framework. Three t-tests were performed to analyze this hypothesis. In the first, group 1 consisted of software organizations that had no investment in a SPI framework. Group 2 consisted of software organizations that had at least 1% of their budget invested in a SPI framework. The results indicate that the difference is statistically significant. Companies investing in and SPI Framework can expect to realize increased benefits for all quality management practices collectively and individually for each except for the “closer to customer” factor. (See Appendix II, Table 2 for details.)

The second t-test separated the groups into software organizations that invested less than 5% in a SPI framework and those investing 5% or more. The companies with below 5% investment in group 1 probably included software organizations which had just recently invested in a SPI framework or had invested marginally in a SPI framework. These results were the same as for the first t-test. Companies investing at least 5 % in a SPI Framework can expect to realize increased benefits for all quality management practices collectively and individually for each except for the “closer to customer” factor. (See Appendix II, Table 3 for details.)

The third t-test analyzed the resultant level of perceived quality management practices based upon the CMM maturity level equivalents. This t-test compared the organizations with a level of 1 with organizations with a level above CMM 1 (or having

achieved ISO 9001 certification). An organization with CMM level 1 is assumed to include organizations that have no investment in a SPI framework and organizations that have recently begun investment in a SPI framework. These results were the same as for the first two t-tests. Companies with CMM Level 2 or above or ISO 9001 certification can expect to realize increased benefits for all quality management practices collectively and individually, including the "closer to customer" factor. (See Appendix II, Table 4 for details.)

Hypothesis 2

$H2_N$: There is no significant difference in the perceived financial performance of software organizations that have invested in a software process framework and software organizations that have not invested in a software process framework.

$H2_A$: There is a significant difference in the perceived financial performance of software organizations that have invested in a software process framework and software organizations that have not invested in a software process framework.

Hypothesis 2 examined the perceived financial performance items for software organizations based upon the same independent variable respondent groupings used in hypothesis 1, the extent of investment in a SPI framework. For each of the groupings, t-tests were performed for the measures of the perceived financial performance variable; namely, change in revenue (also referred to as relief), the change in earnings, and the change in productivity compared to three years prior. All the results for hypothesis 2 failed to support a significant difference for the groups tested (see Appendix II, Tables 5, 6, and 7).

Conclusions and Recommendations

The results support a conclusion that software organizations investing in a SPI framework, such as the CMM or ISO 9001, can expect to realize organizational TQM practices benefits. Even though some of the literature indicated that the CMM may focus primarily on improvement of software processes and not address other areas of TQM (Ravichandran & Rai, 1999), this study found aggregate benefits in organizational TQM practices.

The results of the "closer to customer" TQM factor in the first two t-tests of hypothesis 1 require further explanation. The primary concern for all organizations must be customer satisfaction. The results indicated that customer involvement and customer satisfaction probably is a major policy for most organizations, regardless of their investment in a SPI framework. However, there is a significant difference in the "closer to customer" factor when organizations achieved a higher maturity level or ISO 9000 certification. This research supports benefits which can be expected from having a requirement for ISO certification or a CMM maturity level higher than 1. All organizational TQM factors, even customer involvement and communication, were significantly improved.

For hypothesis 2, the dependent variable, “perceived financial performance”, was not found to be significantly different. These results suggest the measurement of financial results need to be measured more specifically than was possible in this study. Several reasons might explain this lack of significance. If the study were longitudinal, some changes might become evident with respect to financial performance. Comments from some respondents suggested their perception was that the change, or lack of change, was due to external factors, such as poor market conditions or a lack of new business, and was not related to investment in a SPI framework.

Still another consideration may be the customer’s requirement for ISO certification or a higher CMM maturity level. In the case of a customer requirement for SPI framework investment, the justification for investment is based upon the projected return on investment, not the organization’s current or historical financial results. Instead of perceived financial performance, measures based on actual internal financial data would produce more definitive results. However the proprietary nature of these measures precluded the ability to collect them for this study. Additional quality-based measures, such as change in defect rate, cycle time, and software quality cost, may have increased the robustness of the perceived financial performance variable for those organizations that have invested in a SPI framework. Unfortunately, these measures generally are not used by software organizations that have not invested in a SPI framework.

Investment in and implementation of an SPI framework has been demonstrated to be a viable solution for increasing the level of quality practices in an organization. The infrastructure element of human commitment at all levels of the organization, from top to bottom, is essential for attainment of software improvement objectives. Higher quality practices should increase customer satisfaction, due to less defects and rework in the software. This can be expected to have an impact on the company’s financial performance. On the other hand, new organizational processes, such as reviews and audits, initially may increase the software development overhead costs. Comments from the literature support these findings. Krasner (2001) observed that investment in any improvement requires a financial analysis that can demonstrate the added value realized. Strassmann (1997) advised that without a sophisticated financial analysis, investment in software quality improvement can create serious financial problems for an organization.

Adopting an SPI framework requires a substantial initial investment. Software organizations should justify this investment based on both tangible and intangible criteria, a long-term perspective, and consider using complex or novel measures internally to determine the value-added return on their investment. Using basic financial measures may not be adequate to determine a return on investment in a SPI framework. Krasner (2001) concluded that a standard method to determine return on investment for software process improvement programs did not exist.

This study provides empirical corroboration that investment in a SPI framework can provide a software organization with higher quality software, better software

development processes, and stronger assurance of meeting project schedules. Major short-term trade-offs for investment in a SPI framework may be perceived to be reduced project development costs and a higher degree of flexibility for ad-hoc requests. Investment in a SPI framework is not a requirement for developing high quality software. The achievement of a certification or assessment in a SPI framework provides recognition and a possible competitive advantage. Every software organization has different objectives. If investment in a SPI framework can be justified, this study supports the expectation of benefits for the overall level of all TQM practices.

REFERENCES

Black, S. A. & Porter, L. J. (1996). Identification of the critical factors of TQM. Decision Sciences, 27 (1), 1-21.

Carr, S., Mak, Y. T., Needham, J. E. (1997). Differences in strategy, quality management practices and performance reporting systems between ISO accredited and non-ISO accredited companies. Management Accounting Research, 8 (4), 383-403.

Conradi, R., & Fugetta, A. (2002). Improving software process improvement. IEEE Software, 19 (4), 92-99.

Douglas, T. J., & Judge, W. Q., Jr. (2001). Total quality management implementation and competitive advantage: The role of structural control and exploration. Academy of Management Journal, 44 (1), 158-169.

Fayad, M. E., & Laitenen, M. (1997). Process assessment considered wasteful. Communications of the ACM, 40 (11), 125-128.

Flynn, B.B., Schroeder, R.G., & Sakakibara, S. (1994). A framework for quality management research and an associated measurement instrument. Journal of Operations Management, 11, 339-366.

Hunter, R.B. & Thayer, R.H. (Eds). (2001). Software process improvement. Los Alamitos, CA: IEEE Computer Society.

Kordupleski, R. E., Rust, R. T., & Zahorik, A. J. (1993). Why improving quality doesn't improve quality. California Management Review, 35 (3), 82-95.

Krasner, H. (2001). Accumulating the body of evidence for the payoff of software process improvement – 1997. In Hunter, R.B. & Thayer, R.H. (Eds). Software process improvement. (pp. 519-539). Los Alamitos, CA: IEEE Computer Society.

NCSS (Number Cruncher Statistical Software) 2001. [Computer software]. (2003). Kaysville, UT: NCSS.

Oster, A. (2004). Software process improvement frameworks: Perceived impact on the quality management practices and financial performance of software organizations. Unpublished doctoral dissertation, H. Wayne Huizenga Graduate School of Business and Entrepreneurship Nova Southeastern University, Ft. Lauderdale, Florida.

Parzinger, M. J. & Nath, R. (1998). TQM implementation factors for software development: An empirical study. Software Quality Journal, 7 (3-4), 239-260.

Paulk, M. C. (1998). Forward. In Zahran, S. (1998). Software process improvement: Practical guidelines for business success. Harlow, England: Addison-Wesley.

Paulk, M. C., Curtis, B., Chrissis, M. B., & Weber, C. V. (1993). Capability Maturity Model, version 1.1. IEEE Software, 5 (2), 18-27.

.
Paulk, M. C., Weber, C. V., Curtis, B., & Chrissis, M. B. (1994). The CMM: Guidelines for improving the software process. Reading, MA: Addison-Wesley.

Ravichandran, T & Rai, A. (1999). Total quality management in information systems development: Key constructs and relationships. Journal of Management Information Systems, 16 (3), 119-155.

Saiedian, H., & Kusara, R. (1995). SEI Capability Maturity Model's impact on contractors. Computer, 28 (1), 16-26.

Saraph, J. V., Benson, P. G., & Schroeder, R. G. (1989). An instrument for measuring the critical factors of quality management. Decision Sciences, 20 (4), 810-829.

Sheard, S. A. (1997). The frameworks quagmire. Crosstalk, 10 (9), 17-22.

Strassmann, P. (1997). The squandered computer: Evaluating the business alignment of information technologies. New Canaan, CT: Information Economics Press.

Tenner, A. R., & DeToro, I. J. (1992). Total quality management: Three steps to continuous improvement. Reading, MA: Addison-Wesley.

Yang, Y. H. (2001). Software quality management and ISO 9000 implementation. Industrial Management and Data Systems, 101 (7). 329-338.

Zahran, S. (1998). Software process improvement: Practical guidelines for business success. Harlow, England: Addison-Wesley.

APPENDIX I.

SURVEY INSTRUMENT

SURVEY OF QUALITY PROGRAMS IN SOFTWARE ORGANIZATIONS

This survey is a study of quality improvement programs in software organizations. The questions below address quality issues within your organization and, more specifically, in the software development field. All questions refer to the software department within your area of responsibility. If you do not have the statistical information readily available to answer a question, respond with an estimated value.

Please answer all the questions and submit this survey using the "SUBMIT" button at the end of this survey.

SECTION I

Questions in Section I request certain measures within your organization.

1. How much investment has your software organization made in a Quality Improvement Program?
   This investment should be for a program, and should not include general quality improvements.
   Answer based upon percent of total budget (from both external and internal sources) during the last 3 years. (Include any investment or funding in the Capability Maturity Model (CMM), International Organization for Standardization) ISO, etc., if applicable.)

   □ 0 % □ 1 – 4 % □ 5 – 8 % □ 9 – 12 % □ 13 – 16 % □ 17 % or more

2. How much funding has you software organization received for investment in a Software Process Framework, such as the CMM, ISO 9000/9001, etc.?
   Answer based upon percent of total budget during the last 3 years (or since the start of this investment or funding, if less than 3 years).
   (If your organization has not been funded, click 0% below and go to question 3.

   □ 0 % □ 1 – 4 % □ 5 – 8 % □ 9 – 12 % □ 13 – 16 % □ 17 % or more

   A. Please indicate the software process improvement program or framework your organization is implementing (or implemented):

      SEI Capability Maturity Model (CMM)
      ISO 9001, ISO 9000-3
      Other Standard or Software Framework. Please state_____

B. When did your software organization start implementation of the software quality framework? (Month/Year) _____

C. Has your organization achieved a certification or maturity level rating in this quality methodology?
YES NO
Please note the maturity level or certification: _____

3. All respondents – answer question 3.
Rate your organization's change in performance, compared to 3 years ago, for these financial measures:

(1= -51% or less, 2= -11 – 50%, 3= -10% to 0%, 4= 0.1% to + 10%,
5=11 to + 50%, 6= +51% or more

- Change in Revenue or Relief □
- Change in Earnings □
- Change in Productivity (if available) □

Section I Comments?

_____

_____

SECTION II.

Answer ALL items in Section II.

The following section describes specific Quality program characteristics.
Please indicate the extent to which the items below represent practices within your organization/department by selecting a number on the scale of 1 to 5.
Since the answers in this section are subjective, please answer based upon your judgment of the Highest Level possible for that practice.

SCALE FOR QUESTIONS IN SECTION II
5 = Practice at Highest Level
4 = To a High Extent
3 = To a Moderate Extent
2 = To a Small Extent
1 = Very Little or Not at All

EXTENT TO WHICH THESE STATEMENTS REPRESENT PRACTICES WITHIN YOUR ORGANZATION

_____ 1. Measurement of quality performance is defined and tracked throughout the organization.
_____2. SC Manager (or equivalent) has obtained funding and resources for the quality program.
_____3. Team members participate in project planning and design.
_____ 4. Customers give feedback on quality and delivery performance.
_____ 5. A repeatable process in place to improve processes.
_____ 6. Quality of products emphasized in relation to cost or schedule objectives.
_____ 7. Leaders in the SC (or equivalent) communicate a quality commitment to employees.
_____ 8. Quality practices of other software organizations researched.
_____ 9. Management training in quality principles, tools, and techniques.
_____10. A program to ensure consistent repeatable project delivery within time, budget, and quality constraints.

EXTENT TO WHICH THESE STATEMENTS REPRESENT PRACTICES WITHIN YOUR ORGANZATION

_____11. Quality program encompasses all areas of quality improvement within the organization.
_____12. Degree of coordination between the quality department and other departments.
_____ 13. A benchmarking program in place to increase performance for the organization.
_____ 14. SC Manager (or equivalent) has specific objectives for quality performance.
_____ 15. Customers actively participate in determining project requirements.
_____ 16. An oversight structure in place to ensure continuous process improvement.
_____ 17. Quality principles are included in the mission statement.
_____ 18. Employee training in quality principles, tools, techniques.
_____ 19. Measurements used to monitor and improve quality.
_____ 20. Visibility of the quality department.

_____ 21. An effective employee suggestion program is in place.
_____ 22. Strategic plan for the organization is based on the quality program.
_____ 23. SC Manager (or equivalent) encourages employee participation in quality decisions.
_____ 24. Process-based improvement assessments given to evaluate performance.
_____ 25. The quality department's access to upper management within the organization.
_____ 26. Cost of quality is measured and evaluated.
_____ 27. A career development training program is in place.
_____ 28. An active incentive and reward system for quality program successes.
_____ 29. Leaders in the SC (or equivalent) communicate a quality commitment to customers.
_____30. A program in place to reduce rework.

_____ 31. Autonomy of the quality department.
_____ 32. Resources are provided for training in quality.
_____ 33. A program is in place for continuous reduction in defects.
_____ 34. Effectiveness of the quality department in improving quality.
_____ 35. A program is in place to find wasted time and costs in all internal processes.
_____ 36. People in the organization are aware of the quality program.
_____ 37. Customers actively participate through all project phases.

Section II Comments?

_____

APPENDIX II. Data Analysis Tables

Table 1. Profile of Respondents

	n	Percent
Number of Respondents	141	100
Extent of Invest. In Software Process Improvement Framework		
0%	42	29.8
1-4%	78	55.3
5-8%	13	9.2
9+	8	5.7
Extent of Invest. In Quality Improvement Program		
0%	14	9.9
1-4%	77	54.6
5-8%	35	24.8
9+	15	10.6
Count of Surveys with Perceived Financial Performance Items		
Change in Relief or Revenue	130	92.2
Change in Earnings	123	87.2
Change in Productivity	104	73.8
ISO Certification or CMM Level Assessment		
CMM L1	64	45.4
CMM L2	23	16.3
CMM L3	22	15.6
CMM L4	3	2.1
CMM L5	1	0.7
ISO 9000	28	19.9
Organization Size		
0-125	54	38.3
125-250	32	22.7
250-500	29	20.6
500-1000	20	14.2
1000-2500	6	4.26
Geographic Location		
U.S. & Canada	83	58.9
Europe	36	25.5
Other Areas	22	15.6

Methodology for Analyzing Two Sample Populations

When analyzing the results of the statistical tests, certain assumptions must be met if the results are to be valid. For two sample t-tests, the assumptions are the data must be continuous, the distribution of the data must be normal, the variances of the two groups must be equal, the groups must be independent, and the samples must be random. If the data do not follow these assumptions, the following four situations should be applied (NCSS 2001):

1. If the assumptions are met, apply the equal variance t-test.

2. If the data are normal, but the variances tests are rejected, apply the unequal variance t-test.

3. If the data are not normal and the variances tests are accepted, apply the Mann-Whitney U or Wilcoxon Rank-Sum test.

4. If the data are not normal and the variances tests are rejected, use the Kolmogorov-Smirnov test.

When the results of the t-tests are presented in this study, a decision matrix is shown in the tables for each t-test. The decision matrix includes additional columns for data normality, equal variance, and the test applicable for the tests of the assumptions results. All the tests in this study measured statistical significance at the 0.05 level or less. Tables of data results follow.

T-test Results of Hypotheses 1 and 2

Table 2. Hypothesis 1 – T-Test – Group 1 – No Investment in SPI Framework

	Independent Variable – (df=139) Group 1 – No Investment in Software Process Framework (n=42) Group 2 – Investment in Software Process Framework (n=99)				
Dependent Variable	Reject Normal Test	Reject Equal Variance	Statistic Applied	Value	Accept/ Reject H0
Quality measures	Yes	No	Mann-Whitnev U	2.90	Reiect
Executive commitment	No	No	Equal-variance t-test	5.21	Reject
Employee empowerment	No	No	Equal-variance t-test	2.36	Reject
Closer to customer	Yes	No	Mann-Whitney U	4.38	Accept
Process Improvement	No	No	Equal-variance t-test	2.65	Reject
Quality philosophy	Yes	No	Mann-Whitney U	2.71	Reject
Quality training	Yes	No	Mann-Whitney U	4.15	Reject
Process evaluation	No	No	Equal-variance t-test	3.69	Reject
Role Quality Department	Yes	Yes	Kolmogorov-Smirnov	.369	Reject
Aggregate TQM Index	No	No	Equal-variance t-test	4.12	Reject

Table 3. Hypothesis 1 – T-Test – Group 1 – Less than 5% Investment

	Independent Variable – (df =139) Group 1 – Less than 5% in Software Process Framework (n=120) Group 2 – 5% or more in Software Process Framework (n=21)				
Dependent Variable	Reject Normal Test	Reject Equal Variance	Statistic Applied	Value	Accept/ Reject H0
Quality measures	No	No	Equal-variance t-test	3.40	Reject
Executive commitment	No	No	Equal-variance t-test	4.30	Reject
Employee empowerment	No	No	Equal-variance t-test	3.69	Reject
Closer to customer	Yes	No	Mann-Whitney U	1.69	Accept
Process Improvement	No	No	Equal-variance t-test	2.91	Reject
Quality philosophy	Yes	No	Mann-Whitney U	2.66	Reject
Quality training	Yes	No	Mann-Whitney U	3.40	Reject
Process evaluation	No	No	Equal-variance t-test	3.20	Reject
Role Quality Department	Yes	Yes	Kolmogorov-Smirnov	.389	Reject
Aggregate TQM Index	No	No	Equal-variance t-test	3.56	Reject

Table 4.Hypothesis 1 – T-Test – Group 1 – Maturity Level 1

	Independent Variable – (df =139) Group 1 – No ISO or CMM Level Achieved (n=64) Group 2 – ISO 9001 or CMM Level > 1 (n=77)				
Dependent Variable	Reject Normal Test	Reject Equal Variance	Statistic Applied	Value	Accept/ Reject H0
Quality measures	Yes	No	Mann-Whitney U	5.16	Reject
Executive commitment	Yes	Yes	Kolmogorov-Smirnov	.434	Reject
Employee empowerment	No	No	Equal-variance t-test	3.57	Reject
Closer to customer	Yes	No	Mann-Whitney U	3.20	Reject
Process Improvement	No	No	Equal-variance t-test	5.34	Reject
Quality philosophy	Yes	Yes	Kolmogorov-Smirnov	.343	Reject
Quality training	Yes	No	Mann-Whitney U	4.32	Reject
Process evaluation	Yes	Yes	Kolmogorov-Smirnov	.332	Reject
Role Quality Department	Yes	Yes	Kolmogorov-Smirnov	.284	Reject
Aggregate TQM Index	Yes	Yes	Kolmogorov-Smirnov	.400	Reject

Table 5. Hypothesis 2 – T-Test – Group 1 - No Investment

Independent Variable:
Group 1 – No Investment in Software Process Framework
Group 2 – Investment in Software Process Framework

Dependent Variable	n	Reject Normal Test	Reject Equal Variance	Statistic Applied	Value	Accept/ Reject H0
Change in Revenue/ Relief	1 = 39 2 = 91 (d = 128)	No	Yes	Unequal-variance t-test	1.46	Accept
Change in Earnings	1 = 36 2 = 87 (df = 121)	No	Yes	Unequal-variance t-test	1.06	Accept
Change in Productivity	1 = 32 2 = 72 (df = 102)	Yes	Yes	Kolmogorov-Smirnov	0.08	Accept

Table 6. Hypothesis 2 – T-Test – Group 1 – Less than 5% Investment

Independent Variable:
Group 1 – Less than 5% in Software Process Framework
Group 2 – 5% or more in Software Process Framework

Dependent Variable	n	Reject Normal Test	Reject Equal Variance	Statistic Applied	Value	Accept/ Reject H0
Change in Relief / Revenue	1 = 111 2 = 19 (df=128)	No	No	Equal-variance t-test	0.77	Accept
Change in Earnings	1 = 104 2 =19 (df = 121)	Yes	No	Mann-Whitney U	1.28	Accept
Change in Productivity	1 = 87 2 = 17 (df = 102)	Yes	No	Mann-Whitney U	1.66	Accept

Table 7. Hypothesis 2 – T-Test – Group 1 – Maturity Level 1

Independent Variable:
Group 1 – No ISO or CMM Level Achieved
Group 2 – ISO 9001 or CMM Level > 1

Dependent Variable	n	Reject Normal Test	Reject Equal Variance	Statistic Applied	Value	Accept/ Reject H0
Change in Relief / Revenue	1 = 57 2 = 73 (df =128)	No	No	Equal-variance t-test	0.54	Accept
Change in Earnings	1 = 55 2 = 68 (df =121)	No	No	Equal-variance t-test	0.10	Accept
Change in Productivity	1 = 45 2 = 59 (df =1 02)	Yes	No	Mann-Whitney U	1.69	Accept

APPENDIX III. GLOSSARY

Maturity – the degree of experience, capability, seasoning, level-headedness, judgment, and responsibility exercised by an individual or organization. [Hunter & Thayer, 2001]

Perceived quality management practices - all those activities of the overall management function that are considered to comprise the quality policy, objectives, and responsibilities of the TQM philosophy.

Quality management practices **–** all activities of the overall management function that determine the quality policy, objectives, and responsibilities and implement them by means such as quality planning, quality control, quality assurance, and quality improvement, within the quality system. [Hunter & Thayer, 2001; ISO Std 8402: 1986]

Software process **–** a set of activities, methods, practices, and transformations that people use to develop and maintain software and associated products (e.g., Product plans, design documents, code, test cases, and user manuals). [CMU/SEI-93-TR-25]

Software process improvement (SPI) framework – a framework that provides a variety of methods, tools, procedures, document templates, and so forth for enacting the various work processes used in developing a software system. [Hunter & Thayer, 2001] An SPI framework includes currently recognized software standards, such as ISO 9000-3/9001 and software models, such as the capability maturity model (CMM). [Sheard, 1997]

Software process improvement (SPI) – action taken to change an organization's software processes so that they meet the organization's business needs and help it to achieve its business goals more effectively. [ISO/IEC Std TR 15504-9: 1998]

Software quality – the totality of characteristics of the product that bear on its ability to satisfy stated and implied needs. [ISO/IEC 14598-1: 1998]

Total quality management (TQM) – the application of quantitative methods and human resources to improve the material and services supplied to an organization, all the processes within and organization, and the degree to which the needs of the customer are met, now and in the future. [Paulk, Webber, Curtis, & Chrissis, 1994]

Instituting a 360 Degree Metrics Program at Cadence

Mark Noneman
VP, Enterprise Quality
Cadence Design Systems, Inc.
555 River Oaks Parkway
San Jose, California 95134
mnoneman@cadence.com

Abstract

As the saying goes: if you don't measure it, you can't improve it. The quality improvement initiative at Cadence Design Systems had made some progress on the limited data available about product quality. However, it soon became clear that to achieve the next level of improvement a comprehensive measurement program needed to be established. To get a complete view of both product and process quality, we created a 360 degree metrics view; customers, management, engineers, and the product itself all contribute to our understanding of current quality, needs for improvement, and ensuring positive progress is being made.

This paper will describe the process by which metrics were established, measured and reported, how pervasive and continuous review of metrics enabled us to hone in on the most important measures and how those measures are being used to define improvement opportunities and improvement results.

Biography

Mark Noneman is Vice President of Enterprise Quality, Cadence Design Systems. His responsibilities include improving whole-product quality focused on software best practices, especially requirements methods. He has more than 20 years of experience as an electronics engineer, system architect, project manager, and quality expert at TRW, Nokia Mobile Phones, Mentor Graphics, and Cadence Design Systems.

Copyright © 2004 by Mark Noneman

Table of Contents

- Introduction
- Overview of the Quality Initiative at Cadence
- Beginning a Quality Metrics Program
- History: Metrics without Purpose
- Starting With the Basics
- Establishing Release Criteria
- Measuring Release Criteria
- Adding Continuous Improvement Metrics
- Where Do We Get More People?
- Executive View of Quality
- Reporting Metrics
- Conclusions
- References

Introduction

In the past few years, Cadence Design Systems has undertaken a focused activity to improve the quality of its products and services across the company. The company had re-organized from a model based on centralized functions to a collection of business units responsible for product revenue and investment expenses. At the same time, a series of questionable business decisions resulted in several new product development projects being canceled after many millions of dollars of investment and many staff-years of effort. After a critical review by upper management, Cadence started to adopt the well-known Stage-Gate business decision model as a means of getting its important decisions under control.

After several years of successful use of Stage-Gate, we recognized the need for a comprehensive quality improvement metrics program. Previously, lots of attention had been placed on measuring project performance in terms of schedule, resources, and earned-value but no specific goals had been put in place for quality improvement. This paper describes that situation, what Cadence is doing to implement a 360 degree quality metrics program, what we have learned to date, and our plans for improvement.

Overview of the Quality Initiative at Cadence

The current quality initiative at Cadence started in 2001 in an attempt to improve customer satisfaction. Cadence has been surveying customers about their satisfaction for over 10 years. Those survey results are, in fact, the first consistent quality metric in the company. Customer feedback told us that product quality and responsiveness of customer support were among the top issues that customers were concerned about. Since our goal was to improve customer satisfaction in order to improve sales, a holistic improvement initiative was developed in 2002 known as the Enterprise Quality Strategy. The Enterprise Quality Strategy outlined a comprehensive program to address all important customer barriers to adoption and proliferation of Cadence products.

Further discussions with customers made it clear that of all the issues identified by customers, product quality was by far the most important. Even the most important customer support issue, described by customers as "timely and effective solution to reported problems" was, in fact, a product quality issue. By the end of 2002, it became clear that although an "enterprise" approach to quality was necessary for the long term, the most important issue in the short term is product quality.

At the start of 2003, we evolved our approach to quality improvement to focus on four key areas. First, we established a company-wide set of criteria for product release. Second, we instituted a program of continuous improvement based on customer feedback. Third, we began educating our employees to implement software development best practices based on prioritized problem areas defined through root cause analysis as part of our continuous improvement program. And finally, we began a structured metrics program to assess our current quality status and trends toward improvement. This paper will elaborate our metrics program in the context of our holistic quality initiative. First, it is important to establish an understanding of what makes a good metric.

Beginning a Quality Metrics Program

Quality metrics is one of the four key quality programs in the Cadence quality initiative. To ensure rapid and permanent change in customer perceptions of Cadence product quality, we wanted to define "all around" measures (hence, 360 degree metrics) that would allow individual contributors to monitor progress on a day-to-day basis and give executive management a snap-shot view of current quality across the company and where product quality is headed (that is, its trend).

Instead of trying to define all of these metrics all at once, we took an incremental approach. Each of the other three quality programs at Cadence (release criteria, continuous improvement, and best practices) defined metrics for each program. The release criteria program created low-level measures of our products for every release of software to our customers. The continuous improvement program created measures from both a customer perception (external) and an improvement project (internal) perspective. The best practices program creates measures for the productivity and effectiveness of our work. The final element of the metrics program is an executive "dashboard" for a company-wide view of quality and our progress on improvement.

History: Metrics without Purpose

Measuring things for the sake of measurement is not useful and wastes resources; too bad the practice is so common. As described above, Cadence has a long history of surveying customers to measure their perceptions of our performance. However, asking customers about their perceptions was not enough. The data and comments from customer were very high level. For example, we asked customers how important "product quality" was to them and how satisfied they were with it. Unfortunately we didn't define what we meant by "product quality" and customers were left to define the term in any way they wanted. Furthermore, comments collected from customers were similarly high level such as "the most important thing for Cadence is to improve quality"; hardly an actionable request.

Since employees couldn't directly identify with feedback from customers at the high level of the customer satisfaction survey, it was not possible for them to actually do anything to improve satisfaction. We needed a way to make the data "actionable" for employees. In 2003, the customer satisfaction survey was redesigned to collect meaningful, detailed information about customer's perceptions of Cadence products and services. The industry-standard "FURPS" model [1] was used as a framework to ask customers about their experiences in ways that software developers and their managers could actually use. The customer satisfaction program has a clear purpose now: collect customer feedback to continuously improve products and services (with an emphasis on products for the time being).

Another example of metrics without purpose: For several years various "product quality" metrics had been collected and reported to management. One such measurement was the number of currently open defects reported against products. Although measured for years, the number randomly fluctuated from month to month without any particular trend. Since no target had been set (that is, no purpose defined), no change was eminent. Eventually, management created a purpose: "reduce the level of open defects." Immediately, the number of open defects started declining across the board. As expected, this occurred by fixing bugs, closing defects that would never be fixed, and even increasing the threshold for what would be labeled a "defect." Not all these actions improved quality but the purpose was achieved.

Eventually, the level of open defects was considered acceptable and the measure eventually abandoned. Unfortunately, since nothing had been done to address the cause for why open defect levels were so high, the number of open defects soon returned to previous levels or even higher. In this example, although a "purpose" for the metric was eventually defined, the ultimate goal was not stated. The goal should have been to permanently reduce the level of open defects by eliminating their causes. A special effort to reduce the legacy of defects may not have even been necessary.

The essential lesson learned is that metrics without an identified end-goal are at best useless and at worst wasteful. All quality metrics at Cadence are now defined using the Goal—Question—Metric approach outlined below.

Starting With the Basics

We knew from our own experiences and the teachings of others [2] that we needed to create measures only to achieve specific end goals. We used a relatively simple technique known as goal—question—metric (GQM). [3] The most difficult part of using this technique was to define what we really wanted to achieve! In many ways, it's easy to create measures; it can be very difficult to gain agreement on what we want to do with them.

The GQM approach is a structured method to define metrics that address specific business objectives. It is based on two basic principles:

- Metrics affect behavior, and
- We will only measure to achieve a specific business goal(s).

In practice, we need to answer three questions:

1. What business goals are we trying to achieve?

2. What question(s) can we ask that will help us know when we've achieved the goal(s)? In addition, what questions can we ask that will help us know we're not creating unintended behavior?

3. What data will answer the questions?

In some cases, the goals are defined by cross functional teams that together have a complete understanding of the problem at hand. In other cases, executive management may want to achieve specific business goals. Even with a specific identified goal, understanding how we will know it has been achieved can be elusive. Typically, a small team of knowledgeable people must figure it out. Finally, knowing what question to ask must be balanced with what data is available or can be collected economically. (We usually get data analysts involved to determine what data we should use in our metrics.)

Establishing Release Criteria

One of the known problems with Cadence products was the common practice of releasing software to customers that had known problems or that had not completed many basic product requirements. Both customer feedback and internal product release assessments made the extent of this problem clear.

The so-called "Minimum Release Criteria" (MRC) program was created to address this problem. The program goal was often stated as "stop shipping bad stuff." Although not very clear, it got people's attention. The precise long-term program goal is to eliminate basic quality problems from everything we make available to customers.

The MRC program developed criteria for a variety of release types (for example, major releases, patch releases, beta software, etc.) and encompasses all aspects of a whole software product. These aspects include the software itself, documentation, training, services, and protection of intellectual property (ours and 3rd party's).

The criteria focus on <u>what</u> must be achieved but generally does not specify <u>how</u> it is to be done. This is because Cadence has many different types of products that address markets of varying maturity and needs. Flexible development practices are needed to enable product success.

Since the criteria were established generically for all products in the company, we recognized that there might be perfectly good business reasons to violate the criteria from time to time. When we started to test enforcement of the criteria we also identified process shortcomings that prevented compliance to the criteria. For these situations, a "deviation" process was developed to allow the company to continue serving customers. However, for the first time, product development groups were not allowed to unilaterally violate the criteria; they had to get agreement with other parties inside Cadence. These "stakeholders," as they are known, are either responsible for the possible consequences of violating the criteria or act as proxy for the company or customers. For example, if a product groups wants to release a product without copyrighting its documentation, it can do so only if the legal department agrees there is a good

business reason. This deviation process allows for true business needs but minimizes capricious disregard for the criteria.

Each criterion consists of 6 elements: the requirement itself, its rationale, the standard for success, the stakeholder (by role), a suggested implementer (by role), and a suggested milestone for completion. The last two elements are suggestions only and not required. As long as the criteria are completed by release time, the criteria are considered met. Table 1 shows an example of several criteria for different release types.

Table 1. Example of minimum release criteria by release type.

Release Type	Criteria	Rationale	Stakeholder
Base	Migration plan AND target customers for the new release or product have been defined and documented	Sometimes customers are not willing to migrate to new technology because of the extensive migration efforts required. If we identify the actions required to migrate to the new release and document these, it will be easier for customers to plan their effort and adopt the new technology	VP Customer Support
Update	All changes made in the release have been validated to not break the design flow with other relevant products	Customers should be able to use products seamlessly and should not be forced to upgrade due to incompatibility between releases	VP Quality
Hotfix	No new functionality has been included in this release	Customers do not want to change their methodology or use model for a patch so no new functionality should be included	VP Customer Support
Engineering Hotfix	The Engineering Hot Fix release is only shipped to the reported and requested customer(s)	Due to the nature of the release, which is high risk, we do not want to make the fix available to other customers	VP Product Operations

Measuring Release Criteria

To measure the MRC, we are currently focused on compliance and closure of corrective action plans for approved deviations. Every month, each product group reports MRC metrics as shown in Table 2. The compliance metrics (that is, number of deviations by release type) are rolled up for a company-wide report. Since our goal is to eliminate "bad stuff," we need to achieve process capability to be compliant and have zero deviations. Once we achieve that, we can work

to maintain compliance and gradually increase the "minimum" acceptable criteria while we work to develop criteria for other deliverables from Cadence (for example, services).

Table 2. Example monthly report of release compliance to the MRC. (Note: PCR stands for Product Change Request; defect reports in these cases.)

Release			No. MRC Deviations	MRC Deviation Summary (CA#)	Recall (Y/N)
Name	Version	Type			
KEW 5.0	UPD #2	Update	1	PCR names in release notes (1)	N
KEW 5.1	UPD #3	Update	1	PCR names in release notes (1)	N
KEW 5.0	HF #5	Hotfix	1	PCR names in release notes (1)	N

CA	Corrective Action Plan	Completion Date	Current State
1	Establish PCR name process for all releases	July 04	Defined. Training in process

Adding Continuous Improvement Metrics

Of course, just setting a minimum standard for releases isn't enough. We needed to define specific improvement plans if we were to see customer's satisfaction improve. But each product line had very different problems using different processes for different products serving different markets. It was not possible to define a single, company-wide improvement project.

However, we did have information from our customer satisfaction survey and other sources that told us what the most important and common problems were in each area. For example, one product line's most important issue was too many latent defects in releases, another product line's most important issue was missing functionality while another had performance that did not meet customer expectations. (All of those issues are quality issues using the FURPS model.) Therefore, every product line needed to analyze available data to identify what customers considered the most important problems. That, in effect, would define part of the "goal" for GQM.

But knowing the problem was only the start. We also needed to know what customers expected from us for each area. In many cases, we didn't know! To find out, we had to go back to customers and ask them (via a combination of telephone interviews and surveys of the most important customers for that product). Each product area, in developing their continuous improvement plans, went back to customers to find out what they expected from us in the problem areas. We also asked them the relative importance of the identified problem areas; we knew we couldn't improve everything all at once so we needed to focus on the biggest return areas. In practice, we needed to set improvement goals from this information; it was often more practical for us to set intermediate goals that were achievable in reasonable time frames. Table 3 improvement shows the continuous metrics as reported quarterly by each product line.

Table 3. Example quarterly status of improvement for top customer issues.

	Customer Issue	Customer Expectation	Initial State	Target Goal	Current State
1	Random and reproducible crashes	No crashes	Numerous crashes per day	Less than one crash per week	Occasional crashes
2	New and recurring problems found in releases	No recurring problems	Zero regression policy for customer defects	80% of the defects have a test checked in by Q3	Physical design at 75%; simulation at 60%
3	Flows require workarounds	More flow validation	Customer flow validation partnership on need basis	Two customer partnership per six months	(customer) designs have been automated in test suite; Status provided at review
4	Reduce defect response time	Immediate response to each defect	22 calendar days response to P0 defects	14 days response to P0 defects	On target
5	Release Adoption; difficult to move the design environment to a new release	Consolidated documentation & adoption plan	No consolidated migration doc; No customer adopted latest release	A consolidated & complete migration doc for June release; At least 20 customers adopted new release	16 customers adopted new release; second version of the migration doc is in review

Of course, just identifying problem areas and improvement goals wouldn't get us there. We needed to make sure we knew the root cause for each problem, select specific areas to improve based on return on investment, and then plan, and track improvement projects executed in the product groups. Measuring these improvement projects gives us some "leading indicators" to know if we are on track. Seeing the results in the customer-identified problem area and then measuring actual improvement in customer satisfaction takes a long time; in some cases, years. Table 4 shows the management summary of the project status we review monthly. The "traceability" of customer issue to improvement project is documented in the continuous improvement plan.

Table 4. Example continuous improvement metrics for correction of root-cause issues. Corrective actions are summarized for non-green status projects.

#	Project	Expected Result	Completion Date	Current Status	CA
1	Good Unit Test Gate	Faster build closure and tester productivity	June 04	**Yellow**	**1**
2	Interoperable flow release	Deployment productivity and usability	June 04	**Green**	
3	Benchmark flow	Benchmark productivity	June 04	**Green**	
4	Time-to-productivity	Benchmark productivity, winning % rise	June 04	**Yellow**	**2**
5	Publications focus projects	Timing closure, translation, hierarchical flow	June 04	**Green**	

CA	Corrective Action Plan	Completion Date	Current State
1	More rigorous application of unit test gate	June 04	85% effectively deployed
2	Field productivity plan with metrics	July 04	Draft plan, no staffing

Some of the problem areas can be quickly addressed and we have been able to see measurable improvement in customer satisfaction. For example, in one of our product lines "supportability" was clearly identified in the first half of 2003 as an important problem area for customers.

After further investigation, we realized that we had ignored a relatively simple area that when working properly is ignored by customers but when problematic, is a big dissatisfier for them: software installation. Because this product line had been built with several acquired technologies using uncoordinated installation techniques, we found it was taking more than a day for customers to install major releases of this product often requiring numerous calls to customer support or local application engineers; clearly an unacceptable situation.

In discussing the problem with customers, we determined that they expected major installations of this type to take no more than 4 hours without any calls to customer support or application engineers. Given the clear driver of this issue with customers and the relatively small amount of effort required to address the problem, this issue was selected for improvement by the product group.

After only about one month of effort to re-design a consistent installation scheme, time for installing major releases was reduced to about 3 hours. Luckily, adoption rate for new releases in this product line is quite rapid and many customers saw a radical improvement in installation time before the next customer satisfaction survey at the end of 2003. In that survey, the supportability issue for that product line showed dramatic improvement such that it is now considered a non-issue from an improvement perspective.

(Note that the ultimate root cause for this problem was that the product group didn't understand the importance of this issue to customers. Future acquisitions and releases will need to pay attention to this area if we are to maintain satisfaction for customers in this area.)

Where Do We Get More People?

The most common question asked is "are there extra resources to improve quality?" The answer, of course, is "no." A simple analysis of productivity shows there are really plenty of resources available; unfortunately, they are current tied up reworking the product to perform integration and test, fix bugs in old releases, and perhaps most onerous of all, porting new features onto old releases (because customers don't want to adopt new releases!). For historical reasons, Cadence has an R&D timecard system that gives us a rough measure of how our engineers spend their time. From our timecard system, we know that up to 50% of all product development people's time is identified as working in these areas (see Figure 1), none of which add value to Cadence or to our customers. (We don't know how much time is spent in rework in the requirements, design and coding phases.) If we could eliminate some significant portion of this rework, there would be lots of resources available to work on value-adding activities.

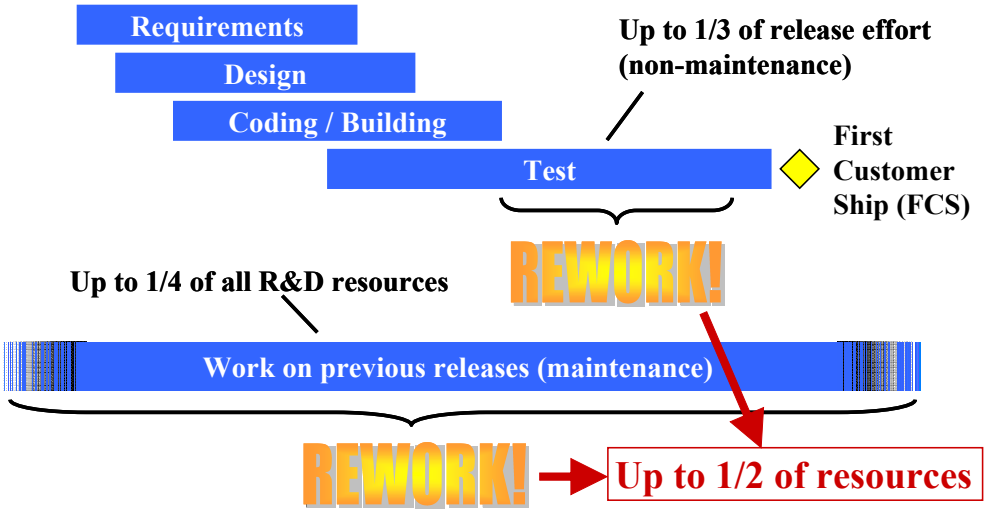


Figure 1. Where will we get resources to improve quality? Reduce rework, of course!

Changes in our timecard system will give us better visibility into where defect prevention and detection is happening. However, the problem with timecard systems is that they invite "game playing"—reporting data according to what people think management wants to see. If we use timecard data alone, we would likely see what we wanted to see: an increase in requirements/design and technical review time with a commensurate reduction in test and eventually maintenance time…whether that is what's really happening or not!

Therefore, we are beginning to collect measurement of requirements management and technical review activities. Metrics such as requirements traceability completeness over time, changes to requirements and test coverage of requirements will help us understand how well we are preventing defects in this area. Metrics such as work products reviewed versus planned and defects found during reviews will help us understand how well we are detecting defects earlier in the development process. Studies have repeatedly shown that these activities save much more time in find-and-fix test cycles than they consume. Our experience so far seems to confirm the savings but it's still early.

While these metrics are in development, we are currently tracing the "adoption" of best practices in each product line. "Standard" best practices include technical reviews, unit testing, requirements management, root-cause analysis, and software failure analysis. Each product line identifies other best practices that address root causes as identified in their continuous improvement plans. Table 5 illustrates the current reporting for best practice adoption showing number of employees trained and the degree to which the practice is executed in the group.

Table 5. Best practice adoption summary reported each month.

Best Practice	BU Adoption Goal	# Personnel Trained	% Process Execution	Plans for Next Period
Requirements Management	Functional coverage analysis and req'ts change management	20 architects and Mgmt trained	<25% in prev release	Reassessing process for next release
Unit Testing	Shorten final integration & test time	No formal training	95%	Rigorous focus to achieve results and compliance across the board
Root cause analysis	Identify and fix true cause of problems	~25 Mgmt and leads	All critical failures	Continue
Software failure analysis	Determine where to improve to prevent problems	No formal training	All critical software failures	Continue
Formal technical reviews	Find defects earlier in the process	~20% of R&D	~1/3 of req/ spec docs reviews	Train all personnel working on next release. Require 100% req'ts and critical code reviews

Executive View of Quality

The fourth major area of metrics development is our executive view of quality. The concerns at the executive level consider customer, business and strategic direction. There are currently four key areas of concern at the executive level:

1. The customer's perception of product reliability; the most important overall product quality issue for customers across Cadence

2. Our responsiveness to customer reported problems; the second most important overall issue for customers across Cadence

3. The total rework challenge for detected defects

4. The need to address customer satisfaction barriers before they become entrenched problems that are difficult to fix.

We have identified five metrics that help us understand the current status in these areas[1]:

A. Total incoming defects per month separated into Cadence-found and customer-found defects (addressing concern #3). See Figure 2.

B. Our defect detection (test) effectiveness as measured by the ratio of customer-found to Cadence-found defects (addressing concern #1); this is the same data as metric A portrayed as a ratio. See Figure 3.

[1] Note that Cadence defines defects as 3 possible "priorities": P0 (highest), P1, and P2 (lowest) and we do not currently separate the severity of the defect from its priority. This leads to many problems in our defect reporting, tracking and measuring that are outside the scope of this paper. Suffice it to say that a replacement system that follows industry norms in this area is planned for late 2004 implementation.

C. The number of "open" defects each month separated into Cadence-found and customer-found defects (addressing concern # 1, 2 and 3). "Open" defects are those for which we have neither committed a fix nor decided that we will not fix it. See Figure 4.

D. The age of customer-reported "open" defects (addressing concern # 2). See Figure 5.

E. Customer support surveyed customer satisfaction for closed support calls categorized into product quality attributes of Functionality, Usability, Reliability, Performance or Supportability (FURPS) (addressing concern #4). See Figure 6.

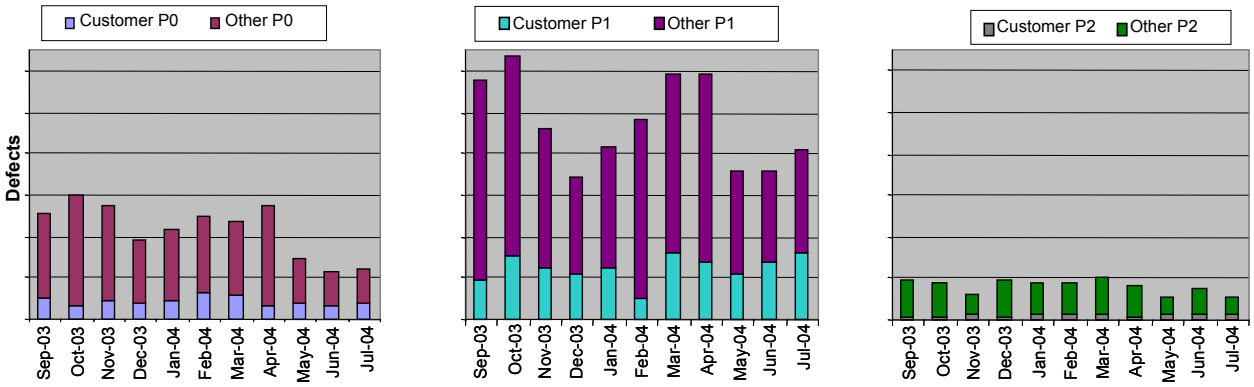


Figure 2. Incoming defect rates for a Cadence product line. Numbers have been removed. Note that we can tell this group has reduced internal testing in the last 3 months because non-customer incoming defects ("Other") have gone down.

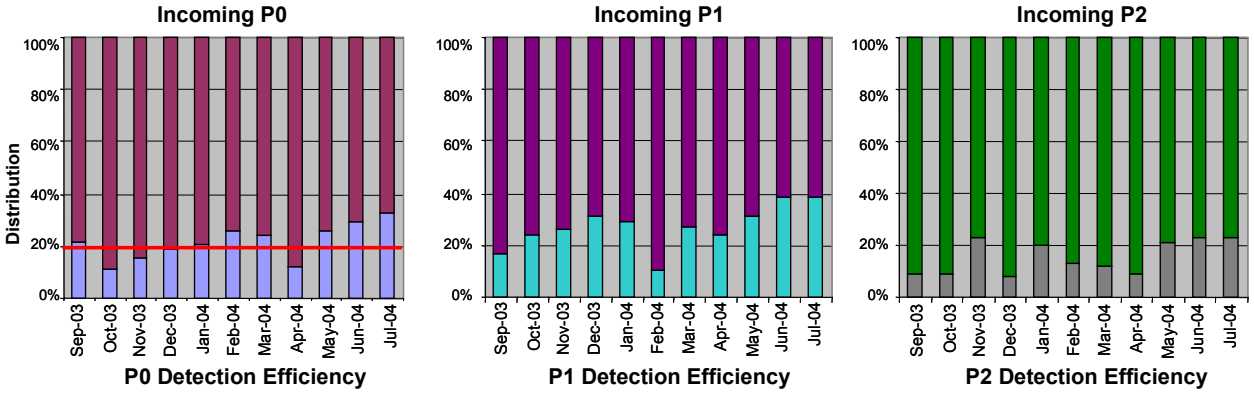


Figure 3. Defect detection efficiency for the same product line. Note the target line for P0 defects: the only company-wide target established so far. Again, since internal testing has gone down recently, defect detection efficiency has also worsened (customers keep finding a consistent level of defects).

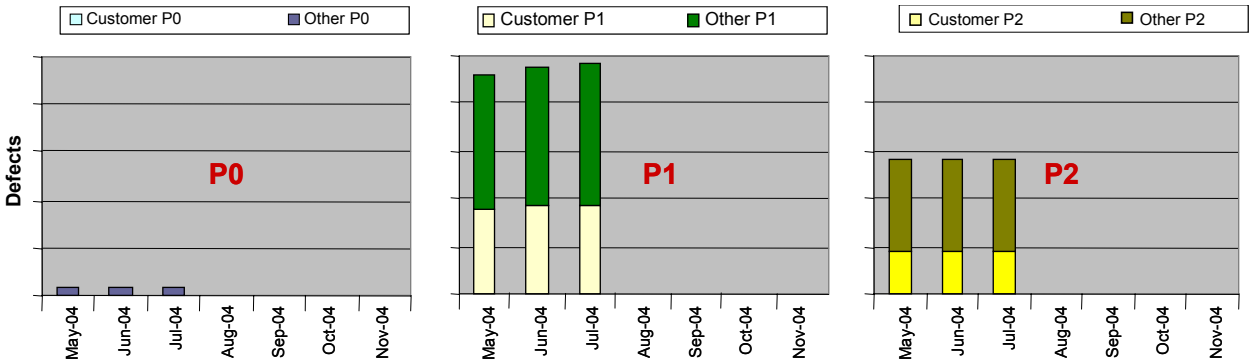


Figure 4. "Open" defects for which we have neither committed a fix nor decided we won't fix them. Clearly, we have a lot of work to do on P1 defects.

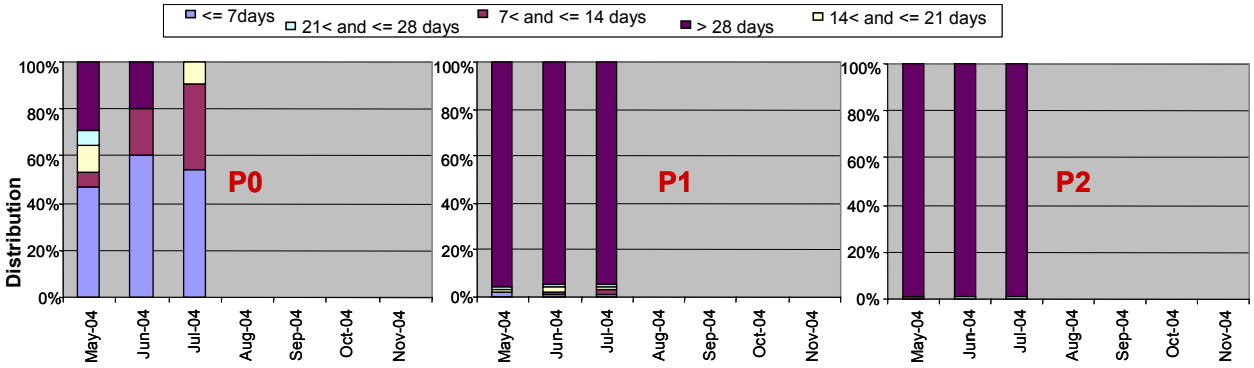


Figure 5. The age of customer "open" defects shown as a stacked histogram. In this product line, the age of customer P0 defects is improving while P1 and P2 defects languish.

It should be noted that the fifth metric (customer support surveyed customer satisfaction) is the least mature of all executive level metrics. We are still in the process of evaluation this measure to see if it is really a leading indicator of customer satisfaction as measured by our corporate customer satisfaction survey. It is a strange but well known phenomena that customers are much more satisfied when they have had a problem that has just been resolved than when we ask them outside of the context of problem resolution. In other words, customers feel better when we've just helped them with something painful to them.

Only one company-wide target has been set for all product lines in the corporate-wide metrics: the target for detection effectiveness for P0 defects is better than 80%. That is, for all reported P0 defects, we should be finding at least 80% of them; customers should find less than 20%.

All other executive level metrics have targets set for the product line based on their measured performance. Our goal is continuous improvement and there is a wide range of performance across the company.

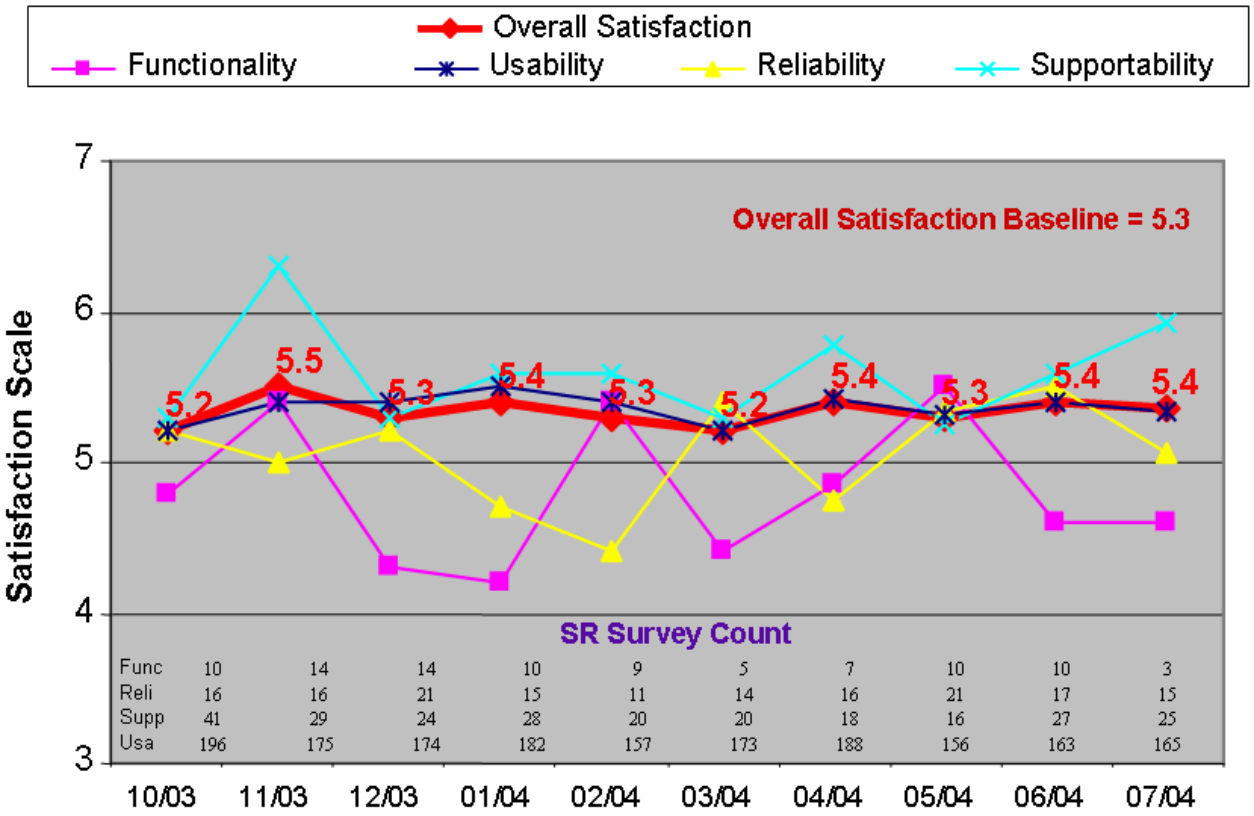


Figure 6. Customer support surveyed customer satisfaction based on support call closure codes. Low survey response count for functionality closure codes cause this metric to be highly variable. Note that there are currently no "Performance" closure codes in customer support. This metric is still being evaluated for its effectiveness as a leading indicator of overall customer satisfaction.

Reporting Metrics

Each metric type is reviewed periodically and, in some cases, based on events. Table 6 shows each metric described in this paper and its reporting requirement(s):

Table 6. Metric reporting requirements at Cadence.

METRIC	REPORTING			
	BY	TO	FREQ.	METHOD
Minimum Release Criteria	Project manager	Corporate Quality	At release	On-line form including approved deviations
	Operations Director	Corporate Quality	Monthly	Summary of monthly releases, deviations, corrective action plans.
	Product Line VP	Executive VP	Quarterly	Statistical summary of MRC compliance

METRIC	REPORTING			
	BY	TO	FREQ.	METHOD
Continuous Improvement Plans	Operations Director	Corporate Quality	Monthly	Status of current improvement projects including corrective action plans as necessary
	Product Line VP	Executive VP	Quarterly	Status of improvement for customer identified issues according to their success criteria.
Best Practices	Operations Director	Corporate Quality	Monthly	Adoption of best practices as identified in Continuous Improvement plan
Common Quality Metrics	Operations Director	Corporate Quality	Monthly	Analysis of current metric status and trends against established targets. Corrective action plan status reviewed.
	Product Line VP	Executive VP	Quarterly	

Conclusions

A structured, holistic metrics program is essential to any continuous improvement program. Careful consideration must be given to the end result (goal) so that appropriate measurements can be made without generating unintended behavior. Using the goal—question—metric method of metric design, we stay focused on the end result. By keeping the customer in focus when developing intended end results, customer satisfaction can be measurably improved. Cadence Design Systems has accelerated its quality improvement progress by carefully developing 360 degree metrics that give the program "teeth." Of course, upper management commitment and involvement in continuous improvement programs and including measurable progress into management objectives is crucial for ultimate success.

References

[1] "Practical Software Metrics For Project Management And Process Improvement," Grady, Robert, Prentice Hall, 1992.

[2] "Successful Software Process Improvement," Grady, Robert B., Prentice Hall PTR, 1997.

[3] "The Goal Question Metric Approach," Basili, Victor R., Caldiera1, Gianluigi, Rombach, H. Dieter, Encyclopedia of Software Engineering. Wiley 1994.

Defect Analysis – A Tool for Improving Software Quality

John Balza
Hewlett-Packard Company
john.balza@hp.com

Abstract:

In Hewlett-Packard's software quality and productivity improvement program, defect analysis was one of our key tools in achieving a 25% reduction in testing schedules and 30% improvement in overall productivity. This paper will be a practical tutorial on how to conduct software defect analysis including several real examples of how to apply the tool. A metric for defect finding effectiveness will be described that provides a management incentive to do a good job at defect analysis.

Biographical Sketch:

John Balza has been the Quality Manager for HP-UX since 1994, leading the 10X quality improvement program. In his 30 year career, he has managed over 50 software projects for Hewlett-Packard at various management levels. He has been exploring software quality and processes since his first failed project, which was a year late and had to be completely rewritten.

© 2004 Hewlett-Packard Company

Why Do Defect Analysis?

In Hewlett-Packard's software quality and productivity program, we utilize the following key practices:

1) Communication of management's commitment to quality including weekly review of key quality metrics (This was discussed in my 2003 PNSQC paper [1])
2) Inspections
3) Retrospectives
4) Defect analysis

Defect analysis seeks to identify the causes of software defects in order to select and address major, recurring causes of defects. In particular, we want to determine how we can find or remove defects earlier in the software process where they are less expensive to fix. For example, our data shows that it takes about 1-2 hours to find and fix a defect found by inspection versus 1-2 weeks to find and fix a defect that is found in system test. The cost of fixing a defect found by a customer is often cited as being 10 times this cost, because of the downtime experienced by the customer, the potential for loss of business, as well as the costs by customer support and in the field to identify and properly address the defect.

A key benefit of defect analysis is that it quickly gets organizational buy-in. The teams doing the analysis decide upon the opportunities they want to explore and make the recommendations on what actions to take. This results in much less resistance than imposing an improvement model like ISO 9000 or the Capability Maturity Model from a third party. Defect analysis is also a great complement to a retrospective process. Our typical retrospective asks what went well, what did not go well, what needs to be improved. Retrospectives tend to deal with subjective data; defect analysis adds some objective data to reinforce what improvements would make a difference. Because the recommendations are very specific from defect analysis, it is easy to measure the defects that are found before and after the implementation of the recommendations, thus reinforcing the value of the analysis.

How to do Defect Analysis – Starting Situation

We utilize different methods of defect analysis, depending on the situation and the maturity of the organization. The starting situation for most teams is that they have completed a project and know what defects were found during that project, but they haven't categorized the types of defects or performed any root cause analysis. Either they have decided that they want to do better next time, or they have been asked to improve their quality, but they don't know where to start. A facilitator familiar with the process is brought in to help the team.

1. Planning

Set the Purpose/Focus of the Analysis

It is important for the participants to have a sense of the exercise's value. "Because someone said so" isn't a very motivating reason. The purpose should be tied to business objectives, such as reducing the number of defects found by customers and partners, becoming more productive by reducing the time we have to spend finding and fixing defects, improving our execution of one of the lifecycle phases such as design, implementation, or testing, etc.

Most generally, the defect analysis identifies causes of software failure in order to identify the process improvements that will prevent such failures in the future. To further focus the analysis, consider where you are in the product life cycle. Perhaps the most common use of defect analysis is at the beginning of a release cycle, to ask the question, "How can we do better this time?" But the analysis can be appropriate at any point in the life cycle. For instance, prior to the implementation phase, you may want to take a look at defects introduced during coding phases in previous releases. Or, because of a schedule slippage from another team, you may suddenly have a few extra engineer months available to spend on process improvement, but you're wondering where to focus the effort. Because of the high cost of patching, you may want to analyze those defects that prompted patches.

