TWENTY-THIRD ANNUAL
PACIFIC NORTHWEST

SOFTWARE QUALITY CONFERENCE

October 10 - 12, 2005

Oregon Convention Center
Portland, Oregon

TABLE OF CONTENTS

Technical Tools Track – October 11

Process Track – October 11

Metric & Test Track – October 12

Soft Skills Track – October 12

Technical Procedures Track – October 12

Open Source Track – October 12

Welcome to the 2005 Pacific Northwest Software Quality Conference

This is the twenty-third year for the Pacific Northwest Software Quality Conference (PNSQC). We think that you will find PNSQC to be one of the best in terms of learning new state-of-the-art and practical methods for improving software quality. The conference features speakers from around the globe, networking events to make contact with other quality-minded individuals, panel discussions on hot topics, workshops to keep your skills up-to-date, vendors to talk with about tools and services, and opportunities to socialize with other participants. For almost a quarter of a century, these aspects have contributed to the popularity of the Pacific Northwest Software Quality Conference!

Our theme this year is "Upgrading Your Toolbox: Adapting and Adopting Tools and Practices." You will hear practical information enabling you to better choose open source tools, deal with limited resources, influence people, improve processes, change your lifecycle, and more.

Our keynotes, Tim Lister and Linda Rising, are dynamic speakers who will give you valuable insight into confronting problems in your organization and influencing people to make changes. Both speakers are experts in their fields and their presentations are likely to affect your thinking.

Paul Dittman
President
PNSQC

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Paul Dittman – PNSQC President and Chair

Debra Schratz – PNSQC Vice President
Intel Corporation

Doug Reynolds – PNSQC Secretary & Program Co-Chair
Tektronix, Inc.

Cindy Oubre – PNSQC Treasurer & Exhibits Chair

David Butt – Program Co- Chair

Rick Clements – Workshops Co-Chair
Pixelworks

Esther Derby – Keynote & Invited Speaker Chair
Esther Derby Associates

Rebecca Gee – Workshop Co-Chair
HOSTS Learning

Cynthia Gens – Program Co-Chair

Shauna Gonzales – Open Space Co-Chair
Nike, Inc.

Randy King – Volunteer Chair
Mentor Graphics, Inc.

Diana Larsen – Open Space Co-Chair
FutureWorks Consulting

Patt Thomasson – Communications & Publicity Chair
McAfee, Inc.

Richard Vireday – Technology Chair
Intel Corporation

2005 CONFERENCE PLANNING COMMITTEE

Rick Anderson
Tektronix, Inc.

John Balza
Hewlett Packard

Balbinda Banga
Portland State University

Albert Dijkstra

Julie Fleischer
Intel Corporation

Manny Gatlin
Timberline Software

Cynthia Gens
Loui Canz - Louis XV

Les Grove
Tektronix, Inc.

Bhushan Gupta
Hewlett Packard

Brian Hansen
Oregon Health Science University

Kathy Iberle
Hewlett Packard

Jason Kelly
Microsoft

Mike Kelly
Consultant

Howard Mercier
Integrated Information Systems

Jonathan Morris
Unicru, Inc.

Elizabeth Ness
Hewlett-Packard

Ian Savage

Eric Schnellman
JanEric Systems

Erik Simmons
Intel Corporation

Wolfgang Strigel
QA Labs

Ruku Tekchandani
Intel Corporation

Ciprian Ticea
QA Labs

Scott Whitmire
ODS Software

Five Uncomfortable Truths

About Making Useful Systems

Tim Lister
Atlantic Systems Guild, Inc.

Tim Lister will present his Five Uncomfortable Truths. These truths are five of his beliefs about why making useful systems is *so* hard. There are truths that you can't say in your organization, so he will say them for you. By recognizing them, we can start to find ways to deal with the truth. Example of an uncomfortable truth: Some projects are sure failures from the day they are born. One of those may be wasting resources in your organization today.