If you do want to further focus your analysis, here are some possible considerations:

- For a large product, you may want to focus the analysis on those parts of the product that receive the most use or that have had the most defects. It's best not to guess at these; instead, use data from your defect tracking system to identify these areas.
- You may want to focus the analysis on a particular dimension of quality besides simply reliability, e.g., functionality, usability, or performance.
- You can examine quality as experienced by a particular customer type (a development partner, system test, customers, a specific customer, etc.).
- You can focus on reducing maintenance costs. This might lead you to look at those defects that were the most expensive to fix.
- You can focus on those areas where a few low cost changes will reduce a large number of defects.

Once you have set the focus for the analysis, be sure to communicate it to all participants.

Plan the Resources

The defect analysis typically takes 3 engineers about 3 days, if you are dealing with hundreds of escaped defects. The person responsible for writing up the results will

spend additional time; moderators will spend less time since they don't have to be continuously involved. A typical schedule looks like this:

Day 1	Day 2	Day 3
Learn process Practice/begin analysis	Complete analysis	Create an affinity diagram of causes Brainstorm solutions Plan actions

For a small number of defects (<100) the process should fit within 2 days.

Moderator

Especially if this is your first analysis, you should ask a moderator to help you plan the analysis and facilitate the meetings. The moderator is involved for the first 2-3 hours and then checks in periodically during the analysis to see if there are any questions. After the analysis, the moderator participates in all activities.

Participants

This process requires only three participants (people doing the defect analysis). Pick three engineers familiar with the product. One should be a senior engineer who knows the product very well. The other two can even be fairly new engineers. It helps to have a range of familiarity because while deep experience can help analyze the defect more quickly/effectively, deep experience can also give one a false sense of knowledge about where the root cause really lies.

Follow-up

Decide up front that you will allocate time and resources to make process changes based on the results of the analysis and hold yourself accountable for completing the improvements. It is extremely de-motivating to spend time analyzing a problem and coming up with solutions only to continue doing things the same old way. Before the analysis, you won't know how much effort will be needed for the improvements, but you should have some sense of who is available for the work and how much time they have available.

Select Defects to Analyze

This process works well for up to about 250 defects. Do not include defects that have not been fixed because these will not typically contain enough information for analysis. Selection criteria should be based on the focus of the analysis. For instance, if you feel your product isn't fitting well with customer needs, you may want to include all customer-reported defects, including those that are classified as enhancement requests or resolved "no change".

Other ways to select defects:

- Severity
- Who found the defect (the development team, other teams, system test or customers)
- Longest time to fix
- Defects inserted during a particular life cycle phase
- Defects in a particularly problematic area of code
- defects that are the most costly to the development team or customer

It helps with paper management during and after the meeting to print each defect report starting on a new sheet of paper and to have all the pages numbered sequentially. Bring one copy of each defect to the meeting to hand out.

2. Analysis

Determine a Root Cause for Each Defect Report

The defect cause description occurs in a moderated meeting lasting 1-2 days. Bring several pads of Post-its® and some pens.

The process is straightforward: scan the defect report and write a short description of what went wrong. In the bottom right hand corner of the Post-it, write the page number of the defect report. This way you can go back and answer any questions if you later forget what you meant.

Practice by doing one defect each and then discussing the result (the cause you've written on the Post-it). When discussing the result, make sure that the "cause" you've written down meets the following criteria:

1. It describes "what went wrong". Think of it as a 1-sentence "plot summary." Here are some examples:

- Message was hard-coded instead of being put in message catalog
- Error message appeared in unforeseen context
- Code was copied but left out an important initialization

2. It does not describe how the defect could have been prevented (e.g., "insufficient testing" or "design was never reviewed" or "lack of developer training"). At this phase in the process, try to avoid thinking about solutions--focus on what went wrong. If, during the analysis, you do have a brilliant idea for a solution, write it down somewhere and forget about it. Pull it out when it is time to brainstorm potential solutions.

3. It does not "pre-categorize." The categories of error will emerge during the next phase, creating an affinity diagram of the causes into categories of failure. So when

describing a cause, avoid writing "Coding error - neglected to free memory". Instead, say "neglected to free memory after a temporary allocation" or something similar.

4. It is specific enough to be understood and categorized later in the process. For example, "memory leak" is too general. What caused the memory leak?

If there isn't enough information in the defect report (or in the room) to know what went wrong, skip the defect and move on. If two people disagree about the cause, write both causes down (on separate Post-its) and move on.

After practicing on one defect report each, try two each, and then discuss the result. The discussion is important because it familiarizes everyone with the defect reports so that the affinity exercise can go quickly, unencumbered with many "What did you mean by this?" questions.

Once everyone is comfortable with the process, work in half-hour blocks, writing causes for a half hour and then reviewing the results for no more than 10 to 15 minutes. Continue until you've finished all of the defects.

Create the Affinity Diagram of the Causes

An affinity diagram is a collection of groupings of like items. When grouping the items, look for a similarity of "what went wrong", e.g., defects resulting from product complexity or from incomplete design. Do not use feature-based groupings.

Group the Post-its on a whiteboard or on flip-chart paper. If a group of causes gets too large (over a dozen or so), split it up into subgroups (unless the causes really are all identical). The rules for creating affinity diagrams apply:

- Don't talk, or at least minimize talking
- If a note keeps getting moved between two groups, make a copy and put it both places
- Prioritize speed over precision

"Shuffle" the Post-its a bit so that you're not just grouping your own notes.

When you are 99% done (there will always be a few that are impossible to categorize), stop and give titles to each group.

3. Making Improvements

Select Cause Categories of Interest

Choose two or three categories of causes you'd most like to address. You’re not trying to solve all the problems in one attempt, its important to prioritize. You can select based on the size of the category, but you should also consider the severity of the problems caused by the defects in that category as well as the motivation of the team to address each category.

You may also want to weight the defects based on the phase that it was inserted. For example, for defects found in test we use the following weights: requirements 14.25, design 6.25, coding 2.5, documentation 1.0.

Generate Solution Alternatives

For each chosen category, brainstorm possible solutions. Think outside the box, not business as usual. Include solutions that are small and cheap as well as more ambitious ones. Record the solutions on the white board.

Select Solution(s) to Pursue

From each list, identify one or two solutions to pursue. Select solutions based on their "return on investment".

What distinguishes a good proposed solution from a poor one?

- Specificity – it is very clear how to implement a proposed solution. For example:
    - "Better peer reviews" isn't actionable, but "involve users of the module in the peer review is specific
    - "Not just "better testing" but specifies what test cases or types of tests.
- Broader than testing - Expect that 2/3rds of the recommendations would be around design, peer reviews, and communication/training.

Develop and Execute Action Plan

If possible, assign owners and due dates for all selected solutions. Often this will require a team meeting or discussions with the project manager.

The recommendations need to be treated like a project. Establish some form of follow-up to ensure that they staffed and executed. In general, we find that a review about 6 months later with senior management tends to ensure that the recommendations are implemented.

How to do Defect Analysis - On-going Method

The methodology in the preceding section was based on a starting situation. But as an organization gains experience with defect analysis, the developers can begin to record the types of defects and root causes of the defects as they are resolved. We typically record the following information about each defect:

- Phase of the lifecycle when the defect was found
- Category of defect (see Figure 1 below from Robert Grady's book [2])
- Phase of the lifecycle when the defect was introduced
- Module where the defect was introduced
- Root cause of the defect (see appendix)

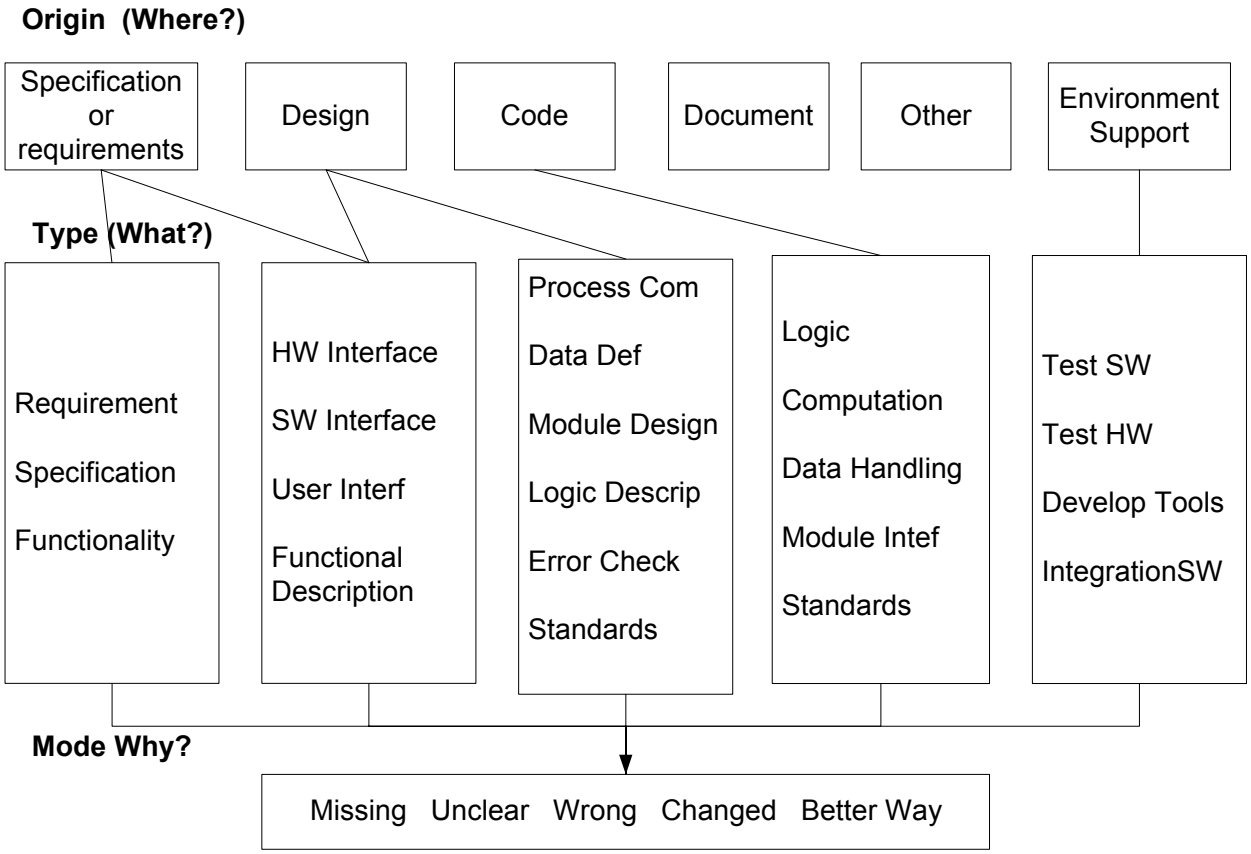


Figure 1 Defect Categories

This allows the 3 day process described above to be reduced to about a ½ day. Half the time should be spent on defect causes and half the time on solutions.

- Review the purpose of the analysis.
- Review the ground rules, e.g., process improvement focus, not blaming.
- Set the schedule for the meeting.
- Discuss the categorization results. Select two or three categorize to address.
- List root causes of the defects in the selected categories. Organize the causes using an affinity diagram.
- Select two or three root causes to address. Select ones which you can reasonably have some control or influence over, and which you have adequate motivation, resources, and expertise to address.
- Brainstorm solutions. Select alternatives to pursue via consensus or a multi-vote process.
- Develop an action plan around the selected solution alternatives. The action plan should include steps, owners, and timeframes.
- Conduct a brief meeting effectiveness review. What went well? What could be improved next time?
- Then follow through on the recommendations, staff them, complete the plan, and review the results with management.

Where to apply defect analysis?

Defect analysis can be done on many types of defects. Most teams start with customer-found defects, but we've also applied it to functional defects found by system test, defects found during testing that should have been found in peer reviews. The Personal Software Process from SEI does defect analysis on an individual's defects found by the compiler or unit test.

Example

One of our teams found that most of their defects occurred in the construction phase (low level design and coding). Packaging scripts were the most frequent type of error, but when weighted by the cost to fix, logic errors were the most costly.

Fig 2 Un-weighted

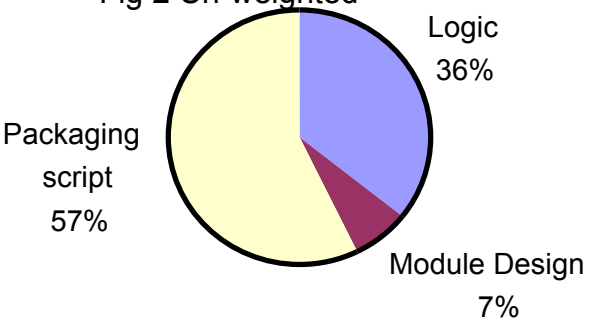
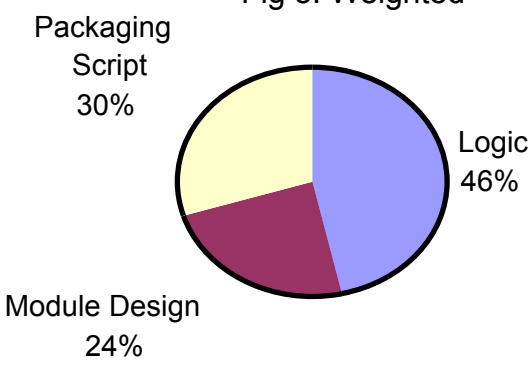


Fig 3. Weighted



They proceeded to identify the root causes of the logic errors and used a fishbone diagram to reach a better understanding.

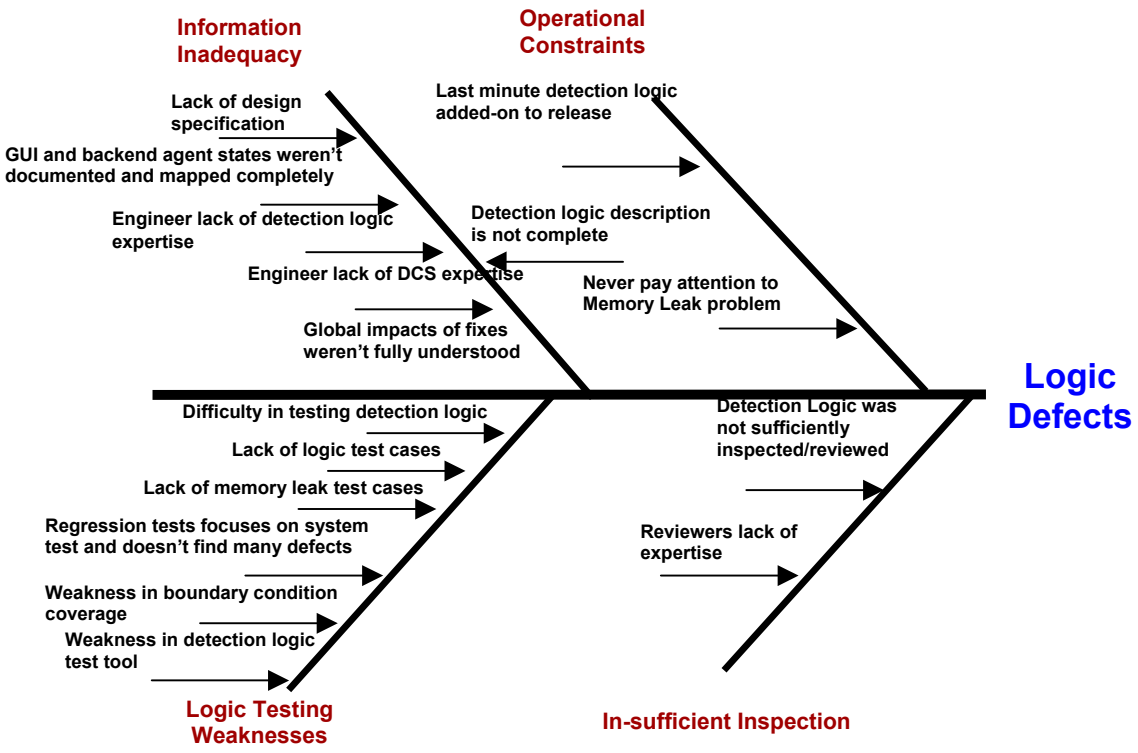


Figure 4: Fishbone Analysis

Then they proceeded to brainstorm solutions to the primary root causes and reached the following recommendations:

1. Ensure the participation of other teams in the inspection/review of our product deliverables
2. Add unit test cases for detection logic
3. Improve regression test with logic test cases
4. Inspect/review/test for memory leaks. Use tool where appropriate

Here are some other typical recommendations that come out of our defect analyses:

- Rewrite module X (which accounted for >70% of the defects)
- Coordinate schedules with a closely related project so that we can do as much jointly as possible
- Establish quality standards for handoffs with a partner we depend upon
- Run module A's tests if module A calls our code, before giving them a new version of our code.
- Involve engineers from the calling module in our peer reviews
- Have a peer review just looking for locking errors (after locking errors were found to be 35% of all defects).
- Enhance our peer review checklist for the most common errors that escape. Assign each reviewer to watch for a set of these errors.
- Provide training on any procedures that are new for this team.

- Install the product like a customer would before doing tests.
- The senior engineer needs to be involved in peer review of defect fixes to look for unintended consequences.
- Improve the code coverage of a module that has low coverage and a high escape rate.

Defect Finding Effectiveness Metric

Key to every process is to have metrics that will drive the correct behavior. Our defect finding effectiveness metric is designed to improve both our customer quality by finding defects before the customer does as well as our productivity by finding defects earlier in the lifecycle. It also drives our organizations to use defect analysis as one of the key ways to improve the metric results.

The metric is defined as the “number and percentage of critical and serious defects found by peer review, developer test, partner/system test, and customers in the last year– measured for every subsystem/product.” A one year time period was chosen to reduce the effects of where projects were in their lifecycle. By using the last year, rather than waiting until the end of the product, we could watch our progress during the project rather than only seeing the results after we were done.

In our environment we have a peer review database that tracks the number of defects found in peer review and a defect tracking system that tracks all defects found after the developer had completed unit testing. The defect submitter can be identified by both their name and project.

One key decision we made was that we would not track defects found and fixed by the developer in unit test. The tracking overhead was considered too high. We do track those found by the developer in regression testing or found after the code has been submitted to the Source Control System.

The metric uses the following definitions:

- "Subsystem Projects" are a set of projects in the defect tracking system that are developing the same 'subsystem'.
- "Associated Projects" to a subsystem are a set of projects that assist the subsystem team with testing before the code is broadly exposed.
- "Developer found defects" are defects where the submitting project is either a subsystem project or one of its associated projects.
- "Partner found defects" are defects where the submitting project is neither a subsystem project nor one of its associated projects and the defect was not submitted by a customer.

We set our goals based on both industry numbers and some of our own best projects. Industry numbers suggest that ‘best in class’ organizations only have a 1% escape ratio to customers. We also had seen several projects where the team had found over 90% of their own defects. So the goals were set:

- 70% peer review found
- 20% developer test found (results in 90% found by team)
- 9% system test or partner found (find 9/10 of the remainder before release)
- 1% customer found (industry “best in class” number)

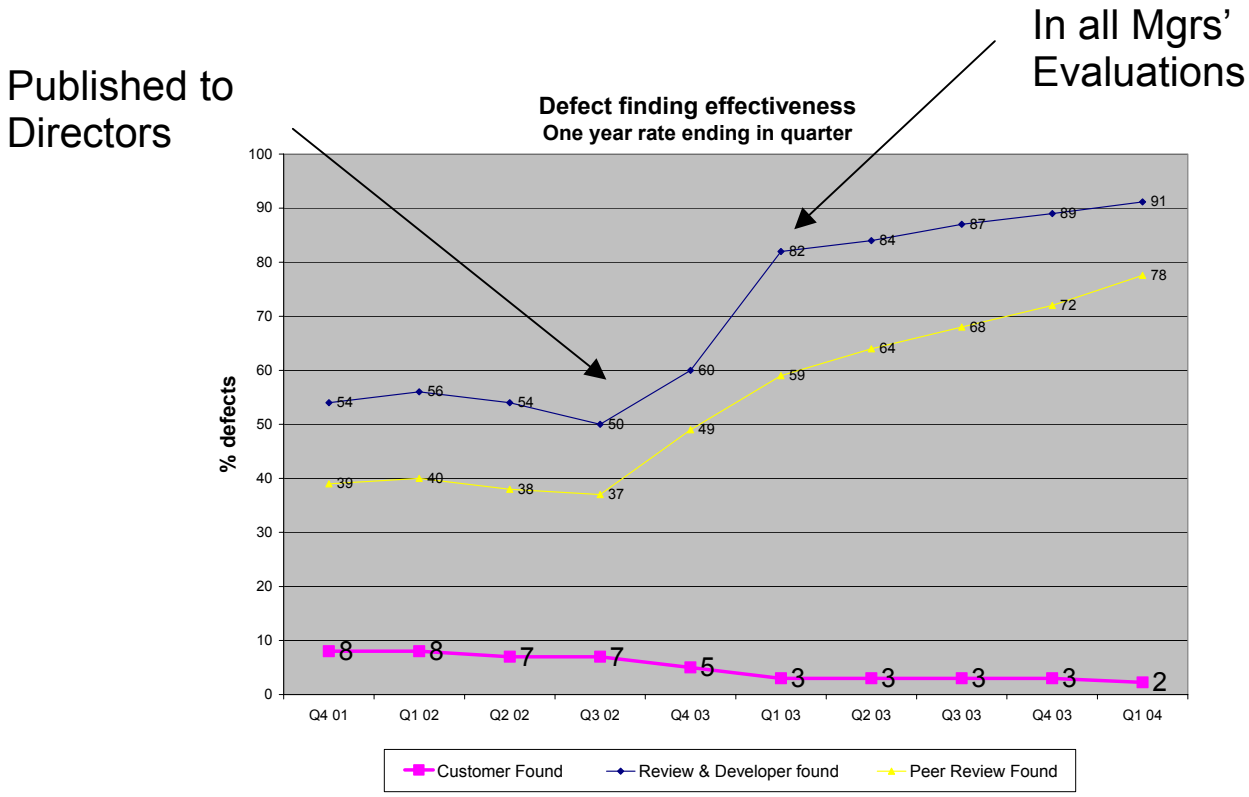


Figure 4 Defect Finding Effectiveness

While we had collected this metric for several years and it had shown modest improvement, notice how once we started publishing this to the lab directors, the metric improved. At first, this was simply the result of more consistent recording of peer reviews in the peer review database. But soon we were seeing more defects being found per hour and more hours spent on peer reviews. This was the result of some of the defect analysis and some of the recommendations for improvement in the peer review process. The reduction in customer found defects has not only been as a percentage, but actually about a 4X reduction in the number submitted each quarter.

Results

Our results show that defect analysis is one of the key methods that contribute to higher quality and productivity. By finding defects earlier, the time to market is reduced and fewer defects are found during the test phase of a project. We are seeing reductions of

75-85% in defects, productivity increases of 30%, and test schedule reductions of 25-50%.

[1] Management Commitment to Quality Requires Measures, John Balza, PNSQC 2003 Proceedings
[2] Practical Software Metrics for Project Management and Process Improvements, Robert B. Grady, Prentice Hall p. 140

APPENDIX

ROOT CAUSE CODES

Communications
This category deals with the verbal and written communications of project-specific information between persons directly (or indirectly) involved in the project. The documents that are produced for the project and how information is disseminated to the team members are dealt with in this category. A document that defines the interface of two modules is a project specific document. Defining how information will be shared within a project is also project specific.

ID	Issue
C1	Unclear requirements document
C2	Unclear design/architecture document
C3	Unclear industry standard
C4	Misunderstanding of the industry standard
C5	Misunderstanding of the requirements
C6	Misunderstanding of the design/architecture document
C7	Misunderstanding between systems engineering and developers
C8	Misunderstanding between developers
C9	Misunderstanding of customer needs
C10	Making product changes without informing appropriate people
C11	Project documentation exists but people are not aware of it
C12	Project information not disseminated

C13	Project information was provided but difficult to realize relevance
C14	Process or methodology information not disseminated
C15	Process or methodology information was provided but difficult to realize relevance
C16	Undocumented information was needed
C0	Other - see elaboration

Education and training
This category deals with the acquisition of knowledge and skills that are needed to do the job. This knowledge or skill is obtained through courses, videotapes, and workshops or through on the job experience. It also may be information about a previous project that is needed to understand the current development project.

ID	Issue
E1	Lack of knowledge in programming language
E2	Lack of knowledge in the use of a specific tool
E3	Lack of knowledge of methodology to be followed
E4	Lack of knowledge about base system
E5	Lack of training in a necessary skill
E6	Training about previous release not available
E7	Training available but not taken
E0	Others, see elaboration

Oversight
Problems that tend to fall into this category are due to carelessness or due to common, simple errors. Attention to detail will usually prevent these types of problems. Checklists could be used in the development process to prevent these types of errors from being introduced into the process. A problem due to editing a file or one due to overlooking some aspect of a logical flow diagram falls into this category.

ID	Issue
O1	Typo
O2	Transcription error

O3	Did not consider all cases from previous phase
O4	Did not consider all cases in current phase
O5	Did not consider boundary conditions or limits
O6	Did not think something applied
O7	Forgot to implement a specific change
O8	Forgot to check impact of change I made
O9	Forgot to remove debug statements
O10	Forgot to do steps in the methodology
O11	Forgot to follow checklist
O0	Other, see elaboration

Project Methodology

This category deals with documents or procedures that guide the project team members through the development process. This information should be independent of people. This means that the general methodology could be adopted by anyone, independent of their background, and the same "standard" output will be produced. The project methodology is adaptable and evolves over time. The project team members must realize that the methodology evolves and is a part of the environment.

ID	**Issue**
P1	Coding standards lacking or insufficient
P2	Requirements standards lacking or insufficient
P3	Design standards are lacking or insufficient
P4	Inspection standards are lacking or insufficient
P5	Checklist lacking or insufficient
P6	Systems architecture standards are lacking or insufficient
P7	Missing steps in methodology
P8	Methodology not sufficient enough to coordinate changes

P9 Steps in methodology were too complex to do correctly

P10 Purposefully did not follow methodology

P11 Steps were manual, but should have been automated

P12 Tool inadequacy

P0 Other, see elaboration

Resources and Planning

Problems are classified as “resources and planning” if the original problems were caused by a lack of resources, up front planning, and/or assignment. Changes in priorities from a project point of view also fall into this category. Management is responsible for resource allocation as well as monitoring and controlling the development process. Both management and staff are responsible for up front planning and setting realistic schedules

ID	Issue
R1	Lack of resources, time, people, or equipment
R2	Task was not assigned
R3	Change in priorities forcing premature delivery
R4	Insufficient planning or scheduling of work items
R5	Not enough resources or time were originally allocated
R6	Reallocation of resources was not done
R7	Inadequate resources or time to respond to changes in inputs (e.g. requirements changed after design finished)
R8	Mismatch of assignments and skills
R0	Other, see elaboration

Effective Team Practices for Facilitative Leaders: Creating and Sustaining Project Momentum

Diana Larsen
FutureWorks Consulting, LLC
dlarsen@futureworksconsulting.com

Identified as a "benchmark" consultant by clients and an exceptional facilitator by colleagues, Diana Larsen works in partnership with leaders of software development and other technical groups to strengthen their ability to create and maintain company culture, manage change and improve project performance.

Drawing on over 15 years of leadership, consulting and facilitation experience, Diana serves as a coach, consultant and facilitator to senior and middle managers, development teams and others in a project community. She conducts readiness assessments and facilitates processes (including project chartering and retrospectives) that support and sustain change initiatives, attain effective team performance and retain organizational learning. Diana speaks at conferences, authors articles on topics affecting project teams, team leadership and organizational change and teaches others to facilitate group processes through Oregon Graduate Institute. Diana also serves on the board of directors of the Agile Alliance.

She earned a BA in Organizational Communication from Lewis and Clark College and has continued her education, studying organizational development, behavior and theory; management and leadership development; team development and group relations; adult learning and adult development theory; instructional design; project management; and agile software development methods. She is a certified Scrum Master and a qualified administrator of the Singer-Loomis Type Deployment Inventory.

Diana is a partner in FutureWorks Consulting LLC and collaborates closely with an extended network of consultant colleagues.

Abstract
With demands for faster delivery and ever more complex applications, software development increasingly must respond with a coordinated team effort. Managers and functional leads need to take on facilitative leadership of the whole team rather than command and control direction of individual contributors. Effective managers don't just assign work and track progress; they pay attention to team development. This paper covers six practices to focus, energize and sustain productivity on your software development team, as well as set the stage for continuous improvement.

©2004 Diana Larsen. All rights reserved. Permission to reproduce in with full text is granted with author's contact information and copyright data retained in the document.

With demands for faster delivery and ever more complex applications, software development increasingly must respond with a coordinated team effort. Managers and functional leads need to take on facilitative leadership of the whole team rather than command and control direction of individual contributors. Effective managers don't just assign work and track progress; they pay attention to team development.

Six practices that focus, energize and sustain productivity on your software development team, as well as set the stage for continuous improvement, include: chartering the project, defining the decision rules, brief daily (or regular and frequent) status meetings, interim retrospectives, rituals for including new members or other significant team transitions, and, finally, a retrospective to conclude each project. These six practices promote team empowerment and increase the chances of the project manager and team delivering a successful result, though paradoxically the project manager may seem less rigidly "in control" of the outcome.

Chartering. Begin each project with a chartering process that includes the project team and representatives from each significant stakeholder group in the project outcome. Through this process, develop a draft charter for the project that includes the following elements: vision, purpose, principles, community, metrics, boundaries, resources and working agreements. The project sponsor and project manager propose a straw charter for discussion and revision by the chartering group. Revising the straw document confers ownership of the project (also known as "buy-in") to the team and other stakeholders and solidifies commitment to achieving the project's purpose.

Decision Rules. As a part of the working agreement discussion, define the various roles and responsibilities on the team including clear understandings of how different kinds of decisions will be determined. Many project managers, and teams, make the mistake of having only one kind of decision-making rule, usually choosing either autocratic or consensus. Neither is appropriate most of the time and both can lead to wasted time and inefficiency. The eight-question decision-tree model proposed by Vroom, Yetton and Jago serves as a helpful guide for determining which decisions are best made autocratically, which decisions a leader makes after consulting with other members of the project community, and which decisions the team should make on its own. Further, the best person to make an autocratic decision may not be the manager but an individual team member with the greatest understanding of the problem and possible solutions. Some decisions can be made by a simple rule, as in flipping a coin. Other team decisions require voting or working toward consensus. Defining the decision rules in advance, before any decision is needed, saves time that can be otherwise lost to misunderstandings, miscommunication and resentment.

Daily Status Meetings. A technique borrowed from Scrum, which borrowed it from many effective teams, the brief daily meeting is a stellar communication practice for any project. Members often stand-up throughout the meeting which tends to ensure its brevity. The goal is a meeting that lasts no longer than 15 minutes. Each member of the team answers three or four questions about the status of their work and the obstacles they need removed and/or resources they will require. If a daily meeting is not possible in your organizational environment, then strive for the greatest frequency you can manage and, whatever frequency you choose, make sure they occur regularly and consistently. This practice promotes joint ownership of the project tasks as well as the kind of camaraderie that leads to a truly synergistic team – one in which the contributions of the whole exceed the individual contributions of the members.

Interim Retrospectives. Find regular checkpoints or milestones during the project to pause briefly and reflect on how things are going. Incremental, iterative projects have the advantage of clear demarcations, but reasonable pause points exist in any type of project life cycle. Use this pause to ask simple questions, such as, "What worked well during this increment?" (WWW) and, "What would we like to do differently?" (DD). An interim retrospective may be as short as thirty minutes or as long as half a day, depending on the length of time and degree of project complexity it covers. Teams that employ extreme programming practices use their iteration retrospective to segue into planning for the next iteration. In addition, interim retrospectives can focus on specific topics or issues that are important to the team.

Inclusion/Transition Rituals. Well-functioning teams often face the crisis of losing or gaining members or other types of organizational changes that impact the team. These transitions can cause a severe drop in productivity if not handled effectively. Smart project managers have a plan for formally including new members as part of the team. Take the time as a whole team to get acquainted and bring the new members up to speed. For example, this is a good moment to pause to review and update the charter, for instance. Other transitions may result from organizational changes that cost the team. They may lose members, committed sponsorship, resources or customers. Some companies have created "funerals" to mark such losses and acknowledge the feelings of grief that can follow.

Project Retrospectives. Unlike a simple project review, the end-of-project retrospective not only captures the lessons learned and best practices from the project just ended, but goes further to seek ways to ensure that those lessons are carried forward as organizational learning on all future projects. Good project managers *never* use a project retrospective to assign blame or confront perceived poor performance. A well-run project retrospective conveys excitement and momentum from the project just ended into the next project just beginning. Instituting the practice of regular project retrospectives creates a climate of on-going, continuous process improvement that leads to well-considered, calculated risk-taking and experimentation that fosters effective innovation.

Bonus Practice: Continuous Learning. I promised six practices and have delivered six practices in this paper; however, I am also offering a bonus practice. Chartering, Decision Rules, Daily Status Meetings, Interim Retrospectives, Inclusion/Transition Rituals and Project Retrospectives all contribute to continuous organizational and team learning. In addition, effective managers look for ways to foster continuous individual learning on their projects, setting aside time for team members to explore new ideas or try new methods. Some teams have instituted mini-sabbaticals; e.g. everyone gets one or two hours a week or a day or two every month to pursue their individual interests wherever they may lead. The manager and the team plan this time into the workload. At the very least, team members' learning can be supported by access to materials and resources to help them follow their curiosities and interests. Where possible they are encouraged to share what they learn with the rest of the team at a brown-bag lunch and learn or other casual forum. Far from being unproductive time, most teams discover that this undirected learning time delivers more energized and productive members.

Bibliography

Catell, Robert, Kenny Moore and Glen Rifkin. *The CEO and The Monk.* Wiley. 2004

Derby, Esther and Diana Larsen. "You're Still Needed." *Software Development*. August, 2004 III. "Immunizing Against Predictable Project Failures." *STQE/Better Software*. January/February 2001, vol. 3, issue 1.
Katzenbach, Jon R. and Douglas K. Smith. *The Wisdom of Teams*. Harvard Business School Press. 1993.
Kerth, Norman L. *Project Retrospectives: A Handbook for Team Reviews*. Dorset House, 2001.
Larsen, Diana. "Embracing Change: A Retrospective," *Cutter IT Journal*. February 2003. Vol. 16, No. 2.
Rising, Linda and Esther Derby. "Singing the Songs of Project Experience: Patterns and Retrospectives." *Cutter IT Journal*, September 2003, vol 16. no. 9.
Schwaber, Ken and Mike Beedle. *Agile Development with Scrum*. Prentice Hall. 2002.
Vroom, V. H., & Jago, A. G. *The New Leadership: Managing Participation in Organizations.* Prentice Hall. 1988.

Are You Hiring Yesterday's Testers?

© 2004 Johanna Rothman

This paper was originally published as "No More Second Class Testers!" in *Better Software*, January 2004.

I've been part of the software industry for over 25 years. I can remember when we didn't have testers. If we were lucky, we had systems engineers, or requirements analysts, but when I first started working, I worked primarily on projects with developers and project managers.

That worked—until the systems we developed were beyond our ability to verify in our project teams. By the time we needed external people to verify our work, testing became a profession. Certainly, even 25 years ago, we had professional testers, but testing didn't become a necessary part of a software project until sometime in the last 20 years.

When I first transitioned into testing from development, I took a developer's approach to testing. I developed automated tests from the API, and tested the user interface last. Even though I was testing operating systems and networks, the systems were smaller and less complex than many of the systems we develop today.

I have the opportunity to meet many development and test groups in my work, and to review their products and product development practices. I've noticed that the largest systems today tend to be larger and more complex than the largest systems of ten years ago, which were larger than the largest systems of twenty years ago. Even the moderately sized systems are technically more complex.

Large and/or complex systems tend to be disproportionately more difficult to develop and to test than the more simple systems I helped develop and test years ago. That means that the people we hire for development and testing need to understand those complex systems.

When I meet these development and test groups, some developers and managers take me aside and say, "We have world-class developers. But our testers are second-class." I hate hearing that. Too often, it's true—and, it's not the testers' fault.

Too often, the people hiring testers have an outdated tester paradigm: "The testers can manually test everything and tell us where the bugs are." But that's not an effective or necessary use of testers, and doesn't help managers know how to hire appropriate testers for their product.

We use people as testers because testing a complex system is just as difficult as creating a complex system. Just creating unit tests that test each path or object individually is not sufficient testing for a complex system. Sometimes, when the answers can be different, even with similar input data, such as in imaging systems, we need not only technical testers but also product experts to test the system. Sometimes, we need people who understand the design of the system, even if they don't have any coding background. Sometimes, we need fabulous exploratory testers, people who want to see how they can break the software.

And, we need testers who can develop maintainable automated tests. Because the larger the system, the more likely the system will hang around for more years than anyone can

believe, and automated regression tests that don't require much maintenance save you money. If the testers don't have the technical background or technical skills to test the system with the help of computers, they test manually. Manual tests are not easy to use as regression tests.

If you don't have access to testers with a wide variety of skills, your testing paradigm hasn't shifted from how we could get away with testing more simple systems to testing the larger more complex systems we have today. The old testing paradigm says it's ok to hire people who can't keep up with developers, people who may be second-class developers or technical wannabes, not first-class testers. Today's complex products demand a new testing paradigm, one that says we need people who understand enough about the product to supply feedback during the entire project.

Different types of developers have different skills. Expert GUI developers have different technical skills than expert network developers who have different skills than expert architects or expert database designers. We may lump them all together as "developers," but most organizations separate the developers by function. However, only the most sophisticated organizations associate specific appropriately skilled testers with each development function.

If you don't associate testers with each development function, or think that testers exist only to find defects, you're not receiving the full potential value of your testers. And the testers you do hire won't be able to keep up with the developers. You'll be creating a second-class test group.

Define the testers' value to your organization

Testers provide value by more than finding and reporting defects; testers provide value to the organization by supplying information about the product. Sometimes the information is test results, defect reports, or data about the system's performance. Sometimes, the information is feedback about requirements or design. The more information your testers provide to the developers, the requirements people, the writers, and anyone else involved in product development, the more valuable they are.

When I work with organizations, here's a test I use to see if their testers are second-class, compared to the developers:

	Yes	No
Are your testers routinely excluded from requirements or design meetings?		
Do your testers have to resort to eavesdropping to hear information about the product?		
Are your testers' requests for tools postponed or ignored?		
Are product testability requirements postponed or ignored?		
Is the testers' per-person training budget significantly less than the developers' budget?		
Are all your testers interchangeable, i.e. they have such similar skills it doesn't matter who you assign to which products?		

Do your testers only work with developers on the code after the product is built, either because they're not brought into the project early enough to work with requirements and design or because they don't know enough about requirements and design to supply feedback?		

If you answered yes to even half of these questions, your testers *are* second-class. They are excluded from key discussions and prevented from obtaining the tools and product expertise they need to do their jobs. Why? In my experience, this occurs when the testers don't know enough about the technical side of testing, don't have the knowledge or skills to test the product adequately, and management is afraid to "waste" money hiring people who have other expertise, because the managers can't perceive an adequate return on their investment.

Not all your testers need to be technical testers. A colleague who tests ultrasound equipment still needs radiologists to test changes to the software, because he has to make sure the software's interpretation of non-deterministic algorithms and systems are correct. He doesn't use radiologists as his only testers; the radiologists can gather enough information about the system for this colleague to manage the risks of not knowing if the changed image interpretation is correct.

Too often I see test groups staffed with smart people — without testing expertise — or people who are experts in the software's problem domain, such as previous customers. Smart people who can't learn enough product details or test techniques cannot speak the same language as the developers, leading to miscommunication or no communication. A test group of customer-focused expert manual black box testers cannot deliver the information the project requires from a test group. In either case, these test groups are not able to adequately assess the product risks, manage the testing time, or determine which tests are necessary.

If you're not sure if your test group supplies sufficient value to the organization, or if your hiring paradigm has kept up with your product, ask yourself these questions:

1. Did your testers find your product's most atrocious problems before release?
2. Do the developers enjoy working with the testers?
3. Do the testers have specific risk and product information they can share at each project team meeting? Are your testers satisfied they have tested enough of the product so that it's not too risky to release?
4. Do the testers accurately predict the time they need for testing and explain why?
5. Do the testers understand that releasing the product is a business decision, and their job is to explain the risks of release?
6. Based on test results, do your projects complete on time? Can you predict you're the project's need for rework based on your testing and results?

If you can't answer yes to all of these questions, your testers aren't a first-class testing organization. There are several reasons your testers don't provide this value to the organization. Your test group might be:

- A homogeneous group, with all the same talents, such as manual black box testers or users with no technical expertise.
- Second-class developers, people who can't make it in development, so someone transferred them to testing. If these people don't have test expertise, they won't be valuable testers.
- Untrained in some aspects of testing that your product requires. For example, if you hired testers who understand functional requirements-based testing, they may not know how to design and perform end-to-end testing (testing that follows data throughout the system, from front to middle to back).
- Unknowledgeable about some of the tools they could use to test the product better or faster.

There are as many reasons for your testers to not supply enough value to the organization as there are testers.

Understand the problems of second-class testers

So, what's the problem with second-class testers? Isn't that just the way of the world? After all, what's testing except a place to put developers who can't quite cut it, or not-so-technical people who can't make it in marketing, or some other place to put people who don't quite fit, but we're sure we need to have.

Brpht. Wrong-o. Testers fulfill an important and critical role in product development:

> Testers provide information about the product under development and test.

To reach this goal, testers exercise different skills than developers.

Testers have at least three unique sets of customers: the developers; the product's users; and the company's management, the people who make release decisions. The developers are customers of the testers' information, and the product's users receive either a better product or a support group that knows the problems in the product, as a benefit of the testers' ability to test the product well. When you have first-class testers, management receives substantial information during the project so they can assess the risk of release.

First-class testers are sufficiently creative to assess the design and architecture of the system before the code is written. While the code is under construction, these testers design and implement their testing harnesses, both automated and manual, creating tests that stress the system in ways the developers do not expect. The testers can measure what they've tested and assess the risk of what they've tested and to know if they've tested enough of the system to help you understand the risks of product release.

Aside from providing information about the product under test, testers interact with the developers, altering the way the developers create the product.

1. When the testers understand the product and find problems early, the developers tend to be more interested in creating a product with fewer defects for the testers to find. Why? Because the testers are their peers, and peer recognition (peers recognizing that we perform superior work) is a significant motivator for developers and other technical staff.

2. When testers develop appropriate tests that detect more problems early, the developers have more flexibility in choosing how and when to fix them. When testers can't detect problems early, the developers are faced with the dilemma of having to choose which one or two of ten significant problems found in the last scheduled week of the project they should fix. No matter what they choose, the developers will be unhappy with their result.

You might think that because developers want to be proud of their products, they would look for problems early and fix them early. Many of the developers I've met do. However, the developers are not testers. They can't detect all the problems in their work products. Developers can't see their own defects. And the more complex and larger the system, the less likely the developers can see their defects.

When testers help developers see their problems early, the developers are more likely to include the testers if other requirements and design discussions. And, the developers are much more likely to build testability into the product by defining APIs or other hooks for testing.

Create a first-class test group

We don't live in a perfect world, but one way to improve your test group's skill is to imagine what a test group would look like in a perfect world, and then to hire or train against those requirements:

- What would your testers do, and when?
- What knowledge do your tester require?
- How can you assess tester knowledge, especially during hiring?
- If you had to hire people without some of the capabilities, how would you choose among capabilities?

What would your testers do?

Testers don't just test the product; testers can fulfill numerous other roles on the project. I've used testers as project coordinators, as technical review readers and moderators, as part of a design team, and as testware developers.

When I staff a test group, I hire a majority of people who can acquire solution-space expertise—deep knowledge about the architecture and design of the product— who can use a variety of test techniques, including automation. Developers can't adequately test their own code. Even when they think they have "fully" tested their code, only 55%-60% of the logic has been tested [Glass]. Even if you don't believe the assertion that systematic black box testing tests less than 30% of the code [Jones], black box testing can't test everything. So even with "full" developer and black box testing, there's a ton of code that we don't know anything about—and we know nothing about the code the developers forgot to insert (missing logic).

Think about your product. What sorts of testing does your product require? If you're not familiar with the internal design of your product, ask the developers. I don't hire testers who don't already know about or have the ability to learn about these kinds of testing:

boundary condition testing, equivalence partitioning, combinatorial testing, exploratory testing, and testing the product from start to end, not just testing requirement-by-requirement. I can train people on test techniques if they have the ability to understand the product design or look into the code and read it. I can't train people on the kinds of test techniques I want them to perform without their ability to understand product architecture and design, or their ability to read code.

Kaner [Kaner] suggests that product experts are a valid component of your test group. For many complex products, he's correct. Unfortunately, I've met many test teams comprised of only expert product users (which is not Kaner's intent)—and those teams are second-class. Expert users who test in conjunction with more technical testers can be extremely effective. Expert users alone, or even worse, manual black box testers who don't know how the system is developed or how the users use the system are insufficient to test the product.

Since our systems today tend to be larger and more complex than even just five or ten years ago, it's worth your while to think about the talents and skills you require in your testers now.

I worked with testers working on an insurance system a few years ago. The testers were testing completely from the GUI, even though it would have been faster, easier, and cheaper to check the answers if they'd tested primarily from the product's API (Application Program Interface). The developers hadn't created a formal API, because the testers didn't know enough about testing to ask for one. The testers didn't understand the architecture of the product, and they couldn't read or write code. The testers had no internal system expertise, although they did understand most of the ins and outs of the insurance system. The testers were not expert users however; the field people regularly called in explaining why something that had worked in the previous release was now broken. The testers' inability to create tests quickly enough to provide feedback to the developers was part of the reason that group's projects were late and the customers found many defects.

You could argue that the developers should have created the API as a reasonable development practice, and I would agree with you. But, if the organization had hired appropriately technical testers, they would have insisted on the API early in the product's lifetime, and the developers would have been happy to define an API for faster testing feedback.

Expert users weren't necessary to test this insurance system. In fact, having smart people who thought they were expert users did not help them, the testers, find problems faster. Their expertise slowed the testing down, because they didn't understand anything about the product except for a superficial knowledge of insurance. Without testing expertise, the testing took longer and found fewer problems.

Contrast the insurance system with the ultrasound system. Because image interpretation is a part of the product, expert users are needed for sufficient testing. The system is too complex to list all the test cases and run through them, because the system requires an infinite number of test cases to "completely" test. No one can completely test the ultrasound system, but expert users can help reduce the risk of releasing with defects obvious to expert users.

What knowledge do your testers require?

Outwardly, the insurance system testers and ultrasound testers look as if they're performing the same work: planning the testing, writing tests, and running tests. However, the knowledge they require and use is different.

I assess four areas of technical expertise when assessing technical expertise:

Functional knowledge	How well the person knows the technical part of their job. For example, I'll assess how well the tester knows how to test (which test techniques), or how well the developer knows how to develop software (design and debugging techniques).
Product domain expertise	How well the person applies their functional knowledge to the product? I'll assess how well the person has learned other products in the past, or if the person can explain how they chose which techniques to use when, depending on the product. I'll also look for ideas about how quickly the person can learn the product (solution-space).
Tool/technology experience	How well the person understands the tools used in or available for our environment. People who already have tool or technology experience can be more productive more quickly than people without. While this is important to discover, I have not found tool or technology experience to be a useful discriminating factor when assess what my project staff knows.
Industry expertise	How well the person understands what the customers of this particular industry expect, and how well can they articulate that knowledge, and apply it to their functional work. Does the person understand the problem-space?

The larger or more complex products requires testers with more knowledge of different test techniques, and testers who can learn how the product works, so they can apply those techniques. Depending on your industry, you may need testers who understand the industry. And, depending on how much you're willing to automate the testing, the testers may need to understand your internal tools and any test tools you use.

If you hire manual black box testers without significant functional test knowledge, who can't learn the product internals easily, then you're not hiring appropriate technical people for testing. You're hiring second-class testers.

For less complex products, you may be able to rely more on testers who have industry expertise. However, if you don't also have testers who understand testing (high functional skill), then you may not be testing as much of the product as you think you are. If you create a test group composed entirely of testers with industry expertise, even for these products, then you have created an entire group of second-class testers, and your testing will be woefully inadequate.

Tool or technology expertise is easily taught and learned by technical people. However, tools and technology don't help testers create good tests quickly, tests that go to the heart of the product and help the developers see what they've accomplished.

How can you detect the tester's abilities?

If you're hiring a test team, part of your problem is knowing how to assess a candidate's abilities. Aside from behavior-description interviewing (questions that help you understand how understand how a candidate has performed in previous jobs), if you want testers who can develop testware (software that helps them test), give them the same audition that you give developers. (If you're not auditioning your technical staff now, you're missing out on a tremendous technique for assessing the candidate's adaptability to your environment.)

Behavior-description questions are open-ended, and ask the candidate to tell you the story how they've worked in the past. An open-ended question is "Tell me about your testing experience." A behavior-description question is, "Tell me about a time you tested a product from the API…"

With auditions, candidates literally show you how they work. When you define an audition, first you define the behaviors you require in a position, and then create an audition, using your products or open source products to see the person at work.

If you aren't looking for people who can write code, but who can read code and then apply different testing techniques, create an audition using your code, and ask them to discuss the testing techniques they would choose and how to apply those techniques to the code.

Or, you may be looking for people who have significant test technique knowledge, who can understand the product's architecture and design, to choose and apply appropriate test techniques. You can ask those candidates to sit in on a design discussion, or lead a design discussion, and then ask how they would test the product, based on the design.

If you're interested in risk assessment, ask the tester to discuss some of their concerns on a project in the past and what they did about it. Show the tester some of your problems and ask how the tester would explain the risk of release to a developer or project manager.

How do you choose tradeoffs among the testers?

Not all the testers you interview will be appropriate for your testing jobs. Some of them will know more about the product internals, some will know more about the industry. I use quantitative and qualitative measurements as part of my analysis for making tradeoffs:

1. What was our defect escape rate out of system test? How many outrageous defects did our customers detect? Could I have prevented that with different testing, or applying different test techniques earlier? If different test techniques or more inside knowledge is required, I'll look for candidates with higher functional testing skills and the ability to look inside the product. If more product expertise is required, I'll look for experts in the product domain.

2. How adaptable is this candidate? Can the candidate understand how to test the different products or different pieces of our products? I want to consider the tradeoffs between expert users and testers with broader knowledge.
3. How much automated regression test development do I think we need? Is the system written in so that we can automate below the GUI? If not, what can we do below the GUI? (GUI-based testing means everything has to be manual, because the GUI changes until the moment the product ships.) For automation, I'll look at people who understand test techniques (functional skills), people who can learn the product quickly, and people who know how to use some test or development tools.

Summary

Testing is a critical part of software product development, and the test group is not a place for second- or third-rate developers, or for people who can't learn enough about the product to be useful to you. Properly done, testing will reduce your cost to market, the risks of releasing with outrageous defects, and the cost of ongoing maintenance.

If you're already managing a test group, think about where your testers are successful and where they are not successful. Supplement the staff you have with other talent when it's time for you to hire, and make sure you have auditions or other interviewing techniques to hire the kinds of testers you need.

Testers don't have to be second-class in your organization. Hire and train your staff appropriately so that they fulfill the needs the organization requires.

Acknowledgements:

I thank all the people who read and commented on this article: Jim Batterson, Michael Bolton, Laurent Bossavit, James Bullock, Dennis Cadena, Esther Derby, Bill Hamaker, Elisabeth Hendrickson, Sherry Heinze, Scott Killops, Bob Lee, Dave Liebrich, Sue Petersen, Dwayne Phillips, Keith Ray, Jay Sen, Shannon Severance, Jerry Weinberg.

References

Glass, Robert L. *Facts and Fallacies of Software Engineering*, Addison Wesley, Boston, 2003.

Jones, Capers T. *Assessment and Control of Software Risks*, Yourdon Press, Englewood Cliffs, NJ. 1994.

Kaner, Cem. *Recruiting Software Testers*, Software Development, December 1999.

Just Unit Testing

Case Studies and Examples

Brian G. Smith
Xmcell.com

Richard P. Vireday
Intel Corporation

Abstract

This paper came about for the simple reason that we had to test some new software we had not developed. This software (the Intel Transport Interface) was a library with a public API, and was to be part of a kit included with pre-production systems. Digging into the system, we found that the developer did not have any unit tests, so we wrote some as part of our test plan. We decided to use an Open Source framework and Mock Objects. Along the way, we found some interesting problems using the most popular C++ Unit Test framework (CppUnit), and that organizational problems can be a more significant issue than we had realized. Ever-changing market trends, a vast selection of business technologies, and process theory inundation all create environments that must be adapted to quickly and impacted the decisions we made. For instance, we have learned to downplay much of the latest process terminology, not necessarily thinking about “Extreme” or “Agile”, even though that is what we eventually end up doing. We have not convinced the ITI developer to use unit tests, and doubt we ever will before the project eventually reaches end of life, but the lessons we have learned from this experience are influencing our future development plans.

1 Introduction

Unit testing is viewed with loathing by many development teams. The thinking may be that “talents are wasted by performing such menial activities.” In reality, it is mostly viewed as just plain boring. Testing does not look productive, it is not creative, it doesn’t provide the same quality of excitement as the activity of constructing an operational piece of code. But the basic testing requirements for effective code-based unit testing are very similar to those necessary for, as an example, end-user GUI acceptance testing: understanding enough of the functionality to devise and provide inputs, and then comparing observed outputs to expected or required behavior. In both cases the tester is provided with an interface, whether code-based or graphical, and they are responsible for exercising it.

All software interfaces are “user” interfaces to somebody, although in both GUI and API testing specific skills are required that are not necessary for the other. In one case, sensitivity to the end-user marketability, distinctive look and feel, and to high-level functionality and usability; in the other, coding language syntax, stacks, heaps, registers, pointer management, performance and stability implications for the next user of the interface, and inevitably assembly code.

Complex software can be developed by small teams. All that is needed to build relatively complex and functional software is a good architect, a computer scientist/manager, and a bunch of programmers with a little experience. The average size of development teams both within companies and on Open Source projects around the world seems to be at this small mix. Of course, this assumes that developers are responsible unit testers. Unit testing is essentially an integral part of software development.

The appearance of the Java language, due to its design and popularity, has been largely responsible for the general appearance of modern tools designed for assertion-based unit testing. Junit, the most popular tool for this type of testing, was both conceived by and evolved with the Extreme Programming class of test practitioners and is still a free download from JUnit.org. A hunch is that the original author of JUnit realized that Java’s reflection facility provided a well-suited environment for solving the problems of uninterrupted test suite completion through well-designed exception handling and convenient run-time access to object information. Also, Java’s innate extensibility and polymorphism allow the derivation, from production code objects, of test objects not bound to that code. This means that product functionality can remain untainted by the presence of tests.

As stated in the Abstract, this paper came about for the simple reason that we had to test some new software we had not developed. We decided to use an Open Source framework and Mock Objects. Along the way, we found some interesting problems using the most popular C++ Unit Test framework (CppUnit), and also that organizational problems can be a more significant issue than we had realized. The technology and organizational lessons we have learned from this experience are influencing our future development plans.

2 A Typical Learning Experience

Unit testing is not a new practice. Early efforts have consisted of a variety of adequate solutions but have also suffered from considerable drawbacks. Take, for example, an old ADT programming project in C++, utilizing “print and diff” techniques.

The test program consists of a main which ‘#include’s the header file of the ADT to be exercised.

```
#include <iostream.h>
#include <iomanip.h>
```

```
#include "Date.hpp"

void main()
        {
        cout << "DEFAULT CONSTR (1/1/1950)...\n";
        Date date_a;
        cout << date_a << endl;

        cout << "3-ARG CONSTR (1/13/1969, 12/2/1990, 1/12/2000)...\n";
        Date date_d(1, 12, 2000);
        cout << date_d << endl;
etc…
}
```

which requires overloading the ostream '<<' operator, like so,

```
        ostream& operator <<(ostream& out, const Date d)
                {
                out << d.the_month() << "/"
                         << d.the_day() << "/"
                         << d.the_year();

                return out;
        }
```

and saved to file. As development on the ADT continues, the test program is run at regular intervals where the current output is compared to ("diffed") with the output previously saved to file.

Unfortunately (in this case), the C++ Standard has changed since the program was written. The iostream class is now template-based rather than requiring the class to be overloaded. After attempting to update the test program, alternately weighing the ease of either upgrading to the new API or using stdio utilities (getch, gets, fgets, printf, etc.) it was decided that this is another great argument for assertion-based unit test frameworks. Had the tests for this code been developed based on assert.h functionality, these old stale tests would have almost certainly compiled and run right away today:

```
        Test_date(d, 1, 12, 2000);

        Test_date (Date d, int d, int m, int y){
                Assert(d.the_day() == d);
                Assert(d.the_month() == m);
                Assert(d.the_year()==y);
        }
```

There are surely practitioners today who would say, "I told you so", but assertion-based testing has not always been the most expedient method when attempting to focus on ***developing* the *product***. Assertions, depending on the vendor implementation, throw exceptions which tend to stop the entire program from running, making automation and "continuous testing" time consuming and troublesome.

Another method is to "#ifdef" source files with various sections of "DEBUG" code, but this can lead to unexpected behavior in the final product. While these methods are certainly useful and in some cases necessary, such as with very low-level or primitive languages, such close coupling of product and test code can be dangerous. Most modern object-oriented languages allow loose coupling and allow the development of shipping code that has no build or run-time dependence on invisible test harness scaffolding.

3 More Modern Unit Testing

This section is on two aspects of modern unit testing, providing a comparison of available frameworks and a helpful design pattern. Different types of testing technologies must often be combined together to test any significantly complex software.

3.1 Unit Test Framework Comparison

Unit test frameworks exist for every programming language as 1) commercial packages, 2) free implementations downloaded from the web, and 3) as proprietary ad hoc tools developed by programmers themselves. Deciding which route to take depends to a large degree on the nature of the software under test and the experience of the team.

From the inception of a software development effort, the ability to create objects and data structures that are easy to test by the developer is essential to the efficient verification of the structural integrity of system components. By exercising individual components such as classes, methods/functions, and block-level structures such as loops and statements, component correctness is ensured independent of the larger program.

Our problem was how to go about selecting a tool that would be most appropriate for the job, would integrate into our existing automation infrastructure and would continue be useful going forward. There has been an explosion in the number of different unit test harnesses in recent years. Most are inspired by the JUnit framework style of assertion-based testing.

Which unit test framework should one use? The decision is initially guided by the language under test and by the nature of the API to be exercised. Generally the framework chosen should be written in roughly the same language as the application code being tested. There are times, when using Tcl for instance, that the language used for the bulk of test writing is different. When using Tcl to test a C++ application the framework ultimately consists of a library of functionality written in C which “glues”, through static or dynamic linking, the code under test to the Tcl-controlled wrapper which provides an interpreter for the scripting language.

Most often, however, simple, effective unit test frameworks should be in the same language. This is due to several factors:

1. The developers already know the language
2. You do not need special tools or other support, since the language is already supported and can run on the developers’ machines
3. It can be maintained and developed in parallel with product code.

For a comparison of C/C++ unit test frameworks available freely on the web, we weighed factors such as setup effort, necessary prerequisites (e.g., compiler features or system libraries), and features such as documentation and results reporting. The results of this extended analysis are summarized in the appendix (see Appendix 9.1).

A single constructor for a home-spun date class was used as the test target; each suite we created with a unit test framework was required to test only the default constructor by verifying that 3 data members were successfully created according to the class specification. That is, correct values for the day, month, and year would be 1, 1, and 1950 respectively. If all assertions within the tests pass, the overall suite passes with a simple print to stdout: “OK”. Otherwise, individual results are summarized when the tests are completed (see Appendix 9.2)

3.2 Mock Objects

In general, “mock object” refers to a piece of software, that simulates the presence and/or behavior of a real thing. They are basically ‘stubs’ designed to provide contrived but deterministic output in order to look like and behave like a real piece of software or even hardware. They are generally simple and do not involve the complexities of the object they mimic.

While the date class constructor provided us with a simple constant for comparing unit test frameworks, our experience with the ITI provides a better example to help demonstrate how an assertion-based framework can be used with mock objects for testing more complex systems.

3.3 Master Control

As part of our QA activities, we use a master control framework of PerlF and Regr, discussed in (Vireday2003). There has to be something that controls the overall test suite, and we use these tools for that. Our tasks primarily involve configuring pre-production systems in different ways, and then running tests against them. So for that, our tools are best for us. We can quickly inject them into systems without installation, and get testing.

Most software development teams using Make or Ant tools will have a **'test'** rule. When a component or product is built, regression tests can be run quickly and easily. From the authors' experiences, having a significant set of tests that run after a build helps teams save hours. You can reject non-working builds! You know what has minimal functionality! Every team that spends some amount of time on this finds the payback is crucial to success.

A side, but another significant point is that the developers should be able to run every test, especially on their development system. Due to hardware constraints and other system configuration factors, that obviously is not always possible. But even Databases can be emulated these days, and Mock Objects can help at least let developers check basic functionality. We wish more schools would start teaching test first approaches.

4 ITI Case Study

Recently one of our development teams was in a tight release schedule and our QA group found it necessary to contribute to a significant part of the project. This meant creating a suite of unit tests (normally provided directly by the developer) for the Intel Transport Interface (ITI) - a task our test infrastructure was not immediately ready to handle.

Briefly, the ITI is an API for managing network adapter information and events, allowing applications to know and respond to changes in connectivity or changes in physical location and network transitions (such as unplugging a CAT5 cable in the cubicle and automatically reconnecting via WiFi in the cafeteria). For example, an application detecting such a transition may need to know a different gateway address, change from a 10/100 NIC to 802.1x network interface, and finally change an icon on a real time network topology map. Surprisingly, the correct operation of such a complex piece of software *can* be assured through unit testing.

The final structure of the ITI tests is relatively simple. CxxUnit uses a Perl script to generate .cpp files from CxxUnit-drived classes that the user creates. A single (command line) executable (iti_tester.exe) is then generated by linking the tests and the CxxUnit automation functionality (Appendix 9.4).

An example of one ITI 1.0 API function declaration (from the iti.h header file):

```
ITIPREFIX DLLSPEC ITI_RET ITI_GetNetworkAdapterConnection (
               ITI_ADAPTER_INFO *hAdapter,
               ITI_ADAPTER_INFO_GROUP *hConnectedAdapters,
               iti_uint32 *iSize );
```

For testing the Intel Transport Interface it was necessary to emulate the existence of a network interface card (NIC) in a system. Creating a mock object for that device provided two benefits:

1. The properties of the device were always known (they were created during the instantiation of the object) while real NICs all have different MAC addresses, for instance,

2. The test bed (machine and operating system) did not require external manipulation to stop and start, or shutdown and startup cycles in order to physically replace the card and register it as active. Clearly this enables automation as well.

In sum, the VirtualAdapter provided everything necessary for testing what mattered, independent of the operating system, NIC type or rev, motherboard, etc. A code example for the VirtualAdapter is provided in the appendix (Appendix 9.5).

5 Process and Structure

Adopting a particular process is not the entire answer to successful software development nor is doing so absolutely necessary. Since time constraints and individual responsibilities are not always conducive to process ideals, having an efficient team structure and an organization, each with a variety of capable team members, can help ensure a product's success in even unconventional ways. From our experience, it can be invaluable to have an agile test and integration verification team that is able to contribute to absolutely essential development-level tasks such as software unit testing. Ironically, for this to be possible a modest degree of team autonomy may be required in order for its members to contribute in the most valuable ways.

Despite a great deal of attention given to new development processes in recent years, we still see everything from reluctance to passive hostility from many development teams to adopt simple and potentially rewarding practices such as Unit Testing. Even though Extreme Programming, Agile Testing, The Spiral Development Model and other so-known "lightweight" software development models promise better products, faster time-to-market, and even happier teams, as developers of test automation systems, we understand the reluctance from our own personal experiences.

Developers are too often needlessly burdened by formalities that don't offer the immediate results that draw software programmers to their profession. Too often even lightweight methodologies threaten to become burdensome, time and energy consuming processes with specialized jargon, fan clubs, books, and scientific endeavors that again alienate the very professionals for whom such practices are intended. Named things, whether methodologies (e.g., Extreme, Agile) or techniques (e.g., Mock Objects, Patterns) are definitely useful insofar as they provide new languages to facilitate discussion. Whether the things they name are useful as strict recipes is another question.

We are now taking a different approach that downplays the formality and complexity of new processes and emphasizes the most productive practices, not as an additional responsibility or step, but as a practice inseparable from the overall practice of software development.

Rather than there being one group of especially important Engineers and another group of less important testers, the lines have blurred, the groups have coalesced, people feel more important, and overall quality of work life and work product are beginning to benefit. The reason is that *roles* are changing. Hybridization of skills and responsibilities is occurring. Scientists set up machines… Quality Specialists write code and fix network problems... Developers test… a little variety goes a long way toward a happier, more interesting workplace.

Much of the change is certainly due to increased emphasis on lightweight processes. However, these processes, coinciding with newer ready-to-use reusable software components and many small ventures attempting to realize an idea, have also changed the *shape* of development teams. Smaller teams have resulted naturally in closer collaboration among team members. Consider the start-up boom of the 1990's, largely coinciding with the simultaneous appearance of Java, the WWW explosion, and thousands of technology ventures seeing the possibility for quick profit. Fortunately even sophisticated, highly specialized, and high quality systems can be developed by a very small number of individuals given that good practices are habitual.

6 Conclusions

What we learned from the ITI process was both technical and organizational. There is no doubt that unit tests are ideally developed before or at least in parallel with production code. This

clearly enables the developer, who is naturally the person most familiar with the software's requirements and internal functionality to find problems early, reducing rework and workarounds later in the development cycle. Furthermore, if a test and integration team is delivered software with a high degree of quality, the total cycle of system verification becomes much less expensive. The benefits are clear: shorter development cycle for products (or prototypes in an R&D context) translate directly to faster time-to-market, increased productivity, and competitive edge.

What started out as a simple problem of testing a new product led us down some interesting paths and to the cutting edges of current testing. We found a variety of possible solutions and settled on the one most appropriate for our problem. After our detailed analysis of various unit test harnesses, we chose CxxTest for the following reasons:

- No dependence on RTTI (Runtime Type Information (standardized))
- No dependence on C++ templates
- No dependence on exception handling mechanisms
- No dependence on external libraries for memory management, I/O or graphics

We also realized that tools and processes do not constitute the whole of a successfully solved problem. Combined with a small, carefully selected team, unit testing should be an integral test activity performed in parallel to writing end-product code. And with such a relatively inexpensive "buy-in" of commitment, the ability for professionals dedicated to quality to do their job improves as well. Through the ITI unit testing experience our team proved that quality professionals can contribute to development responsibilities when the product developers could not. We also showed that the practice of unit testing is a crucial aspect of quality-laden programming that can not be ignored.

Not all engineers find themselves working in an ideal environment. Whether you're delivered software that has been pre-tested or not, you still have a job to do. This is really the primary impetus (and emphasis) for this paper. The Quality Assurance professional is responsible for preventing a sub-quality product from becoming a customer deliverable. The cynical adage regarding the tester's conundrum may be true: developers are rewarded for great software, testers are punished for bad software. Don't let this continue. Regardless of the quality of the software thrown on your plate, it must be tested in the correct sequence, that is, bottom-up. The testing process must begin at the "source", even if it happens to be mid-stream.

The popular lightweight processes of today emphasize a chain of responsibility. Unit testing does not necessitate new ideas, proprietary vocabularies, or expensive tools. Developers are responsible for unit testing and there are many easy-to-use tools and techniques for accomplishing this. In some circumstances software products might also include a suite of customer-executable unit tests. While this is definitely becoming a more common practice, it is still not common enough.

In sum, the benefits of unit testing should be obvious. The verification team can be confident of a baseline of initial quality. The end customer can verify that the software they've invested in is working in their environment and are reassured that the company who created this software is confident of their product. Support engineers who are responsible for debugging customer installations benefit from an in-place tool for debugging. Unit testing is totally under the control of individual developers, is independent of distracting trends and inevitably produces positive results that translate into improved productivity and higher quality systems for the lifetime of a product. Furthermore, it is just as fun and creative as utilizing the APIs exposed by Java, .NET, PHP, or any future high level programming languages. In fact, unit testing facilitates the most rewarding outcomes available because the developer delivers more functionality with confidence and quality by doing something *their own way.*

The requirements of our QA group are changing. We understand that quality software begins at the source and believe that in order to expedite this quality unit tests must be used. In the future if no unit tests accompany software deliverables, *somebody* must create them.

7 Acknowledgments

The authors would like to graciously acknowledge the help of our primary reviewer, Julie Fleischer. With her excellent editing and oversight, we rewrote the paper several times - each time better than the previous. Thanks Julie!

8 About the Authors

Brian has a Bachelor of Science in Philosophy and an Associate of Applied Science in Software Engineering Technology. He has been programming in a variety of languages and environments since the early 1990's, has authored various papers on the software development process, and is currently looking for a full-time "permanent" position in the software development industry. He can be reached at brian@xmcell.com.

Richard P Vireday, B.S. and M.S. in Computer Science and Engineering, is a Senior Software Engineer, who has been with Intel Corporation since 1984. He has worked for Intel on a variety of projects. He can be reached at richard.vireday@intel.com.

9 Appendix

9.1 Unit Test Framework Comparison (for C++)

Framework Features	CxxTest	CppUnit	Cunit	Cutest	Cut	Cutee	Check	TUT
Priority (our need)	1	1	2	2	2	3	1	2
Setup/teardown	Yes	Yes	No			No		Yes
Reflection	No	Emulated	No	No	No	No	No	No
Templates	Yes	Yes	Yes	No	No	No	No	Yes
R/T Reporting	Fair					No		
Report summary	Good	Good	Good	Ok	None	OK	OK	
Debugging	Easy	Ok		Ok		No		
Documentation	Good	Good	Ok	Good		Ok		Fair
O/S Support	Any	Any	*nix	Any		Any		Any
Other requirement	Perl		Curses		Python	None		
Exception Based	No	No		No	?	No		Yes
Archive size	124K	706K	35K	11K	20K	8K	35K	28K
Version	3.9.1	1.10.0	11.02.2000	1.4	2.3	0.4.2	0.9.0	2004-03-26
Setup time	45min	45min		15min		30min		
IDE/GUI	No/yes	No/yes	No	No	No	No	No	
Thread safe	?	Possible				N/A		

Note: Data in the table are subjective opinions of some features. We would welcome any corrections or updates. Most of the tools can be found quickly via a WWW search.

9.2 Modern Date class unit testing

```
Test Program:
//DateConstructorTest_CxxUnit.h

#include <cxxtest/TestSuite.h>
#include "date.hpp"
```

```
class dateConstructorTest : public CxxTest::TestSuite
{
        public:
                void testDateConstructorDefault(void) {
                        Date defdate;

                        TS_ASSERT_EQUALS(defdate.the_day(), 1);
                        TS_ASSERT_EQUALS(defdate.the_month(), 1);
                        TS_ASSERT_EQUALS(defdate.the_year(), 1950);
                }
};
```

Test output:
Running 1 test.OK!

9.3 Framework Notes

9.3.1 Some CxxUnit Test Notes

It is often tricky to get started with a new tool, and CxxUnit was too. The README.txt is very basic, but some extended documentation is available in HTML format. Extra efforts are necessary in order to test programs that link with external libraries or, as was the case with the Date class, additional object files. It quickly became necessary to create a batch/Makefile file in order to consistently execute the setup steps in the correct order

```
perl cxxtestgen.pl --error-printer -o %1.cpp %1.h

REM now compile into the executable test
cl /nologo /c -I./  -GX %1.cpp
cl /nologo /c -I./  -GX date.cpp

echo creating %1.exe
link /nologo  /out:%1.exe .\date.obj .\%1.obj
```

Bang for the buck: CxxUnit automates test "coding".

	Iti_tester	CxxUnit test framework	ITI Application
Contents raw LOC (Lines of Code)	cpp, 359 .h, 590 .pl, 186 .bat, 33	.c, 142 .cpp, 305 .h, 5305 Makefile, 309 .pl, 1027	.c, 706 .cpp, 9023 .h, 2217 .rc, 706 .rc2, 26
LOC Totals	1168	7088	12678

For roughly the same amount of code as the library itself, but with a lot generated by the CxxUnit package, we get a full regression test of every function in the library. The entire unit test suite is automated and controlled using our PerlF framework introduced at PNSQC 2003 [Vireday2003].

9.4 Partial ITI Unit Test Suite

```
//iti_connect.h
#include <cxxtest/TestSuite.h>
#include "iti.h"
#include <../iti_vadapter/v_adapter.h>

/*
Focus on ITI Connection operations
```

```
*/

class itiTestSuiteConnections : public CxxTest::TestSuite
{
public:
	VirtualAdapter* m_vaptr;
	ITI_ADAPTER_INFO* m_Aptr;
	ITI_RET m_ret;
	ITI_ADAPTER_INFO_GROUP* m_Gptr;

public:
	void setUp() {
		/*Create an Adapter Group with room for 5 Adapters.
		Virtual Adapters are simply wrappers around ITI_ADAPTER_INFO*/

		//new group
		m_Gptr = new ITI_ADAPTER_INFO_GROUP;

		m_Gptr->version=9;
		m_Gptr->iNumberOfAdapters=0;

		m_Gptr->pAdapterInfo = new ITI_ADAPTER_INFO[5];

		//new va
		m_vaptr = new VirtualAdapter;
		m_addAdapterToGroup(m_vaptr);

		printf("\n#########################\n");
	}

	void tearDown(){
		delete m_vaptr;
		delete m_Gptr;
	}

	void testGetNetworkAdapterConnection_001_101() {
		printf("testGetNetworkAdapterConnection_001_101...\n\n");

		iti_uint32 size;
		//this is the '001'
		printf("sizeof G: %d\n", sizeof(ITI_ADAPTER_INFO_GROUP));
		printf("sizeof A: %d\n", sizeof(ITI_ADAPTER_INFO));
		ITI_RET ret = ITI_GetNetworkAdapterConnection(NULL,NULL,&size);
		printf("size: %d\n", size);
		TS_ASSERT (size > sizeof(ITI_ADAPTER_INFO_GROUP));

		VirtualAdapter* myVA = new VirtualAdapter;
		myVA->setProp_AdapterName((TCHAR*) "Beta");
		myVA->setProp_AdapterType(0);
		m_addAdapterToGroup(myVA);

		//this is the '101'
		//ITI actually validates your adapter group!!!(i.e.,???)
		//So a virtual adapter won't be found via the device comms,
		//and should be INVALID
		size += (m_Gptr->iNumberOfAdapters * sizeof(ITI_ADAPTER_INFO));
		m_Gptr->size = size;
		printf("%d, %d\n",ret,size);
		ret = ITI_GetNetworkAdapterConnection(myVA->getAdapter(),NULL,&size);
		m_printAdapterGroup();

		printf("%d, %d\n",ret,size);
			TS_ASSERT(ret == ITI_INVALID_ADAPTER);
	}

	void testGetNetworkAdapterConnection011() {
		printf("testGetNetworkAdapterConnection_111...\n\n");

		iti_uint32 size;

		ITI_RET ret = ITI_GetNetworkAdapterConnection(NULL,NULL,&size);

		VirtualAdapter* va = new VirtualAdapter;
```

```
                va->setProp_Connected(true);
                va->setProp_DevicePresent(true);
                va->setProp_AdapterName((TCHAR*)"Wacky Willys Adapter World\0");
                va->setProp_AdapterMAC((TCHAR*)"00:03:47:FC:41:04\0");
                va->setProp_AdapterIP((TCHAR*)"123.123.123.123\0");
                va->setProp_AdapterType(0);

                m_addAdapterToGroup(va);
                //111
                ret = ITI_GetNetworkAdapterConnection(va->getAdapter(), m_Gptr, &size);

                m_printAdapterGroup();

                printf("%d, %d\n",ret,size);
                TS_ASSERT (ret == ITI_TRUE);
        }

        void xtestRegisterNewConnection() {
                //see iti_callback.h
        }

        void xtestUnregisterNewConnection() {
                //see iti_callback.h
        }

}; //itiTestSuiteConnections
//iti_connect.h
```

9.5 VirtualAdapter Mock Object definition

```
#ifndef __V_ADAPTER_H__
#define __V_ADAPTER_H__

#include <iti.h>
#include <iostream>
#include <assert.h>

class VirtualAdapter {

                private:
                ITI_ADAPTER_INFO m_vadapter;
                bool m_set;

        public:
                VirtualAdapter()
                        :m_set(false)
                {
                        //initialize
                        TCHAR* str= "new adapter\0";
                        strcpy(m_vadapter.szAdapterName, str);

                        str = "0000\0";
                        strcpy(m_vadapter.szNetConnectionID, str);

                        str = "00.00.00.00.00.00\0";
                        strcpy(m_vadapter.szAdapterMACAddress, str);

                        str = "0.0.0.0\0";
                        strcpy(m_vadapter.szIPAddress, str);

                        m_vadapter.bConnected = 0;
                        m_vadapter.bDevicePresent = true;
                        m_vadapter.iType = ITI_NONE;
                        m_vadapter.iNetConnectionStatus = -1;
                        m_vadapter.iWMIIndex = -1;
                        m_vadapter.iWin32Status = -1;
                        m_vadapter.iRasConnectState = RASCS_OpenPort;

                        str = "raw virtual connection\0";
                        strcpy(m_vadapter.szRasConnectionType, str);
```

```
        m_vadapter.iWin32Index = -1;
}

~VirtualAdapter() {
}

ITI_ADAPTER_INFO* getAdapter() {
        return &m_vadapter;
}
```

[etc… a bunch of set and accessor functions for a v_adapter instance]
};

10 References

[Vireday2003] PerlF, Regr & Other Cheap Testing Tools, Richard Vireday, Arne Bowman, Neel Patel, & Arunarasu Somasundaram, PNSQC 2003 Proceedings, www.pnsqc.org.

More information on some of the topics discussed in the paper can be found at the following web sites.

The New Methodology
http://www.martinfowler.com/articles/newMethodology.html#N400361

Using java.lang.reflect.proxy to Interpose on Java Class methods
http://java.sun.com/developer/technicalArticles/JavaLP/Interposing/

History of Java
http://ei.cs.vt.edu/~wwwbtb/book/chap1/java_hist.html

Inversion of Control
http://www.sys-con.com/story/?storyid=38102&DE=1

Unit Test Tool Survey
http://tejasconsulting.com/open-testware/feature/unit-test-tool-survey.html

Open Source Testing
http://www.opensourcetesting.org/unit_c.php

Junit.org
http://www.junit.org

Approaches to Mocking
http://www.onjava.com/pub/a/onjava/2004/02/11/mocks.html

Java 2 Platform SE v1.3.1
http://java.sun.com/j2se/1.3/docs/api/index.html

Extreme Programming
http://www.extremeprogramming.org/ and http://c2.com

Intercultural Needs Assessment Project

Ms. Balbinder K. Banga
Portland State University (bsaran@yahoo.com)

Bio of Author:

Balbinder K. Banga is a Master's degree candidate in Engineering Management (with an emphasis in Project Management) at Portland State University. She is presently completing her second internship with Intel Corporation. Prior to that, she was a Graduate Research Assistant at Oregon Health Sciences University.

Balbinder has presented process papers at the Digital Government Conference (Los Angeles) and FORREX conference in Vancouver, BC and was also invited to present at the Information Resources Management Association conference in Philadelphia. She has also been a contributor to the following papers: *"Business Models for Business Portals," Knowledge Management: A Reassessment" and "Harvesting information to Sustain our Forests."*

Balbinder also worked as a Project Manager at Pivotal Software and Corillian Corporation. She has her Bachelor's degree in Communications from Simon Fraser University and a diploma in IT. She resides in Portland, Oregon with her husband and two children.

Abstract

The purpose of this paper is to illustrate the process and findings generated by an intercultural needs assessment project at Intel Corporation. As Intel expands globally, the company is aware of the distinct differences in doing business around the world. Being a proactive company, Intel is making a conscious effort to confront and find solutions to issues that it (and many similar high technology companies) are facing as they expand their operations globally

The goal of the Intercultural Needs Assessment project was to discover what intercultural issues employees in the emerging markets of Intel were facing that may be preventing them from doing their jobs effectively. It is important to note that the topic of this paper is to specifically focus on the intercultural issues uncovered. Numerous strength areas were also apparent during the survey however the focus of this paper will be mainly on the issues that were uncovered.

The project gathered the opinions of a statistically significant segment of the population so areas that need attention could be uncovered. The main topic areas of study were leadership, teamwork, communications, work effectiveness and Intel Culture.

The method chosen to uncover the issues was a survey/ interview approach (this paper will trace the survey approach). The survey would identify the areas that need attention (via a statistically significant sample) while the in depth interviews would focus on recommendations and how these areas can be addressed (i.e. via training, working with other geographies, etc).

This paper will begin with a general overview of survey projects and detail different methods for collecting data. The bulk of the paper will walk the reader through the six steps in the Intercultural Needs Assessment project, which are:

1. Planning
2. Target Market
3. Survey Development and Administration
4. Data Analysis
5. Report Out and Communication
6. Take Action on the Findings

This paper will briefly cover the solutions Intel has put together to deal with some of the issues uncovered. An in-depth look at the solution space is outside the scope of this paper.

Survey Projects

Intel University uses the data gathered from a needs assessment project (or survey) to solicit public opinion about training needs and possible solutions. Although there are a number of different strategies one can use to conduct research on the survey population (use existing or secondary data, conduct an in-depth case study, do content analysis, or interview people), IU has decided that the approach and process that they will utilize will be a random survey approach. The goals generally are to get opinions of the population which are statistically significant and this objective can best be obtained from surveys. Intel believes so strongly in the data that is attained from surveys that they have hired a team of statisticians whose main purpose is to conduct survey projects for various business groups within the company. Intel is a data driven company so the investment in this type of effort (to them) is well worth it.

Why do Surveys?

Survey research is a powerful tool for gathering accurate and useful information. The value of a survey comes from the idea of drawing probability samples from a large population. For example, suppose we have a large bag of colored marbles that contains thousands of blue and red marbles and we want to know what percentage of the marbles are red and what percentage are blue.

The question is: Can we draw a sample out of the bag in such a way that we draw very nearly the same percentage of red or blue that exists in the bag? The answer is definitely, yes. However, in order to be effective, the sample size must be large and we must select each marble randomly. Each time we draw one out of the bag; all marbles should have an equal or at least known chance of being drawn. As long as there is a known chance of drawing any one of the marbles on any draw, the scientific principles of probability theory can be applied to the sample.

The key reason why we can use a sample to estimate the characteristics of the population as a whole is because we have a random sample. Of course, when we're interviewing people and getting the opinions of people, we are talking about something entirely different.

In their book entitled, "How to Conduct your Own Survey," Salant and Dillman state that to make accurate estimates based on a human sample; we try and meet four requirements[1]:

- The sample is large enough to yield the desired level of precision. (For the Intercultural Needs, we were shooting for a 90% confidence level with an

[1] **How to Conduct Your Own Survey** by Priscilla Salant and Don Dillman, Wadsworth Publishing , Belmont, California 1996

error rate of + or – 5%. We were looking for a 20% response rate therefore we sent out 5x the number of surveys we were expected to get back. We used the random survey calculator found at the following link to calculate our numbers: http://www.custominsight.com/articles/random-sample-calculator.asp

- Everyone in the population has an equal (or known) chance of being selected for the sample. (For the Intercultural Needs Survey, our target group was Engineers, Managers, Individual Contributors, Planners and Sales and Marketing employees. Since our response rate goal was 20%, we ended up sending surveys to everyone in the target group).

- Questions are asked in ways that enable the people in the sample to respond willingly and accurately. (For the Intercultural Needs Survey, participation was strictly voluntary and the responses were anonymous. We feel due to the anonymity of the survey, we could be fairly confident about the accuracy of the responses).

- The characteristics of people selected in the sampling process but who do not participate in the survey are similar to the characteristics of those who do.

Salant and Dillman say that the above principles are critical to making surveys work and are essential to understand. The closer the survey comes to meeting these four requirements, the more confidence we can place on the results.

Other Data Gathering Methods:

While Intel has chosen the survey approach to data gathering however, surveys are not the only method available. In some cases, other methods are more useful and utilized at Intel. Four alternatives that should be considered are: [2]

- **Use existing or secondary data**: Secondary data is past efforts that have already been conducted. There is no reason to repeat efforts if the data is already available elsewhere. For the Intercultural Needs Survey, we had quite a bit of information available to us however all past efforts did not garner statistically significant results therefore; we decided to go ahead with a full scale survey effort.

- **Conduct an In-depth Case Study:** The purpose of a case study is to understand all the factors related to the research question, but only for a specific case. The results generate ideas that can be explored in depth later on. However these results can not be generalized across the populations. Our goals with the Needs Analysis were to make conclusions

[2] **Designing and Using Organizational Surveys: A Seven Step Process**, Jossey-Bass Publishers. San Francisco, California, 1998

that could be generalized across the population therefore this approach was not sufficient for our purposes.

- **Content Analysis:** Content analysis is a systematic study of written or other types of communication. These might include government documents, media analysis or research from other departments. Content analysis is often used to study policy issues.

- **Interviews:** Interviews are conducted with individuals or groups who are selected for their particular characteristics rather than at random. Such individuals are often selected because they are knowledgeable about the research question. Like case studies, the results of interviews with people who are not selected randomly cannot be generalized but can be used to generate ideas for later study. For the Needs Assessment Survey, we followed up the survey with in depth interviews. The purpose was to validate or invalidate the conclusions that were drawn in the survey as well as discover any solutions from the interviewees.

Each of the four information gathering techniques discussed is well suited to address a particular data gathering goal. However, if the goal is to find out what percentage of some population has a particular attribute or opinion, and the information is not available from secondary sources, then survey research is the only appropriate method. Accordingly, Intel University has launched a full scale effort with the survey approach.

The remainder of this paper will detail the Intercultural Needs Assessment Survey project at Intel Corporation.

Project Management of ITNA:

A Project Manager was hired to see to it that the Needs Assessment was completed with statistically significant results. Many departments at Intel had been working on gathering needs for intercultural business effectiveness but there was no large scale, statistically significant sample of employee's requirements and issues. To this end, the main goal of the project was to gather a statistically significant sample of the population's issues when it came to intercultural business.

Intel developed a survey process guide which was followed for this project. As per the survey process guide, the ITNA project was executed in the following general steps:

1. Planning
2. Target Market
3. Survey Development and Administration
4. Data Analysis
5. Report Out and Communication
6. Take Action on the Findings

The remainder of this paper will go over each of the steps in detail.

1. Planning

Putting together the project plan is one of the most important tasks for a Project Manager. The planning section of the project allows all stakeholders to get on the same page and ensure all stakeholders are not only in agreement with the strategy but to ensure the needs and goals of each stakeholder is taken into consideration when planning the execution of the project.

Process documentation operated as an excellent guide to cover all planning steps of the project. The sections that were included in the published Project Proposal document (attached) were:

- **Project Scope, goals and objectives**: to put together a statistically significant report detailing what issues are preventing emerging markets from doing business effectively.
- **Resources:** Roles, responsibilities, bandwidth were identified.
- **Timeline**: MS Project plan was put together. Tight timeline of June 15 to finish and deliver project findings.
- **Budget**: Budget was set by larger project of which I had funding and resources provided to me. The only cost directly to this project was $500 for survey setup which was utilized.

- **High Level Methodology:** Through much debate and delay, it was decided a survey approach followed by Interviews would be done.
- **Deliverables**: document key areas for intercultural development and training for emerging geographies, provide sufficient information to put together recommendations and produce a statistically significant (20% return rate for survey) report.
- **Risks:** Large risk was not meeting timeline.

The planning of this project can sometimes be riddled with setbacks as not all stakeholders may agree on the plan. For this project, one of our primary goals was to get a statistically significant sample therefore the best way to achieve that goal was to conduct a survey. Some stakeholders felt in depth interviews were also needed therefore, in the end, a hybrid approach was decided upon. A survey would be sent out to a statistically significant sample of the population and follow up interviews were conducted as a second step.

2. Target Market

Now that a plan was in place, the next step was to move onto working out the details of the target market. In order to make sure all stakeholders' needs were met as far as the target audience, a target market meeting was set up to determine the exact requirements of the target audience.

Since everyone had agreed to sign off on the Project Proposal document, all of the population who dealt with global teams (Engineers, Managers, Individual Contributors, etc.) were surveyed and those who wanted to specifically target a particular segment of the population could filter data according to job titles.

3. Survey Development and Administration

Developing the survey strategy is one of the most time consuming and important aspects of the Needs Analysis project. The survey was designed with the goals and objectives of the project in mind, that was to uncover issues that are preventing emerging markets from being effective. Data was also used to set the foundation for deeper levels of learning.

The first step was to gather preliminary information about past intercultural needs efforts. All past effort documentation was analyzed and the conclusion was that although the information was important, there was still insufficient data to provide any conclusions. Therefore, it was decided we would progress with an in depth survey approach to validate the information from a statistically significant perspective.

Based on preliminary information and survey goals, we identified five key behaviors to be assessed in the survey: Leadership, Teamwork, Communication,

Work Effectiveness and Intel Culture. We felt these areas specifically analyzed specific cultural issues that may be preventing work effectiveness at Intel.

We made sure that we could splice data according to target audience requirements. We chose to be able to splice data according to geographic location, amount of experience on global teams and job title.

Based on the above, we wrote a comprehensive list of questions to ask participants. After numerous reviews, the final survey was 37 questions. We wanted to make sure the survey was easy for the user to fill out so we established an agreement with Intel's IT Flex Services. IT Flex Services hosted the survey on their survey and put together two reports for data analysis: Survey Count and Averages and Raw Data. Throughout the project, we were able to view the reports as responses came in.

The project moved fairly quickly on the survey development and administration portion. The survey was sent to numerous reviewers and got frank and honest feedback. Managers were happy to see adequate teamwork on a global perspective.

The response rate goal for the overall project was to get a 20% response rate across all geographies and the project exceeded that goal by attaining a 31% response rate:

	PRC (People's Republic of China)	LAR Latin America Region	India	Russia	Total
Surveys Sent	1085	592	731	219	**2618**
Surveys Received	353 (33%)	237 (40%)	170 (23%)	51 (24%)	**811 (31%)**

The survey was administered with the help of two employees in IT Flex Services who provided good customer service even though their servers were down consistently. At times, data analysts were not able to get reports which caused minor delays to the schedule. IT Flex Services always came up with a workaround to the problems; therefore, overall, the service provided was quite good.

4. Data Analysis

The goal of the project was to collect data so that trends in intercultural issues could be identified. It is important to note that the purpose of this study was to

specifically focus on the issue areas so that enhancement efforts could take place during the solutions part of the project. The project certainly uncovered many areas which were working quite well. For example, across the various geographies it was uncovered that teamwork is quite strong and employees enjoy working on cross cultural teams. With the framework of good teamwork set, the other issue areas have great potential in the solution space. The second part of the Needs Assessment focused on the solution space however a detailed discussion of the solution space is outside the scope of this paper. We will present some of the solutions at a high level in the "What to do with the findings" section of the paper.

An in depth analysis of the data with correlations, cross tabulations, etc could not be conducted however our main focus was to discover:

1. Any issues that were apparent with regards to the categories described above
2. Any consistencies across geographies.
3. A comparison of issues according to geographies.

Evidence of issues in various geographies was gathered. In summary, there were immense consistencies across geographies with the following issues identified:

Cultural understanding is weak: We asked two separate questions on the team and the leader understanding cultural differences and both questions earned a low rating amongst the participants. The leaders were described as "cross cultural team leaders." These leaders did not come from a particular geography; rather they were leaders of a team which included members from multiple geographies. Unfortunately the use of the term "cross cultural team leader" to describe team leads caused some confusion amongst the respondents as the did not understand the title.

Below is a summary of the responses that were received in regards to cultural understanding of the team leader. As illustrated, this point was identified as an issue across the four emerging markets surveyed. Respondents had a chance to provide open ended remarks and the trend was that leaders need to get better educated on differences in culture. In turn, business relations would improve.

Issue: Cross Cultural Team Leader does not understand cultural differences

India	PRC	LAR	Russia	Overall
30% (51/170)	**37%** (131/353)	**38%** (90/237)	**37%** (19/51)	**36%** (292/811)

Represents % of respondents who identified this question as an issue

- **General Comments from open ended responses:**
- Most team leaders assume (wrongly) that there is only 1 culture at work - Intel culture. Little effort or emphasis is placed on understanding differences due to different cultures outside work.
- I do not think we have done a good job in sharing and training our employees on what are those cultural differences we should take into account when leading cross cultural teams. (IC, LAR)
- Cross cultural team leader should have a good understanding of different cultures, work culture, religion influences, etc., otherwise there is bound to be mistrust in a team. (Mid Manager, India)
- The cross cultural team leader does not sit down with the EEs to understand their culture (China, IC)

Key:

- Issue (>30%)
- Borderline Issue (28-29%)
- Non-Issue (<28%)

- **TRENDS (repetitive concerns across geographies):**
- More training is needed for Leaders on cultural understanding
- Leaders need to be more proactive in understanding and overcoming cultural differences.
- Leaders need to recognize that cultural understanding is important part of doing business

A separate question was asked with regards to the team understanding cultural differences and similar results to the ones mentioned above were noticed. Below is a summary slide of the comments received with regards to cultural understanding of the team. As noted above, respondents felt more training/ knowledge was needed on differences in cultural viewpoints:

Issue: the team does not take advantage of different perspectives and experiences and is not aware of differences in cultural viewpoints

India	PRC	LAR	Russia	Overall
36% (63/170)	**36%** (127/353)	**43%** (101/237)	**40%** (20/51)	**39%** (316/811)

Represents % of respondents who identified this question as an issue

- **General Comments from open ended responses:**
- There is no training or documentation to help teams mature in a cross cultural environment. It takes living in a different culture to see the stark difference. We tend to value homogeneity rather than differences as it is easy execution. (Snr Mgr, LAR)
- Cultural differences are frequently not an important part of innovation - the differences to be leveraged are from the experiences those individuals have to draw upon. (Senior Mgr, China)
- So far it has been that the team in US make all decisions as "tell" other teams what to do rather than taking advantage of skills and experiences world-wide to optimize our global working model (Senior Mgr, India)
- "N.A.+ the Intel Culture is very open especially in constructive confrontation or doing straightforward business. In LAR you might hurt peoples feeling's personally or derail the process, because people in the Latin cultures might become afraid thinking they did something wrong, while all that was done was bring in improvement solutions." (IC, LAR).

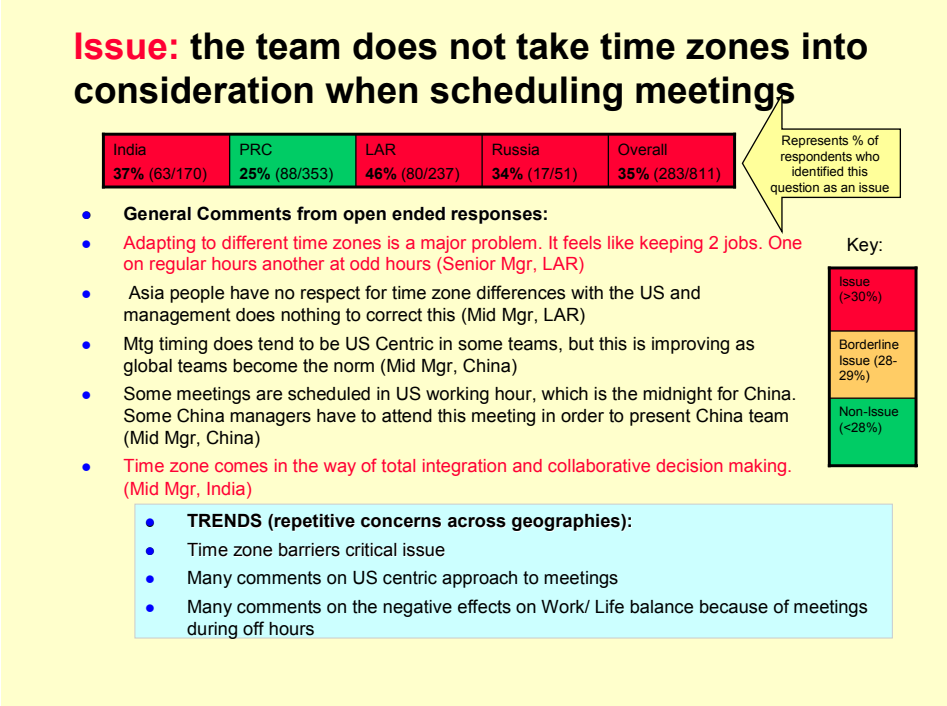
Key:

- Issue (>30%)
- Borderline Issue (28-29%)
- Non-Issue (<28%)

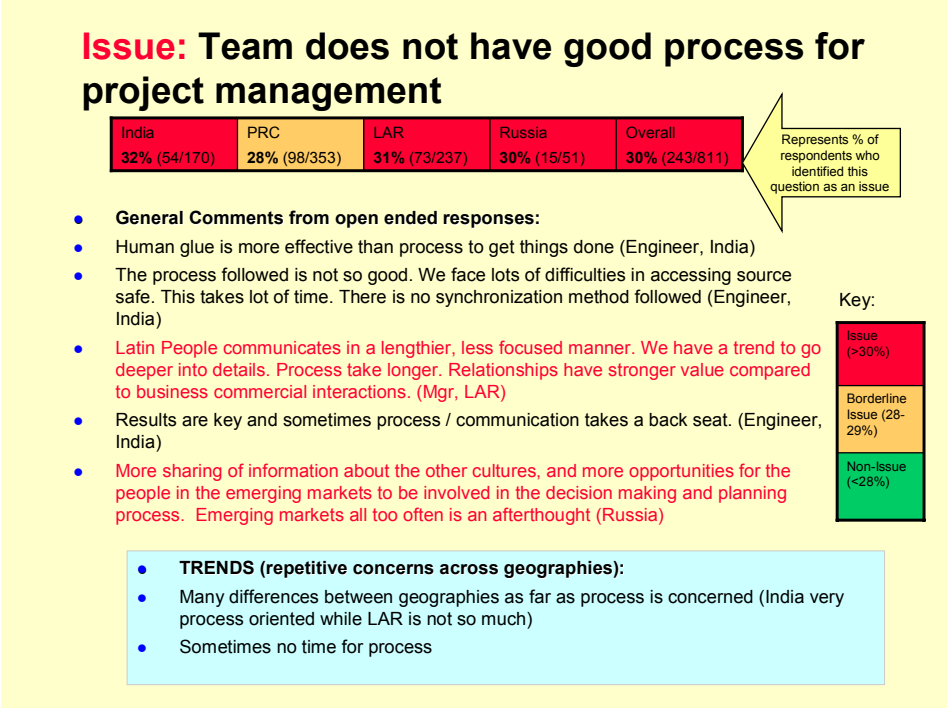
- **TRENDS (repetitive concerns across geographies):**
- U.S centric approach
- Important to leverage cultural differences as everyone benefits in the end
- Need to work cultural training into the agenda of the team

US Centric approach is apparent: Intel is rapidly expanding into the global arena however there is still a catch up game taking place with regards to reassigning responsibilities to the emerging markets. Some in the emerging markets commented that Intel is U.S. centric in their approach to doing business.

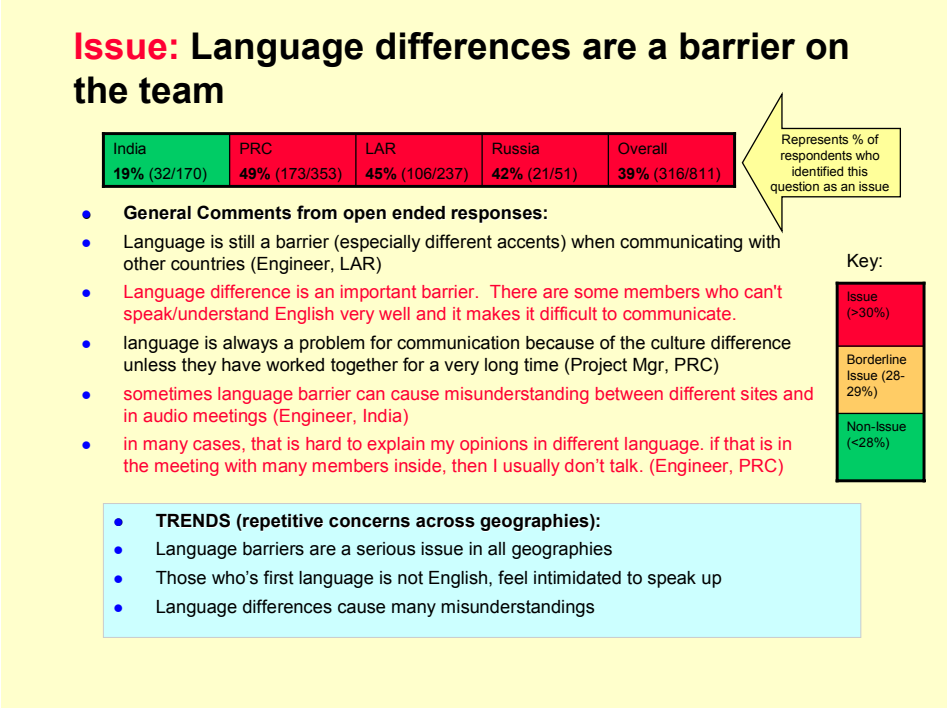
Time zone/ schedule issues: One of the main issues many global companies are tackling with regards to doing business globally is to come to a mutually acceptable middle ground on conducting meetings. Comments stated that there is an effect on the work/ life balance of employees due to having to work all hours. Below is a summary of the findings with regards to this issue:



Project management issues with working remotely and across geographies: Intel is a company known throughout the industry as being very process oriented. Intel has been very successful with its copy exact policies in its manufacturing plants around the world. As new sites develop and expand, Intel is investing a great deal of effort into project management procedures. At the present time, the survey uncovered some areas that need improvement:



Language issues: One of the common issues companies are facing when doing business globally is understanding language differences. Accents and language differences can play a role in the breakdown of effective communication. The severity of the issue of language differences really depends on the geography. As noted below in the results slide, India, where English is spoken fluently has little issues with language differences however other geographies have a critical problem with regards to language differences:



Intel values were also explored at the different geographies to determine if Intel Values are similar or different than the values among the emerging geographies. As a background to these results, it is important to define Intel values. There are 6 core values that Intel strives for:

Risk Taking —

- Foster innovation and creative thinking.
- Embrace change and challenge the status quo.
- Listen to all ideas and viewpoints.
- Learn from our successes and mistakes.
- Encourage and reward informed risk taking.

Quality —

- Achieve the highest standards of excellence.
- Do the right things right.
- Continuously learn, develop and improve.
- Take pride in our work.

Great Place To Work —

- Be open and direct.
- Promote a challenging work environment that develops our diverse workforce.
- Work as a team with respect and trust for each other.

- Win and have fun.
- Recognize and reward accomplishments.
- Manage performance fairly and firmly.
- Be an asset to our communities worldwide.

Discipline —

- Conduct business with uncompromising integrity and professionalism.
- Ensure a safe, clean and injury-free workplace.
- Make and meet commitments.
- Properly plan, fund and staff projects.
- Pay attention to detail.

Results Orientation —

- Set challenging and competitive goals.
- Focus on output.
- Assume responsibility.
- Constructively confront and solve problems.
- Execute flawlessly.

Customer Orientation —

- Listen and respond to our customers, suppliers and stakeholders.
- Clearly communicate mutual intentions and expectations.
- Deliver innovative and competitive products and services.
- Make it easy to work with us.
- Excel at customer satisfaction.

Intel employees in the emerging markets were asked if their geography values were similar or different that the values practiced by Intel Corporation (for example, as shown in the table below, it was uncovered that risk taking is not an inherent quality in India region. Meaning that employees in India are not natural risk takers based on their cultural characteristics).,Below is a summary of the results :

	Similarities	**Differences**
LAR (Latin America Region)	Results Orientation	Discipline Risk Taking
China	Results Orientation Customer Orientation	Risk Taking Quality Discipline
Russia	Quality Results Oriented Risk Taking	Discipline Customer Orientation

	Similarities	Differences
India	Customer Orientation Discipline Results Orientation	Risk Taking

In general we found that cultural characteristics of the emerging market have a great deal to do with whether or not Intel values are successfully being executed at the various geographies. For example, numerous studies at Intel have uncovered that employees from China and India have difficulties with constructive confrontations. Intel defines the practice of Constructive Confrontation as encouraging employees to identify issues and get them into the open where they can be resolved quickly. Constructive Confrontations are an intrinsic part of the Intel Culture. Many commented that, with time and proper role modeling, the Intel culture could be successfully adopted at the emerging sites.

Data analysis is an arduous task and time consuming task however is the most important part of the survey project. Considering the survey size was quite large, there was an extensive amount of data to be looked at. The objectives of the project were fulfilled through the data analysis effort however there is quite a bit of raw data which still can be analyzed further to gather more knowledge about the survey population.

5. Report Out

A plan was devised to report out the findings of the project to numerous departments within Intel. The report's findings will be filtered out to quite a few departments as the content effected a large number of workers at Intel. The Program Sponsor put together a data rollout plan which included presentations to the Intercultural Training Team, Employee Development Team, Training Managers' Council, Product Line Managers and various business groups at Intel.

The report out section was very well executed with a solid plan to disseminate information to all those effected. It was refreshing to know that, although it seemed like we were working in isolation during the execution of the project, the data findings will be shared with a large number of people.

6. What to do with the Findings

"Many clients and practitioners pay little attention to this phase once they reach it, feeling instead that when the survey is done and all the feedback is delivered, it can be forgotten. The fact is that this stage can make or break the survey effort… Even if the survey construction, administration, communication, analysis and delivery are conducted with the utmost competence, if the survey results are not absorbed and ultimately used by

organization members to make key decisions, the survey cannot be considered to have been maximally effective" (Church & Waclawski, 1998)

The main purpose of conducting a Survey Needs Assessment was to provide recommendations and solutions to the issues identified. The first step in the goals of the Intercultural Team was to identify issues (which the survey did) and the second step was to plan and scope recommendations and solutions to the issues identified. The survey achieved what it set out to do and the next step is to take action on the findings.

Intel Corporation has invested a great deal of effort and resources into identifying practical ways to solve some of the intercultural issues its emerging markets are facing. Intel commissioned a team of employees to work specifically on implementing solutions to the issues identified through the Needs Assessment project. Subsequently, the company has also developed Emerging Market Ramp Teams who's sole purpose is to analyze and develop programs to help bring the emerging markets up to speed. Intel is being recognized for its leadership efforts in this area and many companies are looking to Intel for Best Known Practices for solving intercultural issues.

Below is a high level look at some of the solutions the Intel team is currently working on to address these issues:

Some Proposed Solutions:

- HR participation
  - Offer escalation process
  - Facilitation of critical issues for key strategic global teams
  - Offer a process to help avoid intercultural issues
  - Offer support to managers to coach their teams
- Proliferate customer related BKMs
  - Use forums like ISMC to share intercultural BKMs with our customers, fellow travelers, channels, etc.
- Help facilitate move of US centric decision making to emerging markets where appropriate
- Global rotation program/ mentoring programs
- Assessment of the practice of Intel Culture and development programs as a result of assessment
- Hire the right people to succeed in intercultural teams
- Consider very flexible schedules to help alleviate time zone differences related issues

Books and Articles:

- Church, A.H. & Waclawski, J. (1998). *Designing and Using Organizational Surveys: A Seven-Step Process.* San Francisco, CA: Jossey-Bass Publishers.

- Brinkerhoff, Robert, O. (2001) *High Impact Learning: Strategies for Leveraging Business Results from Training* Perseus Publishing.

- Kirpatrick, Donald. (1998) *Evaluation Training Programs* San Fransisco, CA: Berrett-Koehler Publishers, Inc.

- Rea, L.M. & Parker, R.A. (1997). *Designing and Conducting Survey Research: A Comprehensive Guide.* San Francisco, CA: Jossey-Bass Publishers.

- Macey, W.H. (1996). Dealing with the Data: Collecting, Processing, and Analysis. In A. Kraut's (Ed.) *Organizational Surveys: Tools for Assessment and Change.* (pp. 204-232). San Francisco, CA: Jossey-Bass Publishers.

- Edwards, J.E., Thomas, M.D., Rosenfeld, P.I. & Booth-Kewley, S. (1997). *How to conduct organizational surveys: A step-by-step guide*. Thousand Oaks, CA: SAGE Publications.

- Kraut, A.I. (1996). *Organizational Surveys: Tools for Assessment and Change*. San Francisco, CA: Jossey-Bass Publishers.

- Fowler, F. J. (1993). Survey Research Methods (2nd edition). Newbury Park, CA: SAGE Publications.

Can you hear me? … Can you hear me now? …

Challenges in Managing an Offshore Test Team

Yana Mezher
Software SETT Corporation

Abstract

IT offshore outsourcing is a significant factor in the U.S. job market today. Software SETT had the opportunity to greet this challenge in 2003. This paper presents several projects that were led by two U.S.-based Test Leads and it reveals their hard-earned lessons. The obstacles of time, culture and distance require a very different management practice and commitment, one that many U.S. managers may be unwilling to make. In this paper, we briefly showcase the projects, but focus in greater detail on the unique issues we faced and how these were solved at the ground level. Covered issues include tracking progress at a reliable level of detail, getting the information you need to make the right decisions, and building trust and accountability. This paper does not discuss the general principles behind offshore outsourcing, but rather presents hands-on experience with this important way of doing business in IT/software testing today.

About the Author

Ms. Mezher is a test lead at Software SETT Corporation. Over the past eight years she contributed to Software SETT's on-line courseware as well as supporting many projects with mentoring, team training and in-classroom development. Ms. Mezher has supported many large and small Silicon Valley companies covering technologies such as People Soft, data warehousing, voice infrastructure, web applications, matching services and on-line auctions.

Mailing address
Software SETT Corporation
P.O. Box 718
Los Gatos, CA 95031-0718

Copyright © 2004. Software SETT Corporation. All Rights Reserved

Introduction

Managing an offshore team poses various managerial challenges. Overcoming these obstacles and breaking the barriers between groups of people that are located across the ocean enables project success, a better understanding of each other's cultures, and a feeling of accomplishment and achievement. Some of the challenges faced by a manager of the offshore team are:

1. **Dependence on formal methods of communication** (e-mails, documents, publications, etc.). Written communication was the primary mode of interaction for most teams. A less often used alternative was to rely primarily on telephone conversations with some use of e-mail.
2. **Loss of informal communication.** Opportunities for quick resolution, quick clarifications and more thorough problem solving could not be pursued.
3. **Rigid hierarchy of offshore teams.** Top-down directives were over utilized. Unnecessary intermediate steps were performed before work products could be delivered. Collaborative work was uneven.
4. **Outdated and inflexible training.** Technology had advanced beyond the processes, development and testing methods that were applied.
5. **Limited control over staffing changes.** Lack of availability of qualified individuals (it's a hot job market in India) and lack of control exerted by the staffing coordinator made changes to staffing difficult.
6. **Cultural and distance limitations.** The free flow of information sometimes did not occur because of cultural differences and the distance between team members, e.g., individuals will "silently fail" or fear passing along "bad" news; tasks take exactly as long as the schedule allows

Even though cultural differences posed some of the most difficult and hard to solve challenges and issues to the managers of teams located in various parts of the world, we will not be addressing them in this paper directly, as the focus of the paper will be processes, techniques and practical advice on how to deal with concerns and problems that are typical for projects with distributed teams. The subject of how information gets processed and communicated by different cultures will be touched on in various areas of this discussion, when approaches and solutions to common outsourcing problems are addressed.

Our Projects

Projects that utilize offshore staffing vary in structure and approach. Our experience with the offshore teams ranges from the pioneers in offshore outsourcing to the most current experiences.

On some of the earlier projects, offshore resources were brought to the U.S. to train in software testing with the eventual return back. For those projects, very structured testing processes were established by the U.S. teams with the offshore team expecting to do just one function of the testing--test execution. For another project, the U.S. testing team was the outsourcing solution for a company located in one of the Asian countries. It was a reverse offshore project where only the testing team was located in U.S., with the rest of the company in Asia. Most current projects supported different types of global structure, where several teams were located in different locations.

Regardless of the differences in structure, all the offshore projects have many common elements and challenges that we will be addressing and discussing in this paper. Two projects will be showcased in the paper as the ones that can serve as good examples of different types of offshoring. These projects prove to contain most of the typical offshore testing challenges and dilemmas, even though they are different in structure and size. The first type of project is termed "truly offshore." This type of project has an entire function that is located outside of the US. The second type of project is termed a "global team." This type of project has its team members and its constituent groups dispersed around the globe.

A Truly Offshore Project

Two of these projects that we led were part of the IT group. The entire management team and many of the developers were located in United States. The entire testing team and the remainder of the developers were located offshore. Projects were part of the IT group and all of the team members had direct reporting within this group. Business teams worked closely with the IT group, especially with the Analyst and the Program manager. These business teams did not conduct any system testing activities and had no direct responsibilities during system test on the projects.

Each project had a Project Manager and/or Program Manager, an Analyst, 1-5 developers, a database administrator, and a Test Lead. All of these individuals were located in U.S. Some of the development of the application functionality was outsourced to the offshore team. A team of 2 Testers per project was located offshore. Within the offshore testing team, Testers were working at different locations.

The same exact structure existed for another parallel project. Each Tester was assigned to a particular project and was expected to work on that project for the duration of their assignment. All system testing was expected to be done offshore. Testers were the only ones who had access to the testing tools and the special test set-up. Development and test environments were placed in a location that was different from the location of the development teams and these environments were maintained remotely. This project was a good example of a set-up where only one function is completely outsourced offshore, while the other functions were kept in a single U.S. location.

Global Team Project

The global team project was driven by the business team, where the business team actively participated in the system testing process and also conducted the majority of testing themselves. The project team was not centrally located, nor was any functional part of the project team. Project managers and team leads were positioned in various locations around the world. For example – the project manager was from Germany, the development lead was located on the West Coast of the U.S., and the "Move to" production manager was located on the East Coast of the U.S. Most of the functions on the project had two or more people assigned to them. There were three database administrators located on the West Coast of the U.S., in England, and in Malaysia. The U.S. development team had counterparts in Europe and in China. The testing team consisted mostly of business users of the application. There were more then 200 Testers that were assigned for system testing that were located all over the world.

This project structure was put in place to ensure 24 hour a day turnaround of all project issues and to minimize the impact of the loss of team members due to unpredictable circumstances. This was a truly global project where some of the functions were completely outsourced to the offshore team, while others functions were duplicated at various sites.

Dependence on Formal Methods of Communication

When managing a team that is located across the ocean and in a different time zone, it becomes apparent that you have to adjust and modify your communication style. Formal methods of communication become the most common, and sometimes it is the only method of communication between the teams. In this case, written communication with the offshore team should be well defined, clear and comprehensive, free of slang, and minimize possible interpretations.

Some of the formal methods of communication that the managing team can utilize when working with the offshore team include:

- **Day to day communications**--e-mails, status reports, regular updates
- **Project specifics**--design and analysis documents, specifications, and schedules
- **Test processes**--test case writing instructions, test execution routines, testing templates.
- **Technical communications**--instructions on how to execute tests and processes. File locations and access levels.
- **Reports** –weekly, monthly, by build, by stage, formal reports to the management team.

Day-to-day Communications

Day-to-day communications become extremely important as they substitute for all informal contact. They fill this niche of establishing a personal connection with your offshore team as well as providing a method to exchange information.

You can decide on the format of your day-to-day communications based on the structure of your team and existing processes.

Most teams rely on the combination of e-mail communications and regular telephone updates. Some projects manage to use mostly e-mail combined with very infrequent telephone contact. In either case, day-to-day communications should be well defined and structured. Expectations of the team members should be stated up front.

When you decide on the format of your communication, keep in mind that you are dealing with time differences. If you want to incorporate telephone conferencing or have regular status updates by phone, it probably should be scheduled for either early in the morning or late at night in order to minimize work schedule adjustments.

Your day-to-day communications should follow the same format and/or pattern. If you decide to update your team on the project development status every day, you should follow the same routine at all times. Your team relies on your updates and expects them on a regular basis. Very often, the offshore team feels removed and uninformed about project development and would appreciate just a note that no new development is in progress. Using templates for your status reports and updates will help the team members organize the information and be able to quickly search for it mentally and electronically when needed.

Regular updates can be organized differently based on the structure, level of expertise of your team, your personal preferences, and the overall management style of your project. On our project the "reply to all" feature on the e-mail was utilized consistently. This ensured that all of the updates and replies were communicated directly and timely to all of the team members. On the other hand, the offshore Testers were flooded with the e-mails that did not pertain to them and their responsibilities. They were not sure which tasks were assigned to them directly and which ones were only for their information. This caused additional unnecessary e-mail exchange between the Test Lead and the Testers. In this case, it helped to have the Test Lead filter and organize information before delivering it to the test team. (It also helped to have the subject line or the first line of the e-mail clearly state what the e-mail was about and who it was intended for.)

Templates used during regular updates help to group similar items together and serve as reminders about certain issues. If your status report includes an "Issues" sections, it forces Testers to think about any problems that they need to inform you about, regardless of how minor they are. Using templates also cuts down on the time and effort that is applied to maintain regular communications. It becomes an easy and fast task to fill out obvious information, it helps to remember minor or routine assignments, and it focuses attention on the important items. Our testing team did not particularly enjoy filling out the "issues" part of the daily report and usually left it blank. It took active encouragement on the part of the Test Lead to make Testers alert the lead and therefore other team members about issues and problems that they were facing. As they started doing it, it became easier to filter out potential problems and help the Testers with the tasks that they identified as problematic.

The test team might be used to a very formal writing style that you will have to adjust to. Formal English is used in most countries where English is studied as the second language. Using slang and American expressions can be confusing for the offshore team. Some of the words and expressions might be misunderstood and cause tasks to get executed when they shouldn't be or cause them to be executed incorrectly. On the flip side--sometimes communications received from the offshore team might require some "translation" to U.S. English. We consistently received mails where Testers expressed doubts and questions about the product's functionality, but it took some time to figure out that they actually had questions about functionality of the product.

Developing your own communication style will be similar to developing your own management style when you are working with the onsite team. Very often you can learn about your team by studying their written communications. You can learn how detailed and organized a person is by finding details of their assignment in weekly e-mail updates. E-mail from one of your Tester can have perfect formatting that could lead you to believe that this person is a good candidate to create various templates and documents. E-mails very often become the only way to discover and understand each person's personality, strengths and weaknesses. And a Test Lead can benefit from learning about the team members and taking this knowledge into consideration when distributing assignments and tasks.

Project Documentation

When working with the offshore team, all members of the project have to ensure that project documentation is current, complete, detailed enough, and contains information that is necessary for the various stakeholders. Very often documentation is the only resource that the offshore team can obtain access to. Their access to people may be very limited as they might be located at another site or because they are located in different time zones.

Participation in meetings can be restricted for the same reasons. Knowledge transfer between the offshore team itself and other groups usually relies heavily on good project documentation. If it is not available, then big gaps in project knowledge can exist. The offshore team typically works independently. They receive documentation; review it and use it to achieve their objectives. If there are questions regarding any documentation, then it becomes a very tedious and time-consuming process to get answers and necessary clarifications. It can be done in a form of e-mail with questions and answers going back and forth between members of the team and between the team and other stakeholders. In the beginning of the project when there was not enough good documentation, we sometimes had e-mail threads with one small question that was being answered and investigated for days at a time. Many times the subject got dropped just because it became clear after many days of researching that it was not a critical issue. But on other occasions the subject turned out to be of great importance for Testers and could have been lost if it had not been addressed in the e-mail.

We had good luck with using the tracking feature in Microsoft Word when multiple people were adding their comments and questions to the existing document. This feature allows the team members to post different points of view at different times and keep track of who is the author of the changes. We use this feature also within the test group when inputs from different test reviews are incorporated into a single document.

Various forms of online meetings can be used to get clarifications for the entire team. If the changes that need to be implemented require simultaneous input and discussion of several people, we found that an on-line meeting is the fastest approach. Of course, all of the key people have to attend the meeting. Attendance can sometimes be a problem where members are located in more than one time zone.

It is important that documents have a similar format so that the offshore team can easily find areas in which they are interested. As the projects and team membership change, consistency in the documentation and details help the offshore team as well as the rest of the project team to maintain momentum and pace of the information flow. On both of the projects that we are discussing, the initial set up of the basic project documentation was there, so we did not have to start creating templates and forms from scratch. It helped that one project manager was setting up the initial project documentation and only later new people were arriving to bring their ideas to the child projects that sprung from the parent project. Project forms and templates became a collaborative effort of several groups of people working on the project at different times. New ideas were added and invalid material was eliminated while the basic form and content was maintained. As Test Leads, we tried to encourage project managers and the rest of the team members to maintain the consistency of all project documentation as much as possible, as we realized that this helps the Testers to become more efficient.

Test Processes

On several projects on which we participated, we found well-organized basic testing documentation, but a lack of test processes in general. Testing was done differently from project to project, depending on the team members, and their expertise and experience. A typical situation on the project would be when one Test Lead left and another one started and testing processes changed as a result. For our Global team project, Test cases would be written in one format under one Test Lead's coordination, but when a new Test Lead arrived, she would introduce another test case format. That would bring some confusion to the test team and create additional work for them. The new Test Lead insisted on creating test cases in a different format for a variety of reasons that made sense at the time.

At the end of the test cases writing phase, two sets of the test cases existed in two different formats--the original tests cases and the new ones for the current release. When it was time to start executing them and documenting the results, it caused the Testers to become confused. As Testers for that project were in multiple locations, they had trouble understanding which test cases were new and which ones were old and how to document results for them. Regional Test Leads spent a lot of time trying to figure out why there were two sets of tests and what was the important part of each of the tests. Then they tried to get confirmation of their assumptions from the overall Test Lead. A lot of time and effort was spent just on figuring out what the test process should be based on the current status of the documentation. This discussion brought up a question about how to set up good processes at the start of the project in order to accommodate the needs and specifics of the particular project.

For that project, it was decided that the original test cases needed to be rewritten in a new format, as they did not match other test documents. Rewriting the tests required additional time and effort. For the next phase of the project, to avoid this unnecessary work, we created a process for writing the test cases, updating them, and performing future modifications. Additionally, test case execution instructions were created to match the test case format.

Technical Documentation

As we started identifying the need for setting up testing processes, it became clear that documentation needed to be written concurrently and these processes needed to be made available to all teams.

To guide Testers with test case execution, some test case execution guidelines were created. Since Testers were located in different places, they were unable to learn from each other. Specifically, they were unable to transfer working knowledge of some of the common execution routines. Instructions that addressed basic as well as some advanced or corner cases processes were established.

On the other project, the Truly Offshore project, Testers were replaced frequently and not all of the details of test case execution were transferred between team members. Documenting these instructions in detail helped to save a lot of the Test Lead's time because these questions about test execution were usually reported back to the Test Lead, who lacked access to the test environment that was necessary to verify the answers. Therefore, the Test Lead had to pose the same questions to the developer(s), who might be unavailable, and deliver the answers back to the Testers.

We also discovered that a process should exist that would update all of the instructions with new information at the beginning of the new project. Access levels, login and password information, permissions, and file locations change with each project. It helped to go over instructions and request in advance all of the information that was necessary for test execution.

On most of the projects, obtaining access to certain tools, drives and functionalities required a formal process that involved management approval. Knowing and dealing with these formalities upfront proved to be more efficient than waiting until it was time to execute the tests.

Creating good bug tracking instructions proved to be one of the most useful test preparation activities. Because the bug-tracking tool was used by multiple groups, it became the most commonly used tool on the project. The instructions for using it, can be as simple or as comprehensive as you want them to be or as your process requires. (It is best to write a basic set of instructions that requires the person who enters the defect report to provide the following information: description of the defect, the tested software version, a numbered procedure for duplicating the issue, input files that were used in revealing the defect, and any output files that resulted.)

On the project where there were more then 200 Testers worldwide, the bug-tracking tool included detailed instructions about how to deal with local issues and timing. On the "Truly Offshore Project," the instructions dealt mostly with the process of fixing and retesting and delivering information to and from developers. The bug tracking tool instructions are one of the documents that will be reused repeatedly by various team members, so it makes sense to invest more time and effort into its creation and maintenance and to do this maintenance regularly as the project scope changes.

Reports and Reporting

Having a team in multiple locations requires the Test Lead to have a very good reporting process in place. This process is beneficial to all of the team members – Testers, Test Leads, Project Management, business partners, and other team members. On the project where the Test Lead was managing all day-to-day activities of the small team--A Truly Offshore project, daily status reports became the norm. Appropriate templates were established by the Test Lead to deliver information to the Testers. In return, the Testers provided status to the Test Lead in the daily reports. On the parallel project, daily reports were presented in an informal format and the format of the weekly reports was formally sorted and grouped to include descriptions of all the activities.

Various reports were identified and created for the duration of the projects that we participated in. Needs for some reports arose only at a specific time, such as at the completion of a milestone, while others were on going. One standard type of on-going status report to management lists the team members' achievements and alerts management of problems and issues that may require attention.

For the team that was distributed globally and had various locations, reporting was more high level and one way. For the most part, reports dealt with the number and nature of the bugs reported and progress of testing in a specific area. In the case of a big testing group, regional Team Leads were collecting the input for the reports. Later on, these reports were assembled into one status report for the entire testing activity. The reporting process usually takes time, especially if a big group of people is involved.