Tim Lister is a software consultant at the Atlantic Systems Guild, Inc., based in the New York office. He divides his time between consulting, teaching, and writing. Tim is co-author with his partner, Tom DeMarco, of the book, *Waltzing With Bears: Managing Risk on Software Projects* (Dorset House, 2003) that won Software Development magazine's Jolt Award as General Computing Book of the Year for 2003-2004. Tim Lister and Tom DeMarco are also co-authors of *Peopleware: Productive Projects and Teams*, (Dorset House, 1999) now available in 14 languages.

Tim is currently a member of the Cutter IT Trends Council. He is a member of the I.E.E.E. and the A.C.M. He is in his 20th year as a panelist for the American Arbitration Association, arbitrating disputes involving software and software services.

PACIFIC NW SOFTWARE QUALITY CONFERENCE

Five Uncomfortable Truths About Making Useful Systems

Tim Lister

"Alice laughed: "There's no use trying," she said; "one can't believe impossible things."
"I daresay you haven't had much practice," said the Queen. "When I was younger, I always did it for half an hour a day. Why, sometimes I've believed as many as six impossible things before breakfast."
Alice in Wonderland.

Five Uncomfortable Truths To Talk About

- Managers
- QA
- Requirements
- Estimating
- Software

But First…

The Dead Fish of Failure… It sits on the table of far too many projects.

How can we accept projects formulated to fail?

Everybody smells it right away.
Everybody hunkers down.

The joy of success. It lets you:

- ★ try hard
- ★ learn
- ★ experiment
- ★ have pride

For starters:

We all need to declare the smell as soon as it is wafting in the project area.

We need to all discuss the damage the Dead Fish does.

Is it too much to ask that 50% of our projects should be outright winners?

Managers Are Lonely

Managers Are Lonely

- **How do they learn?**
- **Feel like imposters.**
- **No collegial environment**
- **Not truly a teammate**
- **What is a P.M.?**
- **What are their responsibilities?**
- **Classes in stuff that doesn't matter**

Q.A. *Is* a Bottleneck

Q.A. *Is* a Bottleneck

- **Separate the role, okay, but separate the work?**
- **Q.A. reacts to the cacophony of what goes on before it.**
- **Dependent on the V model**
- **A thankless job.**

Getting Agreement on Requirements is Not Neat…

Gathering requirements???

Here's a pretty orange requirement. I'll take it back to the office.

- **What is the problem?**

Getting Agreement on Requirements is Messy…

- **Everybody starts out wrong - model and prototype.**
- **"We've been thinking."**

Estimating is Overwhelmed By Expectations

Estimating is Overwhelmed By Expectations

- **skew**
- **precision**
- **cargo overboard**

Everyone estimates… all the time.

Software (In The General) Is Uninteresting…

Watch Out For Anti-Intellectualism

Making toast

or

Hard work ?

- **The endless search for the "right way."**
- **Meta- methods like "agile."**

Never lose the satisfying feeling of making something valuable together with your colleagues.

The Guild

Tim Lister
The Atlantic Systems Guild, Inc.
353 West 12 Street
New York NY 10014
212 620-4282
lister@acm.org
systemsguild.com

The Art, the Science,

and the Magic of Influence

Linda Rising
Independent Consultant and Author

Those of us who struggle to "find a better way" are hindered by our beliefs in several myths:

- Good ideas stand on their own and shouldn't need "selling"
- We are logical decision-makers, so the best way to convince others is to outline a logical argument
- We are intelligent people and, therefore, not susceptible to the blatant approaches of marketing and advertising.

Linda Rising will reveal some surprises that will help overcome long-held beliefs in these and other myths.

Linda Rising has a Ph.D. from Arizona State University in object-based design metrics. Her background includes university teaching as well as work in industry including telecommunications, avionics, and strategic weapons systems. She is a presenter of topics related to patterns, retrospectives, and the change process. As an author, Linda has written numerous articles, and published four books: *Design Patterns in Communications, The Pattern Almanac 2000*, and *A Patterns Handbook*. *Fearless Change: Patterns For Introducing New Ideas* is her latest book, written with Mary Lynn Manns.

www.lindarising.org

The Art, the Science, and the Magic of Influence

Linda Rising
linda@lindarising.org
www.lindarising.org

Introduction

- **Who am I?**
- **Why am I here?**
- **What do I know anyway?**

Shortcuts

- **Complexity, complexity, complexity**
- **Limited time, ability, energy**
- **Evolved to help us**
- **Work well *most* of the time**

Ethics

- **Can tools be “good” or “evil”?**
- **Can influence be used in the short-term?**
- **We don’t think we are influenced.**
- **Experiments always have a control group.**

What's the plan for the talk?