Some intermediate steps can be taken and/or different types of reports can be established in order to reduce the report preparation time. Overall, all team members benefit from receiving information about what other team members are working on and what issues they are dealing with.

Outdated and Inflexible Training

When we first started on the projects, we found that the existing training procedures were outdated and inflexible. From our experience, the offshore test team members were well trained in the technology, but lacked an understanding of both software development and testing methods and processes. The existing training plan for the offshore team had the following problems:

- The focus was too narrow and technical
- Testers were only prepared for immediate tasks
- Testers were unfamiliar with failure analysis and how to apply it
- The plan was created by someone that was no longer involved in the projects

Training Focus

The original training plan was designed when the test team was expected to spend most of their time conducting automated testing, without having the Testers participate in any other areas of the testing process. As the project demands evolved and a greater degree of involvement was required by the offshore team, the Testers became overwhelmed and found themselves unprepared when the need to work on additional software testing tasks was raised. We identified this gap and added intensive training of overall test procedures for all projects. All team members had the same training on the software testing subject matter and testing processes were defined for the whole group.

The training plan also was not broad enough in that it did not prepare the Test Engineers for all of the projects in which they were expected to participate. It just covered the training for their primary project assignment. Each testing project was organized in its own unique way using a different directory, document, and terminology structure. As the functional changes became more ambitious and the pace for delivery increased, it became very important to have processes as consistent and standard as possible to allow team members to move from one project to another without spending time adjusting to the new procedures. Therefore, all of the projects were organized similarly to allow smooth transitions between them. While working on the test process for multiple projects, the Test Lead also needed to keep in mind how to organize it in order to create consistency in the documentation, test structure, and implementation.

Failure Analysis Training

Not only was a thorough grounding in testing procedures missing, the Testers also had little training in failure analysis and how it applied to the code. Much of the Testers' past training was based on software development, building functionality that worked. Test analysis conducted by the Testers showed a strong leaning towards validation – much like what a software developer would do during unit testing. When pushed for a stronger analysis, it was difficult for the Testers to see beyond the functionality and how it should work, not how it could break. Complex interplay in the code, processing sequences, unique and unusual conditions, and common failure points were all new, unfamiliar concepts. The training plan was modified to include a stronger emphasis and practice with test analysis and test design.

Manual testing was identified as a big part in the overall testing strategy and was represented in the training plan. A special section was added that described ad hoc practices and their place in the testing strategy for specific projects. As a result of these changes, the Testers felt much more comfortable and proficient during the testing process when they were applying unusual conditions, creative data combinations, and ad hoc techniques. A great number of defects were found this way that led to a higher quality product and also to improved Tester satisfaction and enhanced motivation to continue looking for hidden defects.

Current Project Demands

Another problem with the original training plan was that its author, who had a fairly good knowledge of the technical requirements of the job, didn't have enough information about the current project demands. When it was written, the training plan was quite thorough and was maintained solely by the staffing coordinator. Unfortunately, the coordinator had little to no recent involvement in the project. The coordinator's training plan addressed various areas of testing, but it missed the details of the project that only a person involved in it could have known.

It became apparent that a Test Lead's lack of involvement in the training of new hires might have consequences for the project at a later time. Therefore, the Test Lead had to become more involved in reviewing the training tasks as well as determining their assigned durations. It is expected that the Test Lead have some involvement and participation in training on the subjects that required in-depth knowledge of the project details and the software testing processes. With the new training plan, recently hired Testers were able to learn up-front the details of the project to which they were assigned. This early training eliminated the need for them to later stumble over some important piece of project information. It also allowed team members to participate more confidently in project discussions and gain better understanding of their testing assignments.

Prepare for Project Flexibility

The original training plan did not address the need for the Testers to be able to move smoothly and easily from one project to another within the group. The plan focused solely on one project. This focus greatly limited the Test Lead's and the Project Manager's ability to utilize resources efficiently. After learning the dynamics of the parallel projects, similarities, differences and demands, it became clear that the training plan should address all of these issues and prepare the Testers to be flexible when they move from project to project.

To create a training plan that meets these expectations, a Test Lead must constantly monitor the training process, and update it with new project details and information. While working on the test process for multiple projects, the Test lead also needs to keep in mind how to organize it in order to create consistency in the documentation, test structure, and details.
The training plan that was created, incorporated the following requirements: (a) Testers must be able to participate in multiple projects, (b) Testers must be prepared for several assignments, and (c) the plan must include enough detail of the assignments while also including an appropriately high level view of the assignments. Following these requirements became extremely crucial when one project urgently required additional resources and a Tester from a parallel project was able to complete this transition seamlessly, thanks to the receiving the planned training.

Positive Results

Managing an offshore team is not an easy or trivial task. It is very trying and challenging at times. It may require taking a new approach to planning, communicating, and training. At a minimum, it requires an approach towards management from a new angle where events are viewed and analyzed from a global perspective. As with many difficult challenges, when you achieve success, it can be very rewarding. Although it was often more frustrating for team members that were located offshore than it was for those that were located in the U.S., the increased challenge of being located offshore brought out the best in these team members.

Our offshore team was very responsive to the micromanaging techniques that were applied during the most stressful and demanding times on the project. The team not only complied with all of the little details of the project, but they volunteered even more information when they thought it was necessary. The team provided comprehensive answers to the questions that were posted by the Team Lead day after day after day. They kept fairly good track of their assignments for the near future and were also able to respond to the questions that related to the past tasks.

Following day-to-day directions was not done blindly on their part. As Test Leads became more comfortable with each test team member's skills, they started giving more open-ended assignments to the entire team. That proved to be very successful.

Our test team was very bright and very well educated. Once they learned the dynamics of the projects, they began showing their unique skills and volunteered to provide new ways of doing things. One day the Test Lead was pleasantly surprised to find a nicely organized spreadsheet with references to many test documents after she mentioned in e-mail that there was a need to create some sort of cross-reference document. One of the Testers voluntarily took responsibility for this task and created a document that is now used on all projects. This Tester not only formatted the document in the most pleasing and easy to use way, but also continued to maintain the spreadsheet. This document has now become one of the major test tracking tools and it has been expanded to many worksheets documents.

As projects progress, Testers started to take on more and more responsibility, very often without the Test Lead's coordination. They started noticing things that had to be done and proceeded to doing them without reminders from the Test Lead. Some new documents were created this way as the Testers felt a need for them. The Testers suggested some interesting test techniques when old methods could not support new test demands.

Overall, the team proved to have good traditional education in engineering and had good knowledge of modern programming practices. This knowledge became very helpful when we needed a liaison between two groups that worked in two time zones. One of our Testers became a link between a development group that worked in the U.S. and a group in India. His knowledge of the processes, environment set up, and details of the project became essential. And even though it was not part of his original set of responsibilities, this assignment became very crucial to both teams and to the project at large.

Conclusion

Offshore outsourcing is here to stay and as we are learning how to best manage it, we realize that to make it a success, we need to fine-tune our own practices first. Offshore teams are part of the overall project team and benefit or suffer from the same project pitfalls and accomplishments as the local team. A phenomenon of offshore managing is that all the problems and struggles of the project get amplified by the fact that the teams are far removed, isolated, and located in another time zone and within different cultures.

To make projects that are using offshore teams successful, every team that we worked with came to the realization that they needed to review and clarify their own internal U.S. practices first. It was impossible to create a successfully working process for any offshore team without first identifying a good working process for the whole project. Having efficient communications just within the offshore team is unattainable if the rest of the project has a limited and irregular communication style. The offshore team cannot be capable of creating, maintaining and following good documentation if the rest of the project team relies on a disorganized assortment of tools, processes, and documentation procedures.

It all starts here, in the basics of project management and coordination. Projects that right from the start identify their needs, follow up the plan to resolve their issues, and have realistic expectations of what is achievable and what is not, are more likely to be successful in their offshore management and to achieve project success.

Leading a Global Test Team - From Terror to Bliss

Author: Kathleen Kallhoff
IBM Corporation
IBM Printing Systems Division
May 2004

e-mail: kallhoff@us.ibm.com

About the Author:

Kathleen Kallhoff has worked for IBM for 26 years. She is currently the team lead for the function and system test team of Infoprint Manager, which is a client/server software offering of the IBM Printing Systems Division. She leads teams located in Boulder, Colorado, Timisoara, Romania, and Yamato, Japan. Her background includes software testing, software tools administration, build and packaging support and Advanced Technical Support for IBM Printing Systems. She received a BS in Medical Technology and Microbiology from Colorado State University in 1976.

Abstract:

The ability to communicate effectively with remote teams is one of the keys to being successful in a global business environment. Organizations faced with the possibility of working with a global development or test team may, at the least, feel anxious regarding the relationship, or at the worst, feel threatened that their jobs may ultimately be at risk. What is often not realized by the local team is that by off-loading some aspects of the project, the door is opened for increased productivity, workload balance, and an opportunity for the local team to tackle more challenges. With creativity, communication, patience, flexibility, and with a little humor thrown in for good measure, it is possible to successfully blend local and remote teams and create a win-win relationship.

Introduction

Opinions vary on the success of sending software work offshore. The obvious issue, in my experience, is fear that local jobs will be lost and people will be laid off. Other issues can center on the local team's suspicion that the remote team will be difficult to work with, due to language, cultural differences, or technical misunderstandings. The facts are, however, that we live in an increasingly global world. The internet and other types of technology have made it possible to work more easily across remote locations. In fact, many companies have teams and divisions around the world, so it is possible to submit that working with a separate company in another location to share workload is little different than working with remote teams within the same company. My division of IBM, the Printing Systems Division, made the strategic decision to use a global team to provide the development and test for portions of our software offerings. I accepted the opportunity to integrate a local and remote team to perform the function test phase of a complex client/server product. My task was to build team unity and set up the environment to make the transition successful. My experience working with a remote team has been excellent. In this paper, I describe the activities and provide some perspectives that made my experience successful.

In 2002, I accepted the assignment of transferring some of the Function Verification Testing (FVT) for a software product developed by the IBM Printing Systems division (PSD) from the U.S. location to a company based in Romania. PSD initially entered into an agreement with the remote company to perform the software development and function test of a few self-contained product offerings. As this relationship proved to be successful, PSD management became interested in expanding the business agreement to cover other software activities. The project in which I was involved was significantly challenging because there was a steep learning curve for the Romanian team to build expertise on a complex client/server software product and I had to blend the local and remote teams into single, cohesive team. I needed to develop a smooth process for managing the workload, and we needed excellent collaboration between the remote and local teams.

The prospect of taking on this challenge was intimidating, to say the least. Many peers voiced skepticism of the viability of this project and many said that this project was doomed to failure. The reality is that the project has been extremely successful, beyond my wildest imaginings. The not-so-secret factors that led to developing the successful relationships and team interactions were:

- Support of management
- Local training of the remote team at the start of the project
- Communication, communication, communication
- Recognition
- Trust – on both sides
- Bridging differences
- Putting human faces to the team
- Understanding local culture
- Using the time difference to an advantage
- Flexibility
- Humor

Getting started on the right foot

Starting the collaboration between the teams involved the following:

- **Management and Peer support**

I had excellent support from my local management chain. When I started to doubt the viability of this project, my immediate manager was there to provide input, suggestions, and support. When we needed to escalate issues related to concerns over system security and access for the remote team to project related documentation and system databases, my local management team was extremely responsive and acted quickly to resolve issues. I soon realized that support from my immediate management chain was not going to be a problem. Instead, the challenge was getting support and buy-in from management and key leaders across multiple organizations.

As a member of a team that spans software development, information development, system test, and customer support, it took time and diligence on my part to help members of these teams feel comfortable with the remote team and with the processes that I developed to support the transition of responsibilities. I knew that as the team lead, it was my job to be an advocate for the remote team, in order to develop the support necessary from local peers. I developed processes to ensure that information, whether an e-mail, the details in a code defect, or a proposed test scenario would be well understood by all concerned. For example, the local teams were familiar with each other. It was common to for a tester to walk over to a developer's office and discuss a possible defect scenario. In addition, they both spoke the same language, so they understood each other's slang and phrases. Of course, this approach did not work so well with the remote team! I overcame this obstacle by establishing that all e-mail communications, from either side, would come to me first. I would make sure that I understood the issue, and then I would forward it to the appropriate person. Initially this approach was a lot of work for me. Teams from each side, however, readily accepted that there was value in this approach, to make sure that the right person received the email and that the problem or issue was clear and well understood. Using this technique, I was able to avoid potential communication problems. This process also helped to establish the credibility of remote team, as I was able to sanity check the issues they saw and to weed out any incorrect ones before they went too far. Once the team established credibility with key development members, buy-in to the viability of this project occurred as a natural consequence.

Management support through out the organization was also key in the success of the project. Managers needed to provide approval for the remote team's access to project databases, the defect and code libraries, e-mail systems, and controlled access labs.

- **Preparation of the teams**

Having the right people for the task and ensuring that all participants are prepared, is key to the success of any project. The remote team consisted of four test engineers, one of whom was the team lead and the person I would interface with the most. As soon as I had information about the remote team, I took the opportunity to introduce myself through email, and I sent them documentation about our product so that they could begin to become familiar with it. In my project, we had the luxury of bringing the remote team to our site, so that we could provide hands-on training. The timing of the trip corresponded with the projected start of Function

Verification Test (FVT) for a new release of the product. By participating in the FVT, the remote team worked with my local team, learned our protocol for writing test designs and scenarios, experienced hands-on work with the software, performed testing on an actual product, and were able to interact with the local teams. In my opinion, the local training was key to the success of this project. In addition, during their visit, I took every opportunity that I could find to expose the team to any other ad-hoc educational opportunities. For example:

- Education related to our defect tracking system
- Writing the test design and test scenarios for a new function item in the release
- Orthogonal Defect Classification
- Standards for writing defects
- Ad-hoc sessions with the software developers of the product
- Use of other tools and education, including on-line education about our product

The local test team was gracious and was willing to embrace the new members. As they began to work together in the test lab, the commonality of understanding computing and software systems went a long way to create team cohesiveness.

- **Putting a face to the name**

Aside from the obvious advantages the hands-on training had for the remote team, it gave the local teams an unparalleled opportunity to get acquainted in person, to have a better understanding of the abilities, personalities, and work ethic of the each member of the remote team. I made sure to introduce them to everyone involved with the project. These introductions usually took place in the hallways and were informal, but went a long way to break the ice. They attended project-related team meetings, formal design reviews, and any ad-hoc discussions in which I could involve them.

We went to lunch together and we got together after work to get to know each other better. It is much easier to work together when you know more about an individual on a personal level. I have seen the alternative side of this first hand with other projects assigned from PSD to the remote company. Some of these projects have teams where neither side has met face to face. What I observed in this situation is that there is an increased chance for misunderstandings, false starts, and finger pointing when things go downhill. Although the projects have ultimately been successful, members from both sides frequently contact me to help resolve issues that they could handle easily if they knew each other personally. I realize that working with people you have never met can be daunting, especially if they are in another country. It may not always be possible to travel to enable personal contact. I am extremely lucky to have been able to travel to Romania to work with the team there, and some members of the team have returned to our location for additional training.

Putting a face to the name has helped immensely with the inevitable challenge of using e-mail as the primary method of communication. An e-mail ‘conversation’ differs from one held face to face in many ways. In a face-to-face exchange, the participants can interrupt each other to correct mistaken impressions or to short-circuit confusion and long-winded digressions. In an e-mail exchange, by the time you read someone's message, the sender has already finished composing it. You cannot interrupt to correct misconceptions or to clarify your point of view. E-mail tends to be very impersonal. Even with the use of emoticons, (e.g., smiley face graphics), the inability to detect each other's body language tends to sensitize us and put us on guard,

making it easy to take offense. I encouraged the team to make our e-mails more personal using emoticons, colors, and graphics. This technique has gone a long way to help avoid e-mail pitfalls and misunderstandings, because it increases the trust and confidence that we understand each other.

- **Be a tireless advocate for the remote team**

Since I have been able to know the team, and have established a cordial working relationship with them, becoming an advocate for them has become easy. Advocacy takes many forms:

- I made sure that the remote team had access to the same data and resources that the local team has. I know that this statement must seem like a 'no brainer'. However, I found that though my management team was committed to the project's success, they left it up to me to figure out how to accomplish it. In the early stages of the project, it was clear that the only way I could ensure success was to make sure that the remote team had access to all product documentation, including design documentation, local discussion team rooms, and defect data. The data is stored in various project team rooms, the defect tracking system and code libraries, and some hardcopy. I frequently had to work through issues from product developers and project management, worried that we were sharing too much information, but with management support and as the team gained credibility, that concern went away.
- I gave credit where credit is due. When I presented test status at project team meetings, I made sure to list the contributions of the remote team.
- I passed along appropriate project information to the remote team. Information gleaned from project meetings, e-mail, informal discussions with developers, and any other source I can find, I send to the appropriate team members. I find myself in meetings thinking, "I need to send that information to Romania."

A significant by-product of being an advocate for the team is enhanced team unity. The remote team feels valued; they know that they can rely on me to ensure that they have access to the information they need, and that I will work to keep the local teams aware of their contributions.

Building and keeping momentum going forward

A number of factors contributed to the ability to sustain momentum on this project:

- **Communication, Communication, Communication**

I went out of my way to act as an 'interpreter' for each side. My interpreter role included e-mail and teleconferences. I tried to 'put myself in the other guy's shoes' to make sure that each side understood what the other was saying. In doing this, I became a better listener, and I was able to add value to discussions. Many times it is obvious to a listener that the other participants are coming from opposite experiences and that a misunderstanding is soon to occur, and it is possible to get both sides on the same page with careful listening and interpretation. This is particularly important when English is not the native language for part of the team. Use of slang, colloquialisms, and analogies is fine as long as everyone understands what they mean. I learned that our rampant use of sports analogies is confusing to people who are not familiar with the particular sport. For example, the Romanian team knows nothing about baseball or golf, so

when I would use the phrase “Par for the course”, or “We missed the base with that idea”, I would get blank looks from everyone. One time when I tried to explain the analogy, one of my team members thought for a minute, and then translated a common Romanian phrase into English, and said “Well, that’s like putting good cheese into a dog’s guts”, and they all laughed at my look of confusion.

Communication that involves different cultures often surprises the different participants in the conversation. In her article, Judith Olsen, author of ‘Culture Surprises in Remote Software Development Teams’.[1] writes, “Our own culture is invisible to us. Culture is acquired. It helps people categorize and predict their world by teaching them habits, rules, and expectations from the behavior of others. It helps people "read" the world's signals—the meaning of symbols of artifacts, gestures, and accoutrements of others. Culture also molds the way people think: what their motivations are, how they categorize things, what inference and decision procedures they use, and the basis on which they evaluate themselves. It sets the gestures, space, and timing of interactions.”

"We don't see our own ways of doing things as conditioned in the cradle," writes Esther Wanning, author of Culture Shock! USA. "We see them as correct, and we conclude that people from other countries have grave failings."[2]

It takes effort to make sure that others can understand what you mean. In my experience, rather than admit that they did not understand, the members of the remote team would either not reply, or would make some non-committal response. I learned that I had to read faces and be comfortable asking, “Did you understand that?”, or “Am I making myself clear?” I also had to develop comfort when I did not understand what they were trying to say. Over time, we both developed enough confidence to ask for clarification when our conversations were not well understood. The team was highly motivated to improve their skills in English, and we came to a mutual understanding that I was free to correct grammar and word usage.

In addition to e-mail and telephone conversations, I found the following communication methods particularly useful:

- Instant messaging, particularly with graphical icons is very effective. I use this method of communicating almost every day. It occurs in ‘real time’, which allows for a more fluid conversation, and it can be personalized with great graphics, which give personality to the ‘conversation’.
  Unlike e-mail, where the emoticons are ASCII characters, graphics can look like  and even  .

  Many free versions of instant messaging are available. I use Yahoo and AOL with the remote team. It is important to note that there may be confidentiality concerns with using non-secure instant messaging. Be aware of the level of your conversation and take into account that this may not be a secure way to share information. We are careful not to discuss sensitive or internal confidential information via this channel.

- Teleconferences are another effective tool. I conduct a weekly teleconference with the remote team during which we discuss status, plans, and any issues that arise.
- A web conference with the addition of a teleconference is a technique we have recently employed. This is a particularly effective method of conducting a code or test plan review. The web conference tool we use, IBM Sametime Connect, allows multiple participants to log into a screen sharing facility. We control access using userids and passwords. When logged in, a moderator can control who 'has the floor' so to speak. A single document or program is shared at any given time. Each participant can 'see' the document being shared, but only one person has control of the document. The person who has control can edit the document, the other participants can see the edited version instantly, and the moderator can pass control at any time. I have found this to be extremely effective and much more efficient than reviewing via email. There is increased collaboration and interaction between the teams because of utilizing this technology.

- **Flexibility and Humor**

I can't say enough about being flexible and using humor to establish a successful working relationship. Of course, this topic is not unique to blending local and global teams, but rather it fits into successful teaming in general. However, in my situation, it was key to establishing an easy working relationship and helping the remote team feel accepted. When the remote team arrived in Boulder to begin their on-site training, it was arranged for them to visit the IBM facility on a Saturday morning to introduce each other, to help orient them to the workplace, and to set some initial expectations. The team from IBM consisted of two managers from my management chain, a member of the local test team, and me. The members of the remote team were the owner of the company and the four test engineers. I knew that I, at the very least, was very nervous. I soon found out that I was not the only one. As we walked down the hall to the conference room, my counterpart on the Romanian team, Lucian, confided in me that three of the four members of the team had never flown before, that only one of them could drive legally, that none of them had been in such a large facility before, and that they were all really nervous and scared. He was charming and genuine; I could not help but respond in kind. I told him that I was impressed with his command of English and that it was a lot better than my command of Romanian. Thus we began our working relationship with a small exchange of 'human-ness' that has carried along to this day. On another occasion, in the first week, Lucian mentioned that he wanted to be sure that they understood the 'rules' so that they could make sure that they were doing everything right. I jokingly responded: 'Lucian, there are no rules… I'm making this up as I go along'! It went a long way to break the ice, helped us feel like we were collaborating, as teammates, rather than as leader and subordinate.

We took the team out to lunch for their last day in the US. We took them to a local Sushi bar, and they gamely tried the various forms of Sushi and Sashimi. Japanese food is not common in Romania; the looks on their faces as they ate the first bites of raw tuna were priceless! In return for this 'favor', I had to promise that when I traveled to Romania, I would be willing to eat one

of their local dishes, mashed pork brain! Which I did, but it does not rank high on my list of a repeat item. It does not taste like chicken.

Flexibility is required of both the local and the remote team. We take for granted the processes and systems we have at our local facility. I had to have flexibility to work through security issues to get the appropriate access to our IT systems for the remote team, to develop processes for transferring large amounts of data to the remote team, to manage how the time difference between the two teams affected our efficiency. The remote team had to be both flexible and patient as the above issues were resolved. Working through the issues due to the differences in time zones provides an excellent example of how flexibility is required of both teams. For example, there is a nine-hour difference between the time zones, so when it is 8:00 AM locally, it is 5:00 PM in Romania. The Romanian company required their employees to shift their work schedule to be from 9:00 AM – 6:00 PM so that we could maximize the overlap time when both companies would be at work. In addition, the local teams are willing to be flexible; we often start teleconferences and web conferences at 7:00 AM to avoid extending the day too long for the Romanian team. The Romanian team invariably works past their scheduled stop time, as do we when necessary, to continue email or instant message conversations, resolve problems and finalize other issues. It is a natural consequence due to fact that we are starting our day, discovering issues or problems via email, and need to pass along information and get problems resolved, or set the stage for them to work on items the next day.

My willingness to be their advocate required me to be flexible and innovative, as I had to find ways to promote them and keep their interests in my line of sight. The recognition that I was working as their advocate enabled them to be patient and trust that I was working toward our mutual benefit.

- **Trust and Recognition**

Trust builds over time and usually comes as a natural consequence of respect and observing results. As the local and remote teams built trust in each other, they were more able and willing to promote and recognize each other's accomplishments. I learned to give credit, when appropriate, for the remote team's contributions, to copy their management on chain e-mails, just as I would with the local team. For example, the team lead of the remote team did an excellent job of reviewing the product documentation. He provided screen captures, found, and reported inconsistencies, and made helpful suggestions. I received a note of appreciation from the lead Information Developer on the product, and I immediately passed it along to the remote team, copying their management and our local management. He deserved the recognition, and it helped cement the trust between the teams.

I worked hard to avoid 'us and them' dynamics. The following quote, taken from our internal web site, speaks to the interaction of trust and recognition:

"A company based on values: Our values as IBMers shape everything we do, every choice we make on behalf of this company. Having a shared set of values helps us make decisions and, in the process, makes our company great. But their real influence occurs when we apply these values to our personal work and our interactions with one another and the wider world."

- *Dedication to every client's success*
- *Innovation that matters – for our company and for the world*
- *Trust and personal responsibility in all relationships*

I am proud to work for a company that is based on values. I believed in putting these values to work when I began working with the remote team. I strove to earn their trust, and as a result we work together effectively in a mutually supportive fashion.

- **Bridging Differences**

Communication, flexibility, humor, and trust are all key to bridging differences. It is sometimes challenging to work as a team when you all share the same geographic location. It is even more challenging to bond with people you rarely, if ever, meet in person. Even with internet-based technology, it is hard to build trust with a group of faceless voices on a conference call. Challenges that integrated teams face across geographies include:

- It can be difficult to form team identity. There can be the 'I'm here... You're there' mentality. This may be manifested in the attitude of always waiting for the other guy to move first. The team leader must take responsibility and move with authority to remove this roadblock. For example, after the Romanian team returned home and continued to execute function tests, there was sometimes distrust among the local team that the defects they opened were valid. I spent many hours recreating the defects in our lab to demonstrate to the local team that these people really knew what they were doing. This issue has gone away as time and results have proven that their work is reliable and trustworthy.
- The economic benefit of moving development work offshore may be offset by the potential higher costs associated with remote teams. For example, travel costs, duplication of equipment, and project management focus may be more costly when managing the integration of global teams. This is an excellent example of why effective communication will affect the successful integration of global teams. It may not be necessary to travel as frequently with effective communication tools. Reduction in project management challenges is possible when members of the global team communicate well and rely on flexibility and humor in their interactions.
- Assumptions based on cultural differences may affect team unity. It can be difficult to build trust, understand motivation, and build a sense of shared commitment when cultural differences get in the way of effective interaction. It is extremely important that the team lead work to eliminate these problems. Strive to understand the attitudes, beliefs, and motivations that influence the team.

It is critical that the local organization develops an awareness of the cultural differences that affect how to give effective directions, how to effectively resolve problems and delegate responsibility. Of course, this awareness goes both ways, but in my opinion in the US, we are so 'American centric' that we often do not look at the nuances of other cultures. For example, the Romanian culture is hierarchical. They are familiar with a workflow that consists of either a manager or team lead delegating work downward. The team members do not expect to have a lot of input with regard to what they will be working on; this decision is left up to the team lead / manager. In my company, we often 'work by committee', hold many meetings to resolve issues,

gather everyone's input, and then make a decision. The Romanian team wonders how we accomplish anything; the local team wonders whether the Romanian team members feel left out of the process. In fact, each team functions well individually and it is left to the respective team leads to understand the nuances and effectively assign workload.

According to Judith Olson, there are two classes of basic cultural differences that may arise in multicultural teams: (1) Team Composition – the members of the team, what motivates them, and how they develop trust in each other, and (2) Teamwork – the ways in which the activity progresses, how the planning occurs, the process of decision making, and how responsibility is awarded and accepted. [1]

On my project, the team composition was set from the beginning. I had no control of whom I would be working with, but I was able to determine what motivated them and how to establish trust. Establishing teamwork came as a natural process of working together over time, as well as incorporating humor, flexibility, communication skills and maintaining respect in all cases.

Despite these challenges, team members **must** collaborate to accomplish their goals and function seamlessly. The following suggestions may help:

- ***Redefine whom "we" represents.*** It is vital for a team to have a psychological togetherness when they cannot be together physically. A team leader can and should promote a 'we' mindset by looking for ways to promote unity. When teammates include each other in decisions and information, team unity will form. Foster an attitude that 'we' represents a group defined by purpose, not necessarily geography. The attitude of 'we' will help form a team identity.
- ***Face-to-Face meetings.*** As mentioned at the beginning of this article, key to the success of my project was bringing the team to our site. We have continued to travel between our facilities; this has strengthened the bond and the relationships, and has allowed for additional training and clear goal setting.
- ***Use technology to your advantage***. E-mail, teleconferences, instant messaging, and web conferences are examples of technology that support remote teams, and which can help reduce the higher cost of co-teaming. Be sure to set appropriate expectations with regard to using the different technologies. Be aware of how easy it is to misinterpret an e-mail or an instant message, particularly with cultural and language differences in the mix. Use graphical icons to personalize and convey emotion.
- ***Establish expectations.*** Set expectations for working together: when and how is it appropriate to communicate by email or by telephone. Establish clear response times, whom to carbon copy, and set expectations for inclusion of and full participation by all team members. Be aware of the cultural differences that affect assumptions. Successful teams will create their own culture, creating a version of 'us' that supports team cohesion and commitment.
- ***Use the time difference to your advantage, but don't take advantage of it!*** Be aware of the time difference between the remote teams. Plan the schedule for conference calls and e-meetings at a mutually convenient time. This may mean that the local team comes in early, or stays late, and visa-versa. If the remote team is in Europe, and the local team is in the US, be careful to avoid prolonging their workday. Be

sensitive to the fact that although your day might be starting, theirs is ending. Avoid the temptation to prolong an Instant Message session or to send an urgent e-mail to which you expect a reply. Work with the local support teams to ensure that system availability takes into account the remote team's work hours. However, you can use the time differences to your advantage by sending work or requests via e-mail at the end of the day, knowing that there will be a response when you get back to work the next day.

- ***Keep executives informed.*** Involving and informing the executives associated with the project helps keep them committed to the team's efforts. Executive support and a focused vision helps the team compete with other demands on team member's time, attention, and efforts.

Conclusion:

This project has been successful beyond my wildest dreams. The following items illustrate:

- The business decision to engage the Romanian company was purely economic. We had more work to do than we could afford to do with IBM employees. We were able to deliver more function in the product, in a shorter time, by transferring part of the work to Romania.
- We were able to reduce changes in project scope by utilizing the remote team effectively. It is very important for this product to be responsive to input from customers with regard to new functionality. Before engaging the Romanian team, the project manager might need to reduce or remove the planned new functions due to limitation of resources for testing, or make a risk-based decision regarding what to test.
- This project has been on going for 2 ½ years. As the Romanian team gained experience, I was able to assign items that are more complex to them. The trust and confidence that we have in each other has grown as has their abilities.

Because of the success of this project, PSD development management has gradually increased the number and scope of projects assigned to the Romanian company. Some of these are development in nature; some are test efforts. Each project has a PSD subject matter expert (SME) to take on the lead role and interface between Romania and Boulder. This role is crucial in the ultimate success of the project. It is the responsibility of the SME, as well as the local management team, to ensure that all teams clearly understand the requirements of each project deliverable, how to resolve issues, problems, and answer questions.

My growth as a team lead has surpassed what I expected because of my experience with this project. The success I enjoyed is due to incorporating the factors outlined in this paper. However, I look to the accomplishments, integrity, and capabilities of the members of the team, both local and remote, who contributed greatly to the project's success. When the team is successful in reaching its goals, individual success follows. I took the approach that I had to be the advocate for the team. Most of the time this required more work for me; however, it paid off many times over. I feel enriched by this experience and feel privileged to have been part of it. I look forward to continuing my work with the global team. It is a fascinating and rewarding job.

Acknowledgements
This paper is a result of my own experiences co-joining a local and remote team. I had no experience in this area prior to taking this assignment. I gratefully accepted hints and tips from other individuals who had already been down this path. In particular, I'd like to acknowledge Mike McDonald and Andy Hunter. Without their guidance and help, particularly when we traveled to Romania, my job would have been much more difficult. Thanks also to all the local testers in PSD who were willing to accept the remote team and have been so gracious to share their expertise with them. Thanks to my local management, team for their support and encouragement, in particular Ernie Kumar, without whom I would not have this fantastic job!

Tremendous thanks to David Curtis, Jeannie Birndorf, and Debra Schratz for their invaluable assistance reviewing this paper and for making significant contributions to help make it more readable.

References

1. Culture Surprises in Remote Software Development Teams ACM Queue vol. 1, no. 9 - December/January 2003-2004
by Judith S. Olson, University of Michigan; Gary M. Olson, University of Michigan and Collaboratory for Research on Electronic Work
http://www.acmqueue.com/modules.php?name=Content&pa=showpage&pid=105&page=1

2. Wanning, E. Culture Shock USA: A Guide to Customs and Etiquette. Singapore: Graphic Arts Center Publishing Company, 1997.

Effective Solutions for Global Software Development

Robert F. Roggio
Professor
Dept of Comp and Info Sciences
University of North Florida
Jacksonville, FL

Rahul Dighe
Inteliant Technologies, Inc.
Santa Clara, CA

Sharada Bobbali
Information Engineer
CACI Federal Inc.
Jacksonville, FL

Abstract
Economic benefits, manpower availability, and round-the-clock development, are resulting in increased global development and maintenance of software applications. Evidence all around us clearly indicates that large-scale software development is continuing to evolve from a single-site activity to a multi-site, distributed activity and in many cases to an enterprise-wide, multi-country activity. But is this apparent transition without serious stakeholder problems? Empirical data suggests that multi-site activity tends to increase development cycle time. Extended development times are often directly attributed to problems in communications, inter-site coordination, process and project management issues, and a myriad of other technical issues. These difficulties become exacerbated when multi-site activity spans national and cultural boundaries. Although economics is almost always offered as the most recognizable rationale in support of off-shore development, companies engaged in Global Software Development (GSD) report widely varying experiences: both positive and negative. Challenges faced while developing software globally appears to span a breadth of issues that include multi-cultural communications, process and project coordination, geographic distribution of persons and facilities, and lack of a consistent supporting infrastructure. This paper formulates and defines principles of success for developing software projects globally.

1. Background
To many practitioners the rise of global software development has arisen as part of the natural maturing of an industry, as supply and demand of technical resources and business arrangements, such as strategic partnerships, joint ventures, and global companies continue to become more and more commonplace internationally. It some cases global efforts may be attributed to the national policy of some countries where the government, as a client, may require suppliers to locate the development facility in that country as a condition of a contract or to qualify for a favorable tax treatment. According to a report by the National Association of Software and Service Companies, 185 of Fortune 500 companies outsourced software development to India alone and, for example, in 1999-2000, the amount of outsourcing in India grew at a rate of 53 percent [12]. Until the early 1980s, approximately 75 to 80 percent of the world's software production took place in the US, and most of the labor supply was domestic [26]. By the mid-1990's, however, the need for these professionals had increased to the point at which there were not enough resources to meet demand [10]. As companies competed for resources, development and labor costs soared. Given this backdrop, many businesses are continuing to find it economically attractive to develop software around the world where there is the best confluence of economics, technical expertise and infrastructure.

1.1 Salary Differentials
The data in Table 1 shows the salary disparity between a US engineer and those in emerging countries. In most cases, it is possible to locate skilled, English-speaking, computing professionals in a number of countries at 10 – 20 percent of what it would cost to hire a similar person in the US. Even if the high cost of operating an offshore center is considered, the salary differential is significant enough to motivate US based companies to relocate some software development off shore.

. Country	Purchasing Power Parity	Annual Salary
United States	1.0	$70,000
Hungary	0.367	$25,690
China	0.216	$15,120
Russia	0.206	$14,420
India	0.194	$13,580
Source – Ron Hira, Columbia University; http://www.ieeeusa.org/forum/POLICY/2003/061803.html		

Table1: Annual Salary Requirements for an Engineer in Selected Countries

1.2 Lack of Skilled Work Force

Availability of skilled software professionals was a problem in US in the booming 90's. Enrollment in US universities and colleges was insufficient to meet spiraling industry demands. Table 2 presents a comparison of US graduate production compared to emerging countries. (To show a percentage increase in degrees awarded in various countries, we computed the "percent Diff" column).

Country	BA and BS Degrees			MA, MS and PhD Degrees		
	1989	1999	Percent Diff	1989	1999	Percent Diff
China	127,000	322,000	153	19,000	41,000	115
India	165,000	251,000	52	64,000	63,000	-1.5
Philippines	40,000	66,000	65	255	937	267
Mexico	32,000	57,000	78	340	63,000	18429
United States	196,000	220,000	12	61,000	77,000	26
Source–National Science Foundation: http://www.ieeeusa.org/forum/POLICY/2003/061803.html						

Table 2: Science and Engineering Degree Production in Selected Countries

During this period, US software firms experienced a scarcity of experienced software professionals. This led US software firms to lobby congress to allow more foreigners to immigrate to the US for employment. But in some cases it was not possible to relocate people, and a number of companies opened work centers where software professionals were available. This trend continued and businesses started development sites off shore, significantly reducing the additional effort, time, and expense involved in hiring well-trained programmers from overseas.

1.3 Continuous Software Development

Planning and budgeting personnel and time to meet schedule demands is an arduous task. Project estimates in person hours cannot be merely divided by the available manpower to calculate a delivery date. While some development and testing activities may be run in parallel, others must be synchronized, as there may be clear dependencies among various components. This becomes particularly true in testing, where identifying and correcting defects are usually sequential activities. In such cases, a global development and testing team can work in tandem. Typically, developers in the US may correct deficiencies and have the product available for testing at the start of the offshore team's workday. The testing team in India, China or Russia, may then test and pass results back to developers in the US.

One of the first assumptions in person-hour estimates is that increasing the time-effort dedicated to the project will reduce the total cycle-time required for implementation. Varying time zones in the world can make this a reality. Theoretically, a geographically dispersed team can work three eight-

hour shifts. For example, Beijing, London and California are separated by a time zone difference of eight hours, making it possible for teams to theoretically hand-off work to each other.

1.4 Localization benefits to achieve market opportunities

The world is growing smaller. Companies producing such software frequently witness the use of their products extended into countries other than where it was developed. International users, among other things, want software that has been further customized to work in their locale and language. As a strategy to market products internationally, companies may set up development centers where the software is to be marketed. Because the operation is in close proximity to the target market, it is best equipped to adapting the software to the local customer needs and expectations.

Companies are also likely to relocate offices where there is already a concentration of similar industries. This might be driven by the availability of infrastructure and access to skilled professionals from nearby universities. Some regions of the world are at the forefront when it comes to adopting emerging technologies. As an example, mobile standards have consistently been driven from Europe. Competing companies have established presence in the region to demonstrate that they are an integral part of the European community. [13].

1.5 Mergers and Acquisition (Partnership)

Many client organizations prefer companies that can offer single vendor solutions. More and more software is packaged in a way that spans enterprise needs. Because new software development is always a risky venture with unpredictable costs and untested markets, companies sometimes find it beneficial to merge with or acquire other companies and integrate their product suites. Merged companies may or may not be in the same country, and in many cases a business merger or acquisition does not necessarily translate into a physical merger. This may be brought about because of the necessity to serve local customers or because of inability or unwillingness of the people to relocate.

There has also been a general trend among top consulting firms in the US to setup development centers offshore. This allows these companies to offer the same cost benefits that their competition in emerging countries can offer to customers in the US. Interestingly, the reverse is sometimes true, as some companies from emerging countries are buying local firms in the US to increase their local presence in order to offer services that must be handled locally. In both of these situations software development has to be managed as a global, multi-site activity.

1.6 Pitfalls – A Duality

Successful software development projects, whether local or global must satisfy the basic criteria of being delivered on time, according to schedule, meet standards, satisfy stakeholders, and be within budget. Although companies can and do realize benefits attributed to global software development it is not without its drawbacks.

The drawbacks can be attributed to how global software development is actually carried out in the company. Given time-zone differences, there is a potential for locating teams such that while it is time for a team in one country to leave work, the second team is just starting its day and can continue where the first team left off. Theoretically, a workday gets extended to 24 hours, and work can continue round-the-clock.

In marked contrast, there is considerable evidence that indicates [2] that compared to same-site work, cross-site work takes much longer, and requires more people for projects of equal size and complexity. Although it appears round-the-clock development is possible for some tasks, it is not always suitable. Highly cohesive tasks can always be most efficiently performed at collocated centers. In the presence of this duality, what may render distributing software development economically viable is the price/salary disparity and availability of persons of equitable skills, who speak English in emerging countries.

2. Global Software Development Issues and Resolutions: Seven Practical Issues

Global Software development, if done right, can have tremendous benefits both in terms of economics and quality. We discuss below some approaches to developing software globally.

2.1 Practical Issue: Time Overlap

To make the development of a complete software system manageable and successful, software designers and developers often must cooperate efficiently as a team, and this requires a considerable amount of personal communication. In an ideal situation, work could proceed on a round-the-clock basis due to time zone differences between the US and offshore development centers. To leverage the time zone difference effectively, the requirements and enhancements can be completed and communicated overnight by an offshore team and the changes submitted by the on-shore team may be ready for review the following business day. A team in New Jersey can hand off work at the end of their workday to be continued by team members in India. The Indian team can progress on the tasks while the New Jersey staff sleeps.

Communication between the offshore and on-shore teams is critically important and should not be constrained by time-zone differences. These differences influence real-time communication that can take place only during a few overlapping hours of the workday [10]. Specifically, office hours in New York and Europe only overlap by two hours, which occurs between 9 and 11 a.m. EST. As time zones grow farther apart, time slots available during normal business hours are reduced along with the opportunity to communicate in real time.

In Global Software projects, teams separated from East to West have less work overlap time than teams separated from North to South. Even small time zone differences can bring surprising difficulties. A study on Global Software Team[8] coordination observed that a time-zone difference of one hour between two sites substantially affected the team's ability to communicate interactively because it reduces their overlap time by four hours – one hour at the beginning of the day, one hour at the end of the day, and one hour during each site's lunch break.

Global software development efforts must pay considerable attention to time zone differences. These differences, when coupled with other factors such as different workweeks, religious holidays, and national holidays in the offshore development center substantially reduce overlapping hours. Diminishing overlapping hours become non-overlapping days. Various American-Israeli collaborations reported an "Israeli black-out period". The Israeli workweek is Sunday through Thursday. When combined with the seven-hour time difference relative to the US east coast, the US site is out of touch with the Israelis for two days of their workweek [4]. Since there are only a few overlapping hours each working day, global software teams separated by time zones must adjust their schedules to compensate for non-overlapping hours. Teams have to plan convenient timeslots to communicate and coordinate within the limited overlap hours available. They should adjust their work practices to deal with temporal factors – such as establishing overlapping work hour windows; use liaison personnel whose work hours are the same as the other site; and program work to deliver output in batches to the other site at the end of the day [9]. If effectively managed, the coordination costs will be reduced due to increased overlap hours and if essential communication occurs during the overlap hours. Therefore, it is advisable to agree on a few hours each day that all team members will be available for real-time contact. In addition to the adjustment in work schedules, the team manager should travel periodically to interact face-to-face with the offsite team.

2.2 Practical Issue: Cultural Differences

"A typical American yuppie drinks French wine, listens to Beethoven on a Japanese audio system, uses the Internet to buy Persian textiles from a dealer in London, watches Hollywood movies funded by foreign capital and filmed by a European director, and vacations in Bali; an upper-middle-class Japanese may do much the same". [14] An American software engineer in Silicon valley might start his day taking yoga lessons at the gym, eat Indian food at a nearby restaurant for lunch; his

counterpart in India might start his day in a aerobic studio and choose from KFC, Pizza Hut, Mc Donald's for lunch and stop by for some Starbucks-inspired coffee houses on the way home. All this might seem to suggest to some that we aren't much different after all!!

The answer to the question, how different are various cultures within the world, depends on to whom you pose the question. "Confronted with apparent differences between other people's ideas and practices and their own, anthropologists have historically reacted in one of three ways. Some, known as Universalists, have sought ways to minimize or erase the appearance of difference or to deny that any significant differences exist, and to treat otherness as an illusion. Some, known as Developmentalists, have perceived in the encounter between cultures, a story about a civilizing process. Still other anthropologists, known as Pluralists have argued that cultures can be different but equal, and they have cautioned against cultural imperialism, suggesting that one's own local cultural evaluation should not be confused with Universal Scientific, Practical, or Moral Reason". [22]

Ways of communicating and expressiveness are as much an individual traits as they are based on national cultures. In the US, we can see heterogeneous teams composed of people of various cultures and social backgrounds; "Microsoft by some accounts has 20 percent of its employees of Indian origin" [25]. The success of such heterogeneous teams in IT organizations across America is witness to the fact that cultural issues can be minimized on greater exposure to one another. Some of the ways by which we could alleviate this are:

2.2.1 Employee Rotation (Liaisons)

In a personal experience of one of the authors, it was clear that rotating team members between the development centers located in India and in the USA was very beneficial. The Indian team was primarily responsible for testing a US developed product. On a regular basis the US team would deploy builds that the team in India would test overnight. Since the center in India was new and communication mainly through use of email and product documentation was not up to date, feedback cycles were large, and overall confidence of the development team for the testing team in India was low. There were instances when the Indian teams did not understand the importance of acknowledging email, and while the worked diligently on implementing email suggestions, they didn't feel it necessary to acknowledge that they had done so. Rotating team members from India gave the Indian team as well as the American team a chance to interact with each other and foster team spirit. It also helped clarify expectations that people had of each other with regards to communication, process, and points of contact. Returning team members acted as cultural liaisons for the Indian team members that hadn't been to the US yet. "This greatly helped reduce the "we vs. they" culture and a lack of trust and respect for team members in other functions or at other locations". [15]

2.2.2 Recognize Differences and Avoid Cultural Dominance

Accepting the view of the Universalist or Developmentalist school of thought leads one to believe that little or no cultural differences exist. This leads us to the question, "Does an individual display behavior, which is dominant even while under the influence of various cultures? The answer may be found in Hosftede's seminal study that involved a large sampling of IBM personnel in 40 countries on the topic of work values. The study concluded that in spite of IBM's strong corporate culture, national cultural differences do surface. [4]

"The Western view of the world is in terms of individuals as opposed to groups, to which most eastern people view the world". [16] The individualistic view of the world leads one to believe that cultural differences do not matter, as people by nature are inherently different, unique and individualistic. "When cultural differences are ignored, cultural dominance is the result. It may come from sheer numbers, such as having a vast majority of team members from one culture who impose their own style over the rest of the team. It also comes in the form of a hierarchy, as is the case when teams have some members from headquarters and other members representing subsidiaries. In this case even if its team representatives are the minority, the headquarters standards and style tend to get imposed on the others" [16].

Whether it's an American company doing business with India, China, or Russia; or whether it's the Russian, Chinese, and Indian doing business with an American company, cultural dominance can be avoided if both the software supplier and the client organization recognize the need to acknowledge clear cultural differences. It should be the responsibility of neither the software supplier nor the client to have singular responsibility to adapt to the other's organizational culture. [17]

2.2.3 Open Communications across Cultures

Even though we can say that English has more or less become the de-facto standard for business communication across the world, the way people express themselves varies widely. Communication is also hampered because people don't have a way to meet with each other in person. In American schools and workplaces we see numerous examples of cross-cultural teams functioning in harmony. Much of it is due to the fact that people have had a means to work with each other and understand each other's idiosyncrasies. In a cross-cultural team separated by space and distance informal communication is limited. "Westerners – and perhaps especially Americans – are apt to find Asians hard to read because Asians are likely to assume that their point has been made indirectly and with finesse. Meanwhile, the Westerner is in fact very much in the dark. Asians, in turn, are apt to find Westerners – perhaps especially Americans – direct to the point of condescension or even rudeness". [19] In countries that revere hierarchy and strive to promote group harmony, instances of individuals critiquing each other in an open forum are rare. In a virtual collaboration between teams at MIT, USA and Centro de Investigacion Cientifica y de Educacion Superior de Ensenada (CICESE), Mexico, a designer in CICESE wanted to criticize the work of an MIT designer but, instead of telling the person directly, asked the CICESE project manager to relay the information to the MIT project manager to handle the situation. [28].

"Debate is almost as uncommon in modern Asia as in ancient China. In fact, the whole rhetoric of argumentation that is second nature to Westerners is largely absent in Asia." [19] For example, British managers in an outsourcing relationship with a particular Indian software supplier found that the Indian programmers, in deference to authority, would not voice criticism in face-to-face meetings but would sometimes send their opinions in email messages after the meetings had disbanded. The British managers, used intense interaction and development of ideas through meetings, felt frustrated at this "polite" behavior [29]. An explicit understanding of how people from various cultures express their opinions, criticisms, and likewise respond to them, will go a long way to aid communication

2.3 Practical Issue: Task Distribution Strategy in Global Software Development

With software teams distributed across time zones, the answer to the question, "How do we divide a software task to make best use of the skill and resources available within a certain team to get maximum productivity?" becomes critical. It is a matching game between the software task at hand and the skills and resources, which are distributed. For example, "one joint venture that developed embedded software was literally forced to find the few people on the planet that had the know-how to develop the project. Management had no choice but to mold together disparate sites- in the United States, United Kingdom, India, and Israel – some with only two or three developers, and make them into a productive team" [4]. Sometimes tasks must be split in such a way as to maximize economic benefits. Quality Assurance requires fewer people than Software Testing. It makes economic sense in such circumstances to have software tested in low wage paying countries, so as to have has many hands on the job as possible, but still manage to get a decent ROI.

Such arrangements may still have pitfalls. Consider a story in which a misunderstanding over a single keystroke sent a software developer from Swindon, England, where he wrote his code, to Nuremberg, Germany, where the tester worked. "They kept telling us it didn't work according to the spec," says Frank Wales, a technical manager at Lucent Technologies Inc. in Swindon. "Eventually, we had to fly the developer to meet with the tester. He said, 'OK, show me what you are doing.' The test spec said to type a blank when you got to a certain point. It meant to hit the space key, but the tester was typing 'b-l-a-n-k.' "[13].

If work on the task cannot be continued because of inputs required from other side of the world, delays may invariably causes schedules to slip. Such feedback cycles can be larger during the initial

phases of the project when all issues have not been clarified or during integration if interfaces haven't been clearly defined. With the inevitability of global teams is it possible to divide the work between various teams, such that the feedback cycles get reduced? The following sections offer some suggestions to that effect.

2.3.1 Modularize to reduce dependency among cross-sites

In the paper "Distance, Dependencies and Delay in Global Collaboration" [3] the authors studied (among other things) the delay that allows us to compare single-site work with cross-site work. The study involved tracking a unit called a Modification Request (MR), which is simply a request to incorporate specific functionality into the software. The authors compared all single site MRs (sites in which the originator of the request and those who carry out the request are collocated) to MRs that involved at least two sites. The results are shown in Figure 1.

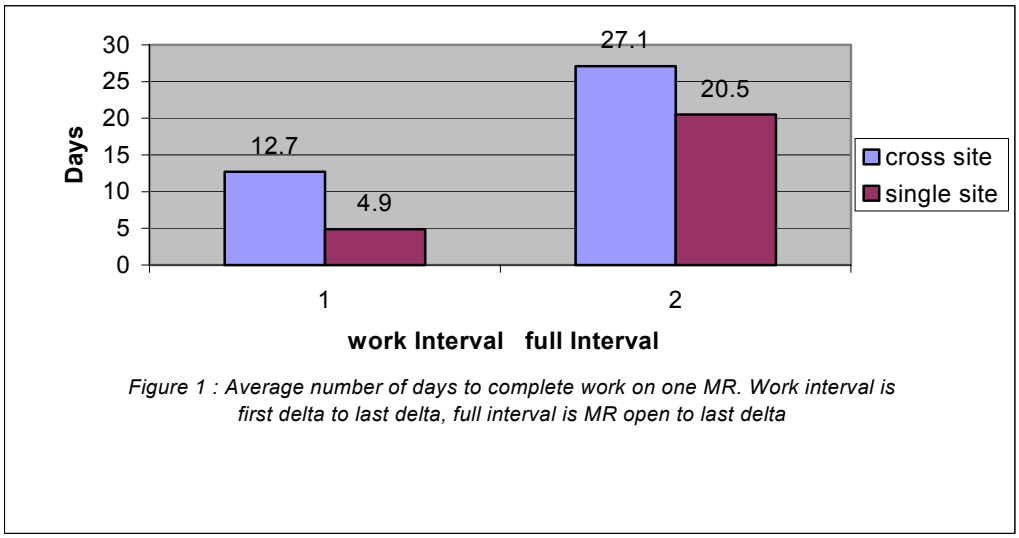


Figure 1 : Average number of days to complete work on one MR. Work interval is first delta to last delta, full interval is MR open to last delta

Figure1. Comparisons of Work Interval Times

The figure shows that the average single-site MR took about five days to complete; from the time the work actually began until the last change was made (work interval) and contrasted with multi-site MRs where it took 12.7 days, more than 2.5 times as long, to complete.

2.4 Practical Issue: Formal and Informal Communications

Software development requires much communication among the team members particularly in the initial phases [1]. "In contrast to the image of software developers as relatively introverted and secluded, they in fact spend a large proportion of their time communicating"[2]. "Email communication addresses this to some extent but email does not permit developers to carry on the conversations as effectively as possible when they are physically collocated" [3]. In general, software team members communicate through two major types of communication, namely formal and informal.

The formal communication is the official type of communication that needs a formal clear interface. The important issues of the project development for example, the update of project progress, definitely need formal methods of communication. Effective formal communications requires team members to exchange information by organizing and attending weekly meetings and by use of verbal messages through sending and receiving text messages. However, non-verbal messages help the receiver to interpret the sender's intention and for the sender to ensure that the message was received and understood.

Communication and coordination among team members are achieved with a balance between formal mechanisms, such as meetings, and informal mechanisms such as developers discussing the

problematic issues when they meet at the coffee machine, cafeteria during lunch or maybe through a chat in the hallway, parking lot and so forth. Dewane Perry [1] found that developers spend 75 minutes per day in unplanned, informal, interpersonal interactions. Programmers and analysts use these informal interactions for problem solving, informal code reviews and for analytical purposes. Informal communication contributes to creativity, quick problem solving and team bonding. These informal face-to-face meetings can also aid in clearing up misunderstandings and misjudgments.

For teams separated by geographical distance however, informal communication is impeded due to distance. Team members working at geographically distributed sites are unable to coordinate by peeking around the cubicle wall, nor can project members control the team members by strolling down the hall and visiting team members' work areas. Tom Allen [6], in a study of engineers tabulated the communication frequency of over 500 individuals in seven organizations over the course of six months and developed a profound relationship between distance and communication. He found communication drops precipitously when engineers' offices are about 25 meters or more apart from one another as illustrated in the graph of Figure 2.

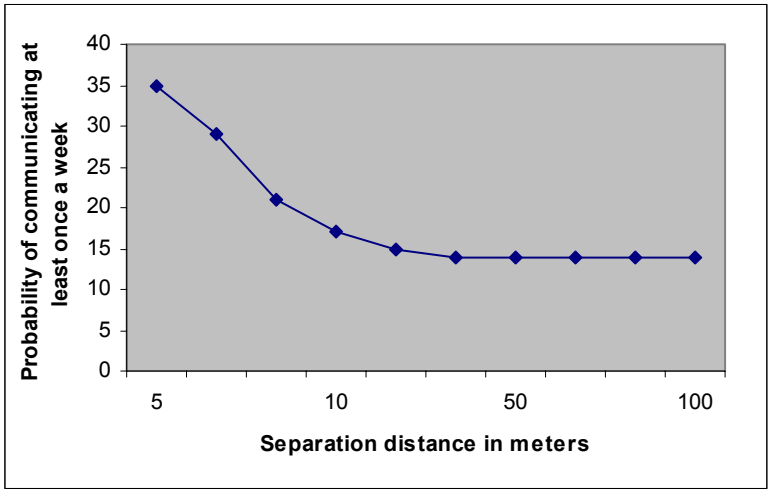


Figure 2: Relationship of distance to probability of communication

The absence of ongoing informal conversations among team members in a Global software Development can lead to potential misalignment and rework. Sometimes project issues can go unrecognized or lie dormant and remain unresolved for extended periods. Global teams must therefore encourage communication that is more informal so that communication patterns over distance emulate the single-site environment. The Push and Pull communication strategy should be encouraged. "A push strategy aims to complete the transaction through a pre-designed communication flow, often referred to as a single or multiple stepped strategy whereas a pull strategy informs the recipient of new information and allows him the freedom to access the sources of information in his own time" [11]. In Global software Development, good documentation skills and practices among developers should be insisted on as a push strategy. This should not only include updating the various artifacts but also revising and informing to team members about the new components incorporated into the project as a pull strategy. Participants must focus on their ability to communicate with other group members and follow the team's progress.

2.4.1 Lateral communication

Project Managers must encourage lateral communication, wherein communication between on-site team members occurs directly with their counterparts working in the off-site as shown in Figure 3 [4]. E-mail naturally facilitates lateral communication and leads to the integration of tasks effectively and problems tend to be immediately Lateral coordination is flexible, and assists in making more responsive decisions.

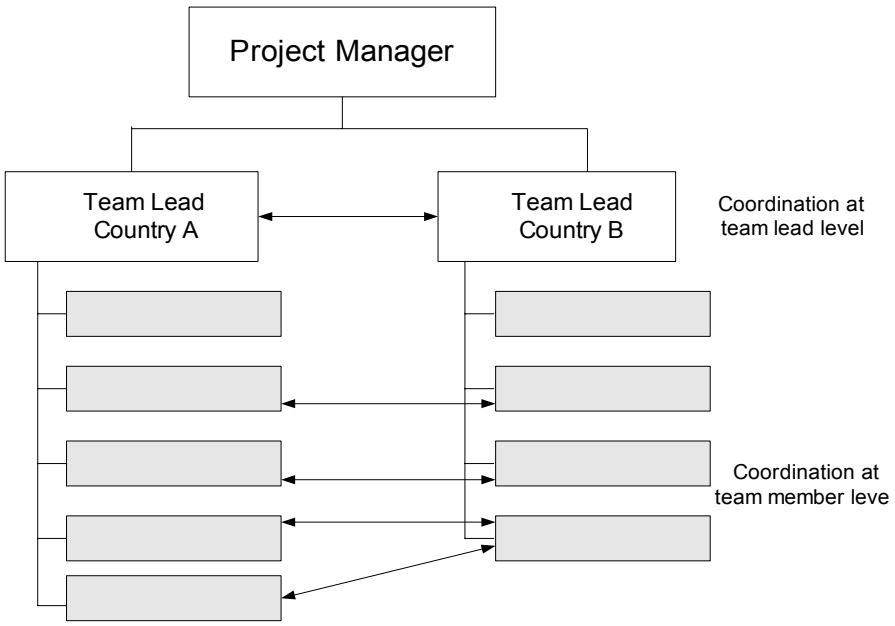


Figure 3: Lateral coordination and communication

Galbraith offers three techniques of lateralism that are applicable to global software teams:

- "Initiate inter-team events such as meetings or celebrations, e.g., kick-off meetings
- Exchange people via inter-site and inter-country rotation; team members should travel to establish personal relationships. Each team member should meet his counterparts at the offshore site to improve the inter-site cohesion and communication. Developers return from these trips with a new perspective for the people, organization, and culture which they were exposed to.
- Create a mirror organization. A mirror organization is a symmetrical organization at each site. The organizational structure and its formal roles are identical across sites. This design makes it easy for a team member of one site to identify her counterpart in another site. Then it is easier for each team member to solve problems and to build a relationship with a distant counterpart" [5].

2.4.2 Collaborative Technology

In a collocated environment, team members operate in an environment that is inherently shared. For example, in a design meeting, members may use a drawing board to convey their architectural or detailed design proposals. In teams that are distributed such collaboration is not as easy. To facilitate a feeling of shared environment, technologies such as online white boards and videoconferencing help the team members to communicate effectively with a feeling as though they are working at the same location. A collaborative technology is a "tool that enables individuals to jointly engage in active production of shared knowledge" [8]. This technology represents the "path to location transparency – bringing the distant developers together" [4].

The objectives of collaborative technology include fostering informal inter-site communication and bringing about new forms of formal inter-site communication. It supports the team's information access like finding, sorting, processing and retrieving the information that members need to solve a problem. Collaborative Technology includes asynchronous tools such as e-mail, groupware platforms, discussion lists, project management tools, software configuration management, and issue and defect-tracking databases, as well as synchronous tools such as audio conference, video conferencing, and electronic-whiteboard.

2.4.3 New Collaborative Tools

Bell Labs encountered quality problems with geographically dispersed software development projects when they used the existing communication tools such as videoconferencing and electronic whiteboards. For example, video conferencing has to be planned well ahead and there are always technical glitches such as poor audio quality, and low bandwidth. To address these problems, Bell Labs introduced four new communication tools that can be used in multi-site projects [13].

- "Rear View Mirror is a "presence-awareness and discussion tool". This is a chat room tool that displays images of all of the people in a group, highlighting those who are logged. The Rear View Mirror collaboration tool offers instant messaging and a threaded discussion board for members of a development team. Users can click on a photo to establish a voice telephone connection.

- CalendarBot is a Web-based scheduling tool that shows the planned whereabouts of project personnel by day and by month.

- Expertise Browser finds and displays relationships among people, organizations, and the code they produce.

- Collaborative Interactive Viewing Environment (CIVE), a Web-based tool box that shows the local time at each site, holidays at each site, people in a given group and CalendarBots for those people. It also contains a directory of phone numbers and e-mail addresses for the group's members." [13]

2.5 Practical Issue: Project and Process Management Issues

When teams are composed of heterogeneous individuals from different socio-cultural and educational backgrounds, separated by space and time, working for organizations that follow different processes, and ensuring a consistent process acceptable to all; a project manager seasoned for such a role becomes critical.

2.5.1 Process Management

A standardized and widely understood software development process is the key to predictable software development. Many European governments require companies bidding a contract to be ISO-certified. Similarly, companies and government contracts in US may require adherence to the CMM as a requirement.

"Process maturity positively impacts global development. CMM Level 2 is an absolute minimum to achieve sound project management (which is a prerequisite for distributed development). However, only a standardized organizational process framework allows a seamless integration of different development centers and ensures that interfaces for various work products – including work flow management – facilitate an exchange of results and shared contributions". [18]. Global companies, such as IBM, Oracle, and Microsoft, are in a position to adopt a standard company wide process policy and thus alleviate the problems experienced while working in a hybrid process environment. Having different processes to achieve the same goal, however, can be a balancing act between organizations. Such situation can be exacerbated if the situation involves client – supplier relationship following different processes, where the supplier working in tandem with the client develops part of the software. Having a common process ensures at least predictable software development. According to Tyrrell [20], when it comes to gauging the quality of the software being produced, "if we have two developers each working according to their own processes, defining their own tests along the way, we have no objective method of comparing their work either with each other, or, more important, with a set quality criteria"

Some authors [7] have argued that imposing a common process can interfere with a development center becoming immediately productive, as the center must go through a steep learning curve until it becomes familiar with the process. According to the Silicon and Software Systems (S3) Group, a

global software development company and an IT services company based in Georgia), the aim of the process should be "*to facilitate the engineer doing the job well rather than to prevent them from doing it badly.*" However, it is our view that unless the arrangement is such that a development center is working independently, working towards a common process will ensure such global teams can reap the maximum benefits that come with having a standardized software development process.

2.5.2 Project Management

The role of the traditional project manager and the global project manager are quite different. ”The traditional project manager skills are often based on the assumption that the employees are located just down the hall; that they are all there at the same time; and that they share a common culture”. [17]. Global managers, in contrast, must deal with people who are not only separated by space and time, but are from different cultures. Such varied people are used to different styles of management that are prevalent in their country. In some countries a sense of hierarchy is important, whereas in some countries companies are trying to promote a flat structure. Carmel in his book Global Software Teams mentions the example of an American manager who had to put on shows of yelling and screaming when he visited a Russian site, the reason being that Russians' had been living in a authoritarian regime for so long that he needed to show them who is the boss. [4]

According to Garrett [21], in looking for successful global project managers, companies should seek individuals that display the following characteristics:

(1) Look for the big picture and multiple possibilities.
(2) Understand the world is very complex and business is interdependent.
(3) Are process-oriented, understanding that all business follows a process of inputs, tools and techniques, and outputs.
(4) Consider diversity in people as a real asset and valued resource, know how to work effectively in multicultural teams.
(5) Are comfortable with change and ambiguity, and
(6) Are open to new experiences and enjoy a challenge.

Although there are numerous tools available that enhance the communication and coordination among global software teams, these tools are just means to an end not an end in itself. An effective global manager needs to be sensitive to effective use of these tools when communicating with diverse people. People whose first language is different may find it difficult to communicate synchronously over the telephone or in video conferencing. Such situations may be reduced by emphasizing email communication, where each person has more time to comprehend and formulate a reply. When teams are separated by wide time zones, a regular work hour for one team could mean midnight at some other location. In such a situation a global manager should plan the schedule in a way that each location alternates with regards to attending meetings during a convenient time.

Teams separate by distance are naturally prone to ambiguities. Such ambiguities can arise from reasons such as major decisions not being communicated to all concerned parties or communication which is misunderstood. A global manager who sets a clear vision for the project can affect such ambiguities caused by space, time, or culture to be diminished, focusing everyone's attention on the task at hand.

A global project management has to try to clear any ambiguities as to the roles and responsibilities between various teams. He/she should strive to make engineers in teams define how exactly what they are working on fits into the overall scheme of things. If the roles and responsibilities are not defined it can lead to hassles in integrating various components, in some cases it can also lead to overlap, reducing potential for reuse. “A global manager who has the right balance of global, managerial and technical skill” [4] becomes all the more important.

2.6 Practical Issue: Outsourcing

Outsourcing occurs when an organization transfers control of the business process to a supplier and then purchases the results of the process with no regard to the means of production. So the internal process (or parts of the process) that might be accomplished by a buyer are relinquished.

While the buyer should not attempt to have a direct influence on the precise means of production, it is an imperative that the buyer monitor the results - ideally via an individual with liaison responsibilities. The buyer can thus control the output through contract. This allows the supplier to do what it does better and cheaper than the buyer and concomitantly allows the buyer to do what it does best, namely to focus on the core of its business to maximize profits.

An alternative to outsourcing is contracting, where the main difference lies in placement of control. There is a contract arrangement if the buyer retains control of the process; an outsourcing arrangement if the supplier has control of the means of the process.

Outsourcing, in contrast to contracting, requires considerable startup costs to the supplier. Significant expenses may be incurred in establishing and developing an infrastructure (not a major factor in contracting). Expenses born by the supplier to establish infrastructure are normally factored into costs to the buyer and thus is designed to discourage early termination of their arrangement, which a buyer always has the right to stop. While contracting readily allows a buyer to move quickly to another supplier, a buyer should not enter into an outsourcing arrangement with this option in mind.

There are many different motivations to enter into an outsourcing arrangement, and while the most common reason to outsource is cost, there are a host of other reasons. “Companies outsource: to reduce and control operating costs, to improve company focus, to improve quality to access capabilities not otherwise available, to reduce cycle time, to make capital funds available, to obtain cash infusion to reduce risk, to gain flexibility, to turn fixed costs into variable costs, to stabilize an unstable situation, to engage an outside agent of change, and to free internal resources for other purposes” [24] In general, if some other company can perform a function that is not core to the business organization and can do it better, more efficiently or faster; then that function is a candidate for outsourcing.

2.7 Practical Issue: Configuration Management

Configuration Management (CM) can be used to maintain the integrity and continuity of effort of hundreds of people or that of just a few. Applied globally to software development, CM is daunting!

Evolutionary tools work well in developing organizations that are concerned with specific areas of CM. Should they later incorporate added features, it will be necessary that they either purchase the added functions or develop them on their own. Dart explains that evolutionary tools can be sufficient in some situations due to the financial considerations and reduced training required. No evolutionary combination will provide the seamless integration that is provided by full-process products though.

All companies that are even moderately successful in the software industry use CM in some form, and have automated the process to a great degree. To be successful, companies must incorporate automated configuration management into their operations. There are a wide variety of tools available that provide a varying degree of functionality. See Figure 5 taken from [23], but renamed.

There is no best tool to automate CM since any tool is only as good as the organization that uses it. But, several issues should be taken into account when choosing a tool. Certainly, the user interface should be adaptable to the type of user so that features not needed are hidden while other users might require other functions. This keeps training requirements to a minimum. Those features used should be “intuitively obvious” and all the features should be usable. According to Dart, “The best CM tool is the tool that a company can easily deploy, as well as make use of all the potential it offers.” [23] For multi-site development of non-trivial applications, managing and integrating an automated configuration management tool into the process is essential.

Business and Technical Benefits of Configuration Management

Business Benefits	Technical Benefits
Support of reactive or proactive business strategy	All objects versioned
	Metadata in repository
Insurance against the unknown	Failure recovery, rollback support
Resistance to faults and failure	Everything under version control; Repeatability of all steps; rollback
Adaptable to changes in business strategy and processes	Customizable, multi-process support
Very easy audits	Queries and audit log
More responsive to customers	Faster change cycles
Better forecasting and planning	Visibility into all work status; history
More independence	Less reliance on a single person
Foundation for process and quality improvement	Enable standards certification since all require CM
Eliminate avoidable mistakes	Process enforcement
Fewer bugs in released products and Web site	Only tested Web content is published or released
Automatic quality control	Automation of processes/workflow to give the effect of an assembly-line production
More product lines	Minimal change complexity; variant management; change propagation
Teamwork optimization	Parallel development; concurrent baselines; private workspaces
Enables survival and thriving	Control over everything

Figure 4 Business and Technical Benefits of Configuration Management [23]

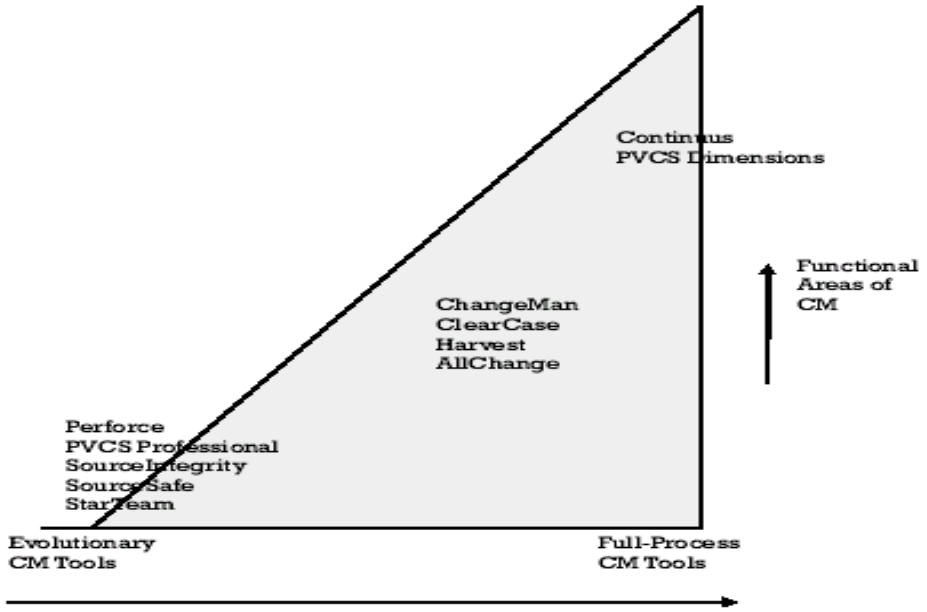


Figure 5 Configuration Automated Tool Support

3. Conclusion

Software development centers composed of heterogeneous individuals separated by space and time will become more and more commonplace. Because of economics or availability of required talent,

software development that is carried out by distant teams must do so in a very carefully, coordinated, managed manner. The tasks are indeed fraught with pitfalls, as teams may not share much in common except their means of communication and tools.

To make global software development successful and predictable we would like to suggest the following heuristics that are based on successful ventures:

- Ensure team member interaction in a more informal level.
- Use cultural liaisons to act as mediators between various teams to help teams reduce the "we vs. they" culture.
- Be sensitive to language differences when communication.
- Avoid cultural dominance. Global teams should adjust to coexist and understand members of other teams rather than impose one team's culture on the other.
- Work diligently to reduce and eliminate work delays arising from slow feedback from users located at different sites. (Feedback cycles must be reduced)
- Break down architecture into modules to minimize communication overhead between various sites, with interfaces clearly defined.
- Plan for real time contact with those in different time zones. Global teams should plan convenient timeslots to maximize the overlapping work hours.
- Encourage more informal interactions and communications between developers and analysts for problem solving and analytical purposes.
- Implement an automated global configuration management effort to the point that team members will not be burdened with it. It should be usable by all users.
- Keep emails short, very precise, and limited to one page in length. Audio conferences and video conferences should be planned during the overlapping work hours between the global sites.
- Proceed carefully. Companies with no experience in off-shoring or not familiar with rules and regulations of countries in which they want to set up an off-short development center, should try experimenting by getting into a contract with an off-shore development center and later consider setting up its own off-shore center, either by buying the company it had a contract with, or starting off on its own.

Global outsourcing has become the norm for large organizations, but it does have drawbacks. The decision to outsource globally should be considered in light of other solution possibilities and the long-term effects. Significant effort is required to establish relationships, maintain communication, develop infrastructure and coordinate tasks.

BIBLIOGRAPHY

[1] D.E. Perry., N.A. Staudenmayer., L.G. Votta. 994. "People Organizations and Process Improvements", *IEEE Software*, 11, 4, 36-45.

[2] James D. Herbsleb, Audris Mockus, Rebecca Grinter, "An Empriical Study of Global Software Development," *Proceedings of the 23rd International Conference on Software Engineering*, (ICSE '01), May 2001, pp 81-90.

[3] Herbsleb, J.D., Audris Mockus, Rebecca Grinter, "Distance, Dependencies, and Delay in a Global Collaboration," *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work,* Philadelphia, PA, 2000 ISBN 1-58113-222-0, pp 319-328.

[4] Erran Carmel. 1999. "*Global Software Teams",* Prentice-Hall.

[5] Galbraith, J. R. Designing Organizations, San Francisco, CA: Jossey-Bass Publishers, 1995

[6] Allen, Tom, Managing the flow of Technology, Cambridge MA: MIT Press, 1997.

[7] Rober D. Battin, Ron Crocker, Joe Kreidler, K.Subrmanian. 2001, "Leveraging Resources in Global Software Development", *IEEE Software,*18,2,70-77.

[8] Grinter, R.E., J.D. Herbsleb, and D.E. Perry. *The Geography of Coordination: Dealing with Distance in R&D Work*. in *International ACM SIGGROUP Conference on Supporting Group Work (Group 99)*. 1999. Phoenix, Arizona: ACM Press.

[9] The Effect of Time Separation on Coordination Costs in Global Software Teams: A Dyad Model 0-7695-

2056-1/04 2004 IEEE
[10] D.W. Karolak. 1998. *Global Software Development*. John Wiley and Sons, Inc.
[11] "The Confused State of New Media Deployment ", http://www.one1.com.sg/Resource/newmedia.htm
[12] 2000 Annual NASSCOM Report. http://www.nasscom.org
[13] "Software Development Goes Global": News Story by Gary H. Anthes. June 26, 2000 http://www.computerworld.com/softwaretopics/software/story/ 0, 10801, 46187, 00.html
[14] Tyler Cowen. Sept 23,2002. *Cultural Destruction: How Globalization is Changing the World's Cultures.*
[15] Oppenheimer, H. Project Management Issues in Globally Distributed Development
[16] 2001. llya Adler, http://www.mexconnect.com/mex_/travel/bzm/bzmadler30.html
[17] 2000. Kimball Fisher., Mareen Duncan Fisher. The Distance Manager: A Hands-On Guide to Managing Off-Site Employees and Virtual Teams.
[18] Christof Ebert., Philip De Neve. 2001. "Surviving Global Software Development", *IEEE Software,*18,2,62-69.
[19] Richard Nisbett. March 3, 2003. *The Geography of Thought : How Asians and Westerners Think Differently… and Why.*
[20] The Many Dimensions of Software Process: Sebastian Tyrrell http://www.acm.org/crossroads/xrds6-4/software.html
[21] Gergory A. Garrett. 2004. "Global Project Management: Best Practices", C*ontract Management*.
[22] Richard A. Shweder. April 2003. *Why do Men Barbecue? Recipes for Cultural Psychology*
[23] Dart, Susan, Configuration Management: The Missing Link in Web Engineering, Boston, MA, Artech House, Inc., 2000.
[24] Turning Lead to Gold: The Demystification of Outsourcing, Bendor-Samual, Peter, Provo, Utah Executive Excellence 2000
[25] http://www.theinquirer.net/?article=6010
[26] C.L. Miller. 1994. "Transborder Tips and Traps", *Byte,*19,6,93-94.
[27] Rober D. Battin., Ron Crocker., Joe Kreidler., K.Subrmanian. 2001. "Leveraging Resources in Global Software Development", *IEEE Software,* 18, 2, 70-77.
[28] Jesus Favela., Feniosky Pena-Mora. 2001. "An Experience in Collaborative Software Education", *IEEE Software* 18, 2, pp47-53.
[29] S. Krishna., Sundeep Sahay., Geoff Walsham. 2004. "Managing Cross-Cultural Issues in Global Software Outsourcing", *Communications of the ACM,* 47, 4, 62-66.

Simulators, or How to Turn Your System's World Upside Down

David Liebreich
Independent Consultant
Dave@DaveLiebreich.com

Bio:

Dave Liebreich is an independent consultant with over fifteen years of experience supporting the development of complex, distributed, networked systems using non-commercial tools. Dave loves figuring out “How are we going to test *that*?” as both a manager and an individual contributor.

Abstract:

Have you ever wanted to test what your system would do if it tried to talk to a broken web server? What if the database server was so bogged down, it didn't respond for over 2 minutes? And what if the web service returned an internal fatal error?

Simulators are an answer. Like mock objects at the unit test level, simulators at the system test level provide a mechanism to control the world your system talks to.

I've used simulators in much of my testing of networking and computing infrastructure products, both at the functional and the system level. In this paper, I will describe my experiences with one of the simulators that I developed.

Copyright © 2004 Dave Liebreich

Background

In 2000, I was working at CacheFlow Inc. as a lead test engineer for a product named **Director**. CacheFlow's flagship product was a web cache appliance - a box that sat on the network between end users and the Internet and reduced bandwidth usage by caching popular web pages. Director was a kind of network management appliance for these caches, allowing cache administrators to monitor and update multiple caches using a single interface.

As we built out the test lab, we realized that we could not get enough caches to do sufficient load testing. Even if we could have obtained the boxes, we did not have enough rack space or power to keep them up and running. So I decided to write a program that would simulate a single cache. We could run multiple instances of that program on a single PC running some form of Unix (Linux, NetBSD, FreeBSD, ...), saving space in the lab and allowing us to "point" a Director at hundreds of these virtual caches.

Caches offered web, Simple Network Management Protocol (SNMP), and CLI (Command Line Interface) based administrative interfaces. Most administrators used the web interface for configuration, and SNMP within HP OpenView for monitoring. Director accessed the caches using the CLI, which made it easier for me to write a cache simulator as I only had to reproduce that one interface.

I chose to write this simulator in Perl, but could have used a number of other languages just as effectively. My main requirements for the language were:

- regular expression support
- small-enough runtime footprint
- already known by the rest of the development team

I do not have access to the original simulator source, so all code examples in this paper are simplified versions of what I can remember.

Experience

My first task was getting the basic communications working. The cache CLI was accessed via telnet or ssh, much like remote access to a Unix system. Rather than write the telnet and ssh server-side code as part of my simulator, I set up regular user accounts on a Unix box with the cache simulator program as the login shell.

The “dialog” when connecting to a cache using the telnet protocol looked like this (user input in bold):

```
username: username
password: ******
CacheOS>
```

The cache echoed the username in clear text, and echoed an asterisk for each character of the password. This is slightly different from the standard Unix login sequence:

```
Welcome to SuSE Linux 8.2 (i586)
username: username
password:
Last login date of last login
shell $
```

where there is leading and trailing text, and nothing is echoed for the password. I modified the system configuration to suppress the extra text, and luckily the code in Director did not care whether there were asterisks echoed at the password prompt.

Since the ssh protocol itself handles transmission of the username and password (or other authentication information), the first response by the cache to a ssh connection attempt with valid username and password was

```
CacheOS>
```

The same changes on the Unix box to suppress extra text during telnet logins also worked for ssh logins.

With these configuration changes, the simulator code looked like:

```
#!/usr/bin/perl
print "CacheOS> ";
```

The standard dialog between Director and a cache continued as follows (commands sent by Director to the cache are in bold type):

```
CacheOS> enable
password: *******
# show version
version information
#
```

So I added to the simulator code:

```
#!/usr/bin/perl

use strict;
use warnings;

my $linevar;

$| = 1;

print "CacheOS> ";
$linevar = <>;
print "$linevar";

print "password: ";
$linevar = <>;
print "********";

print "# ";
$linevar = <>;
print "$linevar";

print "CacheOS Version X.Y.Z\n";

print "# ";
```

The next batch of commands I needed to implement were those sent by Director to the cache every five minutes or so. These commands both reset the idle timeout counter for the cache CLI session and provided cache status information to the Director monitoring engine.

Up until this point, I had been relying on conversations with the programmers, reading the Director source code, scanning the Director logs, and watching packet traces from a network sniffer to get through integration problems. I decided to add some basic logging to the simulator itself in hopes of increasing my development speed.

I replaced all the `read` and `print` statements with calls to new subroutines `read_log` and `print_log`, which looked like:

```
$remoteaddr = $ENV{"REMOTEHOST"} || "local_hostname";

sub log_it {
  print LOGFILE scalar localtime() . " $remoteaddr @_\n";
}

sub print_log {
  print "@_";
  log_it "sent >>@_<<";
}

sub read_log {
  my $line = <>;
  print $line;
  log_it "received >>$line<<";
  return $line;
}
```

With this change, I could see what was sent or received, to or from which Director, and at what time. Most of my remaining integration issues were diagnosed and fixed using information from the simulator log and Director's log, including the time I inadvertently pointed two Directors at the same simulator instance (oops). Later, I integrated the log contents into the pass/fail logic of some of the test cases.

Back to the periodic commands. I knew I'd eventually want to turn the simulator architecture into a state machine, but I didn't want to do that until I had a feature that required it. So the new commands were implemented like:

```
while ($_=read_log()) {

  /^show cpu/  && do { print_log "CPU 55%\n# ";    next;};
  /^show disk/ && do { print_log "Disk 80%\n# ";   next;};
  /^exit/      && exit;

  print_log '%Invalid Command' . "\n# ";

}
```

At this point, I had a tool I could use for simple capacity testing of Director. I configured 100 instances of the simulator on a single Unix host, then configured a Director unit to manage those 100 simulated caches. We had tested Director configurations managing over 500 caches, but those caches did not exist on the network so all the “connections” were timeouts with retries. Using the simulator, I found a number of bugs even when Director was only talking to 10 caches.

A number of our existing tests covered the detailed interactions between Director and caches. Once the Director code was fixed to handle 100 caches, I used the simulator to augment those tests even though the simulator code could not handle the commands the test generated. I configured a Director to manage 3 live caches and 97 simulated caches, then ran those tests using only the 3 live caches as targets for the interactions. This found bugs that did not occur when just the 3 live caches were used, even though the 97 simulated caches were “idle.”

My next task was to implement the content management commands. These commands were used to pre-load specific web pages on to the cache, query the cache about the status of these pages, and delete pages from the cache.

The command to pre-load a specific web page was:

```
content distribute url url
```

The cache returned an error code if the specified url did not pass some rudimentary syntax checks. If the url was OK, then the cache queued the fetch request and printed the current time before returning the CLI prompt.

Another command:

```
show content outstanding-requests
```

listed all the fetch requests currently queued on the cache. Once a fetch command completed, it was removed from the queue and no longer was listed in this output.

The command:

```
show content url url
```

displayed information about the specified url. This information included a status field which specified if the url was already in the cache, queued for a fetch, or currently being loaded into the cache. If the last fetch attempt resulted in an error, the status field contained the error code from the failed attempt.

Queued or in-progress fetch requests could be canceled in bulk or on an url-by-url basis with the commands:

```
content cancel outstanding-requests
content cancel url url
```

and content could be removed from the cache with the commands:

```
content delete url url
content delete regex regex
```

On a live cache, both the cancel and delete commands added requests to an internal queue, then returned to the CLI prompt, so there was a small window between the time the cancel or delete command was received and the time when an in-progress fetch was shut down or the content actually disappeared from the cache. This lag only affected interactions between the cache and browsers or web servers, not between the cache and Director, so I did not try to reproduce it in the simulator.

I used the URL as a covert channel to control the behavior of the simulator content management commands. The default behavior was to accept the URL as-is and respond as if there were a 5 second delay before contacting the web server, a 20 second delay while the content was downloaded, and a final result of a successful download. This behavior could be overridden by embedding certain strings in the URL itself:

If the URL contained	***The simulator responded***
vvREJECT=*nnn*vv	The command was rejected immediately with error code *nnn*
vvDELAY=*nnn*vv	The initial delay was set to *nnn* seconds (instead of 5)
vvDL=*nnn*vv	The download time was set to *nnn* seconds
vvRES=*ccc*vv	The end result was set to the standard HTTP response code *ccc* – if the code started with “2” then the download was marked successful; otherwise it was marked as a failure

So the pre-load of the URL http://some.web.server/doc/vvRES=404vv was a simulation of a 404 response from the web server, and the URL rtsp://some.web.server/stream/vvREJECT=15vv was rejected immediately by the simulator with the error code 15.

Now I could use the simulator to test Director's behavior when issuing actual content management commands. I could also cause the simulated cache to return any error code for any content management command, which was very difficult to do on a live cache. And I found a lot of bugs.

Influenced by this success, I wanted to add to the simulator the ability to return all possible errors for all possible commands, or simulate degenerate cache behaviors like heavy load (very slow response), freezes (cache no longer responds to any command, but the connection stays up), and crashes (connection is dropped).

Some of these errors were triggered by embedded strings in URLs or passwords or other text fields, just like the error behaviors for the content management commands. Embedding the string “vvFREEZE=12” in the enable password configured the simulator to cease responding to new commands after processing 12 commands from Director. Setting the SNMP read community string to “SLOW=45” simulated a heavily loaded cache, where there was a 45 second delay before any command response.

Other behaviors were triggered by the contents of a configuration file that the simulator read at startup. Overriding default values, prepopulating the simulated cache with outstanding content requests, setting the cache software version or hardware configuration, or immediately simulating a cache crash could not be done with in-band commands because the simulator behavior needed to change before any commands were received.

One additional feature I added was the ability to specify a different embedded-string delimiter in the configuration file. For example, I would leave some caches set up to use the default delimiter (“vv”), some to use a different delimiter (“ww”), and some to use a regular expression that matched both (“vv|ww”). A fetch of the URL “http://some.domain.here/wwRES=401ww” sent to all of the caches would succeed on some and fail the fetch with HTTP error 401 on others.

Our group maintained a suite of automated tests that were run by a simple test harness against every build. The harness handled basic setup and teardown of the testbed, some resource allocation, and a simple set of scripts to compare test run output and log files against known-pass versions. In order to help other testers write test scripts that used the simulator, I needed to integrate some basic housekeeping commands into the harness.

On other projects, I had written tools that used out-of-band control channels. The harness would connect to the tool on a predefined network port, and could transfer or reset logs, change configuration parameters on the fly, or query various statistics. But the code to do this was harder to maintain, and we did not have any tests that required on-the-fly changes to the simulator configuration.

Most of the test scripts needed only to reset the simulator log and copy over a specific configuration file before starting the actual testing, then copy back the resulting log file at the end of the test and scan its contents for unexpected commands sent from Director. I wrote some simple scripts (in Perl, shell, and Expect) that would log in to the simulator host machine as a particular userid, then move, copy, or delete files as needed. These scripts, installed in the standard harness utility directories, were enough to get us through all the testing on two releases of Director.

Analysis

Some of the lessons I learned and/or applied were:

Simplicity

> Simple code that works is a good goal for any software project, but test tool development needs simple, easy to understand code more than most.
>
> If you find a bug using the test tool, the programmer investigating the bug needs not only to understand the code under test, but also needs to understand the test tool operation. Simple test tool code makes this easier on the programmer, where complex test tool code may earn the resentment of the programmer and worsen the tester-programmer relationship.
>
> Simple code is easier to test, and you need to test the test tool extensively. Test tool bugs are among the most frustrating a tester can discover – don't wait until the actual testing to find them in your code.

Work on the communication interface first

> The communication interface between the simulator and the system under test is likely one of the most complex areas of the simulator. Whether that interface is a text CLI, email, HTTP, SOAP/XML-RPC, or something else, you don't want to be debugging the connection while you are trying to implement new simulator features. So get the connection working solidly with the simplest case possible before moving on.

Deploy early and deploy often

> As soon as the simulator could handle a new test, I released it into “production.” In general, test tool development projects thrive with an iterative development style.

Choose the right configuration techniques

> The major configuration techniques are in-band covert channels, out-of-band static, and out-of-band dynamic. You can use all three, or just one, but pick the ones that best meet the detailed need.
>
> An in-band covert channel is any mechanism to pass information through the regular interaction of the system under test to the simulator, without affecting the operation of the system under test. Text fields like URLs or names or passwords can be used as covert channels, as can certain sequences of operations. The benefits of in-band covert channel configuration include deterministic timing of actions (the test may run so fast that it would be difficult for an external source to change the simulator config at the right time), the same simulator can be used by multiple systems under test (since each session is configured independently using in-band methods), and the test scripts are usually easier to understand (create a new user with the username “ThisWillFail” is fairly straightforward).
>
> Out-of-band static configuration sources are things like init files, registry keys, and database contents. The advantages of out-of-band configuration include greater bandwidth (in-band channels are constrained by the length of the text fields), and you can configure the behavior of the simulator during the initial connection phase (before any in-band channels might become available), and it's easier for the test harness to interact with the simulator if it does not have to emulate the system under test.
>
> Out-of-band configuration can be made dynamic by adding a command to re-read the configuration sources so that the simulator configuration can be changed on the fly. You could also include a SNMP agent in the simulator code, or provide a separate configuration communication interface. The dynamic channel can also be two-way, providing a mechanism for an external source to query the state (including statistics) of the simulator while it is running.

You don’t need to write everything from scratch

> Search far and wide for publicly available code that does what you want. When writing the cache simulator, I was able to use the Unix login shell framework to handle initial authentication, client identification (the remote ip address of the Director was in the utmp file), and multiple connection management. There are plenty of HTTP, SMTP, SNMP, SOAP/XML-RPC, and other protocol implementations available – use them.

Don't try to make an exact match

> In most cases, you don't need to duplicate the original functionality completely and exactly. Implement the minimum necessary good paths through the interactions, and focus on the bad paths that are hard to induce in the original. The first (minimum good paths) gives you enough functionality in the simulator to augment load and stress tests, and the second (bad paths) expands your coverage of how the system under test responds to problems in the environment

Use originals with simulators when doing load and stress testing

> Regarding load and stress testing, do not rely solely on simulators when building up the testbeds. There might be bugs in the original component that only show up when interacting with the system under test, and these bugs are probably not in the simulator. The original component may behave differently enough from the simulator that additional bugs in the system under test are exposed. And your customers – the development and management teams – will probably want to hear that you are doing “real-world testing” with the actual components the customers will be using.

Log the interaction, not the internal state

> Once you are using the simulator in your tests of the system, the simulator log's primary function is to help determine test pass or fail. So don't record simulator internal state indicators in the log; instead, record only that information that exposes the interaction between the system under test and the simulator. That information is usually who sent what when (the id of the system under test, what the simulator received, and the timestamp of the interaction) and what was sent back to whom when (the reply that was sent, the timestamp of the interaction, and the id of the system under test).

Let the test harness help, but not too much

> The test harness is good for resetting logs and creating external configuration files, but make sure anything that the test harness does can be quickly done manually as well. I've seen tools that were so tightly integrated with the harness that they could not be used stand-alone. Programmers and testers who wanted to explore system behavior ended up not using the tool because they did not want the additional overhead of the full test harness.

Use version control and labeled releases

> Source code control is such a small overhead that I recommend using it for all code used in testing. Labeled releases help recreate the testbed where a bug was found, and allow certain simulator versions to be tied with specific regression suites.

Summary

When I left the project, the simulator could be controlled by the startup file, a library of template files, and special embedded strings in various text fields of cache commands. We used the simulator in functional testing where it was difficult to induce specific, deterministic behavior in an actual cache. We also pointed the Director at a mix of live and simulated caches during stress and load testing. Using the simulator allowed us to find bugs that would have been difficult or impractical to find using only live caches.

Keeping the simulator simple increased our confidence that when we saw an unexpected behavior, the bug was in the system under test rather than the test tool code. And keeping the simulator loosely coupled to the test lab environment (including the test harness) made it easier for developers and other test groups to use the tool. The feedback from this increased exposure was invaluable in deciding what new features to add.

How to Make your Bugs Lonely
Tips on Bug Isolation

Danny R. Faught
Tejas Software Consulting
faught@tejasconsulting.com

Bio: Danny R. Faught frequently writes and speaks about software testing, including papers and courses that cover bug isolation. He is the Defect Tracking Technical Advisor for StickyMinds.com. Danny has 11 years of testing experience.

Abstract: Yikes! Testers often trip across software bugs. The more we can isolate the problems we encounter, the more likely it is that we can help the programmers who fix the bugs, and the more likely it is that the problems will be fixed at all. From the first symptom to reporting and then debugging, we go through a process of isolation to zero in on where the bug lies. Bug isolation is a highly creative process, but also one that requires a great deal of discipline. Sometimes this takes many hours of effort. Have you thought about making that process more efficient? This paper will highlight practices that help to efficiently narrow down the possible causes of a bug. It will discuss the top down vs. bottom up approach and which one often send us in the wrong direction. There will be examples of using the bisection technique to quickly narrow down a precise quantitative value. There is a gray area between the defect isolation performed by a tester and the debugging that the programmer does. Finally, we'll explore the pros and cons of the different ways to split this task between the two roles.

Copyright 2004 by Tejas Software Consulting

1. Introduction

"A problem well stated is a problem half solved." –Charles Kettering[1]

Bug reporting starts with a symptom—something in the software doesn't work right. Some testers would report that symptom and move on, leaving it up to a programmer to trace the symptom to its cause. After all, there's a lot more testing to be done, right? But most testers could do much more to isolate the bug before they report it, and their organization would likely be better off for it. In this paper, I will give you some tips on how you can discover additional details about the bugs you find, and thus write more specific bug reports that are more likely to result in getting the bugs fixed.

The introductory quotation captures the fact that reporting a bug can be fully half of the process of fixing it, perhaps more. But only if the tester takes the initiative to do the work to isolate the bug, so that the bug report is an array of spotlights pointing to a very lonely bug with very little cover to hide behind.

I took a couple of shortcuts to simplify the wording in this paper, assuming that you're using a bug tracking database, assuming that you might be testing a client/server system rather than a simple desktop application, and using terms related to a GUI or web-based interface. If your situation is different, the concepts presented here should still apply. Also, I use the term "bug" where others may say "defect" to mean the same thing.

2. What happened?

Your quest to isolate a software bug starts with the "Oops!" You notice some symptom that indicates that something might not be right. It's your job as a tester to learn more about what happened.

At first, treat each bug as if it were a rare occurrence, and carefully collect any data that you can. Get into a habit of never dismissing an error dialog without considering whether you should capture it. Save the exact text of any error messages you see, and take screen shots. If you get a string of errors, save all of them. Save as much as you can – disk space is cheap, and it only takes a few moments to copy and paste text into a file, or to dump the state of the screen. Have a favorite text editor and image editing software close at hand.

Why take such care? Sometimes you really might not be able to reproduce the bug. The evidence that you collect at the scene of the first symptom may be the only data that you're able to report. Having the details exactly right is far better than trying to remember what an error message said. Also, you might get slightly different symptoms each time. By keeping careful records, you can gather important information about the nature of the bug that you're chasing. You might even want to use a video capture tool that creates a full-motion video of your interaction with the program.

After you have gathered your evidence, you still should be reluctant to reset the system or the software. Try to reproduce the bug without changing anything that you don't have to. There might be some sort of state that's critical for reproducing the bug. If you can reproduce it now but not later, you know that there is some sort of state involved, but at least you'll know that you saw the bug more than once. Save any input data that you're using, in case you make changes later that cause the bug to mysteriously go away.

What if you simply can't reproduce the problem after the first occurrence? You should report the bug anyway, using all the information you know. Unless your organization persecutes people who report unreproducible bugs, you want to make a record of what happened. Clearly indicate in the report that you were not able to reproduce the bug. Someone may be able to gather further information if the bug recurs later.

With several years of testing experience, you can develop a sense for which bugs will be difficult to reproduce. For example, when I get a crash that comes out of the blue, perhaps while not giving the application any input at all or just typing text, I've learned that I'm not likely to reproduce the bug easily. You'll notice other patterns as you get experience with bug isolation.

If you've reproduced the bug several times, and you're getting the same failure each time, you can start to play with the state. Here's the ideal – completely reinstall the software under test and the operating system on your computer and all servers involved, restart everything, and then show that you can still reproduce the bug. Now realistically, the times you'll need to go that far will be rare if ever. But remember that if you don't go to that extreme, there might be something about the current state of the system that's necessary for reproducing the bug but isn't there when the programmer tries to find the bug later. So consider restarting the application, restarting the software on the server, and perhaps even rebooting one or more of the systems involved. Or, if you have access to a different system

running the software, try to reproduce the bug there. The more varied the scenarios you've seen the bug, and the less your current state is affected by the states the system happened to be in when you were testing, the more likely it is that you have a simple bug that can be easily reproduced.

I'll use an example of a real bug I recently found in the Mantis bug tracking system as an example through the next several sections. The example will be a running narrative in italics throughout the paper. Hopefully it won't be too confusing talking about reporting bugs that are in the bug tracking system itself. I was testing Mantis during the course of writing a review. Mantis has a web-based user interface.

> *I had been doing exploratory testing on Mantis for a while. When I try to submit a new bug, I get the error: "APPLICATION ERROR #1303, Invalid value for field."*
>
> *I copy the text of the error and paste it into a text editor that I'm using to take notes. I click the Back button and try to submit the bug again, and I get the exact same error. I do it one more time, and still the same error. So it looks like I have a reproducible bug, at least with the current state of the system.*

3. Simplify

Think about the sequence of events that you just took the software through to eventually get to the problem. Ideally, you started testing by clicking one button, and then saw the problem immediately. More likely, though, you had been testing for a while, possibly for hours. Perhaps the last thing you did is the only thing required to reproduce the bug, or maybe you have to repeat hours of testing. Until you can prove that a simpler scenario is sufficient, you have to assume that every detail of your testing session is relevant. Your task is to rule out as many of those details as you can as not being relevant to the problem.

There are many approaches that you can take to simplify your scenario. You want to form hypotheses and then conduct experiments to prove or disprove those hypotheses. Here's a suggested approach that I've refined over the course of reporting hundreds of bugs.

3.1. What to simplify?

There are several different things that are subject to simplification. Consider:

- Procedures. This is usually what testers focus on – shortening the step-by-step interaction with the system.
- Inputs. This is all the data that you feed to the program, such as a command-line argument, a text field in a GUI interface, a file, or database. You want to also reduce this data to the smallest data set that still reproduces the problem.
- Configuration. What options have you selected that are different from the default configuration? If you can't reproduce the problem the way the software is configured out of the box, find the few crucial settings that are necessary for the problem to show up.
- Platforms. Can you reproduce the problem on all of the operating systems, operating system versions, and hardware combinations that are supported? If not, then you've found an important clue. Also, what about other software that is running, and their versions? Many bugs are not platform-specific, and testing on other platforms can sometimes be difficult, so this area often isn't thoroughly explored.
- Other state information. The items above probably don't capture every possible relevant variable. Look for other things that might vary from one system to another and cause the bug to manifest on some systems but not others.

> *In our example, there were the procedures I followed to submit the bug, plus many things I had done before that that might be relevant. There were inputs I typed into the web form, plus the existing data in the database that might be affecting the results. There were a large number of configuration variables that I could set on the server, though I had only modified a few of them. Also, I had added a custom field using the administration interface.*
>
> *Mantis can run on many different operating systems and hardware configurations, and it requires a web server (several different ones will do), MySQL database, and PHP, each of which has several supported versions. I was using a 100 megabit network with only a router in between; no firewall or switches involved. I had tried unsuccessfully to install Mantis on a different system, so I didn't have another test bed easily accessible. I did have access to a production Mantis installation—the one used to track bugs against*

Mantis itself, but it wouldn't have been appropriate to corrupt that database with test data.

3.2. The bottom-up approach - Are you feeling lucky?

You might have a hunch about a particular small part of your scenario that is the key for reproducing the problem. If so, go ahead and try those steps in isolation, and you might be able to remove a large chunk of irrelevant details very quickly[2].

These hunches often lead to dead ends, though, so if it turns out that your simplified scenario does not reproduce the bug, you've probably removed a necessary step somewhere. The best thing to do at this point is to take a more systematic approach. Otherwise, you could spend a lot of time following an unproductive path.

> *In our example, I remember that I had recently created a custom field, and I 'm pretty sure that this is the first time I had tried to report a bug since then. So I have a hunch that this custom field is an important factor in what's going on. I remove the custom field from the project, and sure enough, I can submit a bug without seeing the error. So I'm well on my way toward figuring this out.*

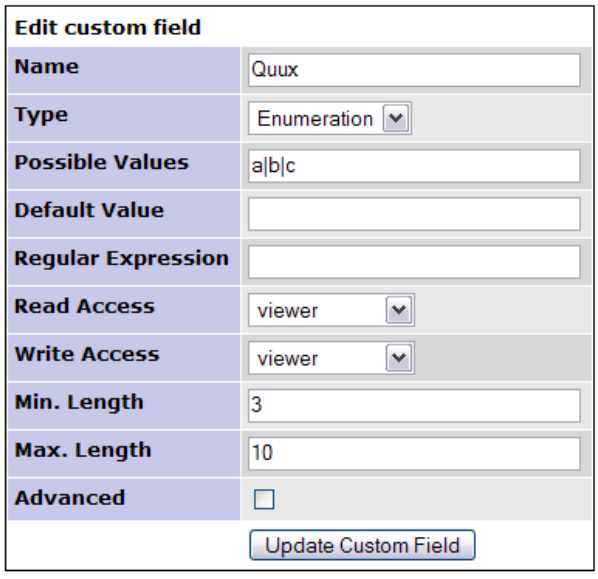
3.3. The top-down approach

When your hunches haven't reduced your scenario sufficiently, it's time to take a more objective approach. The top-down approach is a systematic method for removing irrelevant details from your scenario, and it takes a lot of discipline to do well. I even try to think like a computer when I do this, to guard against getting trapped by assumptions I might make that would send me on a wild goose chase.

Consider all of the factors involved as you simplify—procedures, inputs, configuration, and platform. Don't change too many variables at once, or you could get confused. You want track two scenarios—the simplest scenario so far that reproduces the bug, and the most complex scenario that doesn't reproduce the bug. Your goal then is to get these two scenarios to converge so that the only difference between them is the critical variables that you need to know to reproduce the bug.

If a feature is completely broken, then you won't find a scenario involving that feature that doesn't reproduce the bug. It's important to try several different ways to get the feature to work before you declare it completely inoperable.

> *In our example, we have the scenario without the custom field where we don't see the bug, and the one with the custom field where we're hitting the bug. I turn the custom field back on, and confirm that I get the same error. So I go look at the details of how I set up the custom field to see if I get can a clue about what's going on:*



Hmmm, any clues here? Say, what does “Min. Length” and “Max. Length” mean here? I change the Min. Length to 1, try to submit a bug, and sure enough, I don't get the error. I change it to 2 and confirm that I get the error again. Now I've found the threshold, and it's pretty clear that the Min. Length is being applied to the size of the enumeration value (“a”, “b”, or “c”). I then remove the value for Max. Length and reproduce the bug again, which further simplifies the scenario.

I could also put just one or two values in the enumeration - “a” or “a|b” instead of “a|b|c”, but that seems to be overkill, and just having “a” makes it look like a scenario that a real user wouldn't try.

I report bug 3793 against Mantis (http://bugs.mantisbt.org/bug_view_page.php?bug_id=0003793):

```
Summary: Enumeration values might be impossible to select

Description: On the Manage->Manage Custom Fields->Edit Custom Field
page, it's possible to create a value for an enumeration that can never
be selected. This happens if the length of the enumeration value is
less than the Min. Length, longer than the Max. Length, or doesn't
match the regular expression.

These checks don't make sense for enumerations. Consider issuing a
warning if they are defined when an enumeration is selected as the data
type for the custom field.

The result when selecting an enumeration value that fails these checks
is the unhelpful message:

APPLICATION ERROR 0001303
Invalid value for field
```

3.4. The bisection technique

Bisection, also called a binary search, is an efficient algorithm for searching sorted lists. But with some creative thinking, it can also be a big help with bug isolation. Here's how you'd use the algorithm to find a word in the dictionary: open the dictionary to the page that's in the middle of the book. Check to see if the word is earlier or later in the dictionary. Then find the middle of the first half or last half of the book, depending on which direction you need to go. Repeat, by checking the middle of successively smaller and smaller sections, until you get to the right page. You'll find your word quickly, especially compared to searching linearly starting at page 1.

Now let's apply it to testing. Let's say you generate a data file full of a million random characters, and your application crashes when you give that file as input. You should try to find the smallest possible file that reproduces the crash. After all, a bug report about a 10 or 1 byte file causing a crash is much more impressive than a bug report about the large file. Using the bisection technique, you can quickly narrow down the exact file size that reproduces the bug, such that a file that's one byte smaller works fine.

In practice, it’s not critical that you follow the bisection algorithm exactly, just like you don’t have to divide the dictionary pages exactly in half in order to quickly find a word. In fact, I often will jump toward the far end of a range by a factor of ten or more when I’m first getting started, as the example shows.

As another real example, consider this testing session for Mantis that illustrates one of my favorite attacks.

> *I use a Perl script one-liner to generate a text file containing a million “x” characters on a single line. (See the article “A Lesson in Scripting”[4] for more about that one-liner.) Then I upload that file as an attachment to a Mantis bug. It uploads okay, but when I try to download the attachment, the browser just gives me a blank screen. (Good thing I didn't assume all was fine when the upload worked.) This is what it looks like—notice the horizontal scroll bar:*

Here's what I did to find the smallest file size that fails, false starts and all. Each item lists the file size I used and the results. Each iteration takes less than half a minute.

1,000,000–Empty page. I notice that Mantis displays the size of the attachments, which gives a nice verification that I got the file size right. While the bisection technique says I should try 500,000 next, instead I follow my intuition that the number might be orders of magnitude smaller than a million.

10,000–Empty page. I copy file path into the clipboard so I can easily enter it into the upload form next time. I notice that I inadvertently verified that Mantis can load multiple attachments with the same file name. Next I'll make another big jump.

100–Works. All the characters are visible. I'll indicate the largest working file size (100) and the smallest problem file size (10,000) as "[100-10,000]" as we narrow it down.

1,000–Works. The range is now [1,000-10,000].

5,000–Empty page. [1,000-5,000] I click and drag on the area of the browser window where the text would be, and notice that it actually shows up when it's highlighted. So we're probably looking at a browser bug here. I load the data file I generated directly from the local disk into the browser, and it displays fine. Interesting. But I want to get this number narrowed down before I explore that any further.

3,000–Works. [3,000-5,000]

4,000–Works. [4,000-5,000]

4,500–The text is again invisible. [4,000-4,500]

4,200–Invisible. [4,000-4,200]

4,100–Invisible. [4,000-4,100]

4,096–Invisible. [4,000-4,096] I realize that we're close to a power of two, which is a likely place for a bug to lurk. 4096 is 2^{12}.

4,095–Invisible [4,000-4,095] So the bug isn't right on the power of two boundary. Back to a more mechanical approach.

4,050–Works. [4,050-4,095] I still think it could be close to 4,096, so I'll jump further in that direction.

4,080–Works. [4,080-4,095] We're getting close now.

4,090–Works. [4,090-4,095]

4,093–Works. [4,093-4,095]

4,094–Works. [4,094-4,095] Looks like we've found the critical point, one off from a power of two boundary. Let's check the other side of the boundary to verify that it isn't moving.

4,095–Invisible. Yup, we got it! We can report that a line 4,095 characters long or longer doesn't render properly in the browser. But not yet, so stay tuned.

Narrowing down a failure to a precise failure point like this may be misleading. For example, see bug 3798 that I filed against Mantis[3]. I identified the threshold where the error started occurring, but after restarting the web server, the threshold moved. I suspect that cases like this occur when the failure depends on how many system resources are

available, such as memory. The only way I could have found this out was by determining the precise failure point each time, and the bisection method made this fairly easy to do.

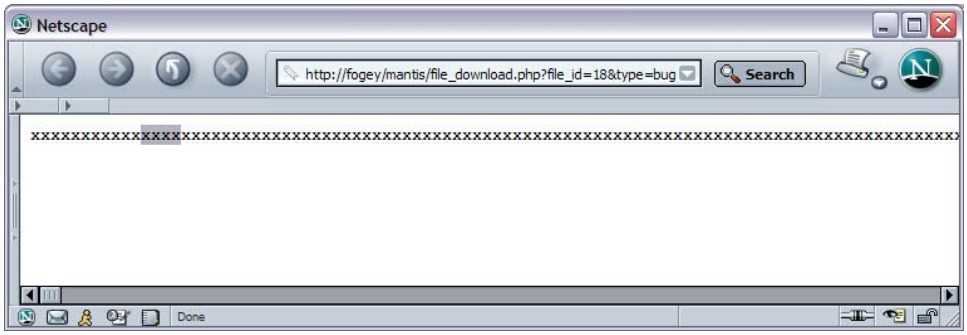
3.5. Refinement

Once we have isolated the bug, we still have the opportunity to discover additional information about it that will help with characterizing the severity of the bug and jump start the bug fixing process.

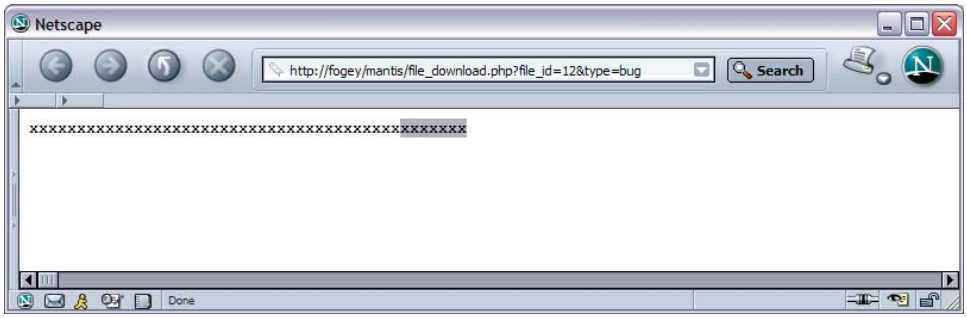
Let's go right back to the example where we left off. Again, the numbers on the left indicate the size of the file that I upload.

> *When I check the 4,095 character file again, I notice that when I highlight a character, all of the characters on the page become visible. Earlier I remember that only the characters to the left of the selected text was visible, and I don't remember when the behavior changed. So I decide to go back and track this down, at first using the attachments I've already created.*

4,096–All text is visible when a few characters are selected. It looks like this:



5,000–Text is only visible to the left of the selection. I see this:



4,500–Only visible to the left.

4,200–Only visible to the left.

4,100–All text is visible when I make a small selection. I need to upload additional files from here on because I'm converging on a different range than before.

4,150–All text is visible when I make a small selection.

4,175–Only visible to the left.

4,160–All text is visible when I make a small selection.

4,170–Only visible to the left.

4,165–Only visible to the left.

4,168–Only visible to the left.

4,169–Only visible to the left. I'm back to 4,170, which I already tested. Wait a minute! 4,170 gets the same result as 4,169. I've somehow gone in the wrong direction. I got sloppy by not keeping careful records. The range is [4,160-4,165].

4,162–Only visible to the left. [4,160-4,162]

4,161–Only visible to the left. [4,160-4,161] It looks like we have it! Let's make sure.

4,160–Only visible to the left. No fair! That's not the same result I got last time. Sometimes a target will move while I'm testing, and sometimes my record keeping is just wrong. So I'll back up and see if anything else is different now.

4,150–All text is visible when I make a small selection. Same as before.

4,155–Only visible to the left. Looks like the range is [4,150-4,155] now.

4,152–Only visible to the left. [4,150-4,152]

4,151–Only visible to the left. [4,150-4,151] Do we have it now? I don't get my hopes up.

4,150–All text is visible when I make a small selection. All looks okay, but I'm still suspicious.

4,151–All text is visible when I make a small selection – a different result than before! I close several extraneous browser windows to see if this might be a memory-related problem.

4,151–Aha! I discover that the result depends on how I select. If I drag the selection right far enough, all text becomes visible. I was being sloppy with how I was selecting text. I should have done it exactly the same way every time.

5,000–All text is visible if I select 904 characters or more starting on the left. A quick calculation shows that this leaves exactly 4096 characters to the right, which is right at a power of 2.

4,095–I just need to select one character to get all the text to show up.

4,096–The text to the right is displayed if I select the two characters at left end, or one character elsewhere.

4,100–I can make all the text visible if I select any character from sixth position from the left onward. I think I see the pattern.

At this point I copy the data file to the web server and view it as a static file instead of going through Mantis. That reproduces the bug, so I know it has nothing to do with Mantis. It looks like the bug is in my browser, Netscape. I don't hold out much hope of getting a commercial vendor to care about a bug report like this, so I install the latest version of Mozilla and find that it has the same problem. I check the bug database for Mozilla and find several similar, though not identical, issues already filed. I decide that bug 92193, "Very long line is not fully rendered," is close enough and I will add a comment there (http://bugzilla.mozilla.org/show_bug.cgi?id=92193).

Getting ready to report my findings almost always inspires me to conjure up a few more experiments. Based on comments about Linux in the bugs I looked at, I tried accessing the test file from a Windows-based web server, proving that the problem isn't specific to a Linux web server. Comments for this bug report point out an interesting environmental variation that I didn't try before–the thresholds move if I change the font size—which I confirm with my test data but don't try to characterize further. I also think to try double-clicking the invisible text, which gives interesting results. And one more thing, in an attempt to see if this is an underlying operating system problem, I try to reproduce the bug with Internet Explorer, and indeed, it also has trouble with disappearing text with very long lines. But the symptoms aren't quite the same, and the line has to be much longer than with Netscape and Mozilla. I decide to make a note of this but not explore further.

I see that no one else has isolated the bug to the extent that I have, so I file this comment on Mozilla bug 92193:

```
I just isolated a bug in Mozilla 1.7 (originally seen in Netscape 7.1)
that seems to be this same issue.  I'm running on Windows XP.  I'm
viewing text files hosted on an Apache server - I tried Apache on both
```

XP and Linux with the same results.

The text files contain a single long line of "x" characters. If they contain 4095 or more characters, the page comes up blank. (I'll attach this one as a test case.) Then if I select any character by blindly clicking where the text should be, they all become visible. When I deselect, all goes blank again.

For a line 4096 characters long, if I select any character from the second one onward, the text becomes visible. If I select the first character, only that one is visible. With 4100 characters, when I select any character from the sixth onward, all becomes visible. But if I select any of characters 1-5, only the selected characters and those to the left of them are visible. This pattern continues for long lines.

If I double-click over the invisible text, it's all highlighted as a solid gray rectangle with the text still not visible.

The thresholds change as I zoom in and out with "Ctrl +" and "Ctrl -".

I've noticed a vaguely similar problem with IE 6 rendering long lines in text files, but the problem occurs at a much higher threshold.

You can see from the example that there may be many opportunities to refine your bug report. I found the precise breaking point for the software, the smallest (and therefore most realistic) data file that illustrated the problem. Then I found a secondary effect and characterized it through a combination of mechanical application of the bisection technique, intuition, and trial and error.

One thing to watch out for is when there is more than one failure mode at different points in a continuum. Especially when one is doing robustness testing, it's not uncommon to find a moderate failure at one point, and a more serious failure with more extreme test data. For more on finding the most serious consequence of a bug, see the "Bug Advocacy" presentation[5], and “The Bug Reporting Processes” paper[6].

4. Where to draw the line?

There is no clear dividing line between the bug isolation and reporting that the tester does and the debugging that the programmer does. In most organizations, there's plenty of room for the testers to put more effort into bug isolation. On the other hand, there may also be reasons to have the testers report the first symptom they see and then move on. Here are some factors in favor of having testers put significant effort into isolating the bugs they report:

- Testers may be more motivated than the programmers to show that the bug has a real impact and should be fixed.
- Because they spend more time on bug reporting than programmers, the testers may be better as bug isolation than the programmers.
- The information that the tester learns about the bug can be put into immediate use for further testing.
- Testers may have access to more resources, such as staff and lab equipment, than the programmers.

However, there are also factors that could indicate that it's best to put most of the weight of bug isolation on the programmers:

- The testers may not be highly skilled or experienced, so they may not be adept at bug isolation.
- Testers might put significant effort into isolating a bug that turns out to be a low priority for fixing.
- Testing resources may be more constrained than development resources.
- It might be more productive to use knowledge of the code to zoom in on the problem.
- Having the testers do significant bug isolation makes their somewhat unpredictable project schedule even more fluid, because it's hard to predict how many bugs will be found and how hard they are to characterize. Of course,

the programmers have the same problem if we shift the work to them instead.

> *Looking back at the last example, you can see a conscientious effort to isolate the bug that went beyond reporting the first observed symptom. In this case, it looks like a good choice. The bug report that was amended had been open for almost three years, and the information already in the report did not characterize the problem well. The new information may lead to some long-awaited progress on this bug.*
>
> *There were still some possibilities for exploration that were stopped short. There are always judgment calls like this, because the opportunities for further exploration are as endless as the number of possible test cases. In this case, there was a correlation with the font size that I acknowledged, but I didn't characterize its effects on the thresholds. This might have led to a theory about the total width of the rendered page being a factor, but I chose to hand off that line of reasoning to the programmer. Also, the fact that Internet Explorer has a similar bug was surprising, and could possibly indicate that a flaw in the operating system is at fault. I decided that the programmer could better judge this because she would know more about the operating system interfaces that were in use. I didn't check other platforms, mostly because the bug was not severe and I felt my time was better spent elsewhere.*

In most of the organizations I've observed, I believe the testers should have been doing more bug isolation, especially for severe bugs. I recommend that you put conscious thought into the factors listed above and help your organization decide on guidelines for where the draw the line.

For more about the bug reporting process, see my articles “Bug Report: But It's a Feature!”[7] and “A Bug Tracking Story.” [8]

5. References

1. Peter, Dr. Laurence J. *Peter's Quotations: Ideas for Our Time*. 1977.

2. Faught, Danny R. and Prathibha Tammana. “Software Defect Isolation.” High-Performance Computing Users Group, March 1998, and InterWorks, April 1998. http://tejasconsulting.com/papers/iworks98/defect_isol.pdf

3. The bug report, titled “Locked out of View Bugs page after large query” can be found at <http://bugs.mantisbt.org/bug_view_page.php?bug_id=0003798>.

4. Faught, Danny R. “A Lesson in Scripting.” *STQE*, March/April 2002. <http://tejasconsulting.com/stqe/a_lesson_in_scripting.pdf>

5. Kaner, Cem. "Bug Advocacy." July, 2000. Slides 17-25. <http://kaner.com/pdfs/bugadvoc.pdf>

6. Black, Rex. “The Bug Reporting Processes.” *Journal of Software Testing Professionals*, 2000. <http://rexblackconsulting.com/publications/Bug%20Reporting%20Processes%20(Article).pdf>

7. Faught, Danny R. “Bug Report: But It's a Feature!” *STQE,* March/April 2003. <http://tejasconsulting.com/stqe/itsafeature.pdf>

8. Faught, Danny R. “A Bug Tracking Story.” ASEE Software Engineering Process Improvement Workshop, February 2002. <http://tejasconsulting.com/papers/bug_tracking_story.pdf> (slides)

Agile Testing

Bret Pettichord

Pettichord Consulting

Many software teams are trying to increase their development agility by developing in smaller iterations and being more flexible about requirements. Agility has gained favor because it allows development teams to change direction based on changing market requirements or emerging software behavior. This presentation discusses three areas of improvement software testing organizations can pursue to become more agile:

- Test automation. We'll review the value and limits of the unit tests developed by agile developers and explore how agile teams are automating their system tests using standard programming languages (rather than vendor-specific scripting languages) and avoiding GUI automation. In general, these methods blur the boundaries between testers and programmers.

- Understanding the business. Testers have long been advocates for the user perspective; an agile software development environment depends on this perspective being articulated even earlier in the process through use cases, user stories, or example scenarios. Rather than depending on others to tell them what the requirements are, testers will need to help define requirements, articulate implied expectations and highlight areas where stakeholders may have conflicting interests. In general, this means that testers will need to be able to function as business analysts; their success will often depend on the depth of their understanding of the underlying business.

- Understanding failure patterns. Software fails in different ways, often as a consequence of the underlying language or architecture. Yet many of classic testing methods make no concessions for such contingencies (e.g., testing for buffer overflows makes little sense with Java code). Testers learn how software can fail while they are testing it. Agility requires this understanding be effectively

communicated and shared with the team; testers can then suggest suitable tests to other testers and prompt programmers to restructure the software in ways that make similar errors less likely. We'll review failure catalogs as one method for communicating failure modes.

In addition to new and changed practices, agile testing may also require teams to discard practices that they have grown accustomed to and comfortable with, such as having an independent test team, collecting comprehensive defect metrics, or maintaining a strict separation between test and production code.

Bret Pettichord is a leading writer and consultant, specializing in agile testing, homebrew test automation, context-driven testing, and software testability. He co-authored Lessons Learned in Software Testing, a Jolt-award finalist, and is host and founder of the annual Austin Workshop on Test Automation. He is a consultant with ThoughtWorks, an international firm specializing in applying agile methods to enterprise software, and a consulting researcher to the Center for Software Testing Education at Florida Institute of Technology.

Getting Started With Automated Testing

Michael Kelly
Mike@MichaelDKelly.com

Bio: I am a software testing consultant for Computer Horizons Corporation with experience in software development and automated testing. I have published numerous articles on topics in test automation and testing in general and have been a technical editor for IBM Rational Software for articles on various testing topics. I am a former adjunct professor in computer science at Indiana Institute of Technology where I taught a course in software testing.

Abstract: This paper outlines the key steps to take as you get started with automated testing. These ideas should be especially useful to you if you are doing automated testing for the first time. This paper is not geared to a specific test tool or tool vendor, and will offer sound advice if you have already selected your tool, if you are looking to purchase a tool, or if you are building your own tools. This paper does not address the day-to-day details of test automation but focuses on developing a strategy for automation, putting together an automation team, and some simple first steps.

Welcome to Automated Testing!
This paper outlines the key steps to take as you get started with automated testing and should be especially useful to you if you're doing automated testing for the first time ever. This paper is not geared to a specific test tool or tool vendor, and offers sound advice if you have already selected your tool, if you are looking to purchase a tool, or if you are building your own tools. This paper does not get into the day-to-day details of test automation but instead, consists of a series of simple steps that help you look at automation at a high level, focusing on developing a strategy for automation and putting together an automation team.

Throughout the paper, I refer to building the right automation team. Just about every team will benefit from *Set Goals for Automation* and *Set Some Standards*. Depending on your context, some of the may be more valid or necessary than others. For example, *Document Everything* may be the most important step you take if you work in an FDA regulated environment, but documenting everything may not be as practical if you are working in an XP environment. There is no particular order in which the steps need to be executed.

This article focuses on team makeup and coordination. Some of the steps will only be applicable for teams of more than one person. Some of the examples I share in the paper are from when I was working as the only automated tester for a project while others are from projects where there were up to eight full-time automated testers (not to mention the other test engineers). The steps discussed in this paper apply primarily to teams of three or more people. If your automation effort has only one or two full-time automated testers there is still good information available to you here, though not all of it will apply.

Terminology
Automated testing is the use of tools assist with a testing effort. There are many types of test automation, including automated unit testing, functional regression testing, and performance testing. Test automation can be as simple as automating the creation of some of your data, or as complex as a series of scripts/programs that do parallel comparative testing of two or more systems with live data. A small sampling of different types of automation available is listed below:

- Load and performance testing
- Installation and configuration testing
- Testing for race conditions
- Endurance testing
- Helping the development effort with smoke tests and unit tests
- Analyzing code coverage and runtime analysis
- Automation of test input generation
- Checking for coding standards and compliance
- Regression testing
- many more…

The main focus of this paper will be on scripted automated testing (be it performance testing, regression testing, unit testing, or some other such type of scripted testing). Some of these steps will be applicable to all of the types of automation listed in the bullet points above (for example *Set Goals for Automation*) while others (such as *Look for Ways to Modularize Your Scripts*) will only apply to scripted development.

Automated testing is a subset of the overall testing profession. There is an overlap of the skill set for those who make good testers and those who make good automated testers. In a couple of places I will talk about skills needed specifically for automated testing, but one should keep in mind that an automated tester should be a tester first. That is, they should be skilled in the arts of black box testing, white box testing, domain testing, performance testing, exploratory testing, and any of the hundreds of other types of testing.

Working with these two suppositions (there are many ways to perform automated testing and automated testers should be testers first) we can now jump into some of the key steps that will make your automated

test team successful. First, we will look at setting some goals for test automation. Setting goals can add focus and a sense of mission to the automation effort. It can also ease the pain of determining your return on investment, as unlike functional regression testing, many of the types of automated testing have no easily quantifiable return.

The first step when getting started with automation is to ensure that you are setting the correct scope for your effort and that your efforts are focused on the right goals. In this next section, we will look at goals for automation and how they will affect the rest of your decisions.

Set Goals for Automation

Again, there are many different types of test automation. One of the problems most teams encounter is that they only focus on a few of the more popular types of automated testing (functional regression testing, unit testing, performance testing, etc…). Setting goals for automation can affect what you automate:

- Do you want to find bugs?
- Do you want to establish traceability for some sort of compliance?
- Do you want to support the development team?
- Do you want to establish scripts that allow you to ensure nothing changes in the software?

These questions are a small sampling of all of the questions that need to be asked in order to determine the scope of your automated testing. Related to your goals, your software development lifecycle and the skills of your testing staff will also affect what you choose to automate. Are you developing in an XP lifecycle with programmers also testing? This would probably lead to more automation[1]. If all of your testers are junior, you probably will use very little automated testing or automated tests that are shallow.

Use your goals to set your strategy and scope for your automation. If your goal is to support your developers, select tools and training to support that goal. You will need testers who know how to program and who can communicate with developers in their language. They will need to look at source code, configure environments, and will probably spend a good deal of time working side-by-side with developers debugging and troubleshooting. In contrast, if your main goal is to support refactoring and change, you will have a team with experience developing regression scripts and need to have team members with both strong business knowledge and strong testing skills. They will need to know how to create scripts that look for changes and will need to know the business in order create scripts that look for underlying changes that may be taking place.

A common mistake when automating for the first time is biting off more than you can chew and consequently missing deadlines or having to work unrealistic hours in order to meet them. Either of these situations will demotivate your testing team and make them look bad in the eyes of the rest of the development team. By setting your goals, you can start small and keep things simple for your first-time automating. In future iterations or projects, you may want to go crazy and automate everything, but by starting small now, you will minimize possible rework later when you start adding new tools or new automation techniques.

When deciding what to automate for your first time, start with small milestones. For example:

- If you're testing a GUI or Web application, start with testing simple functionality. This could include verifying that all the correct controls exist on the screen, the proper fields enable/disable when actions are taken, etc….

[1] In past projects I've worked on, when the test team worked closely with the development team, we had much more automation and the resulting scripts and/or environments were more stable and more robust. On projects where we sat in different buildings or a significant distance apart from them, the test code was less effective and we had more intermittent test bugs due to code workarounds.

- If you're automating performance testing, start with just one virtual user and set your goal at a low number (no more than twenty). When you get one virtual user to work, double this and get two to work. Keep doubling this until you get to twenty. Each increment could present a new set of challenges.

Whatever you choose to test, make sure that it doesn't span more than one part of the application-under-test or more than one or two Web pages. Ideally, you should be able to use record-and-playback features to perform this basic testing or you can do some basic scripting (hand-coding scripting commands into the script files instead of using record-and-playback). These basic scripts will then become the baseline moving forward. Many testing tools will have advanced features such as the ability to add delays and timers, the ability to distribute testing on different machines, and the ability to create graphs for just about everything. Stay away from these as much as possible at first, because they'll just confuse you. Only after your team has had some small successes should you explore these useful and often necessary features.

Once you have determined your goals, you will need to start thinking about what kinds of skills you will need to have on your team. The next step addresses one of the most common oversights when staffing an automated testing team. In your context, depending on your goals, you may need more development-oriented automated testers, or you may need some other special skill.

Have at Least One Programmer on Your Team

The most important thing you can do is select the right team. I recently saw a company do a comprehensive analysis of three market-leading enterprise level test tools. They compared them side-by-side in different scenarios, using their most skilled staff to run the tools through their paces. A couple of months later the team came back and said, “this tool is the tool for our company.” They gave little to no thought about who would be using the tool, their technical experience, where they could get training and how much it would cost, if they had past experience with one of the other tools, etc… This is the biggest mistake I think you can make. Your human resources are your biggest investment and they will be the determining factor for your automation success. A tool is and always will be, only a tool.

To ensure efficient well-planned automated testing, you will need to have at least one experienced programmer in your testing-automation group. You'll soon find out that automated testing *is* code development. If you will be doing a lot of scripting in your automation effort, most likely you will want to build modularity into your scripts. You will want to create some custom functions, extend the test tools, and potentially read test data from databases or data files. Depending on the tool and language, encapsulation, object orientation, or some other method may be used to make the code more maintainable or easier to use. If you are looking at providing code analysis services (performance profiling, code coverage, runtime analysis), you will need someone who knows how to read the code-under-test and who can ask the development team the right questions.

Even if the scripting environment you are using is not Java or C++ (which it may very well be), you're still building systems of scripts, data files, and libraries. Record-and-playback features only offer quick solutions for the most common tasks and controls. For advanced automation of any kind or for any custom controls, you'll need to be able to write your own *maintainable* code. That means employing programmers, not manual testers who learn to code as they go. Having said that, beware you don’t employ only programmers. All team members should be testers first. Skilled in the mental arts of testing as well as armed with the ability to code and design test systems.

That said, once a decision has been made as to what tool will be used, it is important that the users of that tool know how to use it properly. The next step offers some places to look if you are still choosing your tool, and offers some advice on ensuring the automation team knows how to use them.

Get Familiar with the Tools

Depending on what tools you already have in your arsenal and what types of automated testing you will be doing, this section may or may not apply to you. Most automated testing tools are designed for unit testing, some sort of regression testing, or performance testing. Addressing what tools are available and what tools may be best for you are outside of the scope of this paper. However, if you are still reviewing

tools and would like some help selecting the right tools for your specific context, I would recommend using the following resources (which I am in no way affiliated with):

Open Testware Reviews: http://tejasconsulting.com/open-testware/
- A source of information about freely available test tools.

Testing Tool Information: http://www.grove.co.uk/Tool_Information/Choosing_Tools.html
- Short Sharp Advice about how to choose a testing tool.

Test Tool Evaluation Center: http://www.testtoolevaluation.com/index.asp
- An online resource with a lot of info and wizards for tool comparisons.

The scope of your automation effort will depend heavily on the tools you use and have access to. If you already have an investment in an enterprise test tool, you will probably want to leverage it. If you are operating on a tight budget, you will look more at open source tools, shareware, and custom tool development and scripting (virtually *any* scripting language can be it's own full-blown automation tool). Below is a sampling of some of the different names commonly used to label test automation tools:

- Disk imaging tools
- File scanners
- Macro tools
- Memory monitors
- Environmental debuggers
- Requirements verifiers
- Test procedure generators
- Syntax checkers/debuggers
- Runtime error catchers
- Source code testing tools
- Environment testing tools
- Static and dynamic analyzers
- Unit test tools
- Code coverage tools
- Test data generators
- File comparison utilities
- Simulation tools
- Load/Performance testing tools
- Network testing tools
- Test management tools
- GUI testing tools

Regardless of what tool(s) you select, you will need to be sure that you spend time learning how to use them properly. Go through the tutorials that are provided and read through whatever documentation is available. While the tutorials aren't the definitive guides as far as training is concerned, they do get you familiar with the software as well as any vendor specific terminology. Both of these are important. As you'll soon find out most tools are large and complex with feature upon feature. The tutorials will familiarize you with the features you'll be using most.

If you feel you still need more familiarity with the tools after you've completed the tutorials, attend a training class (if they are available), or hire a training consultant to come in and spend some time with you. Having a basic understanding of the tools is essential. Make sure your whole team has had some form of training or some reasonable amount of time playing with the tools before you try to do any real work. (I find that programmers tend to pick up test tools very quickly, while nonprogrammers struggle with some of the programming concepts and need more time.)

Once you know which tools you will be using and have a good idea of how you will be using them, you should start to think about setting some guidelines for how your team will use them together. The next step looks at setting up standards for tool use. These standards will make training easier, make it easier to move team members from project to project, and reduce the cost of ownership for the tools you choose.

Set Some Standards

You'll also find it useful to develop standards for your automated-testing team. This is just as important in testing as it is in conventional software development. Your test system will develop more rapidly and will be easier to maintain if you establish and enforce naming standards, coding standards, environmental standards, and procedures for error and defect tracking. Having these standards documented will also allow people new to the project team to come up to speed faster.

- **Naming standards** for scripts, test logs, directory structures, data structures, and verification points help to keep everyone on the same page. On two of my last three projects, we maintained more than 1,000 scripts and 5,000 data structures for each project. A good naming standard was the only thing that made that possible.

- **Coding standards** should also be developed and enforced. For most companies this is easy — you can just steal the standards used by the development staff. If you don't have this advantage, go online and find some. They're out there, and you can find a reasonable set of standards in about 15 minutes. Once you've used them for a while, you can customize them to fit your needs and the needs of your team.

- **Environmental standards** should ensure that the computers you use all have the same operating system, RAM, hard drive space, and installed software configurations. The only differences should be the specific differences you are looking for in your configuration testing (if applicable). I've found that many of my hard-to-find and expensive-to-fix bugs, for instance, have been due to the fact that a script was developed on a computer that had more resources than the one on which it was executed.

- **Procedures for error and defect tracking** should describe how to log errors in test scripts, submit defects via your defect-tracking tool, code workarounds into scripts, and remove it all after a bug is resolved[2].

Document your team's standards, and be sure your team knows the standards and follows them. This step will also prepare you for the remaining steps as it will potentially provide a framework for decision-making and review. Next is a step focused more on script-based automation, but the principle can still be applied to other types of automation.

Establish Some Baselines

Once you've decided what you're testing, you should establish some simple baselines. As mentioned above this can be done using record-and-playback features, by performing some simple scripting, by gathering small sets of runtime data, etc.… Whatever type of testing you're implementing, figure out what the bare minimum tests are for that type of testing to be successful and implement them in very small, very simple chunks. Remember that this is your first project and most likely your first time using these tools. You will not have sophisticated frameworks and test architectures in place yet. Those will come later as your team matures. You will use these baselines as the foundation for what you will be implementing going forward. Hopefully, you will have some sort of archiving or source control and can always come back to these simple tests if you need to.

In past projects I have established the following baselines:

- For regression testing: a baseline set of scripts that test basic functionality in the system using the most commonly used paths through the application.

- For performance testing: a baseline set of scripts that target each standalone service for the application that run successfully with one virtual user.

- For code coverage analysis: try to develop a series of tests that will allow you to get some level of coverage in each major module of your application.

- For source code checking tools: try to get a small subset of the application code to pass the verifications and modify/refine the rules as needed.

---

[2] I didn't figure this out until a few projects ago. My team had many problems communicating when it came to finding and reporting bugs. One of us would log a bug and develop a workaround in the script without communicating to the rest of the team what we had done. Inevitably, someone else would test the bug when it was fixed, mark it resolved, and never remove the workaround in the code. Usually this lack of communication caused confusion and rework, and cost the team time.

- For establishing traceability: take a small series of test cases and, using your test management tool, establish traceability from the manual/automated scripts, to the test cases, to the requirements. Generate a simple report that shows this traceability in a usable format.

- For test data generation: generate a subset of the overall data you will need for testing and verify the data's correctness and randomness.

Once you have established your baselines you will be better prepared to start looking at more advanced methods of scripting (or data gathering, or scenarios, etc…). This is important as the next two steps focus on optimization of these baselines. As you mature in your processes and experience, you will be able to cut out this step, but be sure you are taking small steps right now.

Look for Ways to Modularize Your Scripts
Now that you have your baseline scripts, grab a good software architect - or your team of testers and a large whiteboard. Start looking through the code in the scripts for repetitive calls or other common actions. What you're doing is looking for ways you can modularize your scripts.

Ideally, you want to optimize your scripts so that maintenance is as easy as possible. I've found that I've *never* regretted spending too much time developing a powerful and robust script, and I've often kicked myself for taking shortcuts in development. A script *will* cost you more to maintain than it will to create, unless you develop the script just as you would the software it's testing. Do it right the first time and reap the rewards in all of the following iterations of the project. After you've planned out what modularization you can do, implement it using whatever method your tools allow for the most reuse (libraries, objects, classes, etc…). More than likely, you'll carry these over to following projects, and they'll evolve and change as you do.

For ideas on how to modularize your scripts, look at user communities for the tools you use, read literature on the topic from any of the major testing websites, talk with your development team, or hire in a consultant to train your staff on what to look for. These resources are also good places to find information about the next step as well.

Use Data Structures and Data-Drive Techniques
Effective and cost-efficient automated testing is data-driven. Keith Zambelich's whitepaper *Totally Data-Driven Automated Testing* (available at www.sqa-test.com) is a must-read for anyone doing automated testing. Data-driven testing simply means that your test cases and test scripts are built around the data that will be entered into the application-under-test at runtime. That data is stored by some method and can be accessed by some key used in your scripts.

This method offers the greatest flexibility when it comes to developing workarounds for bugs and performing maintenance, and it allows for the fastest development of large sets of test cases. Most likely, your tool will have some method for implementing this, or you can use a spreadsheet or a database. As a special bonus, once you get good at data-driven techniques, you can use automated methods to generate some of that test data for you.

Document Everything (that makes sense to document…)
Finally, document why you designed things the way you did. Document what each module does, and what each function in it does. All of this documentation is useful as training material or for future reference, and it helps you keep track of lessons learned. I document as much as time allows. Sometimes I get it all, and sometimes I can't document anything. I've never regretted documenting any information, but on more than a few occasions I *have* regretted not having any clue about how something worked or why I made a certain decision on a project I'd been away from for a couple of months. Sometimes documentation is the only thing that can save project scripts that no one has worked on in a while.

A Review
Even if you follow all the steps above, you'll still struggle the first time you attempt automated testing. Just remember to follow this road map and you should survive:

- Set goals for automation.
- Have at least one experienced programmer in your testing-automation group.
- Get familiar with your tools.
- Develop standards for your team.
- Establish some baselines.
- Modularize and build reusability and maintainability into your scripts.
- Use data-driven testing techniques whenever possible.
- Document what makes sense to document.

The clarity and simplicity of your goals, your understanding of the tools, and the makeup of your team. These are the factors that will determine your success.

As your knowledge of these grows, your goals and priorities for testing will develop. The way you automate your testing will also change as you include more tools, get more experience with them, and read about more complex and innovative ways of using them. Primarily, the quality and success of your testing will grow as the skill level and experience of your team changes. Tools will do nothing if people can't use them or don't know how to test to begin with. It has been my experience that the makeup of the team is the most influential factor in an automation efforts success.

Your goals should be driven by your business knowledge. Your programming knowledge and testing skill reflect your team and how they use their tools. Test automation requires a balance between programming knowledge, testing skill, and business knowledge. Incorporating all of those assets into your team will be the most important step of all.

Test Improvement under Fire

Abstract

Development was starting to lose trust in our test process. Large bugs escaped into production causing major outages. The question "why didn't Test catch this?" rang through our halls. No matter the effort, we were unable to reproduce the problems with our tools and lab. We were fixing, releasing, fixing, and releasing. Morale was low and the team was tired. Something had to change!

This paper discusses one team's journey down the test improvement road. We'll identify the pieces of the Six Sigma, TPI (Test Process Improvement), and structured testing techniques we used. We'll explore what solutions worked, what didn't, and lessons we learned.

Test innovation isn't just about improving the test process. It's a fine balance of managing existing test responsibilities and change. When improvements are weighed against business objects, sometimes the immediate demands of the product outweigh test innovation efforts. Choosing the right areas to invest time and effort is critical to success.

About the Author

Nasa Koski has nine years of experience in the software industry. She has been with Microsoft for five years and is currently a Test Lead in MSN. She currently manages a team that provides system engineering and test though high-scale test environments.

Nasa Koski
Microsoft Corporation
nasak@microsoft.com

Copyright Nasa Koski, 2004

Introduction

Let's face it. Resources and budgets are tightening. The demand for better, quicker software is increasing. Many project teams run to the finish line while also starting the next race. How does Test keep up, let alone innovate?

In many cases, the answer is: improvement in small doses over time. Teams incrementally improve test automation, test methods, and processes so they can reach toward the next level while still shipping on time.

At times though, this is not enough. The business and software complexity can develop more quickly than product groups have time to grow quality practices and processes. In turn, teams who only incrementally mature will eventually find themselves further behind.

So why not leap forward?

The problem is getting the time and resources. For many teams, the people innovating on test practices are the same people testing the products. Management must make trade offs by weighing current business objectives and the means that will achieve them.

Extending schedules to allow for test innovation can be a difficult argument to make if current processes are successful and product innovation is the priority. Since change takes time, good ideas often die or get cut when the demands of the release are high. It's easier for teams to use skills and knowledge that worked in the past even if they are not as effective in the present.

At some point though, you must leap forward or the success will dissipate.

Growing Pains

Several years back I joined a start-up team working to develop a high-scale internet service. We were the typical start-up team: small, agile, and "scrappy." In a test team of four, everyone did a little of everything. We wrote test plans in Word, test cases in Excel, and worked long hours building custom tools.

Our test strategy was fairly basic with black box and exploratory testing. With pressure of getting to market, we concentrated on more complex/risky areas and spent time on hard problems like scaling up and performance.

Over a period of a couple years, we continued to grow our team and practices. We enjoyed the success of our product and worked hard to keep it that way. The service that had once started out with an audience of a few thousand had grown to millions. With it came new challenges around scale, performance, and quality.

We continued to ship and release without many problems, but Test was starting to feel some growing pains.

- We invested our initial resources into stress and performance in anticipation of projected traffic. As a trade off, we neglected automation on the functional side. Over time, the weight of manual test cases slowed our team's ability to maintain a tight schedule.
- Test case management in Excel was unmanageable with a large number of test cases across multiple areas. The excel sheets were not well equipped to provide test case steps and a clear result.
- Test cases were often vague. The expected results were not always explicit, which made automation impossible.
- The number of tools we had grew over time. Unfortunately, they had no common platform. New changes in the product forced re-work across multiple toolsets.
- The team started experiencing natural attrition. However, since early test efforts lacked a structured test method the team's knowledge started leaving with the testers. New testers took a month or more to be effective in the team since most learning was tribal knowledge.
- The number of escaped bugs was increasing with every release. Instead of celebrating our accomplishments, we needed to answer hard questions about why issues were missed.

It became painfully clear that change was needed. Considering our team had never undergone a significant improvement effort, we needed to quickly learn how.

Where to Start?

The first step to any problem is first admitting you have one. No team can bring about change unless they first acknowledge that one is needed. This doesn't only apply to the immediate group, but also to management. If key members don't believe improvement is necessary, it will become difficult overcoming future resistance or obtaining the resources needed to successfully bring about change.

Fortunately, most improvement efforts stem from problematic processes. When our team experienced the problems, we leveraged these pain points to reinforce why a change was necessary. By calling out these points as they occurred, it became easier to combat resistance or disbelief that a problem existed. In this way, we were able to gain agreement that a change was necessary. The next problem was: how to go about it.

Get Organized

When my team had finally acknowledged a change was needed, we had to figure out where to begin. We had never undertaken such a large effort before. We needed to get organized. We asked ourselves, what do we need to get us where we want to go?

- **Acknowledge the past** – We didn't get to the present by happenstance. Many things we had done enabled us to be as successful as we were. We needed to recognize our past successes in order to move forward in the future.
- **Create a vision** – while we all acknowledged change was needed, we all had different ideas about what that meant and what it would accomplish. We soon learned that without a solid vision, you will waste time stumbling in a gray cloud.

- **Determine resources & key stakeholders** – All change requires time and effort. In order to realize your vision, you must identify the cost/benefit of the changes required and get commitment/buy-in from key stakeholders.
- **Get smart** – To renew our perspective, we needed a more diverse look at how we could approach testing. We also realized we probably weren't alone. There were probably existing tools and processes that could be reused and we could learn from.
- **Make an initial plan** – Finally, start wrapping your footwork into an initial plan. Be accountable to it by creating a schedule and getting sponsorship.

Acknowledge the Past

Life isn't a picnic when you realize you're going nowhere fast. For us, it was one of our team's lowest points. Naturally, we didn't want to focus on this. We simply wanted to get out of the rut by acting fast. We headed straight for the vision. We realized later, acknowledging the past is an important step. We had to appreciate it fully in order to gain the wisdom experience was offering. Our team later iterated back to complete this important step.

With low morale, it's easy to draw out the inefficiencies in the team. Yet it was important to note that those now ineffective practices were critical and necessary to our previous success. The failure wasn't necessarily the practices themselves, but that we held on to them too long.

Opening a discussion regarding the situation also provided a platform to discuss our frustrations. It also helped us to come to terms with how we got into this situation. This helped us properly segue into creating a new vision for ourselves.

Define Your Vision

The vision should paint the picture of where the team is heading. People should hear the vision and understand what the end goal is. The vision becomes the soundboard of all tradeoffs and decisions.

Developing a vision statement is like asking: "What do we want to be when we grow up?" In a team meeting I asked this very question to kick off a brainstorm. Testers said everything from work-life balance to training and growth. We threw these phrases on the white board and a picture started emerging. We then started brainstorming words we felt captured this picture such as flexible, innovative, efficiency, quality, etc. We used these words to build draft statements.

In our experience, creating the vision was difficult. While we were all generally on the same page about what we wanted to achieve, capturing it in one sentence was difficult. For us, it wasn't important that the vision statement was perfect. However, it was necessary that everyone understood it, believed it, and could explain it.

Identify Key Stakeholders

Before embarking upon a major change effort, it's important to identify who has a vested interest. You should also assess what level of support you need from them for the change. Without their input or buy-in you may brush up against resistance that can slow down progress or derail the project.

To identify key stakeholders in our organization, we asked ourselves:

- Who will be impacted by the changes? Who will gain or lose from the change?
- Who must we connect with across groups to assure their support?
- What is the "critical mass" needed for the change?

To identify what we needed from the stakeholders we asked ourselves:

- What involvement will they have in the change?
- What must we do to get the level of support needed?

With this in mind, we developed a plan with five categories:

- **Make** - Those in the team that are directly impacted by the change and must live it personally
- **Help** - Those indirectly impacted and can do things that will make it easier for others to enact the change
- **Let** - Those shouldn't get in the way of the change and wait to see how it will go
- **Nothing** - Those covertly against the change and will do nothing to help
- **Against** - Those overtly against the change and will be resistant

By using two symbols (x and Δ), we were able to depict what role stakeholders currently had (x) and where we needed them (Δ). We used the plan as a roadmap to discuss, plan, and implement steps to engage stakeholders appropriately.

Stakeholders	Make	Help	Let	Nothing	Against	Comments
Test Team						
Dev Team						
PM Team						
Test mgmt						
Dev/PM Leads						
Upper mgmt						
Operations team						

Figure 1: Example of stakeholder plan

Getting Smart

Improvement efforts in the past usually took the form of post-release meetings and brainstorming around what went well and could have been better. While still valuable, obviously it wasn't getting us far enough. We needed something new, something different. We also needed success.

The thing about "getting smart" is that sometimes reality gets in the way. How many times have you wanted to take a class, read a book, or take an extra stretch project only to have it cancelled, moved, or interrupted? Our concern wasn't necessarily getting in the know; it was finding the time to do it.

Developing a "Change" Team

We approached the problem with the concept of a "change team." We needed a small group to explore new ideas around test improvement and test practices. We also needed the team to work in parallel with our existing ship cycle and help others ramp up once the current release was over.

The availability of people to the effort was of grave concern. We knew all too often, anyone assigned to test improvement part-time was at risk of getting pulled into the priorities of the product. We wanted someone fully dedicated to the new effort who would continue to drive and make progress even during the most demanding times. In addition, two team members were added part time. In total, three testers took on the additional project of exploring and learning about test improvement methods. The members were chosen for the following mixture of knowledge and skills:

- sincere interest in test improvement and the belief that change was necessary
- drive and perseverance
- social skills such as the ability to advise, negotiate, handle conflict, and positive attitude
- sense of long-term vision
- strong test expertise

The three individuals became a leadership team to help drive the improvement effort. Choosing the individuals was not difficult in our case. Our options were limited by the expertise needed in the current release, and leadership skills available in the team. The people elected were strong contributors who could work independently and accomplish the tasks at hand.

Reflect, Study, Observe

The "how" of getting smart came with observing, studying, and reflecting. With no prior experience in leading a major change, our team needed to get educated quickly. We soaked in information from four main sources:

- **Team feedback**. In an initial brainstorm meeting, our team reflected on our experiences to determine a list of areas that we could work on. We prioritized and ranked these areas similar to a feature list.
- **Company-wide practices**. Working for a large company, we had the advantage to look at different practices and tools within our company. It was not difficult to find teams who had already worked through some growing pains and matured their process.
- **Industry-wide practices**. Based on recommendations from others and general research, we perused existing resources in the industry that provided a holistic view of testing. Given our limited time, we focused on resources that appeared in wide use. Among the resources used the most, we leaned heavily on TMap (Test Management Approach) and TPI (Test Process Improvement). We were excited about TPI due to its step-by-step nature and focus on areas of test process which seemed to match our bigger areas of need. Since TMAP dove-tailed directly with TPI, aimed to use them together.

- **Training & Exposure**. Sending our team to a "Leading Change" course set us off on the right foot. This was internal training for us, but similar classes can be found at external training facilities. In addition, team members attended external events such as conferences or local test forums.

Armed with the information, the Change Team discussed which options were applicable to our situation. We also discussed what we could realistically accomplish on our timeline.

By leveraging and learning existing efforts, our team was able to profit from others' footwork and avoid common pitfalls. Educating ourselves on best practices and up-and-coming methods started our own creative juices flowing. By reflecting on our past, studying available resources, and observing others, we had enough information to form an initial plan.

Make an Initial Plan, Be Accountable

One of the things I've observed about improvement projects is the lack of schedule, deliverables and checkpoints. In order to be successful, we knew we must apply the same rigor we applied to our product to the improvement process. We started with a high-level project plan for our effort.

To build our initial plan we took a rough cut at the anticipated schedule, scope, and staffing. Since we had not yet investigated the problem spaces or proposed solution, our initial plan became an estimate of the target of work we wanted to complete. As we gained more insight, we would continually refine the plan.

Schedule

Our team started our improvement effort at the tail-end of a release. We anticipated that our window of opportunity to implement a change would occur during the next test planning phase. Since we believed we would have little success requesting more time than what the release schedule provided, our improvement schedule was largely dictated by the driving product schedule.

Within the defined product schedule for the upcoming release, we divided up the test planning and preparation phase into 4 sub-phases:

- Phase 1: Clarify improvement scope
- Phase 2: Develop test strategy
- Phase 3: Release Test Planning
- Phase 4: Release Test Execution

In the first phase, we needed to identify what problems we could solve in the space of a single release. If we could have everything, the list of improvement areas was long. We needed to narrow the scope. With limited time, it was important to hit areas that would be the most effective toward our problem space. In addition, we needed to make sure we considered easy wins.

In phase 2, we needed to focus on digging into our prioritized problems and strategizing on how to resolve them. While we had been researching test methods and practices from the beginning, it was time to mix-and-match techniques to our given problem space in master test planning. The

Change Team also needed to educate and train testers and key stakeholders on any new techniques/concepts we would be using.

Phases 3 and 4 followed the basic test planning and execution patterns. Testers wrote test plans and cases using new techniques. Change Team members mentored and supported testers through the process.

Initial Scope

Defining scope was a hard process for us as it's always difficult to cut items you feel are important. Since we had not formalized any major improvement effort in the past, defining scope was like deciding which high priority item was more important than the next.

We approached the problem by first looking at a wide scope. We wanted to get all our ideas on paper. Even if we did not address the issues this time around, the issue was tracked and prioritized. Our first cut was a team-wide brainstorm of pain points. The discussion focused on identifying problems, not solutions. The contributors consisted of the test team, as well as developers, and program management. We grouped feedback into areas of process, methodology, tools, and communication. The categories we developed were somewhat organic and derived from the themes we saw after we had aggregated all the feedback.

We took a good look at our initial brainstorm list of areas and ranked the issues, measuring against our vision. We weighted each item based on its importance, impact, and ease to implement. We looked for the high benefit, low cost items first. We provided estimates for the time it would take to implement each one. Adding this time together and comparing it with the time we had, we established a cut-line.

Staffing Plan

Without knowing exactly what were going to do, we had a general idea of staff needs for our baseline plan:

- The initial number of resources required: Three testers (one tester at 100% and two testers at 50%) would research, learn, and develop a plan for test improvement
- Testers still servicing the current release would pick up some additional areas for a limited period and we would look to obtain help from neighboring teams.
- In a phased approach, more testers would need to get involved as the previous release was finishing
- We needed to budget time for education and training

Our rough outline of a plan (number of testers, estimated time, major milestones, and its benefits) was presented to management for support.

Get Commitment & Sponsorship

When an improvement project is slighted or cancelled, was the problem really time or commitment?

The amount of time spent on improvement plays a role. Yet, commitment to see them to completion is why many ideas don't fully get off the ground. It's easy to attribute these problems

to “not enough time.” Perhaps it was only a symptom of the true problem: lack of management sponsorship.

Business leadership has a major stake in test improvement projects, which use resources that could otherwise be used directly in product cycles. It is critical to get them on board with improvement efforts. If they do not believe the improvement is necessary, it will be easy for them to redirect efforts back to the shipping product.

Once our team completed the stakeholder plan, we opened communications with management and other stakeholders regarding our improvement effort. While we only had an initial plan, it was important to get sponsorship and funding for our change effort. Without commitment of resources and buy-in we could easily fall into a scenario of resources getting pulled from our effort.

In order to sustain the commitment, we sought to understand what information/check-points were needed by management to demonstrate success and benefit. We established these milestones in our schedule.

Clarify Scope

With the initial plan in hand, it was time to get down and dirty. We needed to figure out which issues on our long list we would address in the next release. From our research in Six Sigma, Test Process Improvement (TPI) and Test Management Approach (TMAP), we tried a little of everything. While all were good exercises that helped us clarify our testing problems, TPI provided the most structure and step-by-step process for the big changes we were looking to undertake.

Root Cause Analysis

In root cause analysis, we seek to find the real cause of the problem rather than dealing with its symptoms. We considered root causes as specific underlying causes that can be reasonably identified and within our control to fix. Among the most effective tools used were:

- **Five Whys** – A very simple, yet powerful tool to drill down into root causes. Presented with a problem, the technique consists of asking “why” five times (or levels deep). Each question applies to the varying responses.

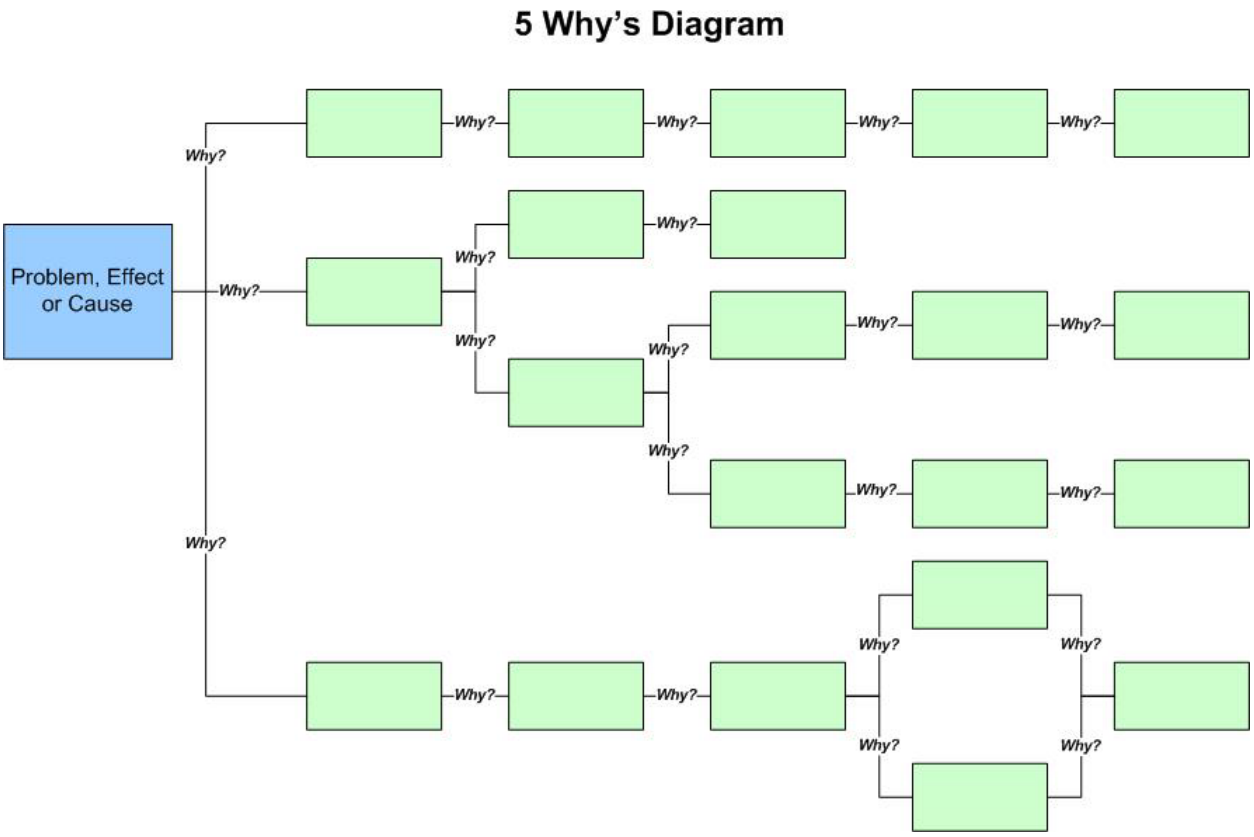


Figure 2: Example of 5-whys diagram

- **Cause/Effect** – In a series of meetings, our team brainstormed potential causes for a problem. We wrote these down and then weighted the most likely causes. The most likely causes were ranked nine, the "might be" causes ranked three, and the least likely ranked one. Coupled with the 5-Whys, this technique was effective at weeding out symptoms from root causes.

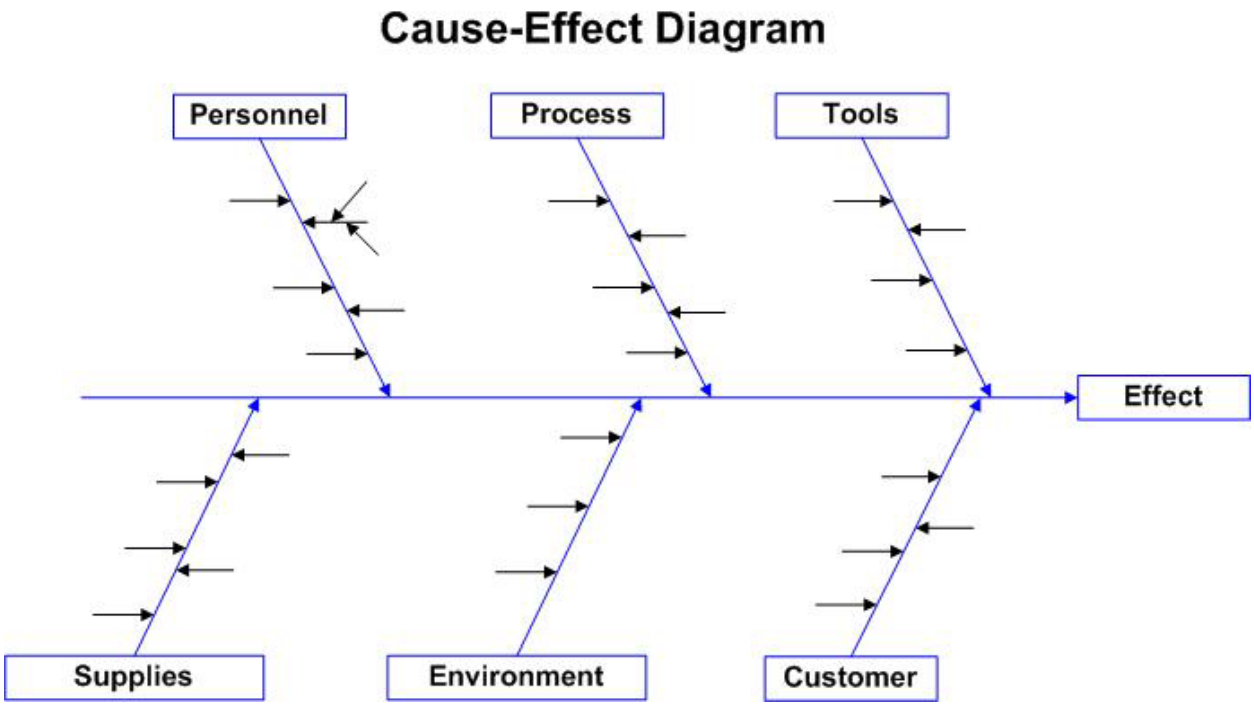


Figure 3: Example of Cause-Effect Diagram

Test Process Improvement (TPI) Model

We used the Test Process Improvement (TPI) Model published by Tim Koomen and Martin Pol to evaluate the maturity of our test processes. This model appealed to us for three main reasons:

a) it was a holistic approach to measuring our current state
b) it was already consumable for test, not requiring interpretation or modification
c) once we identified areas for improvement, a step-by-step guide was needed; TPI dove-tailed directly with such an approach: Test Management Approach (TMAP)

The TPI model contains four elements: key test areas, maturity levels, checkpoints for each maturity level, and improvement suggestions. The matrix itself is comprised of only the test areas and maturity levels. For each test area, we evaluated our team by the provided checkpoints. In order to achieve the level of maturity for an area, all checkpoints for that level must be achieved. The below figure is an example of a TPI matrix.

Area	Milestone												
	Controlled					Efficient					Optimizing		
Test strategy	1					2				3		4	
Life-cycle model	1			2									
Moment of involvement		1				2				3		4	
Estimating and planning			1							2			
Test specification techniques	1		2										
Static test techniques				1		2							
Metrics					1			2			3		4
Test tools				1			2			3			
Test environment			1				2						3
Office environment			1										
Commitment and motivation	1				2						3		
Test functions and training			1			2				3			
Scope of methodology				1						2			3
Communication		1		2							3		
Reporting	1			2		3					4		
Defect management	1				2		3						
Testware management		1			2				3				4
Test process management	1		2								3		
Evaluation						1			2				
Low-level testing				1		2		3					
		-	Current level										

Figure 4: Example of TPI assessment

In this example, the shaded areas depict levels of maturity achieved within the TPI model. The numbers represent each maturity level. Since not all areas and levels are equally important for the overall test process, the TPI model accounts for these dependencies. Some of this can be seen from the distribution of levels across areas. While TPI indicates levels as A-D, we found it was easier to describe them numerically (1-4).

The TPI model allowed us to easily identify areas of improvement for each level as well as the dependency maps between each test area. As we analyzed this data using the TPI matrix, we were able to see that lack of growth in specific areas was preventing growth across others.

We then used the maturity matrix to set a picture of what we wanted to develop. Using the framework of TPI, we reviewed each level in each area and identified what we needed to implement to meet that level. We compared this list to our brainstormed list and found many similarities between the work items.

Area					Milestone								
	Controlled					Efficient					Optimizing		
Test strategy	1					2				3		4	
Life-cycle model	1			2									
Moment of involvement		1				2				3		4	
Estimating and planning			1							2			
Test specification techniques	1		2										
Static test techniques				1		2							
Metrics					1			2			3		4
Test tools				1			2			3			
Test environment			1				2						3
Office environment			1										
Commitment and motivation	1				2						3		
Test functions and training			1			2				3			
Scope of methodology				1						2			3
Communication		1		2							3		
Reporting	1			2		3					4		
Defect management	1				2		3						
Testware management		1			2				3				4
Test process management	1		2								3		
Evaluation						1			2				
Low-level testing				1		2		3					
		-	Current level										
		-	Targeted level										

Figure 5: Example of TPI goal

Formulate a Plan

In formulating the plan, the team starts experiencing the realities of the change. We're not just talking about it anymore, we're actually doing something. No matter the strength of the vision, members of the team will always feel a sense of resistance. Change is unknown and uncertain. It's uncomfortable. It's hard to let go of old habits.

Before starting into the details of the plan, it's important to acknowledge the uncertainty and other feelings that may arise. It's the "goo" of change. Further, unforeseen problems may arise and the team will need to deal with them pragmatically. This gives permission to the team that it's natural to be unsure about what it is embarking on. It's part of the process.

Resistance

The best way to combat resistance is to acknowledge and embrace it. Someone said to me once: "Resistance is energy around your change. It's simply in the opposite direction." Energy about your idea is good. Don't squash it, redirect it.

People can be resistant to change for many reasons, but often it's simply because they don't fully understand what you're trying to achieve. When creating the test plans, it may be difficult to see the forest through the trees. Don't be afraid to revisit the original vision or list of problems you're attempting to solve. In fact, you should continually check the two to make sure you're on track.

Approach the resistance with an open mind. Determine where the disconnect is. Seek to clarify. Look at the change from another point of view and speak in terms of their interests. Sometimes individuals will see holes or flaws in the plan. Getting their input and modifying your plan can make it stronger while also winning their vote.

Structured Testing with TMAP

TMAP is a holistic approach to testing, but we did not have the time or need to use everything. We tuned areas of development to those flushed out by TPI and root cause analysis. In particular: formal test techniques, analysis of risk, and differentiation of test depth. We used TMAP's guidance in quality characteristics, test levels, and test techniques to round out these areas in master test planning and later in test specifications.

What we did use of TMAP was still quite different from our previous methods. Our main resource on the topic was the published book "Software Testing: A Guide to the TMAP® Approach." Since it was not possible for the entire team to study this material, members of the Change team condensed needed material into "test guides". The guides were more specific practices and techniques applicable in the next release. In addition, the change team held many training sessions were the concepts were presented and tried.

TMAP Terminology

Using TPI and TMAP introduced new lingo to the team - everything from the roles in the team to the varying test process and techniques. We noticed this difference right away, but didn't fully acknowledge what a barrier it would be to adoption. To our surprise, the new terms generated a lot of resistance and frustration. To team members, a new vocabulary implied a new way of doing things regardless of whether it was truly changed. The new terminology seemed to tip the scale in regard to what felt too overwhelming. Once we started "translating" and molding the concepts to fit with our existing team culture, the resistance died down.

Inspections & Reviews

TMAP's approach at inspections and spec reviews are interesting and appealed to our team. It was a more structured and measurable process than showing up to a review meeting and providing feedback. It was also a way to identify if the documentation was complete. However, implementing the idea was not easy. The process could not be implemented without the cooperation of the Program Manager (PM) team. The technique called for reports regarding the testability and quality of the documentation. It was intended to identify risks in the document as well as provide structural recommendations to improve in the future. The PM team was completely against this report and we were unsuccessful at convincing them on this point. As a compromise, we dropped the formal report but continued to use the checklists recommended by TMAP. We modified these lists to fit within our context. PMs used the checklist as an "Am I done?" document before handoff for review. For validation, the tester used the checklist to

determine all areas were addressed sufficiently. In the end, both teams were satisfied with the check-list. Since areas that Test looks for was already spelled out, PM had a head start in the review process. Also, testers spent less time iterating on base line information and were able to dig into technical details.

Structured Test Techniques

Using techniques described by TMAP was a great leg up on trying something structured to generate test cases. The step-by-step techniques provided strong exposure to testing in ways other than asking "how can I break this?" Due to the thoroughness of some of the techniques, there was some hesitation on their use. After all, it's easier to ask "how can I break this?" The downside was the training and learning curve on the areas. Due to limits of time, we only applied the techniques to high-risk and completely new areas. We continued to use older techniques and leverage existing test cases in the legacy areas. This was an effective strategy during this particular release since most of the legacy areas had not changed and were at low-risk for regression.

Test matrices using these techniques were more effective in several ways:

- High-risk areas required the use of more thorough techniques that forced testers to document decision paths they were testing. Documenting this within their test specs, Dev, Test, and PM were able to review the test logic and see bugs before the code was tested.
- Test techniques helped balance test skill across the team. Effectiveness of "how can I break this" or exploratory tests are usually dependent on the skill of the tester. Since structured techniques require specific steps, test case quality/thoroughness increased.
- Structured techniques helped to flush out deeper level ambiguity from product specifications
- Tests linked directly to test specifications helped add more visibility into the overall coverage of the feature

Structure & Process

TMAP and TPI introduced an enormous amount of structure and process to testing. After the fact, the team admitted they felt over-burdened with it. I believe some frustration was felt simply turning knowledge into skill, but more work should have been done to mold the process to work for our organization. Even though TMAP provides a step-by-step approach, it is not cookie cutter. In hindsight, I also realize that TPI does not require use of TMAP. You can still use TPI to clarify areas of improvement and use other methods to achieve results.

Execution & Release

For us, executing our test strategy was easier than creating it simply because we'd already hit most resistance and questions in the test planning and design stage. That is not to say we coasted on auto-pilot. During execution, most of the Change Team merged back into their test roles. However, they remained a resource for knowledge, guidance, and mentorship that helped sustain the momentum. Further checkpoints and milestone criteria developed during the planning changes helped to measure progress.

The difficulty of execution was mostly around the complexity of the product and the usual rework that occurs for design changes and unplanned risks. However, early involvement in design, strong planning, and thorough tests around high-risk areas enabled the team to ship a high-quality product with minimal escaped issues.

Preventing the Next Fire

The age of more, better, faster is upon us. Undergoing a big improvement effort is tough and risky. Sometimes you must leap forward to catch up and it's achievable with the proper rigor. Still, taking changes in small doses and running pilots is the most nimble method of improvement. If you're looking to prevent that next fire, here are a few thoughts:

- **Structured analysis**. Post mortems and brainstorming are a great way to identify improvement areas. Yet, the issue at hand may not be the one causing the most problems. In turn, the improvement effort may still be dwarfed by a larger problem. Proper analysis allows us to spend our time in the right places.
- **Get commitment**. Any effort not directly aimed at the product is at risk for redirection. Get management and stakeholder commitment for any effort (large or small). If people believe and understand the investment, they will be less inclined to pull it back.
- **Schedule & Results**. Efforts are easy to dissolve if they don't appear to be yielding results. Don't send testers off to look at improvement areas without a clear objective, specific milestones, and regular deliverables. Continue to get management and stakeholders to re-commit with every step. If momentum and commitment aren't building, consider this a red flag.
- **Stay focused.** Moving on too many fronts can weaken your overall results. Weak results cheapen the efforts and builds skepticism within the team. Focus on key areas you know you can accomplish well. Target high benefit, low cost areas first. If you have committed to solve a problem, finish what you've started.
- **Stay ahead of the curve**. Always keep your ear to the ground for new techniques and ideas. The success of today may be your failure of tomorrow if you hold on too long. Look outside of your team's perspective. Talk to other teams, look at the industry, and become active in the Test community.
- **Sustain the momentum.** Consider yourself a work in progress. A maturity model is a great tool to measure where you are and where you want to go. Continue the improvement processes by committing to achieve new levels.

Acknowledgements

This paper cannot end without my expressed thanks to those who were part of it all.

My heartfelt gratitude to our test team whose sacrifice and dedication made it all possible. Thank you Jose Luis Aguilar, Paul Barnett, Liz Dofredo Murakami, Blaine Hammond, Ric Hernandez, Colin Kushneryk, Wendy Lehman, Thai Pham, Sean Quiriconi, Feng Zhou, and Andrey Zvyagilskiy!

Special thanks to the key contributions of Colin Kushneryk's never-ending research, Blaine Hammond's analytical powers, Paul Barnett's vision and technical depth, Sean Quiriconi's leadership and drive.

Thank you to the Program Management and Development teams who "humored" our ideas, shared their frustrations, and made fun of our yellow book. In particular, I'd like to thank Bama Ramarathnam, Shreedhar Madhavapeddi, Jonathan Forbes, and David Auerbach for helping us improve the product process as a whole.

Of course without the sponsorship of management, we would have been doomed to fail. Thanks to Dan Sivertsen, Michael Connolly, Jeremy Stone, and Steve Liffick who supported our cause, had patience for our learning, and forgave for our mistakes.

The writing of this paper would not have been possible without the support and contribution of others. Thanks to all who reviewed and provided feedback. In particular: Colin Kushneryk, Harry Robinson, Jose Luis Aguilar, and Nancy Monson.

Special thanks to Kathy Iberle for her insightful reviews. Her thought-provoking feedback always stretched me to look at our results from different angles.

Thanks to Pacific Northwest Software Quality Conference for providing the opportunity and platform for our experience to be shared.

Thanks to my manger Rich Owens for his support of this writing project and understanding of my sleepless nights.

Finally, thanks to Harry Robinson for his mentorship, encouragement, and infectious passion for testing. For without his help, this would be one more test experience which stayed in the depths of my mind.

References

Basistha, Bijumon. *Tools for Analysis, A Software Perspective*. July 2003 http://www.stickyminds.com

Bellinger, Gene. *Root Cause Analysis*. 2004. http://www.systems-thinking.org/rca/rootca.htm

Carnegie, Dale. *How to Win Friends & Influence People*. Hauppauge, NY: Simon & Schuster, ISBN 0-671-72365-0

Iberle, Kathy. *But Will It Work For Me?* PNSQC Proceedings 2002. http://www.kiberle.com/2002/communitiesofpracticeNew21.doc

Koomen, Tim and Martin Pol. *Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing.* New York, NY: ACM Press, ISBN 0-201-59624-5

Kotter, John P. *Leading Change*. Boston, Massachusetts: Harvard Business School Press, ISBN 0-87584-747-1

McConnell, Steve. *Software Project Survival Guide*. Redmond, WA: Microsoft Press, ISBN 097-0002186

Pol, Martin, Ruud Teunissen, and Erik Van Veenendaal. *Software Testing: A Guide to the TMAP® Approach*. Addison-Wesley, ISBN 0-201-74571-2

Rooney, James J. and Lee N. Vanden. *Root Cause Analysis for Beginners*. Quality Progress July 2004: 51-53.

Six Sigma Pocket Guide. 10th ed. Massachusetts: Roth & Strong Management Consultants, ISBN 0-9705079-0-9

Does Software Development come under the Total Quality Management umbrella?

Ita Richardson[2*], Gerry Coleman[1], Norah Power[2], Kevin Ryan[2]
[1]Dundalk Institute of Technology, Dundalk, Co. Louth, Ireland,
[2]University of Limerick, Limerick, Ireland
gerry.coleman@dkit.ie, norah.power@ul.ie, ita.richardson@ul.ie, kevin.ryan@ul.ie

Biographies

Mr. Gerry Coleman is a lecturer in the Department of Computing and Mathematics at Dundalk Institute of Technology. He is a part-time PhD student at Dublin City University, where he carries out research within companies using software metrics. Mr. Coleman also supervises M.Sc. students and has secured research funding from a variety of sources.

Dr. Norah Power is a lecturer in the Department of Computer Science and Information Systems at the University of Limerick. Her main research interest is in Software Requirements Engineering. She also lectures at undergraduate and postgraduate level and supervises Ph.D. Requirements Engineering students. She is a researcher on the B4STEP project (www.b4step.ul.ie) which is funded by Science Foundation Ireland.

Dr. Ita Richardson is a senior lecturer in the Department of Computer Science and Information Systems at the University of Limerick where she lectures to undergraduate and postgraduate students. Her main research interests are in Software Process Improvement with a specific focus on small to medium sized enterprises (SMEs). She is currently supervising M.Sc. and Ph.D. students in this area. She is involved in two SFI funded projects – as a researcher on the B4STEP project and as project leader of the SMEs project within the Global Software Development (GSD) Cluster.

Prof. Kevin Ryan is the Professor of Information Technology in the Department of Computer Science and Information Systems at the University of Limerick. His main research interests are in Requirements Engineering and Software Process Improvement. He is Principal Investigator on the SFI funded GSD Cluster and leads the Irish Software Engineering Research Consortium (ISERC). He is also a researcher on B4STEP.

The authors are members of ISERC, the Irish Software Engineering Research Consortium (www.iserc.ie), and are supported by Science Foundation Ireland (www.sfi.ie).

Abstract

We outline the four primary total quality management (TQM) concepts of fitness to standard, fitness to use, fitness of cost and fitness to latent requirement, and then discuss how each can be applied to software development although often under different names. We show how our research in four key areas: software metrics, software requirements engineering, software process improvement, and cost-benefit analysis can be related in turn to the four fitnesses of TQM. Drawing on extensive industrial collaboration, this

* corresponding author

paper demonstrates how each of these software engineering research areas can contribute to bringing software development under the TQM umbrella.

Total Quality Management

For an organisation to be profitable, it must supply quality products. To get quality products, the four basic concepts of quality as defined within total quality management (TQM) should be considered. These are:

- Fitness to Standard
- Fitness to Use
- Fitness of Cost
- Fitness to Latent Requirement.

Fitness to standard and fitness of cost both focus on company issues; fitness to use and fitness to latent requirements focus on customer issues. Depending on what focus the company is taking – companies using TQM fully will be focusing on all concepts - different quality tools are used. The evolution of the methodologies as they relate to Total Quality Management is shown in Figure 1 (Shiba et al., 1993) starting with traditional in-house quality control (QC) and leading, over time, to full Quality Function Deployment (QFD).

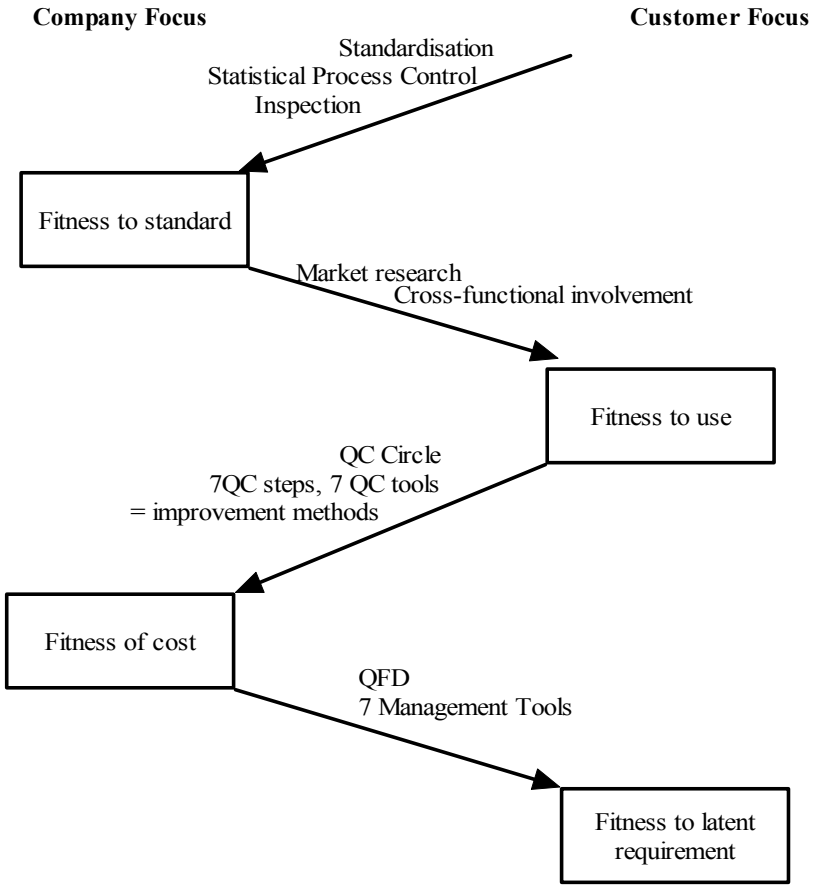


Figure 1 : Evolution of methodologies as they relate to Total Quality Management

Fitness to Standard

Fitness to standard means that the product built conforms to the standards laid down by the designers. When inspecting a product against a standard, it can either pass or fail the inspection. Statistical quality control is usually the evaluation tool used when determining fitness to standard. A weakness of using this concept as the determinant of quality within an organisation is that, in reality, it is very difficult to determine the quality of a product during inspection. Secondly, the product may pass the 'fitness to standard' test, but may not be the product that the customer wants.

Fitness to Use

A customer purchases a product for a particular use. The difficulty faced by designers is that at times the customer will use the product for tasks other than those for which it has been designed. A simple example of this is the use of a screwdriver. Screwdrivers are designed to turn screws – however, another use of a screwdriver is to open cans of paint. Designers must be aware of what the customer is doing with their products and design them to fit those uses. Companies began using market research as a means of determining the fitness to use of products. Quality thinking now urges the designers to go to the 'gemba'- which means getting into a situation where they use the product as a consumer would use it.

Fitness of Cost

Not only do companies need to produce products that comply with the concepts of fitness to standards and fitness to use – they must also comply with fitness of cost. This implies that the product can be produced at as low a cost as possible. The primary means by which this can be achieved is by improving the production process, so that produced goods will pass the inspection process and not be discarded or reworked. Towards this goal, the seven steps for quality control and the seven tools for quality control have been devised[1].

Fitness to Latent Requirements

Fitness to latent requirements implies that companies can gain a competitive advantage by determining customer requirements which the customers themselves do not realise they have. This can give a company the leading edge into the market. One of the popular examples of latent requirements is the development of the Walkman music system. It was a product for which customers had not expressed a need, but since it has been produced it has been highly popular. The seven management and planning tools[2], of which Quality Function Deployment is one, have become standard tools for the determination of latent requirements.

Total Quality Management and Software

Because of the importance of the software industry to the Irish economy, where it employs some 30,000 people and account for 8% of GDP, it is imperative that quality is

---

[1]For further information see Shiba et al. (1993) or Bergman and Klefsjo (1994)

[2] For further reading see King, 1989.

high on the agenda of software development organisations. At the same time many concerns are being expressed about high testing and maintenance costs of software products, and there are many well-known cases where software has conspicuously failed. Much of our recent research has had a quality focus, in one form or another. Recognising the effectiveness and relevance of the total quality management model to quality improvement within other organizations, we have examined how these concepts can be, or are being, applied to software development.

Fitness to Standard – Software Metrics

In software development, measurement is used to determine how well a software product conforms to the standards set. There are two aspects that can be measured, the software product itself and the process used to create the product. The act of measurement produces a series of *software metrics* which supply key information about the quality of both the product and the process. Most software development managers use metrics to control their development effort and to assist in making future predictions.

Types of Software Metrics

Collecting software metrics initially establishes a baseline for software development effort. This baseline creates a foundation against which improvement can be measured (Humphrey, 1995).

Metrics Category
Quality
Size
Productivity
Functionality
Complexity
Reliability
Variance etc.

Table 1 - Types of metrics typically collected

Table 1 shows the types of metrics that are typically collected in software development companies. If we take quality as an example, then the most common way of determining the quality of a software product is by counting the number of defects within it. Put simply, defects occur when products do not meet specifications for designated attributes of product quality (Florac & Carleton, 1999). Any defects uncovered must then be removed and the software repaired. For standardization purposes, and in order to compare the quality of one product with another, defect counts are normalized around product size through the use of metrics such as:

$$\frac{\text{Number of defects discovered in the source code}}{\text{Number of lines of source code}}$$

The resultant figure is usually multiplied by 1,000 giving the number of defects per thousand lines of code or defects per KLOC. Of course there can be difficulties in deciding what to count as a (distinct) defect.

Through techniques such as software inspection and software testing, defect metrics can be collected at a number of stages throughout the software lifecycle, including requirements, design, and compilation and during pre-release testing and post-release operation. By collecting metrics at these various checkpoints the quality of the product can be monitored as it goes through its lifecycle.

Expanding on this, yield figures, including the percentage of total defects removed during a quality control phase, provide an indication of the quality of the process. For example, an Inspection Yield can be defined as:

$$\text{Yield (Inspection)} = 100 * \frac{\text{Number of defects removed by the inspection}}{\text{Total defects (removed by inspection + found later)}}$$

Process quality and conformance to standard can be measured in other ways including productivity and timeliness. Productivity is essentially the rate at which product is produced. Timeliness relates to when the product is delivered to the customer or sponsor and reflects how successfully initial delivery estimates have been adhered to. Productivity can be measured as:

$$\text{Volume of product per hour} = \frac{\text{Volume of product produced}}{\text{Number of hours expended}}$$

Volume itself is generally measured by lines of code, function points or some other size representation and while they pose some difficulties, these measures of productivity assist predictability and enable more confident estimates to be provided to customers.

Timeliness is usually measured as the percentage error between the original estimate and the actual delivery time and can be represented as:

$$\text{Time estimation Error (\%)} = 100 * \frac{\text{Actual time - Original estimate}}{\text{Original estimate}}$$

Use of Software Metrics in practice

We have conducted research within a number of software companies with a view to determining what software metrics are being used in practice. Several of these experiences are detailed below.

Company 1
This organization develops software products for the telecommunications sector and is using the eXtreme Programming (XP) methodology. Within the company, when a new piece of work commences, a group of engineers examine the high-level software

requirements and estimate what the system would cost to develop in 'perfect' engineering days. 'Perfect' engineering days assume that the engineers are working exclusively on the development, without interruptions such as phone calls, meetings or breaks. The resultant metric provides an initial overview of how long the system will take to develop and enables the requirements to be prioritized and allocated to development iterations.

Following this, individuals, or pairs of engineers, take each of the requirements in turn, break them into their constituent tasks and estimate these in 'perfect' engineering hours. Because it is now getting closer to implementation, the estimates become more accurate. Subsequently, as engineers develop the software, they record the actual time taken to satisfy the various requirements.

A second metric collected is the ratio of 'planned' work to 'discovered' work. Planned work is how the engineers originally interpreted the requirements. Discovered work relates to requirements issues not originally foreseen.

The combination of the above metrics, actual development effort, and the planned/discovered work ratio, together push the development time beyond the original 'perfect' estimate to produce the actual delivery time. The actual delivery time, therefore, is a multiple of the original 'perfect' estimate. Over time Company 1 has refined its estimating and delivery capability so that the actual delivery multiple now ranges from 1.5 to 1.7 times the original estimate. By using carefully targeted, metrics-driven improvement efforts, the company is endeavouring to reduce this multiple still further.

Within the XP environment, Company 1 staff gather other metrics including the percentage of time spent in pair programming. They also carry out progress trending, where they compare the rate at which development work is being completed against the rate at which requirements are being added.

On the quality side, they use test-first development, alongside more standard acceptance and regression testing. Defects during development are tracked to repair using the open source tool Bugzilla and additional metrics are collected including the number of defects per iteration.

Company 2

Company 2 produces software solutions for the pharmaceutical industry and is ISO 9001 certified. Staff use the 'V' model for their projects and are especially concerned with quality issues. Source code is kept under tight configuration management and checklist-driven peer reviews are used as quality control mechanisms. Issues found during review are tracked to completion.

Metrics are collected on the number of issues found during these reviews and the amount of time taken to do the review. Ratios are also derived on the number of defects found in system test versus the number found during peer review. At present, there is almost a 1:1 relationship between the number of defects found in peer review and the number found in system test.

These metrics, and those on the actual number of defects found, are then used to set targets for future releases. Where the number of defects found appears high, functionality measures are taken of the relevant system module and compared against historic norms to determine if any additional action or investigation is required.

Company 2 also collects effort figures in terms of staff-hours to complete particular tasks. Using time sheets, staff record how they spend time on various aspects of their work and these figures are recorded in a database to assist future project estimations and projections.

Company 3
Using a tailored iterative development approach, Company 3 creates software products for the banking sector. Staff collect a number of metrics which drive effort estimates in all of the projects.

During requirements capture they produce Use Case documents and count both the number of methods and attributes, estimating how long it will take to design each one of these. Assessments of complexity, in relation to these methods and attributes, are also undertaken, with each being allocated a 'high', 'medium' or 'low' rating. Historic figures, which have been refined over time, are maintained to assist in improving the accuracy of these effort estimates.

From a QA perspective, counts are also kept of the number of times a piece of software goes back and forth between developers and testers. The objective here is to reduce the number of times that these "over and backs" happen. Originally when the company started collecting this data there were on average 6 or 7 "over and backs" and this number has now been reduced to 1 or 2.

Finally, Company 3 also keeps more granular figures, such as the length of time it takes to produce a set of standard product screens and these are used in effort projection at the design stage.

As can be seen from the industry examples above, software metrics can help ensure tight management and control of software development and assist software development organizations in conforming to standard to achieve total quality management.

Fitness to Use – Software Requirements Engineering
Within the software development discipline, requirements engineering (RE) is the development and use of cost-effective technology for the elicitation and analysis of the stakeholder requirements. Stakeholders are people and organizations that have a concern about and/or an interest in the outcome of a software development project. Often seen as sources of requirements, they bring varying degrees of influence to bear on how the requirements are formulated.

Stakeholder requirements comprise functional and what are called non-functional requirements. Functional requirements specify what the software must do if it is to carry out the work for which it is intended (Robertson, 1999), while non-functional requirements are often called quality requirements (Lauesen 2002) and include performance requirements, as well as attributes of the software such as maintainability, security and availability. RE is therefore concerned with the fitness to use of software products and systems.

Requirements engineering (RE) is the development and use of cost-effective technology for the elicitation and analysis of the stakeholder requirements.

Requirements Specification

Software requirements engineering comprises five main processes: requirements elicitation, requirements specification, requirements validation, requirements verification, and requirements management. Requirements specification is recognized as one of the most difficult and most critical tasks in RE (The Standish Group, 1995). It is the process of expressing the requirements for a system or product in a document or other format that can support the verification, validation and management tasks of RE. The requirements specification, properly done, ensures that the outcome of RE can be used in the later (downstream) processes of software development.

There is very little agreement in the RE literature on what the requirements specification is or on exactly what it should contain (Harel, 1987; IEEE, 1998; Kovitz, 1999; Parnas, 1995; Robertson, 1999). Numerous competing methods, techniques, and languages for expressing requirements have been proposed, many of them based on the idea of modeling the application or problem domain, using either a formal or semi-formal notation. On the other hand, the standards and guidelines for structuring a requirements specification assume that requirements are expressed mainly in natural language (IEEE, 1998).

Definition of Quality in a Requirements Specification

In theory, at least, the fitness to use of a software requirements specification depends on certain qualities of the specification. These correspond to the accepted definition of what makes a good software requirements specification. The standard reference for this definition is the IEEE Recommended Practice for Software Requirements Specifications (IEEE, 1998) which lays down the following list of eight 'quality criteria' for a software requirements specification (SRS):

1. Correct,
2. Complete,
3. Unambiguous,
4. Consistent,
5. Ranked for importance and stability,
6. Modifiable,
7. Verifiable,
8. Traceable

In practice, however, completeness in a requirements specification is not always considered important, as the specification document is often seen as a tool for requirements engineering rather than as a product of the RE process. Similarly, ambiguity is often seen as unavoidable in practice, a consequence of using natural language, but not always a problem, because requirements engineers do not rely on written requirements as their sole means of communication in a project. In short, the desirability of these eight qualities of a specification is a function of the situation of use, i.e. their relevance depends on the situation in which the specification is used (Power, 2001).

Requirements Specifications in Practice

The research reported here is based on a qualitative empirical investigation into the industrial practice of RE. It set out to explore the different factors that might influence the approach taken to software requirements specification, including the organizational context of development, the organizational context of use of the software product, the application domain and the system architecture. This investigation was based on twenty-eight semi-structured interviews with a wide range of experienced practitioners, focusing on requirements engineering in general and requirements specification in particular. As a qualitative study, the aim was to sample for variety rather than seeking a traditional representative sample.

The practitioners were drawn from a range of software development organizations, including in-house developers, people working on-site for consultancy firms and those working in software houses and other organizations that specialize in software development. Their products ranged from large and small business information systems to embedded software for consumer electronics or industrial electronic equipment. They were selected by asking other software professionals to recommend someone whom they considered successful in the area of requirements engineering, who would be experienced and authoritative in their responses, who would be able to give frank accounts of how they created and used requirements specifications in their work. In the interviews they were asked to discuss requirements specification practices they had used in successful projects.

One of the aims of the research was to study the different types of requirements specifications that are used in practice and try to account for the differences. Besides the use of ancillary techniques such as prototyping and use cases, which are more useful for eliciting requirements than for specifying them, many different styles of requirements specification were identified. These styles were based on a variety of combinations of specification 'elements' such as problem domain descriptions, features, constraints, goals, issues, and proposed solutions.

Uses of Requirements Specifications

Several different uses for requirements specifications were reported (see Table 2). However, each practitioner put forward a different subset of this list in describing his or her own practices. Some used the requirements specification before prototyping while others would write the specification only after prototyping had been completed. Some used it as input to 'downstream' development processes such as design and testing, while

others used it more as a discussion document, as a means of getting to agreement with key users. Some used it as an end rather than a means, to document a formal contract.

Initial analysis of the interview data suggested that one of the main causes of variation in the uses and roles of the requirements specification was the contractual arrangement (and other aspects of the relationship) between the software developers and the customer, client, or users. Other possible causes of variation, such as product type, application domain and system architecture were rejected, as they were not supported by the interview data.

Uses and Roles of a Requirements Specification
as a discussion document
as a means of communication
as a working document
as the basis of the architecture
as the basis for prototyping
as input to costing
as input to design
as input to project planning
as input to testing
as input to user manual
in looking for solutions
to tell users what they are getting
to evaluate proposals
to document agreement
to get agreement
to articulate knowledge
to express understanding

Table 2

Classification of Requirements Engineering Situations
1. Vertical Market Situation
2. Target Customer Market
3. Sub-Contract for a Product
4. Custom Solution Contract
5. Configured Solution Contract
6. In-house Solution Situation
7. Problem-oriented Situation

Table 3

Table 2 shows some of the initial codes that resulted from the analysis of the interviews, the result of a procedure called 'open coding' in grounded theory analysis (Strauss and Corbin, 1990). Following this, using other coding procedures of grounded theory analysis, a scheme for classifying situations was developed, based on the configuration of sources of requirements such as stakeholders. This led to the identification of seven different types of requirements situations (see Table 3) in the empirical data provided by the practitioners' accounts of their own situations.

Situations of Use

The resulting framework based on these categories is flexible and extendable (Power & Moynihan, 2003). It can be used to classify any project situation, or a set of projects, with respect to the main sources of requirements, such as stakeholders, in that situation. Each type of situation is associated in the framework with a documentation profile, describing the particular specification elements that were typically used by the practitioners working in those types of situations to specify requirements. Rather than expecting 'one size to fit all' it allows for different situations of use to determine the appropriate contents and

therefore the quality attributes of a requirements specification. The framework is based on the practices of skilled and experienced requirements practitioners. For less experienced software practitioners the framework provides practical guidance and suggestions for creating and using requirements specifications in different types of development situations (Power, 2004). Ultimately, this guidance is intended to help practitioners to specify software products that will themselves possess the quality of fitness to use.

Fitness of Cost – Software Process Improvement

Since the late 1980s, Software Process Improvement (SPI) has been gaining popularity as an approach to improving software quality. It is expected that when the software process improves, the software product also improves. It has also been claimed that when the process improves, cost is reduced and delivery schedules become more reliable.

Humphrey (1989) defines a software process as "the set of tools, methods and practices we use to produce a software product". Paulk et al. (1993) expand this definition to "a set of activities, methods, practices and transformations that people use to develop and maintain software and the associated products". An ISO approved process model – Software Process and Capability dEtermination model (SPICE, ISO15504) sub-divides the processes within an organization into five categories – Organisation, Management, Project, Customer-Supplier and Engineering. Processes followed within software development life-cycles are software processes, but there are others. Examples include project management, recruitment and training, configuration management and risk management.

When implementing software process improvement within an organization, the first step is to assess the current process, usually with reference to an established model, for example, Capability Maturity Model (CMM), Capability Maturity Model integrated (CMMi), SPICe or ISO9001-2000. Based on this assessment, the company makes a decision to change specific processes. To make this decision they need to consider various factors including business goals, current state of the process, expected return on investment and the organizational expectations of the process.

How does SPI relate to Fitness of Cost?

In the effort to ensure that a software product can be produced at as low a cost as possible, it is essential to study and improve the software development process, so that produced goods will pass the inspection process and not be discarded or reworked. Our experience has been that, in software development practice, cost can be reduced through improvement of the software process, and therefore, companies aiming for fitness of cost should be interested in software process improvement (SPI).

Use of Software Process Improvement

Qualitative studies were conducted on software process improvement (SPI) projects within a number of small software development companies in Ireland. The improvements made in example process areas are summarized in Table 4 and some are discussed in more detail.

Process	Procedures at start of project	Procedures at end of project
Engineering	Various specifications written giving inconsistent information; Moving engineers between project teams adversely affected management of knowledge.	One specification covered all aspects; Consistent information available; Knowledge loss not as significant.
Project Mgt	Customer services decided deadlines for development; No project plans written for projs; No project reporting by engineers	Deadlines discussed with engineers; Project management and control implemented.
Customer Support	Documentation not supplied; Cust support not documented; Code put live by developers.	Documentation issued with product; Control on customer queries; Configuration mgt introduced.

Table 4: Summary of procedures at start and end of Research Project

In one case, the company focused on improvements to their engineering processes. They introduced a program specification which replaced previous requirements, technical and functional specifications. Software engineers were now working from consistent information and when there was movement between project teams, the project could be completed successfully. This ensured that the project was able to keep within schedule and within cost, something which they had not been able to do prior to the improvement effort.

Where project management was improved, this resulted in more realistic schedules. Schedules were set in conjunction with the software engineers rather than being dictated by the customer services group. Engineers were expected to give the manager feedback and, consequently, he was able to keep tighter control on the project. Again, this had an effect on the reducing overall costs of production.

Another company we worked with improved their customer support processes. When new projects were implemented in customer sites, the engineers were expected to provide the support group with completed documentation, which had not happened prior to this SPI project. They also introduced procedures for dealing with customer queries such as the completion of customer support request forms, ensuring that configuration management was implemented for modified code and the closing of requests when appropriate. This resulted in a faster response to customers, and a reduction in maintenance costs to the company. It also had the effect of allowing software engineers to work on new projects to which they were assigned.

The success of software process improvement within the small software development companies was summarized by one of the managing directors. He attributed the improvement in the company's revenues to the improvement in the software process. In the past, they had many 'irate customers' who were dissatisfied with the product being supplied. Now, they had satisfied customers who were prepared to pay for upgrades to

their products. The improved quality achieved through SPI has indeed resulted in reduced costs and increased revenue for the small software development companies.

Fitness to Latent Requirements – the Cost-Value Approach

Fitness to latent requirements implies that companies can gain a competitive advantage by identifying customer requirements of which the customers themselves were not previously aware. Adding features that are identified in this way can give a company the leading edge in its market. A method developed to assist in prioritizing software requirements was found to be useful in discovering and selecting latent requirements. This was observed in a study carried out in a multi-national telecommunications organization.

A number of requirements management approaches have been developed which help identify, clarify and prioritise the software system requirements. Companies such as Siemens (Gloger and Weber, 1998), Ericsson Radio Systems AB (Karlsson & Ryan, 1997) and Nokia Mobile Phones (Ronney et al., 2000) have found such tools helpful in identifying latent software requirements. The use of the Cost-Value approach at Ericsson Radio is used here to illustrate this phenomenon.

What is the Cost-Value Approach?

The Cost-Value Approach (Karlsson & Ryan 1997) was devised initially to support organizations in deciding which of a set of possible requirements they would chose to satisfy in developing a new system. The basic problem is that such decisions are made in almost every large system project – since we can rarely afford to meet every requirement we would like – but often they are based on nothing but expediency or on individual prejudices and guesswork. The cost-value approach is based on the Analytic Hierarchy Process (Saaty, 1988) which was originally developed for the design in mechanical engineering. The approach supports a joint team of technical staff and customers as they compare pairs of possible requirements in terms both of the cost of implementation and the value to the customer. Each comparison results in a scaled indication of the relative rating. Not all pairs have to be compared and, as the comparisons proceed, it is possible to estimate the degree of consistency in the decisions so far. Stopping rules can be applied which usually allow us to cease making comparisons long before all comparisons have been made. Appropriate software support makes the entire process much more efficient and prototype tools are described in (Karlsson, Olsson & Ryan 1997). From this pair-wise comparison the method can then compute a robust ranking in the two dimensions of cost and value. The results can be plotted on a diagram which graphically illustrates how some requirements represent excellent value for money while others are high-cost but of little value to the customer. (Further details of the commercial use of this approach can be found at www.focalpoint.se who provided the following illustration.)

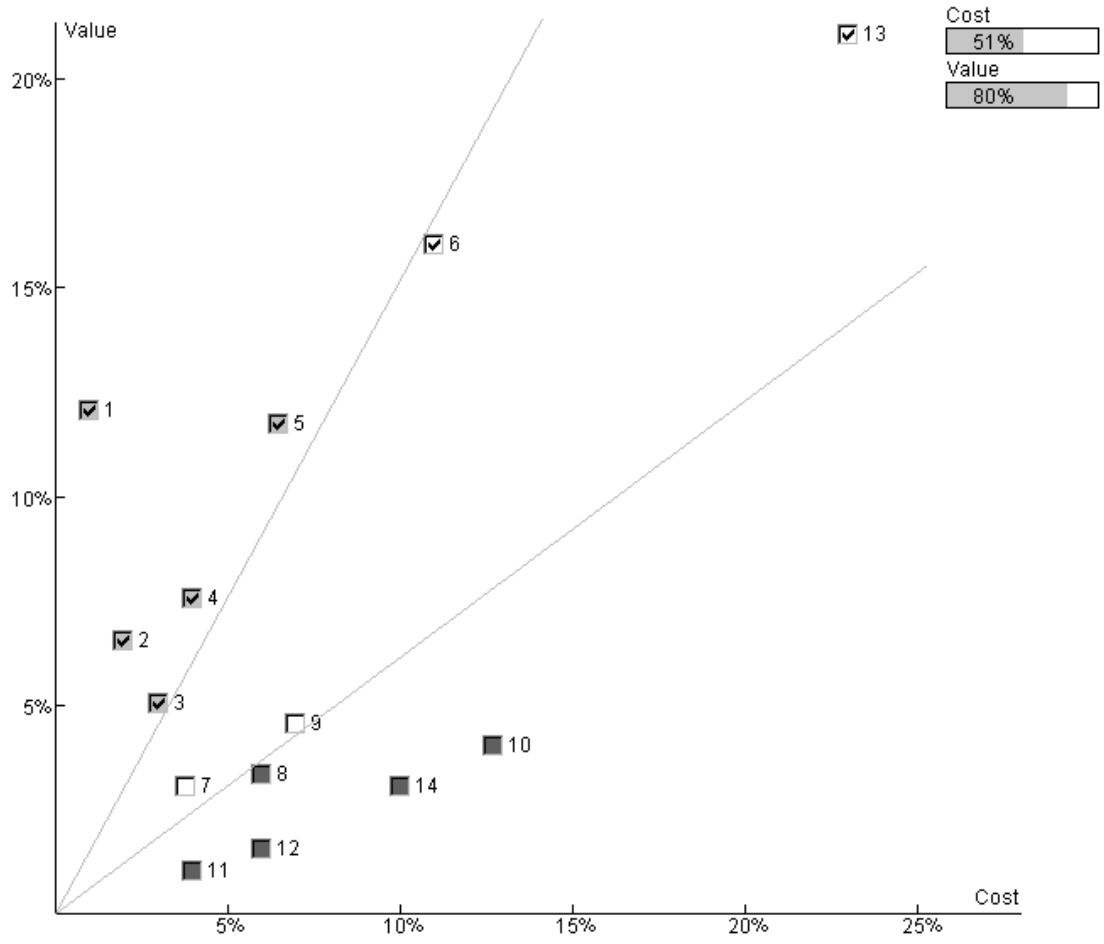


Figure 2 : An example Cost-Value Diagram

In this diagram the 'good value' requirements are located in the top sector while the 'bad value' ones are in the bottom. For example, Requirements 1 and 5 are very good value and while those numbered 10 and 14 are bad value. Requirements proposed for implementation have been ticked, and the tool has computed that for 51% of the potential total costs we can achieve 80% of the potential total value.

The case study reported in (Karlsson and Ryan 1997) was conducted with Ericsson Radio Systems in the development of a telephone base station. A small subset of the potential system requirements was selected and the cost-value process was applied in a manner similar to that shown above. As reported in that paper, and repeated elsewhere, the following two effects were observed.

1. The comparison process served to clarify the requirements both to the developers and to the customers (or the marketing people who acted as surrogates)
2. Costing information, which had not previously been available in such explicit terms to the customers, had a major impact on the eventual decisions about what to implement.

How does the Cost-Value Approach relate to Latent Requirements?

Experience of using the cost-value approach with a number of customers has shown that the clarification process described above led customers to request, or developers to offer, additional features that had not been envisaged at the start of the requirements engineering process. These features were considered for incorporation into later releases of the system. So although the cost-value approach was not originally designed with the intention of uncovering latent requirements, it has definitely had this very useful side-effect when put into industrial practice.

Conclusion and Future Work

This paper has shown that many of the software quality improvement methods and techniques that have been developed over the past ten years can be used, in industry, to help bring the TQM philosophy into the software development process. Although many of these improvements, and much of the research underlying them, were not inspired by TQM they can be used to help achieve the four fitnesses: fitness to standard, fitness to use, fitness of cost and fitness to latent requirement.

No organization that we have encountered uses all, or even most, of the approaches we have described. We believe that future work should aim to bring all of these approaches – and others that have been developed in support of TQM - to bear in a co-coordinated and coherent manner throughout the software development process. Only then can we truly say that software development has been brought under the TQM umbrella.

References

Bergman, Bo and Bengt Klefsjo, 1994. *Quality from Customer Needs to Customer Satisfaction.* Studentlitteratur, Sweden.

Florac, W. A. & A. D. Carleton, 1999. *Measuring the Software Process.* Addison Wesley.

Gloger, Stefan Jockusch and Norbert Weber, 1998. Using QFD for Assessing and Optimising Software Architectures: The System Architecture Analysis Method in *Proceedings of the World Innovation and Strategy Conference incorporating the Fourth International QFD Symposium*, Sydney, Australia, 2nd-5th August, pp 119-127.

Harel, D., 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 231-274.

Humphrey, Watts S., 1989. *Managing the Software Process*. Addison-Wesley, Reading, M.A., U.S.A.

Humphrey, Watts S., 1995, *A Discipline for Software engineering*, Addison-Wesley, Reading, M.A., U.S.A.

IEEE, 1998. *IEEE STD 830-1998: IEEE Recommended Practice for Software* Requirements Specifications. Los Alamitos, CA: IEEE Computer Society Press.

Karlsson, J and Ryan, K, 1997. A Cost-Value Approach for Prioritizing Requirements, *IEEE Software*, Volume 14, Issue 5 pp. 67 - 74

Karlsson, J, Olsson, S and Ryan K, 1997. Improved Practical Support for Large-scale Requirements Prioritizing - *Requirements Engineering Journal* Vol.2 No.1.

King, Bob, 1989. Better Designs in Half the time, GOAL/QPC, 3rd Edition.

Kovitz, B.L., 1999. Practical *Software Requirements: A Manual of Content and Style*. Greenwich, CT: Manning.

Lauesen, S., 2002. *Software Requirements - Styles and Techniques.* Addison-Wesley/Pearson

Parnas, D.L., 1995. Using Mathematical Models in the Inspection of Critical Software. *In*: Hinchey, M.G., Bowen, J.P. (eds.), *Applications of Formal Methods*. Hemel Hempstead: Prentice Hall. 17-31.

Paulk, Mark C., Curtis, Bill, Chrissis, Mary Beth, and Weber, Charles V. 1993. *The Capability Maturity Model for Software, Version 1.1. Technical Report SEI-93-TR-24*, Software Engineering Institute, Carnegie Mellon University, U.S.A.

Power, N., 2001. Variety and Quality in Requirements Documentation. *Proceedings Requirements Engineering, Foundations of Software Quality, REFSQ2001.* June 2001, Interlaken.

Power, N., 2004. The Agreement Dimension in Software Requirements Documentation. *Proceedings Requirements Engineering, Foundations of Software Quality, REFSQ2004.* June 2004, Riga.

Power, N., Moynihan, T., 2003. A Theory of Requirements Documentation Grounded in Practice. *SIGDOC '03, October 12-15, 2003, San Francisco.*

Robertson, S., Robertson, J., 1999. *Mastering the Requirements Process*. Harlow, UK: Addison Wesley.

Ronney, Eric, Olfe, Peter and Mazur, Glenn. 2000. Gemba Research in the Japanese Cellular Phone Market. *Transactions from the 12th Symposium on Quality Function Deployment*, June 5-6, Novi, Michigan, pp358-374.

Saaty, T.L., 1988, *The Analytic hierarchy Process*, McGraw-Hill, New York.

Shiba, S., A. Graham and D. Walden, 1993. *A New American TQM: Four Practical Revolutions in Management*, Productivity Press, Oregon, U.S.A.

Strauss, A., Corbin, J., 1990. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Beverly Hills, CA: Sage Publications.

The Standish Group, 1995. *The Standish Group Report - CHAOS* [online]. Available from: http://www.cs.nmt.edu/~cs328/reading/Standish.pdf [Accessed May 2004].

Encouraging SME Software Companies to Adopt Software Reviews in Their Entirety

Tervonen Ilkka and Harjumaa Lasse
Department of Information Processing Science
University of Oulu, P.O. Box 3000, FIN-90014, Oulun yliopisto, Finland
Tel: +358 8 553 1908, Fax: +358 8 553 1890,
e-mail: ilkka.tervonen@oulu.fi, lasse.harjumaa@oulu.fi

Ilkka Tervonen is a professor of software engineering at the University of Oulu specialized in software quality and quality techniques. He received his PhD in software engineering from the same university in 1994. His current research interests include software quality, software inspection, process improvement and open source software, and he heads the research team i3GO (improved inspection initiative Group in Oulu).

Lasse Harjumaa is a lecturer and PhD student at the University of Oulu. His PhD work focuses on inspection process improvement by means of process patterns. His current research interests include software inspection, agile methods and process improvement, and he is a member of the i3GO research team.

Abstract

Small and medium-sized enterprises (SME) in the field of software have recognized quality as an asset, but they tend to have rather a limited view of possible techniques. They are typically familiar with different types of testing, but are often lacking in knowledge of software reviews and inspections. This paper introduces the results of a long-term research programme conducted with SME software companies which has largely been a learning process, as we walked through the three phases from simple review process tailoring through process improvement to the analysis of motivational factors and obstacles to reviews.

We learned that although the members of the quality group may be interested in review process improvement, this does not guarantee that they can install improvements in projects. This is understandable, because project schedules are very tight and do not allow for process improvement experiments. We came to the same conclusion in our analysis of motivation factors and obstacles. Although interviewees understand that review projects can enable them to find defects earlier, faster and in greater numbers, shortage of time is the main reason which prevents them from using reviews.

1. Introduction

Small and medium-sized enterprises (SME) in the field of software have recognized quality as an asset and a necessity and are increasingly emphasizing the use of software quality techniques. They tend to have rather a limited view of the possible techniques, however. They are typically familiar with different types of testing, but often lacking a knowledge of software reviews and inspections. Even the terminology may cause confusion, and it is often reasonable to start discussions with company staff using the term review and to focus on inspections later, if needed.

Our research group, i3GO <http://www.tol.oulu.fi/i3/>, has participated since 1998 in various local projects and national initiatives focused on software reviews and inspections. In this paper we report converging results on our experiments, which can be classified into three groups, (1) tailoring the review process to more agile cases, (2) improving the review process by means of capability analysis and process patterns, and (3) analysing review techniques used in SME companies.

Our research has consisted of a learning process in which we have walked through the three phases from review process tailoring through more disciplined process improvement to the analysis of motivation factors and obstacles to reviews. We started with simple tailoring cases because companies were interested in light-weight processes and appropriate tools. Since then we have developed a number of supporting tools for inspection, evaluating their usability in a classroom context. We searched for alternative, light-weight inspection processes that could be carried out on the web, so that, as the inspection meeting was typically considered burdensome in small companies, this could be replaced with tool support. After some experiments in 1998-2000 we realized that the saying “a fool with a tool is still a fool” is very true, in the sense that introducing an inspection tool within a company that does not have an established development process will probably fail.

During the next period, from 2001 until now, we have been looking for review improvements in six companies, four of them SME companies and two similar-sized departments of nationwide companies. The work was based on a capability evaluation matrix in which capability was evaluated by means of twelve review/inspection-related activities. A very rough, generalized estimate of the results is that the review process in the SME companies has been typically at a rather low level, because they have recently started work on improving the process. In one company we introduced a set of patterns to direct the improvement tasks. The idea was that companies that desire to improve their processes often lack adequate knowledge to implement such an improvement efficiently. In our experiment the patterns worked as a concrete tool for improvement, although some examples were also needed in order to exploit the patterns fully.

A tentative analysis of twelve small and medium-sized software development departments in the Oulu area based on a short interview process has just started, the main idea being to find out the most important motivation factors and obstacles affecting the adoption if software reviews in their entirety. Another objective of the interview is to find potential companies for further review process improvement measures.

This paper introduces three approaches to the question of stimulating software companies to adopt software reviews. We first report our results in the tool development field, then our

experiences with review process improvements, and finally the first results of our analysis of motivation factors and obstacles when adopting software reviews in companies.

2. Tool support for web-based inspection

Distributed software engineering projects cannot make use of the same traditional methods as do co-located teams, although their communication and quality assurance needs are the same. This is especially apparent in software inspections that have a heavy accent on teamwork. Although tools are enablers for distributed team working, they are also identified as one of the main problems in development [10]. Inspection as a form of teamwork can be a method for sharing ideas, knowledge and work between team members in different locations. The collaborative aspects of inspection are facilitated by web technology, because the web is global and work is not limited to working hours. This not only introduces flexibility into the inspection meetings (in the form of asynchronism and geographical diversity), but it also enables easy, manageable distribution of the artefacts for inspection, including the document to be inspected, checklists, or other related documents.

Even the early review/inspection tools such as CSI (Collaborative Software Inspection) [13] or CSRS (Collaborative Software Review System) [9] supported the major phases of a web-inspection process. Likewise, web-based tools and prototypes such as CoReview and its successor Group Review [5], developed at the University of Arizona, WiP (Web inspection Prototype) [6], developed at the University of Oulu, HyperCode [15], developed in Lucent Technologies, and ReviewPro [16], a commercial tool, support the inspection of text documents with WWW browsers, and the on-line commenting facilities which they provide are convenient and easy to use. These tools are similar in many aspects, as they support inspection by means of checklists, collect defect data, provide links from code lines to these data and allow readable summary reports to be produced.

Based on experiences with WiP, more comprehensive inspection tools called WiT (Web inspection Tool) and InWin (Inspection Window) [17] have been developed. The main feature of these is their ability to inspect any HTML document. The WiT approach is document-oriented and the web page itself is under inspection. The main objectives include automation of the distribution of the inspection material and the enabling of on-line viewing. The difference between the InWin tool and the WiT tool is that the individual and public inspection phases are merged in InWin. All three tools were tested by computer science students taking part in a Reviewing and Testing course at the University of Oulu in 1998-2000. Approximately 50 people used each tool in a distributed manner (typically from home or company sites) to inspect a requirements specification document. All of them had previously performed a traditional inspection with a logging meeting on a similar document. One rather surprising result of the InWin experiment was that students did not like to start the inspection of a product document with marked issues, but preferred starting without other people's markings.

A brief period of experimentation with the InWin tool has been arranged in a company developing Internet applications, where development takes place at a fast pace, the teams are working extremely dynamically and independently of each other in a hectic environment and there

is not necessarily time to carry out the whole inspection cycle. Thus InWin was a more suitable option. The focus of the experiment was on the role of the inspection moderator.

The organisation in which the tool was tested did not have software inspections mentioned at all in its process model before the introduction of the InWin system, nor had it ever done any inspections. The potential of the method had been noticed, however, and the staff were willing to implement inspections on condition that the process was tailored to be more light-weight and flexible than traditional software inspections with face-to-face meetings.

While a one-step inspection is very simple to prepare and manage, it still strengthens communication between developers, thus making the development more transparent and public among the team members. Furthermore, it is easier to make use of outside expertise in the case of web inspections, even requesting comments from people who may be working in other branches of the organisation. The features of the tool that were most readily appreciated in this experiment, were its adaptability to a variety of operating system platforms and its attachment to a web browser. We also realised, however, that the job of moderator may become quite burdensome, as happened in our experiment.

This experiment in the real world led our development work in a new direction, and we understood that we should place more emphasis on rich functionality, flexibility and integration. Rich functionality in this context means functions that typically support the needs of the inspection, such as distribution of the material electronically, facilities for recording and attaching comments (suspended defects) directly in the document, classification of defects according to their severity and category, checklist management and reporting of inspection efficiency. Flexibility, a characteristic already implemented in earlier tools, means independence of time and place combined with tailorability. Integration of the inspection tool also required more concern, because all supporting tools, such as existing data repositories, authoring tools, version management systems, collaborative systems and project management tools, should work together. To demonstrate and evaluate the interoperability aspects in particular, we have implemented a new supporting tool XATI (XML Annotation Tool for Inspection) [8], which supports the inspection of artefacts with varying structures. Suspended defects are attached directly to the structures concerned and recorded in the database to await further handling.

We recognized that it is fairly difficult to introduce companies to a new tool. This is understandable, because companies strive to keep their development process alive and try to avoid all disturbing factors. This was the reason for experimenting with Adobe Acrobat [7], an off-the-shelf product that supports PDF (Portable Document Format) and may thus be of interest from a company viewpoint. It also allows annotations to be shared between team members and thus enables asynchronous discussion of the issues detected. This is useful if the details of the issue (its severity, for example) have not been agreed on during the first reviewing iteration. Both XATI and Adobe Acrobat have been tested with students in the same way as our earlier tools.

All in all, even though the functionality and ease of use of the tools has improved during the development cycle, it is a somewhat challenging task to induce SME companies to make real use of them. It is more plausible that companies will attempt to use existing tools and tailor them to support reviews as well as possible. This means utilizing the characteristics of the Microsoft Office toolset and Adobe Acrobat, for example.

3. Review process improvement

There are many accepted process capability determination models in the area of software engineering, including the Capability Maturity Model (CMM) [14], BOOTSTRAP [12] and SPICE [2]. In addition, there are also testing-tailored models such as the Testing Maturity Model (TMM) [1] and Test Process Improvement Model (TPI) [11]. From a review/inspection process improvement viewpoint none of the general or testing-tailored models is sufficiently oriented towrds inspection evaluation, although BOOTSTRAP and SPICE allow focused process assessment and improvement, i.e. the possibility to choose the process for improvement (e.g. review/inspection).

3.1 Capability evaluation in SME companies

We introduced a capability model tailored especially to review/inspection process evaluation that looks for weak points in a review/inspection through indicators and activities. The indicators can be enabling or verifying ones. The idea is that, if we find indicators of an activity, we then have justifications for its existence. If some activities are missing or at a low level, they should be set as targets for inspection process improvement. The improvement activities are usually based on discussions with company staff. We have also generated a preliminary set of improvement patterns by means of which companies can find improvement activities more easily.

Inspection is traditionally defined in terms of steps such as entry, planning, kick-off meeting, individual inspection, logging (inspection) meeting, edit, follow up, exit and release [3, 4]. The structure of the ideal inspection process, which we use as a reference model in evaluation, is based on these steps. Inspection activities are classified into three sets (cf. Figure 1): supporting activities (at the bottom), which help in carrying out an instance of the inspection process, the core set of activities (the middle bar), which are the essence of the inspection process implementation (defined in inspection books, e.g. [4]), and organisational activities, which ensure continuous improvement and efficient organisation of the inspection process.

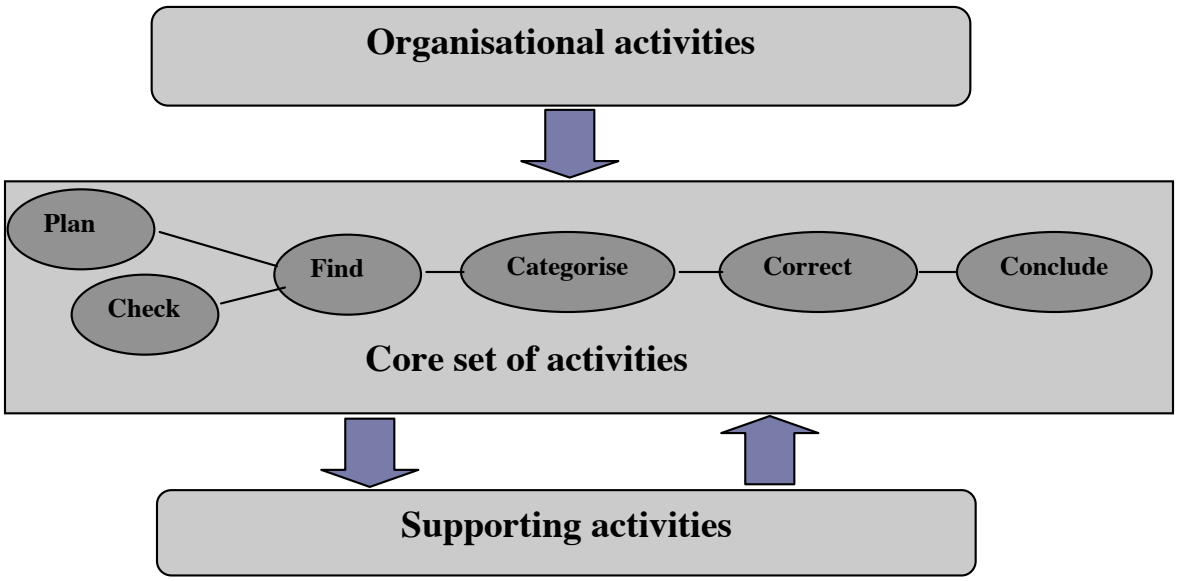
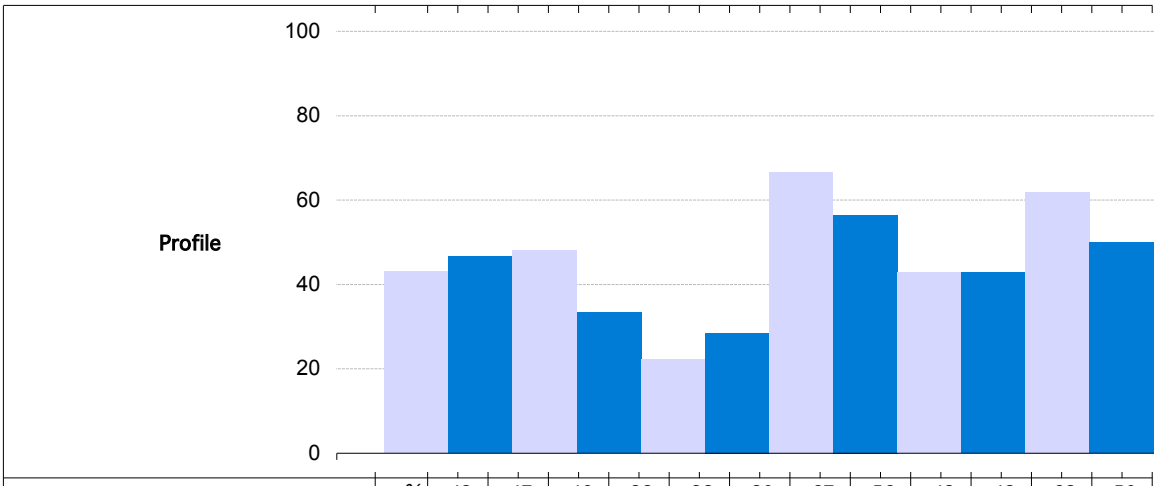


Figure 1. The i3GO capability model for software inspections

The supporting activities are “A.4 Support with computer tools", " A.5 Maintain rules and checklists" and "A.6 Refine information”. The core set of activities includes "A.7 **Check** the preconditions for inspection", " A.8 **Plan** the inspection", " A.9 **Find** issues in the artefact", "A.10 **Categorise** defects", "A.11 Make **correct**ions" and "A.12 **Conclude** the inspection", while the organisational set consists of the rest of the activities, such as “A.1 Establish and improve the inspection process “, "A.2 Organise the inspection" and “A.3 Train the participants". The A.N numbering refers to the numbers of the activities in Figure 2.

The result of a capability evaluation performed in an IT company is also presented in Figure 2. The diagram at the top of the matrix shows an estimate of each activity in graphical form, and the estimate is represented in numerical form at the bottom of the matrix. The company concerned has about 70 employees in Finland and our assessment revealed weak points in the inspection process at the whole company level. Capability determination in the i3GO model is based on the checking of 29 indicators. As we can see from the matrix (Figure 2), one indicator can affect a number of activities. An enabling indicator (E) confirms that the preconditions for a specific activity are met, and the absence of such an enabling indicator strongly suggests that the corresponding activity does not exist. A verifying indicator (V) confirms that a specific activity has produced appropriate and adequate results, and its existence suggests that the corresponding activity may exist, but does not guarantee this. The existence of an indicator is evaluated on a five-grade scale: na = not relevant to the organisation, 0 = not in use at all or only rarely, 1 = partially, exists to some degree, 2 = largely, exists fairly well, and 3 = fully, always in existence.

In the evaluation session we walk through all the indicators, grouping them according to core, supporting and organisational activities, the first 10 indicators applying mainly to the core set of activities, indicators 11-20 being related to all groups of activities and indicators 21-29 being mainly for organisational activities. This order of indicators is justified because it is easier to start the assessment session by evaluating core activities and to go on to supporting and finally organisational activities. In Figure 2 we can see the result of the evaluation in the column score and the multiplied values (score * indicator weight) in the cells. The weights of the indicators vary, an enabling indicator having the weight 1 and a verifying indicator the weight 2. Thus the indicator Entry decision, for example, has the value 4 (2*2) for activity 7 and the value 2 (2*1) for activity 9. The percentage denotes how well the activity is controlled in the company, calculated in terms of the sum value as a proportion of the maximum value, see activity A.12 (100 * 15/ 30 = 50 %)



	%	43	47	48	33	22	29	67	56	43	43	62	50
Indicator	***Score***	**A.1**	**A.2**	**A.3**	**A.4**	**A.5**	**A.6**	**A.7**	**A.8**	**A.9**	**A.10**	**A.11**	**A.12**
1) Entry decision	***2***							**V** 4		**E** 2			
2) Entry criteria	***1***	**V** 2						**E** 1					
3) Schedule	***3***								**V** 6	**E** 3			
4) Inspection resource allocations	***1***		**V** 2						**V** 2	**E** 1			
5) Request for rework on another artefact	***1***											**V** 2	
6) List of corrections	***3***						**E** 3					**V** 6	**E** 3
7) Acceptance of corrections	***1***												**V** 2
8) Request for recorrection	***0***												**V** 0
9) Exit decision	***3***												**V** 6
10) Exit criteria	***1***	**V** 2											**E** 1
11) Forms	***2***	**V** 4	**EV** 6		**EV** 6					**E** 2	**E** 2		
12) Tutorials	***2***	**V** 4		**E** 2							**E** 2		
13) Courses	***1***	**V** 2		**V** 2						**E** 1			
14) Collaborative tools	***1***		**V** 2		**V** 1				**E** 1	**E** 1			
15) Data gathering & anal. tools	***0***		**V** 0		**EV** 0	**E** 0	**E** 0			**V** 0	**E** 0		
16) Rules	***2***	**V** 4				**EV** 6			**E** 2	**E** 2	**E** 2	**E** 2	
17) Role descriptions	***1***	**V** 2		**E** 1		**E** 1			**EV** 3				
18) Checklists	***0***	**V** 0				**EV** 0			**E** 0	**E** 0			
19) Issue list	***3***						**E** 3			**V** 6	**E** 3	**E** 3	**E** 3
20) Categorisation	***0***					**E** 0	**E** 0				**V** 0	**E** 0	**E** 0
21) Pool of inspectors	***2***		**V** 4	**V** 4					**E** 2				
22) Selected inspection leader	***3***		**V** 6					**E** 3	**E** 3				
23) Request for training	***1***		**V** 2	**E** 1									
24) Quality resource allocations	***1***	**V** 2	**V** 2										
25) Reports	***0***	**E** 0	**E** 0	**E** 0	**V** 0	**E** 0	**V** 0						
26) Measured process data	***0***					**E** 0	**E** 0			**V** 0			
27) Improvement suggestions	***1***	**E** 1											
28) Process change requests	***1***	**V** 2	**E** 1			**E** 1							
29) Inspection process definition	***3***	**V** 6	**E** 3	**E** 3	**E** 3				**E** 3				
	sum	31	28	13	10	8	6	8	22	18	9	13	15
	max	72	60	27	30	36	21	12	39	42	21	21	30

Figure 2. Capability determination matrix.

3.2 Process improvement with patterns

After the assessment, each improvement has to be planned and carried out. Inspection improvement patterns are pre-defined guides and procedures for upgrading an inspection process. Seven patterns defined in our catalogue are listed in Table 1.

Table 1. Inspection improvement patterns

Pattern name	Main goal
Greed	Aims at finding more defects during inspections.
Early bird	Aims at finding defects at earlier stages in development.
Substance	Aims at finding more serious defects in inspections.
Comfort	Aims at making the inspection process easier to run.
Promotion	Aims at promoting the process so that it is carried out more often and in a larger number of projects.
Wisdom	Aims at a more understandable, transparent and effective inspection process.
Precision	Aims at making the process more rigorous, and thus more effective.

Each pattern defines an unambiguous goal and a list of actions for making the reviewing process more efficient. The purpose of the pattern catalogue is to aid the assessor and the company representatives in focusing their improvement activities on the most crucial parts of the process and to provide practical guidance for conducting the improvement. The patterns provided in this table address some of the most common deficiencies concerning inspections, and the list is not exhaustive. Additional patterns can be created as needed, or several patterns can be combined to achieve the desired improvement. The Greed pattern is presented as an example in the following:

Pattern name
Greed
Intention
The intention is to upgrade the inspection process in order to uncover more defects during inspections.
Symptoms
Symptoms that suggest using the Greed pattern include the following:

- Only a few errors are revealed in inspections.
- Only cosmetic errors are found in inspections.
- A lot of time is used to achieve poor results.

Problem
Even though inspections are carried out frequently, they can produce poor results. Despite visible investment in the method, its advantages remain unclear. Inspectors may consider the whole process as a waste of time, and eventually the management will think the same way if no observable gains are achieved.

A certain rigor is necessary to get the most out of the process. If this discipline is overlooked, or inspectors are not provided with adequate support for doing their job, inspections will probably reduce productivity and motivation. Reviewing is meaningful and efficient only if it reveals adequate numbers of defects, and if these defects are not trivial.

The reasons for poor performance may vary. First, the artefact to be inspected or material related to the artefact may be unavailable or hard to obtain for the inspectors or there may be shortcomings in the workflow of the process. Second, there should be an adequate number of inspectors to read through the document and they should be given enough time for performing the review. Third, the inspectors must have an up-to-date checklist and tools to support them in the

process. As illustrated in Figure 3, insufficient resources or support can dilute the outcome of an inspection.

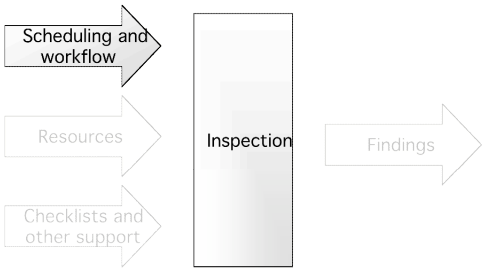


Figure 3. Inadequate resources and support produce poor results

Use of the Greed pattern should balance the process in order to get better results from the inspection.

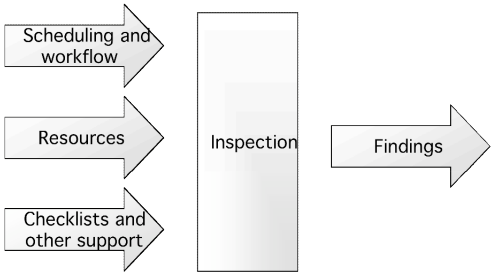


Figure 4. Improvements gained by use of the Greed pattern

Context

The Greed pattern should be used when the inspection process has recently been installed or skills in the use of inspections have diminished, and the process does not perform efficiently enough. Inspection tasks may be seen as extra work and the benefits as vague, or else inspections may be a normal procedure, but arranged and controlled carelessly by the QA group. If inspectors feel that their work is not supported or appreciated adequately, they will probably fail to discover real defects.

Solution

To approach the inspection process with the Greed pattern and upgrade it as follows:

1. Name a person or persons to be responsible for the inspection process. Such persons should pay special attention to the reaching of a common agreement (especially with the management) that inspections are planned, scheduled and resourced in every project. Consider adjusting the project plan document template so that inspections cannot be omitted.
2. Checklists have to be created or updated. Either the person responsible for the process has to take responsibility for maintaining the checklist, or else another person has to be named for the task. It is handy to derive one checklist from one document template – a requirements checklist from a requirements document template, for example. For code,

use the language-specific coding conventions and related guides to create initial checklists. If possible, use historical data (search through a bug database, interview experienced developers) to find typical errors and write them into the checklists.

3. Make artefact distribution more efficient. Study the feasibility of using e-mail or other software tools for this. Ensure that the inspectors really use the checklists and source documents. Observe their work and arrange training if feasible. Ensure that checklists and source documents are distributed along with the artefact, and that the inspectors know how to use them.
4. Consider establishing specialist roles – inspectors who focus on usability, interoperability or code reusability aspects, for example. Create a separate set of checklists for these specialist viewpoints. Train the participants if needed.

Consequences

Implementation of the actions listed in this pattern will affect activities A.5 Maintain rules and checklists, and A.9 Find issues in the artefact, in particular. Organisationally these improvements aim at making the inspection process more visible and establishing responsibility for it. The actions defined in this pattern will strengthen the very essential practices of the process, and may cause modifications to the company's quality handbook, if it does not already adequately address the planning and scheduling of inspection tasks.

4. Analysis of motivation factors and obstacles

The interview was conducted in spring 2004 in twelve software development departments varying in size between 160 and 5 persons. Seven of the companies had a quality group, and five of them maintained that it worked actively, regularly and visibly. Six of the companies possessed a quality handbook, and the interviewees were familiar with it. Among the other questions, we asked about motivation for reviews (Table 2) and obstacles to them (Table 3). The grade of importance is depicted in the tables as follows, VI = very important, I = important, R = rather important, S = some importance, N = not important.

Table 2. How companies are motivated towards reviews

Motivation	VI	I	R	S	N
Sharing expertise	1	7	1	2	1
Sharing information	2	4	5	1	0
Teaching novices	4	3	2	2	1
Controlling project progress	0	4	5	2	1
Process improvement	3	2	3	4	0
Finding more defects	8	3	0	1	0
Finding defects earlier	9	2	0	1	0
Finding defects faster	7	3	0	1	1

As highlighted in Table 2, the interviewees prioritised the "finding defects" tasks as the most important motivation issues, while "sharing expertise" and "sharing information" rose fairly high as motivation issues, and "teaching novices" and "controlling project progress" also received relatively high scores. "Process improvement" was evaluated as a very important source of motivation in three companies, but the majority rated it as rather important or of some importance.

We also studied the reasons why companies do not adopt reviews in their entirety (Table 3). "Shortage of time" and "shortage of staff" were the most important obstacles, and reasons such as "not required", "lack of expertise" and "laborious" were also mentioned.

Table 3. Reasons for avoiding reviews

Obstacles	VI	I	R	S	N
Shortage of time	8	4	0	0	0
Shortage of staff	4	5	1	1	1
Not required	0	2	2	1	7
Costs	0	0	1	3	8
Lack of expertise	0	1	3	0	8
Laborious	0	3	2	3	4
Geography	1	0	0	1	10
No need	0	0	1	1	10

It is also worthwhile noting that a significant number of companies did not see obstacles such as "not required", "costs", "geography" and "no need" as being of any importance.

5. Conclusions

This paper reports three approaches to stimulating the adoption of reviews by software companies. We started with simple tailoring cases, because companies were interested in light-weight processes and appropriate tools. Even though the functionality and ease of use of the tools had improved during the development cycle, it is still a challenging task to induce SME companies to really use them. It is more plausible that companies will attempt to use existing tools and tailor them to support reviews as well as possible.

We also looked for review improvements in six companies, in an analysis based on a capability evaluation matrix. A very rough generalized estimate of the results was that the review process in SME companies was typically at a rather low level, because they had recently started improvement of the process. We also learned that although the members of the quality group might be interested in review process improvement, this did not guarantee that they would install improvements in their projects. This is understandable, because project schedules are very tight and do not allow for process improvement experiments.

Our analysis of motivation factors and obstacles gives the same explanation for why SME companies do not use reviews. Although the interviewees understood that reviews would enable defects to be found earlier, faster and in greater numbers, shortage of time was the main reason which prevented them from using reviews.

In conclusion, we can say that research should proceed in the reverse order from ours. It should start with (1) analysing the review techniques used in companies (including motivation factors and obstacles), then (2) search for weak points in the review process as used in the company (based on capability analysis and process patterns), and finally (3) develop and tailor appropriate supporting tools. Our research will continue in that order. As a result of the analysis of review techniques, we will start process analysis in some companies and move on to improvement by means of existing process patterns. In the light of our earlier experience, we realise that the achieving of improvements through projects is a challenging task and requires support from the management, a quality group who are keen on improvements and the proper allocation of resources – but it is possible.

6. References

[1] Burnstein I., Practical Software Testing: a Process-Oriented Approach, Springer Verlag, 2003

[2] El Emam K., Drouin J. and Melo W. (eds), SPICE: The Theory and Practice of Software Process Improvement and Capability Determination, Wiley, 1997

[3] Fagan M.E., Design and Code Inspection to Reduce Errors in Program Development, IBM Systems Journal, vol 15, no 3, 1976, pp.182-211

[4] Gilb T., and Graham D.: Software Inspection, Addison-Wesley, Wokingham, England, 1993

[5] Group Review, The URL of the Group Review info is http://www.cmi.arizona.edu/collaboration/Group%20Review.html

[6] Harjumaa L., and Tervonen I., A WWW-based Tool for Software Inspection, Proceedings of the 31st HICSS Conference, vol III, IEEE Computer Society Press, Los Alamitos, CA, 1998, pp. 379-388

[7] Harjumaa L., Distributed Software Inspections – an Experiment with Adobe Acrobat, in Proceedings of the IASTED International Conference on Computer Science and Technology (CST 2003), 2003, pp. 26-31

[8] Hedberg H., Hajautetun ohjelmistotarkastuksen tukeminen XML-teknologioilla, Phil. Lic. Thesis, University of Oulu, 2003

[9] Johnson P.M., and Tjahjono D., Improving Software Quality through Computer Supported Collaborative Review, in De Michelis G., Simone C., and Schmidt K. (eds.), Proceedings of the Third European Conference on Computer Supported Cooperative Work, Kluwer Academic Publishers, Dortrecht, Netherlands, 1993, pp. 61-76

[10] Komi-Sirviö S., and Tihinen M., Great Challenges and Opportunities of Distributed Software Development – An Industrial Survey, Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering, 2003

[11] Koomen, T., and Pol M., Test Process Improvement: A practical step-by-step guide to structured testing, Addison-Wesley, 1999

[12] Kuvaja P. et al., Software Process Assessment and Improvement, The BOOTSTRAP Approach, Oxford, Blackwell, 1994

[13] Mashayekhi V., Drake J.M., Tsai W-T., and Riedl J., Distributed, Collaborative Software Inspection, IEEE Software, September 1993, pp. 66-75

[14] Paulk M. et al., The Capability Maturity Model: Guidelines for Improving the Software Process, Addison-Wesley, 1995

[15] Perpich J., Perry D., Porter A., Votta L., and Wade M., Anywhere, Anytime Code Inspections: Using the Web to Remove Inspection Bottlenecks in Large-Scale Software Development, Proceedings of the ICSE-1997 Conference, IEEE Computer Society Press, Los Alamitos, CA, 1997, pp. 14-21

[16] ReviewPro, The URL of the ReviewPro info is http://www.sdtcorp.com/reviewpro.html

[17] Tervonen I., Harjumaa L., and Iisakka J., The Virtual Logging Meeting: a web-based solution to resource problems in software inspection, Proceedings of the 6th European Conference on Software Quality, ADV Handelsgesellschaft m.b.H., 1999, pp. 342-351

[18] Tervonen I., Iisakka J., Harjumaa L., A Tailored Capability Model for Inspection Process Improvement, Proceedings of the Second Asia-Pacific Conference on Software Quality, 2001, pp. 275-282

Open Source Software Engineering Tools

Les Grove, Wells Mathews, Rodney Hickman, Kal Toth
les.grove@tek.com; wells.matthews@biotronik.com; rodney.g.hickman@intel.com;
ktoth@cecs.pdx.edu

Abstract

"Open Source" has been receiving a great deal of buzz in recent years primarily because of the evolution of Linux, Apache, MySQL, and other such free products. The question we address in this experience report is as follows: "Are there any open source software engineering tools mature enough for enterprise use?". We concluded that the answer is "yes".

As part of a recent project for the Oregon Masters of Software Engineering program, we found that there are several open source software engineering tools that can be used to support the software development life cycle. These vary widely as to maturity and quality - from projects that have not yet been "released" to those that have won awards and are of high commercial quality.

We examined several emerging software engineering support tools for requirements management, configuration management, problem tracking, change management, project management, design, integration, build, testing, and release. This paper presents these tools from the perspective of what is currently available and where these tools might be positioned in the very near future. This paper also explores a framework for open source software engineering tools and related tools ripe for open source development and evolution.

Evaluating any technical tool can be time-consuming and prone to subjective opinion. Assessment of open source tools requires additional evaluation criteria over those that might be applied to COTS tools. The evaluation method underlying the work presented herein has been successfully applied elsewhere by the open source community.

Biographies

Les Grove is a software engineer at Tektronix, Inc., in Beaverton, Oregon. Les has 19 years of experience in software development, testing, and process improvement. He has a Masters of Software Engineering degree from the University of Oregon and a Bachelors of Science degree from California Polytechnic State University, San Luis Obispo.

Wells Matthews is a Staff Software Engineer at Micro System Engineering, Inc. Wells has 26 years software development experience including scientific and business data processing, Electronic Design Automation/Test products, and implantable medical device products. Wells has a Masters of Software Engineering degree from the Oregon Graduate Institute and a Bachelors of Arts degree from Reed College.

Rodney Hickman is a software engineer at Intel Corporation, in Hillsboro, Oregon. Rodney has 14 years of experience in engineering and software quality assurance. He has a Bachelors of Science degree from William Penn University, Oskaloosa IA and is expecting to finish his Master of Software Engineering degree from Oregon State University this year.

Kal Toth is an Associate Professor in Computer Science at Portland State University and Associate Director for the Oregon Master of Software Engineering (OMSE). His areas of experience and research interests are in the fields of software engineering and information security. He has a Ph.D. in Computer Systems Engineering from Carleton University and is a P.Eng. (British Columbia) with a Software Engineering designation.

1. Introduction

This paper began as a research and analysis project to investigate the availability and quality of open source software engineering tools and to determine which of these, if any, could usefully support course offerings in the Oregon Master of Software Engineering program. We extended the scope of this R&D project to determine if open source tools could be successfully used in enterprise level[1] software engineering projects at our companies. This led to further questions including:

- What is Open Source?
- What open source software engineering tools are out there?
- What are the key dimensions of open source software and tools?
- How do you evaluate open source tools and the projects that support them?

Our objective for this investigation was to answer these questions with an eye on the "real world". We considered open source software engineering tools currently available and evaluated their maturity levels, capabilities and risks. A range of tools spanning the software development life cycle were selected, including requirements, design, module development, unit testing, integration/build, and configuration management. Finally, we wanted to give something back to the OMSE program and made many of these tools were made available for use in the classroom.

This paper is a summary of our findings, a more complete version (with screen shots and detailed evaluations) can be found at gforge.cs.pdx.edu/download.php/3/OMSE-556-OpenTools-Final-Report.pdf

2. Open Source Software Engineering Tools

We reviewed open source software engineering tools across the software development life cycle including configuration management, requirements management, design modeling, module development, unit test and integration/build. We also explored an emerging framework for integrating open source software engineering tools.

The initial appeal of open source tools may be that they are free. But the term "free" in the open source world has more to do with "freedom" than "no cost". There are indirect costs of using open source tools for software development.

First, there are administrative costs of making tools available on an ongoing basis. We experienced this trying to get the tools installed on the Portland State servers. Although commercial tools also need to be administered, the tool vendor shares some of this responsibility with the enterprise. In the context of open source administration, the enterprise assumes additional responsibilities to ensure that the most appropriate versions of various collaborating tools and applications from multiple sources work properly together. As time moves forward, it becomes the administrator's responsibility to ensure that all open source tools and applications stay up to date and compatible as new releases with potentially new interdependencies become available. This could become a weekly or even a daily task for administrators to keep in touch with all open source tools and versions through updates, news groups, and other tools used by the open source community.

Second, there may be long-term risks associated with adopting certain open source software engineering tools. Should a tool become unsupported or unsupportable, this is likely to negatively impact software maintenance and thereby incur down-stream effort and costs. This, in turn, may jeopardize a company's intellectual property or contract obligations. Generally, core life-cycle development tools (e.g. for design and compilation) do not present such risks as these tend to follow standards (e.g., UML[2], programming language standards). However, software engineering tools such as those used to manage software configurations, problems, changes and releases need to be able to support software products long after the project is complete. Once such support tools begin to adhere to interoperability frameworks and standards, this risk may be reduced.

3. Evaluating Open Source Project / Proxy Measures

Doing a thorough technical evaluation of any tool, especially an open source one, is a non-trivial effort. While exploring the open source landscape, we were fortunate to discover a proxy evaluation method employed by some members of this community. This method relies on reviewing postings made by both users and developers of these

---

[1] "Enterprise level" in the context of this paper refers to software engineering projects undertaken by our companies that are vital to core operations and/or critical product lines of the business.

[2] Unified Modeling Language

tools (note that in many cases, a developer is also a user). The following table summarizes the key dimensions (a.k.a. evaluation criteria) of this method.

Key Dimensions	Description
License	The type of license is important. Typically, the less restrictive the better. The availability of tool support is a factor. Your project might want the ability to pay for support rather than attempt to maintain the software.
User Community technical evaluation	What does the user community think of the software? This is generally a subjective assessment by participants of how well the evaluated tool supports its intended function.
Documentation	What documentation is available? Are there requirements specifications, design specifications or test results available? Release notes? What is the Quality of comments in the code?
Maturity	How well established is the open source project? Proxy measures include time invested/number of releases, user support/responsiveness, various stability of releases, release plan, test suite, commercial support available.
Longevity	How long will the open source project survive? Proxy measures include maturity evaluation (above), dates for releases, recent news postings, development team size, and development team diversity.
Flexibility	How well will the open source project respond to your project's changing needs? Proxy measures include: Available on multiple platforms? Is the software being used in the way it was intended? Are there timely responses in the online forums and mailings lists? Is there an active user community for this tool?

Based on these key dimensions, we rated several open source projects on the following subjective scale: Low, Medium, High, Commercial Grade and Commercial Grade+.

4. Software Engineering Tools Evaluated

This section summarizes our evaluations of software engineering tools that we found to be of particular relevance to the OMSE program and our companies. Each tool was selected based on its ability to represent a particular lifecycle function. The last subsection provides an overview of our exploration of an open source framework for software engineering tools called “OPHELIA”.

4.1 Configuration Management: CVS

For software configuration management, Concurrent Versioning System (CVS) [1] [21] is mentioned in just about every open source article we reviewed. One of the strengths of CVS is its ability to easily connect to the repository whether it is local on a user’s machine or on a distant server. CVS is based on the Unix "diff" and "patch" commands making it a very good tool for managing text (source code) based files. Although CVS can handle binary files, one weakness is that it cannot handle differences in Microsoft Word files as it treats it like any other binary file. Enterprise, commercial tools such as ClearCase can handle this. With large-scale software projects relying on electronic documentation for managing development and communication, this shortcoming is too large to ignore. This may not be an issue for smaller-scaled projects.

As an enterprise tool, CVS is not ready. Enterprise development needs more than simple, loosely integrated CM and bug tracking tools. Enterprise configuration management is becoming increasingly integrated with the software development process where changes are not allowed until specific requests for changes have been approved. This requires a tool that is fully integrated with problem tracking and change management processes that consistently follow a lifecycle process for all configuration items. Enterprise CM needs to provide traceability among all associated components and documents comprising the total project or product. For non-enterprise use, CVS is a great starting point for education and training programs, small projects, and startups. The prerequisite is that the people involved need to be aware of their roles and responsibilities when using the tool and collaborating with the community that supports it.

Some time in the future, CVS, or some similar tool, may provide this sort of enhanced functionality. However, by then, commercial enterprise configuration management tools may be that much further ahead of the curve.

4.2 Requirements Management: DRES

Distributed Requirements Engineering System (DRES) [23] is an open source requirements management tool. The following list summarizes the features available in DRES:

- Web-based forms data entry.
- The DRES requirements repository is implemented either on top of a MySQL database[3] or is integrated within the OPHELIA framework[4] (see Figure 1).
- Includes several forms for data entry and a search capability.
- Requirements are organized into a hierarchical structure of folders and naming convention to uniquely identify each requirement.
- Each leaf requirement is a separate database entry and is separately versioned and lockable.

The authors feel that DRES is useable, but is not yet ready for prime time. Our evaluation of DRES, which was based on posted open source project comments and assessments serving as proxies for maturity, longevity, and flexibility, came up with a "Low" rating.

DRES was primarily evaluated in the context of a graduate school practicum project specifying and managing software requirements. The first revision of the requirements specification, which was completed in an earlier academic term, was then imported into the DRES tool.

Our technical evaluation of tool capabilities and features yielded a “Medium” rating. Observe that this is better than one would expect by gathering postings by the open source community.

The features are a good start, but for enterprise use there is only light documentation and there are key feature, such as traceability and change histories, missing. Nevertheless we felt that the potential for DRES fitting into a software engineering tool framework is quite good. As an example, one of the requested additions to the tool - “Add UML use case diagrams and links to requirements”, potentially integrated with the OPHELIA compliant ArgoUML open source UML drawing tool, would be useful feature.

In 2003, Tektronix evaluated a number of requirements management tools. A set of tool requirements and a guideline that specified how requirements were to be managed were established. Using and internal evaluation procedure, DRES comes up short – especially in areas of traceability and integration with tools that are not compatible with the OPHELIA framework (this includes Microsoft Word). Tools such as DOORS, RTM Workshop, and Cradle met these needs much better. This was not to say that DRES would not catch up some day, but commercial tools are moving towards making requirements management part of a larger integration project support environment with product development lifecycles incorporating design, test automation, and configuration management processes and tools. So it does not seem to be fair to compare DRES with those tools, and these features need to be looked at more closely with within the context of software engineering tool frameworks (see section 4.9).

4.3 Design Modeling: ArgoUML

ArgoUML [17] is an open source UML drawing tool with a few additions. The following list summarizes the features available in ArgoUML:

- GUI to draw UML diagrams. 8 out of 9 diagrams supported according to the web site (however, we did not get all 8 working, probably due to some pilot error on our part).
- Reverse engineering from Java code.
- Code Generation to Java and others with plug-ins.
- Outputs own internal files and XML[5] files.

---

[3] MySQL is an open source database tool that makes use of standard SQL for database access.

[4] OPHELIA, an emerging open source framework for integrating open source software engineering tools, is being jointly developed by a consortium of academic and industry partners (www.opheliadev.org).

[5] XML: eXtensible Markup Language

- Some basic graphics allowed.
- Imports from own internal files, and XML files.
- Can export UML diagrams in a variety of graphics formats.
- Critique/advisor function.
- Fits into the OPHEILA framework.

The authors recommend ArgoUML for student and industry use. Our evaluation drawn from postings resulted in a "High" rating with respect to maturity, longevity, and flexibility. When ArgoUML was evaluated with respect to drawing UML architecture diagrams the rating was "Med-High" to "High".

4.4 Module Development: Mono

Mono [35] is an open source .NET[7] framework project that enables the .NET framework specification to be platform independent. Mono uses a runtime engine to execute. NET framework libraries. The Mono implementation has the availability for adding plug-ins for development of KDE[8] and GNOME[9]. It appears that GUI interfaces will not be OS independent while non-GUI applications are.

A rudimentary demonstration of the alignment of mono class specifications with MS .NET Framework specifications was completed by using a simple HelloWorld program compiled on Linux using the Mono built in C# compiler MCS from command line, and then run using the mono runtime engine on the Linux environment.

The helloworld.exe was copied to a Windows server with MS .NET Framework 1.1 redistributable runtime libraries installed. When executing helloworld.exe from the command prompt the same 'Hello World!' was displayed.

A slightly better test was then completed by compiling the bank example for NUnit described in section using the MCS compiler then tested on the windows server.

4.5 Unit Testing: NUnit

NUnit [39] is an open source framework for development of automated unit testing in a .NET environment. The unit tests can be developed in a set of libraries utilizing the common runtime .NET specifications. NUnit is capable of being run through a GUI, command line, or launched from a build tool such as NAnt.

Test cases are written using the concept of test fixtures with attributes and assertions.

NUnit was tested by using the quick start guide, installed with NUnit, to generate a C# version of a sample bank.dll application with no test fixtures. Two files containing the test fixtures for testing the bank.dll were then created. The three files were compiled on the Mono built in C# compiler; the tests were executed on the windows server where the MS .Net Framework runtime libraries were installed.

The GUI interface for NUnit was launched and the banktest*.dll files, generated above, were used for test execution. Testing was launched by clicking the 'Run' button on the GUI to execute the test cases. The output was subsequently generated and available on the GUI. The GUI will provide test status using a color coding for each test and test set state:

- Gray: Not run.
- Yellow: No test to run.
- Red: Failed test.
- Green: Test passed with status of success.

4.6 Build Management: NAnt

NAnt [38] is a build tool for .NET frameworks. The build is generated based on a configurable build file. This build file is basically an XML build script with the flexibility to accept parameters to define what should be built, the individual source file dependencies, and pre and post build processing. For example the build file can launch

---

[6] XML: eXtensible Markup Language

[7] .NET: Microsoft's solution for Web services. It enables the creation and use of XML and SOA -based applications, processes, and websites as services. [33]

[8] K Desktop Environment, a graphical desktop for Unix and Linux. [29]

[9] GNU Network Object Model Environment, a Windows-like desktop for Unix and Linux.

other activities such as NUnit testing depending on a successful compile of all code. This is to say that the build script will execute the compile block first then if successful will launch the NUnit block of XML instructions.

This was demonstrated by using NAnt to generate the bank.dll and unit test .dll files to test bank.dll, described in section 4.5.

The XML build script is also flexible enough to allow the definition of: compiler, references, manipulation of files upon a successful build and test. The configuration was changed to build the test project using Microsoft's .NET compiler and framework, without changing any source code. By successfully compiling and running unit test and bank source code using both compilers further demonstrated the alignment of Mono and MS .Net specifications.

4.7 Integration Management: CruiseControl

Continuous integration [19] is a concept of fully automating the build, test, and release processes. When referring to the continuous integration, the following processes are completed automatically with no human interaction when file changes are committed to a project:

- Detection of code changes committed into source control.
- Launch of project build.
- Execution of unit testing.
- Results reporting of changed modules build results and test results.

This concept takes the build process from typical change duration as discussed in many shops and migrating it to complement an agile or XP type of programming methodology. This is not to say that these methodologies must be adopted in the strictest sense.

The use of continuous integration allows one to know what objects caused a build to break, enabling troubleshooting to be simplified over a big bang integration build. The use of continuous integration improves the information turnaround time of issues with a build. A developer does not have to wait until the next day to see if the code broke the build then the following day to see if the build is repaired. Rather the build results appear in a more real time basis and can be corrected in a timely manner.

CruiseControl [21] is the final layer to the continuous integration development system for the Mono, Nunit, and Nant exercise. CruiseControl includes a server process that will monitor the source control project and is configured to react to changed files when they are committed to source control. When file changes are detected the system will execute based on predetermined steps that are configured in the CruiseControl configuration file.

CruiseControl also has an ASP.NET web site utility that can be setup for hosting the results summarization. The web site can be used to summarize the resulting files changed in CVS since the last build. The site also reports the results provided by the NAnt build process. The web site will summarize the results from NUnit including test successes, messages for failures and durations for test execution.

The following list summarizes CruiseControl configuration file contents:

- Web site for hosting results.
- Frequency to check source control for committed changes since last build.
- Source control configuration.
- Response configuration (what to do if changes are discovered).
- Results configuration (what to do with results).

4.8 Project Collaboration: GForge

GForge [26] is a web-based project collaboration tool used primarily for open source projects; similar to sourceforge.net [44] and freshmeat.net [25]. GForge is installed on a web server and administered locally where users register and can join projects or create them for others to join. GForge supports keeping track of assigned items and tasks, monitoring forums, viewing projects, source code, and project openings. All public projects are available for viewing, download, and comment. Within a GForge project, defects can be tracked, activities can be managed and monitored, files and documentation can be posted, configurations can be managed, and information can be shared.

Tektronix has been using something similar for internal projects for about three years. Primarily, it has been used for smaller projects with fewer than ten people, and has not been used extensively for larger, enterprise-wide projects. Essentially, it alleviates the need to create an internal project web site.

4.9 Open Framework for Software Engineering Tools: OPHELIA

We discovered an emerging open source software engineering framework in the course of our investigations called OPHELIA[10] [6] [42], an emerging open source framework for integrating open source and proprietary software engineering tools. A consortium of academic and industry partners, sponsored by the European Union, is conducting research aimed at producing an integrating framework supporting the software development lifecycle. OPHELIA provides a mechanism for distributed tool integration through a set of CORBA interfaces. These interfaces describe what type of data a given tool requires and how this data can be accessed.

OPHELIA Architecture
(adapted from figure 1 in [6])

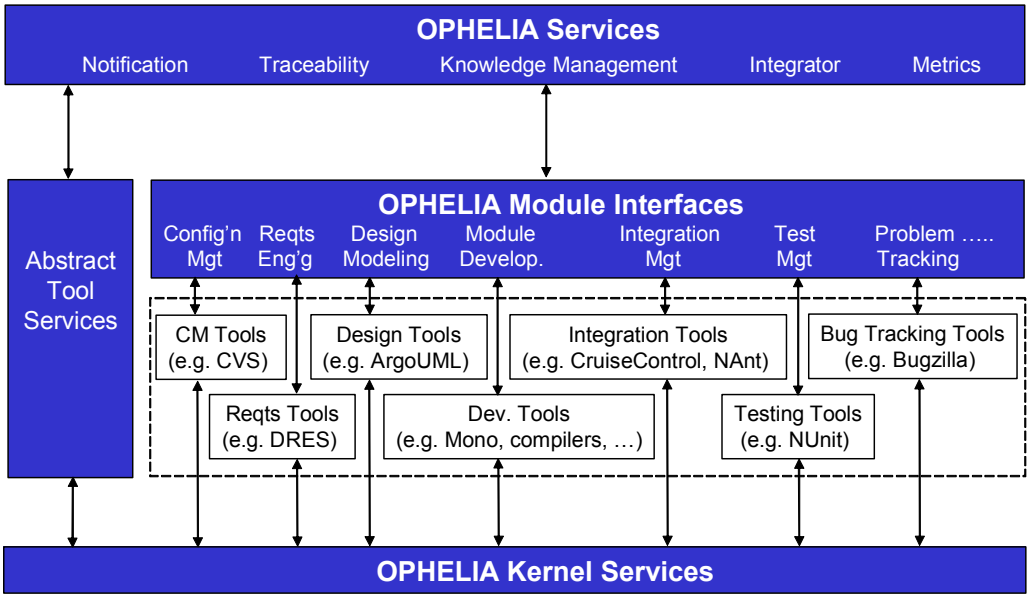


Figure 1

Orpheus is a particular instance of the OPHELIA framework that is being developed to validate the OPHELIA framework. A current Orpheus solution contains DRES, ArgoUML, Bugzilla, CVS and several other software engineering tools. DRES in particular fits with the OPHELIA framework, in that one of the supported back-ends is the OPHELIA requirements module. We did not evaluate Orpheus, or DRES within the context of the Orpheus solution.

5. Conclusions

There are several key dimensions of open source projects. First, developers are users, as most successful open source products are computer science type projects "by developers for developers. Characteristics of notable successes include:

- Single unfragmented solution
- Widely used
- Technically sophisticated
- Large number of stakeholders (e.g., programmers, build managers) depending on it

[10] Open Platform and metHodologies for devELopment tools IntegrAtion

Second, there is a wide range of quality out there, which is not surprising given the large open source project space. Finally, we found that the "top rated" open source products have quality as good as or better than "commercial grade" products.

Using proxy evaluations of open source projects can be fairly accurate and quick, certainly compared to more formal evaluation methods. The most time consuming aspect of evaluating open source tools is the search itself, as finding candidates can be daunting if starting from scratch (a topic not covered in this paper). We actually found one of the more time consuming tasks to be the investigations of related tools that temped us to broaden our efforts.

SW Engineering with open source tools has arrived, maybe. There are several indications that software engineering with open source has approached mainstream use. Indicators include the articles and books widely available, industry awards for open source projects, the creation of the Open Source Initiative (OSI) non-profit organization, and most importantly, the number of widely used high quality open source projects (such as Linux and the GNU compilers).

Proxy Evaluation Summary

Category	Tool	License	Users Technical Evaluations	Documentation	Maturity Proxy Evals[1]	Longevity Proxy Evals[1]	Flexibility Proxy Evals[1]	Overall Proxy Evals[1]
Configuration Management	CVS	High	High- CG	CG	High-CG	High-CG	Med	High-CG
Requirements	DRES	?	Low-N/A	Med-High	Low	Low	Low	Low
Design	ArgoUML	High	High	High	High	High	High	High
IDE	Mono	High	Low-Med[4]	CG	Med	Med-High	Med-High	Med
IDE	Eclipse	High	CG+	CG+	High-CG	High-CG	CG+	CG
Build	NAnt	High	High	High-CG	High	Med-High	Med-High	High
Unit Test	NUnit	High	High	High-CG	High-CG	Med-High	Med-High	High
Continuous Integration	Cruise Control	High	Med-High	High	Med	Med	High	Med-High
Relational DBMS	MySQL[2,3]	High	CG+	CG+	CG	High	High	CG+
Office SW	OpenOffice[2]	CG	CG	CG+	CG-CG+	CG-CG+	CG+	CG-CG+
Web Browser	Mozilla[2]	CG	CG+	High-CG	CG-CG+	CG-CG+	High-CG	CG-CG+

1. Proxy Evaluation Rating: Low, Med, High, Commercial Grade (CG), Commercial Grade+ (CG+).
2. Bundled with Linux.
3. MySQL part of the DRES installation.
4. When project gets to a higher maturity, we expect the User Technical Eval to go up to High

All proxy evaluations are posted on the GForge web site: http://gforge.cs.pdx.edu/projects/open-src

6. Software Evolution in the Classroom

Part of the motivation for investigating open source software engineering tools has been to establish a software engineering tool set for OMSE faculty and students to use thereby enabling them to practice "Software Evolution in the Classroom" on real-word problems.

The Oregon Master of Engineering (OMSE) program has been designed for experienced software professionals interested in attaining an advanced software engineering degree. Students are introduced to a range of software engineering practices. Individual students often work in small teams applying best practices and tools to complete their assignments. Faculty reinforces software engineering practices through the use and demonstration of supporting tools.

Due to limitations of time, software engineering students are typically given fairly basic ("toy") problems to solve with the intent of enhancing their understanding of the principles and processes they learn in class. Students rarely get the opportunity to apply industry practices and tools for evolving software components in a real-world project or product setting.

To address this problem, OMSE students conducting their practicum course work in small teams to evolve and augment their open source software engineering environment. These same tools are used by faculty and students to support classroom instruction and to complete assignments. As illustrated in Figure 2, this approach yields an ongoing software improvement process that provides students with interesting opportunities to evolve their development environment while improving SE tools for software engineering classes. This approach will increase the likelihood that students will achieve positive results when applying SE practices learned in class to problems encountered on the job.

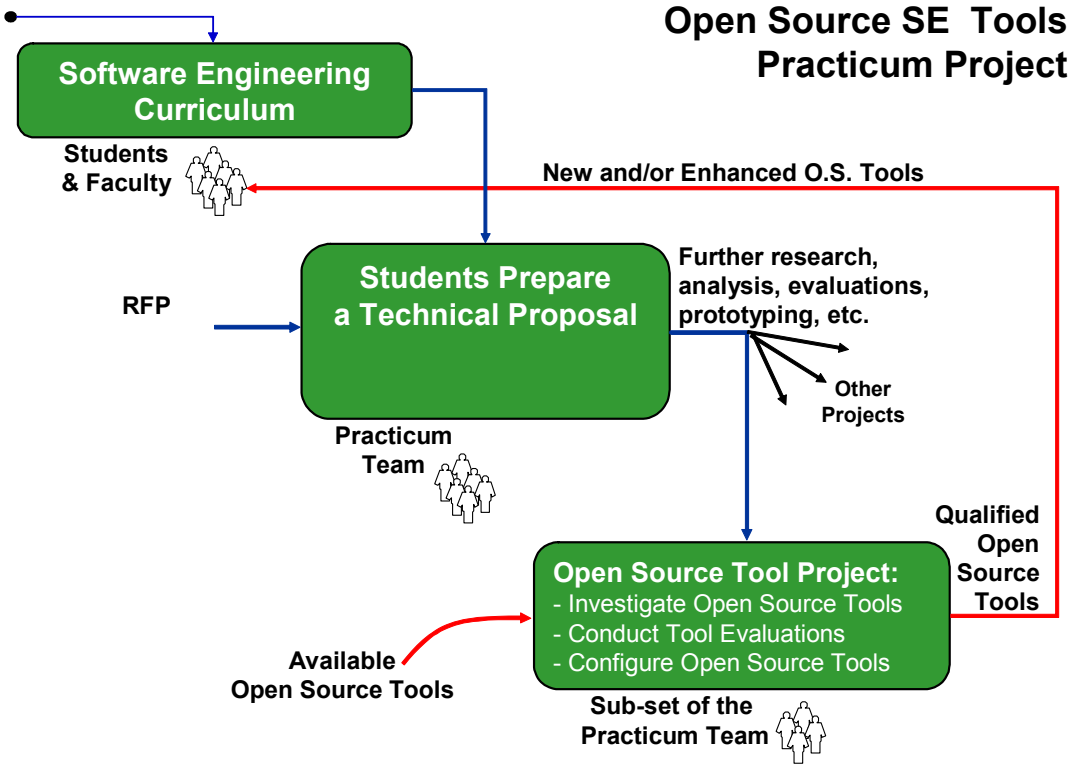


Figure 2

7. References

7.1 Articles

[1] Bar, M., Fogel, K., Open Source Development with CVS, 3rd Edition, Paraglyph Press, 2003.
[2] Chromatic, Myths Open Source Developers Tell Ourselves, O'Reilly OnLamp, www.onlamp.com/pub/a/onlamp/2003/12/11/myths.html, 2003
[3] Fellor J, Fitzgerald B., Raymond, E., Understanding Open Source Software Development, Addison-Wesley, 2001, ISBN 0201734966
[4] Fitzgerald, B., Kenny, T., Developing an Information Systems Infrastructure with Open Source Software, IEEE Software, Jan/Feb 2004.
[5] Gacek, C., Arief, B., The Many Meanings of Open Source, IEEE Software, Jan/Feb 2004.
[6] Hapke, Maciej et.al., OPHELIA, Open Platform for Distributed Software Development, Poznan University of Technology, Institute of Computing Science, Poznan, Poland (see [42] below)
[7] Kshetri, N., Economics of Linux Adoption in Developing Countries, IEEE Software, Jan/Feb 2004.
[8] Lussier, S., New Tricks: How Open Source Changed the Way My Team Works, IEEE Software, Jan/Feb 2004.
[9] Messerschmitt, Counterpoint, Back to the User, IEEE Software January/February 2004, pgs 88-91
[10] Norris, J., Mission-Critical Development with Open Source Software, IEEE Software, Jan/Feb 2004.
[11] Raymond, E., Point, Up from Alchemy, IEEE Software January/February 2004, pgs 88-91
[12] Ruffin, M., Ebert, C., Using Open Source Software in Product Development, IEEE Software, Jan/Feb 2004.
[13] Scacchi, W., Free and Open Source Development Practices in the Game Community, IEEE Software, Jan/Feb 2004.
[14] Serrano, N., et. al., From Proprietary to Open Source Tools in Information Systems Development, IEEE Software, Jan/Feb 2004.
[15] Spinellis, D., Szyperski, C., How Is Open Source Affecting Software Development, IEEE Software, Jan/Feb 2004.

7.2 Web Sites

[16] Apache, httpd.apache.org
[17] ArgoUML, argouml.tigris.org
[18] C# Refactoring Project (Opnieuw), sourceforge.net/projects/opnieuw
[19] Continuous Integration, www.martinfowler.com/articles/continuousIntegration.htm
[20] CPPRefactory Project, sourceforge.net/projects/cpptool
[21] CruiseControl, cruisecontrol.sourceforge.net
[22] CVS, www.cvshome.com
[23] DRES, ophelia.cs.put.poznan.pl/dres
[24] Eclipse, www.eclipse.org
[25] Freshmeat, another open source development web site, www.freshmeat.net
[26] GForge, gforge.org
[27] GForge (PSU), gforge.cs.pdx.edu
[28] GNOME, www.gnome.org
[29] KDE, www.kde.org
[30] "Lamp, The Open Source Web Platform", O'Reilly books, www.onlamp.com
[31] "The Literate Programmer", Donald Knuth. www.literateprogramming.com
[32] Microsoft Cassini Webserver project, www.asp.net/Projects/cassini/download
[33] Microsoft .NET, www.microsoft.com/net
[34] MIT open source "community", opensource.mit.edu
[35] Mono, www.go-mono.com
[36] Mozilla, www.mozilla.org
[37] MySQL, www.mysql.com
[38] NAnt, nant.sourceforge.net
[39] NUnit, www.nunit.org
[40] Open Source Initiative (OSI) non-profit corporation Website, www.opensource.org
[41] OpenOffice, www.openoffice.org
[42] OPHELIA Website, www.opheliadev.org
[43] Slashdot, discussion site on all things under open source", slashdot.org
[44] SourceForge, large open source development, sourceforge.net
[45] Tigris, a major open source software engineering tool site, www.tigris.org

Inherent Risks in Object-Oriented Development

Dr. Peter Hantos
The Aerospace Corporation
P.O. Box 92957 – M1/112
El Segundo, California 90009-2957
Peter.Hantos@aero.org

AUTHOR'S BIOGRAPHY

Peter Hantos is currently a Senior Engineering Specialist in the Software Acquisition and Process Office of the Software Engineering Subdivision at The Aerospace Corporation. He has over 25 years of experience as a professor, researcher, software engineer and manager. He has authored numerous technical papers, U.S. and international conference presentations.

Prior to joining Aerospace, he held various positions at Xerox Corporation. As Principal Scientist at the Xerox Corporate Engineering Center, he developed corporate-wide processes covering the full product development lifecycle for software-intensive systems. Other highlights of his Xerox career include the creation and management of a software technology group to facilitate the technology transfer and productization of software prototypes developed at the Xerox Palo Alto Research Center. Later, as Department Manager, he directed all aspects of hardware/software quality and test for several printer product lines.

Dr. Hantos started his career in the U.S. as Visiting Professor at the Computer Science Department of the University of California, Santa Barbara. He is a Member of ACM, Senior Member of the IEEE, and holds M.S. and Ph.D. degrees in E.E. from the Budapest Institute of Technology, Hungary.

ABSTRACT

The main challenges in determining inherent risk sources are the comprehension of the dynamically changing nature of object-oriented technology, and the distinction of general software development risks from the ones that are specific to object orientation. The key to meeting these challenges is the use of established, well-proven frameworks to inventory the essential attributes of object-oriented technology and project risks. This paper presents a systematic process, mapping Bertrand Meyer's key object-oriented technology concepts into Barry Boehm's top ten software risks to identify inherent object-oriented technology risks. Bertrand Meyer in his referenced book provides an excellent overview of the fundamental object-oriented concepts, and it is easy to show that recent developments in object-oriented technology, such as Java, Use Cases, or UML also fit well into his framework. Barry Boehm in his well-known risk identification checklist summarizes the top ten, "methodology-neutral" software risk sources, based on surveying experienced project managers. The systematic approach presented in this paper allows project managers to more completely understand the cost/benefit aspects of migrating to object-oriented technology, and align their project management strategies better with the organization's business goals.

© 2004 The Aerospace Corporation
THE AEROSPACE CORPORATION

1. Introduction

In this paper the term object-oriented technology (OO) refers to object-oriented development processes and methods, object related standards, and associated products and tools from third party vendors. Object orientation is a software engineering technique that has been around since the late 1970s. However, during the 90s it has become more pervasive, and it is fair to say that currently most new software projects use OO to various extents. Companies developing software are looking to OO as a means to achieve their strategic business objectives. Their expectations are that OO will enable them to build complex systems of superior quality with reduced development time and costs, while providing long-term benefits such as maintainability, reusability, and extensibility. Although OO provides many opportunities, projects adopting it have inherent risks that must be actively managed and mitigated to avoid failure.

Why is it necessary to explore OO-specific risks in software development? Aren't the same risks associated with the introduction of any new technology? With respect to paradigm scope, complexity and depth, OO has far reaching consequences, and for the project manager the decision is not simply whether or not to introduce OO on a particular project. The use of OO permeates all aspects of development and, based on business priorities, managers also have to determine whether the replacement of legacy languages and tools is justified, and what the optimal insertion order and desired penetration of these new OO concepts should be.

2. Object-Oriented Technology

Bertrand Meyer in his 1995 book [1] provides a sound overview of OO fundamentals. According to Meyer, software construction embracing OO is structured around the following concepts[1]:

- M1 - A new way to define architecture and data structure instances
- M2 - Information hiding through abstraction and encapsulation
- M3 - Inheritance to organize related elements
- M4 - Polymorphism to perform operations that can automatically adapt to the type of structure they operate on
- M5 - Specialized analysis and design methods
- M6 - OO Languages
- M7 - Environments that facilitate the creation of OO systems
- M8 - "Design by Contract", a powerful technique to circumvent module boundary and interface problems
- M9 - Memory management that can automatically reclaim unused memory
- M10 - Distributed objects to facilitate the creation of powerful distributed systems
- M11 - Object databases to move beyond the data-type limitations of relational database management systems

Please note that this paper is not intended to be a tutorial on OO; rather, it will examine risk implications associated with all of these concepts. The reader is assumed to be familiar with the basics.

3. Risk Management

Risk Management is acknowledged as a critical process of project management, and has received more and more attention since the 80s. For example, in case of the SEI-developed[2] process improvement framework, during the transition from the SW-CMM®[3] to CMMI®[4], risk management was elevated from a recommended practice to a formal, independent Process Area. Nevertheless, to accommodate a broader audience, the definitions used in the following discussion are based on IEEE Std 1540-2001 and not CMMI material.

Risk is defined as a potential problem, an event, hazard, threat, or situation with undesirable consequences. The non-deterministic nature of risk makes risk management a special challenge for the project manager. During the planning of a project we might be tempted to trying to avoid risks all together, but relying strictly on avoidance as a risk mitigation technique is usually not adequate. The success of a project primarily depends on the project manager's ability to manage the delicate balance between opportunities and risks. Unfortunately, when all risk goes away, so

---

[1] The M1-M11 numbering of concepts did not originate from Meyer; it was introduced to help the mapping process

[2] SEI: Software Engineering Institute, a Federally Funded Research and Development Organization at Carnegie Mellon University, Pittsburgh, PA.

[3] SW-CMM: Software Capability Maturity Model.

[4] CMMI: Capability Maturity Model Integration. CMM and CMMI are registered in the US Patent and Trademark Office by Carnegie Mellon University.

does opportunity. That's why successful project management practices include risk management, a continuous process for systematically addressing risk throughout the life cycle of a product or service.

The risk management process consists of the following activities [2]:
a) Plan and implement risk management
b) Manage the project risk profile[5]
c) Perform risk analysis
d) Perform risk monitoring
e) Perform risk treatment
f) Evaluate risk management process

The focus of this paper is risk identification, a critical aspect of risk analysis. Risk identification, similar to all other elements of continuous risk management, is not a one-time activity. Changes in the risk management context and changing management assumptions represent major risk sources, and need to be continuously monitored as well. IEEE Std 1540-2001 does not prescribe how risks should be identified, but suggests numerous methods, including the use of risk questionnaires or brainstorming. A specialized example of a risk questionnaire, to be used in a J2EE[6] environment, is presented in [3]. Most risk questionnaires are the result of some sort of brainstorming effort; in most cases, the authors interview experienced project managers about their past projects and, after some filtering and processing, they turn the structured risk statements into questions or checklists. For an example of a systematic approach to develop a checklist, see Tony Moynihan's article [4]. Barry Boehm published his Top Ten Software Risks first in 1989 [5], and presented the updated list in his 1995 Software Engineering course with surprisingly few modifications that were based on feedback from the USC/CSE[7] Industrial Affiliate companies (For a published version of the second list please see [6].) Essentially all items, although sometimes named slightly differently, still represented major risk sources, and the name changes can be attributed to changes in popular terminology and not fundamental root causes.

4. Identifying OO Risks

4.1 Consolidating Boehm's Risk Sources

The consolidation process and its result are shown in Figure 1. First, items on the 1989 list were crosschecked with the 1995 one. Item #5, "Gold-plating" from the 1989 list, is clearly a requirements mismatch issue[8]. Finally, on the 1995 list, for the sake of brevity requirements mismatch has been also combined with user interface mismatch, and COTS issues with legacy software issues since they have many similarities with respect to root causes.

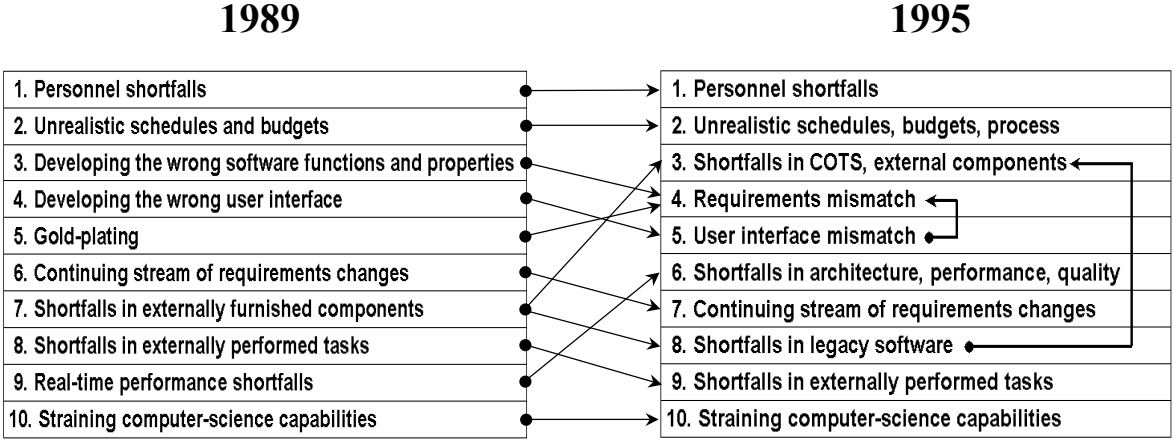


Figure 1. Consolidating Boehm's Top Ten Software Risk List

---

[5] Risk profile: A chronological record of a risk's current and historical state information [2].
[6] J2EE: Java 2 Enterprise Edition.
[7] USC/CSE: University of Southern California/Center for Software Engineering.
[8] Gold-plating is a popular term in software management for implementing features by software engineers that go beyond the scope of actual requirements.

4.2 Mapping and Interpreting Meyer's OO Concepts

The main challenges in determining inherent risk sources are the comprehension of the dynamically changing nature of OO technology, and the distinction of general software development risks from the OO specific ones. The key to meeting these challenges is the use of well-proven frameworks to inventory the essential attributes of OO technology and project risks. Boehm's risk identification checklist was chosen because it is well accepted in the software engineering community. During the mapping process we examined Boehm's consolidated risk list item by item and identified the corresponding OO concepts. The results of this mapping are shown in Figure 2. Due to the consolidation process described in the previous paragraph, Figure 2 shows only 8 risk items. Please note that risks B6 and B7 are caused by customer and contractor behavior, and their successful mitigation does not require any OO-specific considerations.

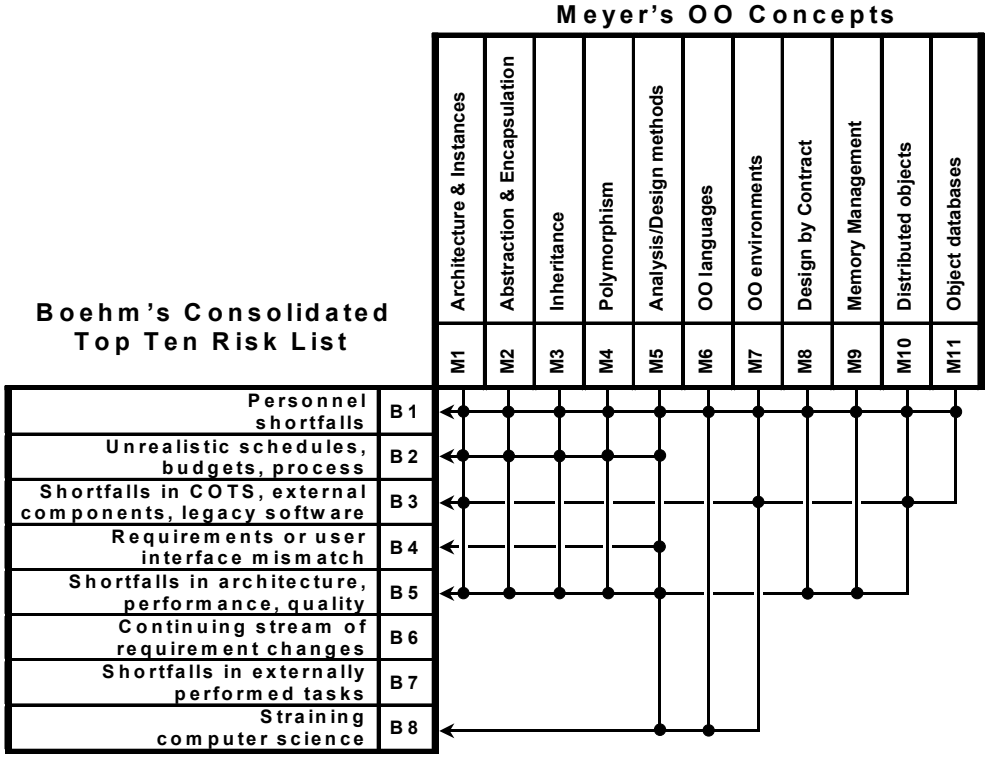


Figure 2. Mapping Meyer's OO Concepts into Boehm's Consolidated Risk List

4.2.1 Personnel Shortfalls (Risk B1)

Software development is a highly labor-intensive process, and its success depends primarily on the people in the organization. Beyond well-known organizational and political issues, several OO-specific concerns need to be explored. The most significant concerns are specialized skills and experience, and that is why all OO concepts are connected to this risk item as shown in Figure 2.

The first issue is the appropriate balance between application domain knowledge and OO knowledge. It is difficult to find people who are skilled in both; hence the collaboration between project personnel with different skill bases is critical. The second issue is the number and distribution of available people. OO knowledge is relevant for all members of the organization, although not to the same extent. In all people-categories, such as managers, architects, developers, and testers, it is important that all personnel have or acquire via training the necessary OO skills. For example, to avoid personnel shortfalls, the executives themselves who create, manage or sponsor the development organization have to understand the essential elements of OO even before staffing starts for the project. While having prior OO experience is an asset for managers, the minimum requirement should be to have a certain level of OO literacy. In fact, Meyer's book, which is used in this analysis, is an excellent tool for this purpose, i.e., educating

managers in OO[9]. The seeding of all teams with OO mentors is also a good approach to distribute OO domain-knowledge and to both jump-start and to facilitate OO development. Not surprisingly, most other sources that have analyzed OO migration have focused on the human dimension as well. Two of the three key items discussed in [3] deal with learning curve and training, and [3] contains further references to other authors addressing the same concern [7], [8].

Personnel issues play an important role in the team context as well. OO requires a new way of thinking and moving away from outdated approaches like using functional decomposition for architecting systems or implementing obsolete programming constructs. For teams with a long heritage of using legacy approaches the paradigm shift is particularly difficult. In fact sometimes we have observed a quiet, "passive resistance" to OO methods, where the people attempted to fake the usage of new methods but at the same time they were continuing "business as usual". A good example for this anomaly is writing "C"-like programs with the use of a "C++" compiler.

4.2.2 Unrealistic Schedules, Budgets, and Process (Risk B2)

Unrealistic expectations, lack of management appreciation for the necessary skills, and the difficulty of the paradigm shift will lead to unrealistic schedules. Similarly, the underestimation of time and cost of necessary training would result in unrealistic schedules and budget. Nevertheless, some key OO items specifically contribute to this problem. Based on E. Flanagan's summary [9], most of the time OO projects are introduced on the following grounds:

- OO is better at organizing inherent complexity, and abstract data types make it easier to model the application. (These statements are building on Concept M1, labeled "Architecture and Instances.")
- OO systems are more resilient to change due to encapsulation and data hiding (per Concept M2).
- OO design often results in smaller systems because of reuse, resulting in overall effort savings. This higher level of reuse in OO systems is attributed to the inheritance feature (per Concept M3).
- It is easier to evolve OO systems over time because of polymorphism (per Concept M4).

However, we can also learn from [5] that, particularly when OO is introduced for the first time, expectations might be exaggerated, and frequently the impact of potential costs and risks are intentionally minimized to claim maximized payback. For example, it might not be made clear to the sponsoring executives that it would take several years for just the previously mentioned four benefits of OO to be fully realized.

One of the side effects of the OO approach is that the design process becomes more important than it was in non-OO projects. Due to encapsulation, data hiding, and reuse, the design complexity moves out of the code space into the design space. The increased design complexity has testing consequences as well. Even if incremental integration is applied, more sophisticated integration test suites need to be created to test systems with potentially large number of highly coupled objects.

It is also an unfortunate fact that while the OO concepts identified make system comprehension easier during analysis and design, they cause testing and debugging to become more difficult, since now all debugging methodologies and tools have to work with those abstract data types and instances. Those organizations that assume that testing OO is like testing any other software are in for a big surprise. R. Binder makes a powerful case for this argument in his article [10]. According to Binder, it is a common myth that only Black Box[10] testing is needed and OO implementation specifics are unimportant. In reality, OO code structure matters, because inheritance, encapsulation and polymorphism present opportunities for errors that do not exist in conventional languages. Also, OO has led to new points of view and representations, and the test design techniques that extract test cases from these representations must also reflect the paradigm change.

4.2.3 Shortfalls in COTS[11], External Components, and Legacy Software (Risk B3)

The use of COTS and other, externally developed or legacy components in OO presents particular difficulties for structural comprehension and architectural design. These external components, their architecture, interfaces, and documentation are not necessarily consistent with the class and object architecture, communication mechanisms, and

---

[9] Consider the book's subtitle: "A manager's guide to object orientation, its impact on the corporation and its use for reengineering the software process".

[10] Black Box testing targets externally observable behavior that is produced from a given input, without using any implementation information.

[11] COTS: Commercial Off-The-Shelf.

view models of the system being developed. A particular OO problem in this area is the interface of Object Database implementations with traditional Relational Database Management Systems. The problem may deepen in situations where multiple new technologies merge, as for example, in the use of Java-specific object-oriented COTS products (Enterprise Java Beans, Java Message Service, etc.) to develop application services on standard IBM, Sun and Oracle platforms.

4.2.4 Requirements or User Interface Mismatch (Risk B4)
The OO source of risk is the fact that Use Cases are used almost exclusively to develop requirements in OO systems. However, Use Cases only capture functional requirements, so additional process steps need to be included to develop and implement quality-related[12], non-functional requirements. An interesting source of Graphical User Interface mismatch is that the Use Case methodology, though well suited for capturing the dynamism of changing screens, is inappropriate for representing screen details.

4.2.5 Shortfalls in Architecture, Performance, Quality (Risk B5)
This is the area where OO approaches present a controversial impact. Data abstraction, encapsulation, polymorphism, and the use of distributed objects, while increasing architectural clarity, all come with a price: substantial overhead due to the introduced layers of indirection. Unless the system is carefully architected and sound performance engineering practices [11] are implemented from the beginning, satisfying performance and quality objectives becomes difficult. All of these issues boil down to the earlier mentioned design challenge. The system architect has to carefully determine the optimal system cohesion, since most real-time performance requirements can be achieved if one is willing to suffer increased coupling and the consequent loss of flexibility.

Another sensitive part of OO systems is memory management in general and the implementation of garbage collection in particular. Garbage collection is an integral part of most OO run-time environments. It is a popular technique to ensure that memory blocks that were dynamically allocated by the programmer are released and returned to the free memory pool when they are no longer needed. A typical OO application of this feature is the dynamic creation and destruction of objects. The problem is that in conventional systems the execution of the main process needs to be interrupted while the garbage collector does its job. This randomly invoked process with variable durations disrupts the real-time behavior of the system. There are two different approaches to the mitigation of this risk. In the case of real-time OO systems, prudent programming practice should include explicit object creation and destruction to eliminate the dependency on garbage collection. Another solution is the implementation of the garbage collector via multi-threading. However, multi-threading is a difficult, advanced concept that itself can be the source of numerous risks. For a complete discussion of multithreading implementation pitfalls in Java see [12].

It is also important to examine the relationship between corporate architecture initiatives such as platform-based product lines and some key OO concepts. Developing and maintaining a product line of multiple products that share a common, managed set of features one product at a time is no longer economically viable if a multi-project business case exists. The implementation of a strategic reuse process becomes the key factor in achieving fast, efficient, predictable, low-cost and high quality products [13], [14]. In the context of reuse, a product line will represent a product family, with a set of related systems that are built from and leveraging off a common set of core assets. More specifically, products of a product line share ("reuse") an architecture and common components. As discussed earlier, OO promises a high level of reuse via the inheritance feature. Nevertheless, OO's practical reuse is not as supportive of the described strategic reuse initiatives as we would like to see, and even the full and uncompromising implementation of OO does not guarantee the satisfaction of the mentioned corporate architecture initiatives.

4.2.6 Straining Computer-Science Capabilities (Risk B8)
The appeal of the concepts M1-M4 (see Figure 2), which are theoretical in nature, inspires system architects to use OO in designing complex systems. Concepts M5-M7 are related to implementation, and their role is to enable and facilitate the use of the theoretical concepts. This risk item refers to the persistent tension between the theoretical concepts and their implementation, and the delicate balance that must be maintained amongst programming languages, developing environments, and analysis/design methods. The viability and feasibility of all these elements

[12] Quality in short is fitness for purpose, the degree to which a system accomplishes its designated functions within constraint. It includes all the "illities", e.g., availability, reliability, security, safety, etc.

have to be continually verified against the developed system's architecture. A recent example is the introduction of a promising new programming technique called Aspect-Oriented Programming (AOP). According to Gregor Kiczales, one of the principal developers of AOP, integrating AOP with OO development environments is difficult [15]. A standard development environment would have facilities for structure browsing, smart editing, refactoring, building, testing, and debugging, but it does not have a way to represent and directly manipulate AOP-specific constructs.

5. Summary

A systematic approach was presented to identify risks in OO development. The fundamental concepts of OO were introduced and matched against a well-known, methodology-neutral list of software risks. This dissection of OO concepts allows project managers to more completely understand the cost/benefit aspects of applying OO, and to align their project management strategies better with the organization's business goals.

6. Acknowledgements

This work would not have been possible without assistance from the following:

- Reviewers
  Richard J. Adams, Sergio Alvarado, Suellen Eslinger, and Joanne Tagami, The Aerospace Corporation
  Scott A. Whitmire, ODS Software, Inc.
- Sponsor
  Michael Zambrana, USAF Space and Missile Center
- Funding Source
  Mission-Oriented Investigation and Experimentation (MOIE) Research Program, S/W Acquisition Task

7. References

[1] Meyer, B. Object Success, Prentice Hall, 1995
[2] IEEE Standard for Software Life Cycle Processes-Risk Management, The Institute of Electrical and Electronics Engineers, Inc. 2001
[3] Merson, P., *Managing J2EE Risks*, Software Development, July 2004
[4] Moynihan, T., *How Experienced Project Managers Assess Risk*, IEEE Software, May/June 1997
[5] Boehm, B., IEEE Tutorial on Software Risk Management, IEEE Computer Society Press, 1989
[6] Boehm, B., *Software Risk Management: Overview and Recent Developments*, COCOMO/Software Cost Modeling Forum, #17 Tutorial, October, 2002
[7] Fichman, R., and Kemerer, C., *The Assimilation of Software Process Innovations: An Organizational Learning Perspective*, Management Science, 1997
[8] Feiman, J., *Migrating Developers to Java: Is it Worth the Cost and Risks?*, Gartner 2000
[9] Flanagan, Elizabeth, B., *Risky Business*, C++ Report, March-April 1995
[10] *Binder,* R. V., *Object-Oriented Testing: Myth and Reality*, Object Magazine, May 1995
[11] Smith, C.U., Performance Engineering of Software Systems, Addison Wesley, 1990
[12] Sandén, B., *Coping with Java Threads*, IEEE Computer, April 2004
[13] Jacobson, I. et al, Software Reuse, ACM Press, 1997
[14] Northrop, L. M., *A Practical Look at Software Product Lines*, CASCON 2003, Ontario, Canada, October 2003
[15] Kiczales, G., and Kersten, M., *Show Me the Structure*, Software Development, April 2004

ROI for Software Development Operations

Wolfgang Strigel

QA Labs, Inc.

In today's economic climate, IT investment decisions are subject to intense scrutiny. Decisions about new initiatives are often based on evidence of positive return on investment (ROI), a widely-used approach for measuring the economic value of new and improved processes such as software process improvements. Vendors and service providers have been quick to frame the benefit of their offering in ROI terms. Unfortunately many of these claims are unsubstantiated and grossly exaggerated; most fail to mention how long it takes for the promised benefits to materialize, rendering the assertions useless.

Investments in IT must consider both short-term tactical goals and long-term strategic goals of the business. In other words, the CIO needs to react to, and align IT with, the current business plan while developing a framework to raise organizational awareness of opportunities where IT can make a difference to the business overall.

This presentation will analyze opportunities for achieving significant ROI from improvements in the development process. Rather than limiting the discussion to cost cutting measures, we will argue that the most significant opportunities come from addressing the overall business objectives and relating IT improvements to opportunities for increased revenue and service provision. The resulting impact on revenues makes ROI arguments a powerful instrument to link IT objectives with corporate goals. We will also consider how improvements must be aligned with cultural and technological readiness for change. Several examples and case studies will be presented to show how traditional and emerging process technologies can generate positive ROI.

Wolfgang B. Strigel is a co-founder and President of QA Labs Inc., a leading North American software testing company offering services in quality assurance, QA outsourcing, consulting, and training. Previously he founded and served as President of the Software Productivity Centre Inc. He also formerly served as a vice president at MacDonald Dettwiler, a Canadian aerospace company.

As a member of IEEE, Mr. Strigel serves on the Editorial Advisory Board for the IEEE Software Magazine. He is also a member of ACM, PMI, ASQC, and the ISO standards council, in addition to several academic advisory boards. He is a frequent speaker at international conferences. Mr. Strigel received a B.Sc. (Munich, Germany) and M.Sc. (McGill University, Montreal, Canada) in Computer Science, and a MBA (SFU, Vancouver, Canada).

ROI for Software Development Operations

Wolfgang Strigel
QA Labs Inc.
Vancouver, Canada

©2004 Software Productivity Center, Inc.

PNSQC.ORG
The Pacific Northwest Software Quality Conference

ROI is in fashion

"We are pleased to extend our alliance with Company X to help customers further drive ROI throughout the IT environment"

©2004 Software Productivity Center, Inc.

"We Are Pleased to Extend Our Alliance With Company X to Help Customers Further Drive ROI Throughout the IT Environment"

- **What** does this mean?
- **How** is ROI achieved?
- **Why** does this generate the expected ROI
- **When** will this ROI be of any significance?
- **Under what conditions** can substantial ROI be achieved?

©2004 Software Productivity Center, Inc.

Definition of ROI

ROI = Relatively Obscure Idea

ROI = (benefit-cost)/cost

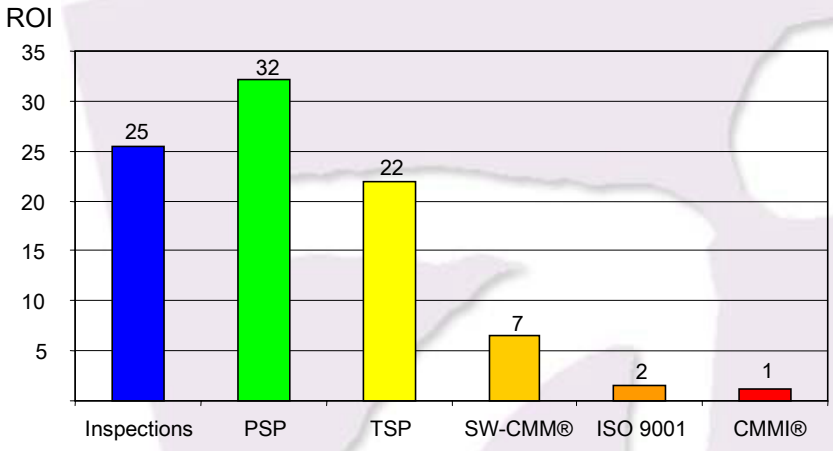
©2004 Software Productivity Center, Inc.

Myths of ROI

- ROI is only about controlling costs
- ROI is only relevant as upfront cost justification
- Tools inevitably create positive ROI
- Process improvement has a defined end point
- Process models like CMM have guaranteed ROI

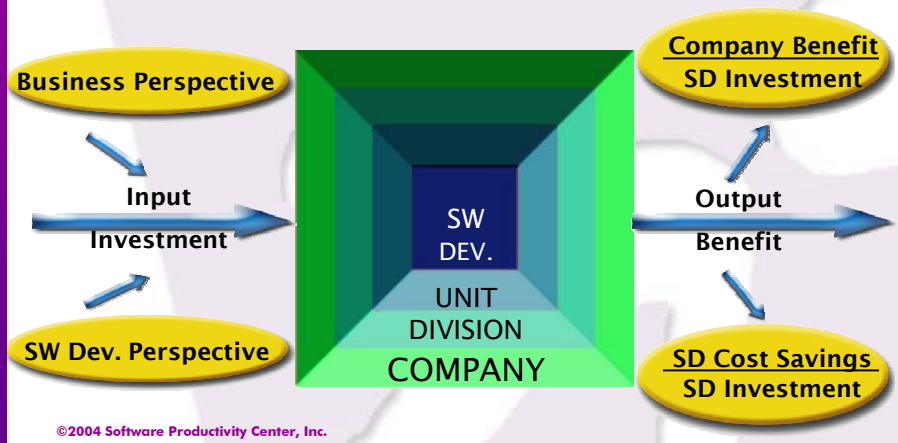
©2004 Software Productivity Center, Inc.

Beware of Statistics



©2004 Software Productivity Center, Inc.

Elevating SW ROI to the Corporate Level



©2004 Software Productivity Center, Inc.

What and How to Measure

Do you have current measurements to compare with future improvements?

©2004 Software Productivity Center, Inc.

Cost/Benefit Measures

Cost Measures

- Improvement effort
- Tool cost
- External consultants
- Training cost

Atomic Measures

- Requ's, Use Cases
- LoC, FP, UI features
- Defects

Benefit Measures

- Increased revenue
- Customer satisfaction
- Reduction in cycle time
- Development cost/schedule reduction
- Reduced maintenance cost
- Lower staff turn-over
- Higher predictability

©2004 Software Productivity Center, Inc.

Example

Consulting & Customer Effort: $60k

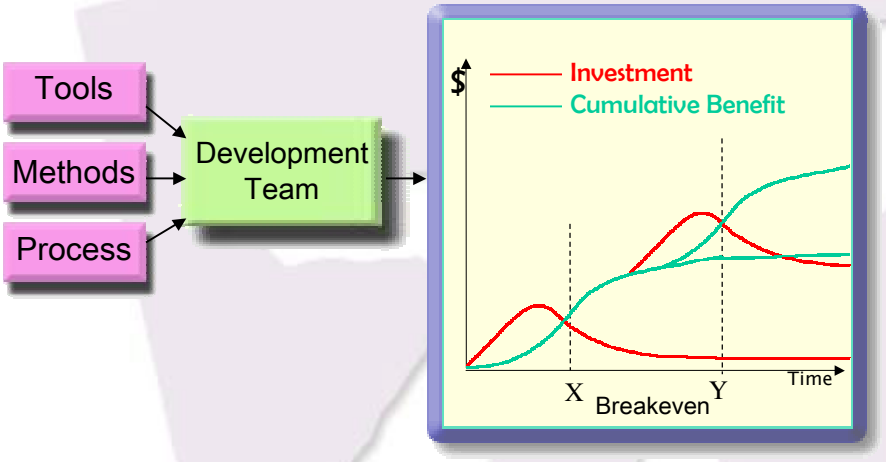
Productivity Improvement: 144%

6 Months Benefit: $648k
(15 Team Members)

$$\frac{\text{Benefit - Cost}}{\text{Cost}} = \frac{\$648k - \$60k}{\$60k} = 9.8 \text{ (ROI)}$$

©2004 Software Productivity Center, Inc.

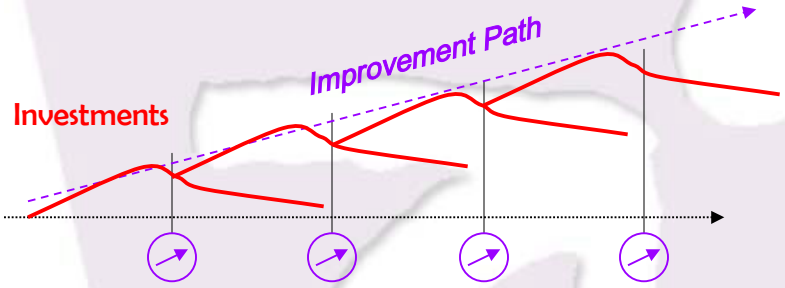
Benefit Buildup



©2004 Software Productivity Center, Inc.

Incremental Improvement

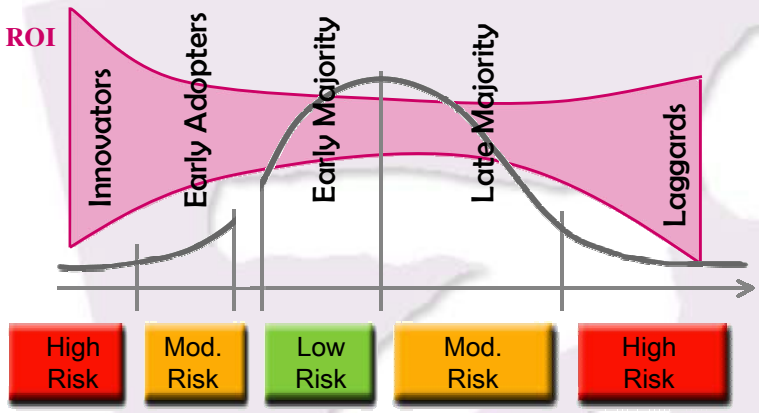
SPI should not be a big one-of-a-kind project



Incremental bite-size steps

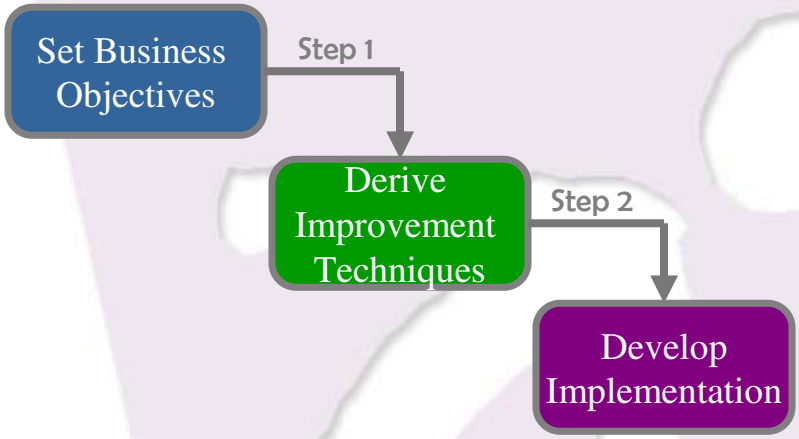
©2004 Software Productivity Center, Inc.

What Is Your Risk Profile?



©2004 Software Productivity Center, Inc.

Objective Setting



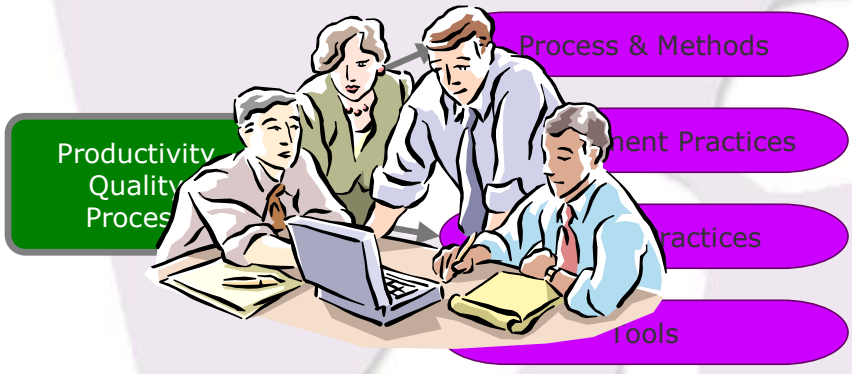
©2004 Software Productivity Center, Inc.

Step One: Improvement Techniques

Business Goals	Impr. Techniques
Accelerate Time to Market	Productivity
Increase Feature Set	Productivity
Reduce Maintenance Cost	Quality
Increase Customer Sat	Quality
Increase Product Life	Architecture
Increase Staff Utility	Training, Process

©2004 Software Productivity Center, Inc.

Step Two: Implementation Methods



©2004 Software Productivity Center, Inc.

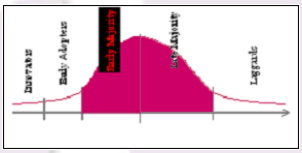
ROI and SPI
Areas of Leverage

Requirements specification

- Planting the right seeds
- Development AND testing depend on it
- Outsourcing needs good requirements

Methods and Tools

- Professional requirements capture and definition (process)
- Requirements management tools
- Change management tools



©2004 Software Productivity Center, Inc.

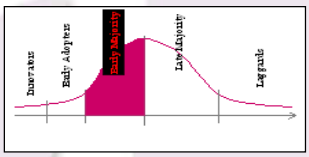
ROI and SPI
Areas of Leverage

Change Management

- Tracking issues and changes
- Ability to roll back
- Maintaining multiple versions

Methods and Tools

- Professional change management process
- Change control board
- Change management tools



©2004 Software Productivity Center, Inc.

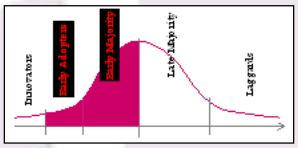
ROI and SPI
Areas of Leverage

Testing

- Start at the beginning of the project
- Automate as much as reasonable
- Maintain defect tracking and prioritization

Methods and Tools

- Professional test process, test plans, test scripts
- Test automation tools
- Bug tracking data base tied into change tracking



©2004 Software Productivity Center, Inc.

Testing ROI Example

Consulting & Training Investment:	$200k
Customer Effort:	$415K
Business Productivity Improvement:	25%
12 Months Benefit: (175 business staff)	$2.1M

$$\frac{\text{Benefit - Cost}}{\text{Cost}} = \frac{\$2.1\text{M} - \$615\text{K}}{\$615\text{K}} = 2.4 \text{ (ROI)}$$

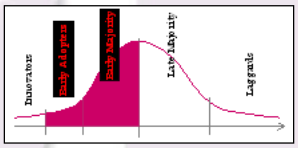
©2004 Software Productivity Center, Inc.

ROI and Outsourcing

- Off-shore or near-shore outsourcing
- Significant ROI is possible

Invest in

- Initial project is likely an investment in learning
- Initial business process reengineering
- Requirements process
- Change tracking tools
- Project management tools for remote access
- Well defined subcontracting process



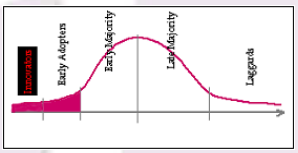
©2004 Software Productivity Center, Inc.

ROI and Agile

- XP, peer programming, extreme testing
- Initial culture shift and risk of misfit
- Significant ROI is possible

Invest in

- Lots of training
- Initial business process reengineering
- Lots of buy-in
- Customer involvement

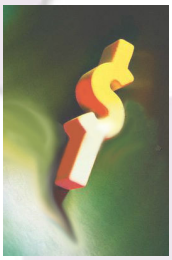


©2004 Software Productivity Center, Inc.

The Big Picture

Improving software operations is not only about cutting costs

→ added value to the business
→ increased competitiveness
→ increased market share
→ reduced maintenance costs
→ keeping up with other parts of the business



©2004 Software Productivity Center, Inc.

Context	Publication	Return on Investment
Unknown	Brodman & Johnson 1996	1.5
Systems	Diaz & King, 2002	2.2
BDN International	Slaughter et al. 1998	3
Unknown	Herbsleb et al. 1994	4
U.S. Navy	Dunaway et al. 1999	4.1
Unknown	Herbsleb et al. 1994	4.2
Hughes Aircraft	Humphrey et al. 1991	5
India	Goyal et al. 2001	5.5
Tinker Air Force Base	Brodman & Johnson 1996	6
Unknown	Herbsleb et al. 1994	6.4
Motorola	Diaz & Sligo 1997	6.77
OC-ALC (Tinker)	Butler 1995	7.5
Philips	Rooijmans et al. 1996	7.5
Raytheon	Dion 1992 & 1993	7.7
Boeing	Yamamura & Wigle 1997	7.75
Unknown	Herbsleb et al. 1994	8.8
Unknown	Brodman & Johnson 1996	10
Hewlett-Packard	Grady & Van Slack 1994	10.4
Northrop Grumman ES	Reifer et al. 2002	12.5
Ogden ALC	Oldham et al. 1999	19
	Average	**7**
	Median	**6.6**

©2004 Software Productivity Center, Inc.

Solingen, R. van, The Cost and Benefits of Software Process Improvement, Proceedings of the Eighth European Conference on IT Evaluation, ISBN 0-9540488-0-6, Sep. 2001

Summary

- First choose a business objective, then set ROI targets.
- Define base measurements (input, output) to calculate ROI.
- Start small, consecutive mini projects. Measure.
- Select techniques acceptable to your culture.
- Be patient – don't expect results in Q1.
- Make sure to have a process before introducing new tools.

©2004 Software Productivity Center, Inc.

Proceedings Order Form

Pacific Northwest Software Quality Conference

Printed Proceedings and CD Roms are available for the following years.
Circle year for the Proceedings that you would like to order.
1999, 2000, 2001, 2003, 2004

To order a copy of the Proceedings and CD, please send a check in the amount of $35.00 each to:

PNSQC/Pacific Agenda
PO Box 10733
Portland, OR 97296-0733

Name_____

Affiliate_____

Mailing
Address_____

City_____

State_____

Zip_____ Phone_____

Using the Electronic Proceedings

Once again, PNSQC is proud to produce the Proceedings in PDF format. Most of the papers from the printed Proceedings are included as well as some slide presentations. We hope that you will find this conference reference material useful. If you have any suggestions for improving the electronic Proceedings, please visit our web site at www.pnsqc.org and send us email.

Adobe Acrobat Reader

The electronic Proceedings are in Adobe Acrobat format. If you do not currently have Acrobat Reader 4 installed, you can download it from Adobe's web site: http://www.adobe.com/acrobat.

Copyright

You can print articles or slides for your own private use, but remember they are copyright to the original author. You cannot redistribute the Acrobat files or their printed copies. If you need extra copies of the printed or electronic Proceedings, please contact Pacific Agenda. An order form appears on the previous page.