- **Myth**
- **Underneath the myth**
- **Quote**
- **Brief explanation and a story**
- **Research**
- **Influence strategy**

Myth #1:
Having a good idea is enough. I don't have to "sell" it.

***Underlying the myth*:**
An idea is "good" or "bad."
Good ideas "should" succeed.

As you sit across the table from the person you'd most like to work for, it is crucial that you relate your skills to what's going on in his head, not merely to what's going on in yours.

What Color Is Your Parachute?
Richard Nelson Bolles

The person you are trying to convince always wants to know, "What's in it for me?" You must answer that question.

Mentor Story

Research

- **Kurt Lewin during WWII**
- **Convince housewives to try organ meats**
- **Lecture vs. discussion led by informed facilitator**
- **3% of lecture attendees vs. 32% of discussion group participants were convinced to eat organ meats**

We communicate on many levels—conscious and unconscious. Having integrity means sending consistent messages. Use a personal touch to genuinely see the other person's concerns.

Myth #2:

The success or failure of my new idea has nothing to do with me. I'm just the messenger.

Underlying the Myth:
We make decisions rationally. We're not affected by the personality of the presenter.

People don't sue doctors they like. We've had people say they want to sue some specialist, and we say, "We don't think that doctor was negligent. We think it's your primary care doctor who was at fault." The client will say, "I don't care. I love her. I'm not suing her."

Malpractice lawyer, Alice Burkin, in *Blink* by Malcolm Gladwell

People buy (ideas, cars, houses, …) from people they like. We like attractive people. We like people who are like us and who *like* us.

Prisoner story

Research

- **Experimenter asked a favor of three groups.**
- **I: received only praise.**
- **II: received only negative comments.**
- **III: received a mixture.**
- **Experimenter had most success with I, even when the group realized that the experimenter stood to gain from their response. Praise produced the same result even when untrue.**

Be fearless. Find something you like about the other person. Not only will that cause the other person to like you, you will come to like them because you have found something you genuinely appreciate.

Teacher evaluations story

Myth #3:

I'm giving my time and energy to tell people about my idea. That's enough.

Underlying the myth:
People have too much stuff. They don't need any more—especially food. More stuff is a waste of resources!

Across all cultures, the giving and receiving of gifts obligates the receiver.

Hare Krishna story

Several studies showed that sending money with a questionnaire increased replies compared to offering the same reward after-the-fact. One study showed that a $5 gift up front was twice as effective at getting returns as a $50 offer for completing the survey.

In any situation, don't say, “Who can help me?” say, "Who can I help?" Don’t just give tokens. Give useful and valuable information. Even if that information turns out to be something they already know, what's important is that you tried to help them and now, to live up to the rule of reciprocity, they are likely to try to help you.

Myth #4:

We can all learn to live together in peace and harmony.

Underlying the Myth:

I'm doing the best I can. It's those few instigators that cause all the trouble. If we get rid of them, everything will be peachy.

All things being equal, you root for your own sex, your own culture, your own locality…and what you want to prove is that you are better than the other person. Whomever you root for represents you, and when he or she wins, you win.

Isaac Asimov

We are constantly labeling others and dividing the world into "us" and "them." In an instant we know who we like (and don't like). To do this we rely on patterns we learn all our lives.

T-shirts in a retrospective story

Research

- **Boys at camp in two cabins**
- **Letting them assign names to the groups**
- **Competitive games**
- **Shared pleasant activities had no effect**
- **Successful joint efforts were created where competition would have harmed all interests and cooperation was necessary to reach mutual goals.**
- **Similar results for other groups. Cooperation leads to greater liking and greater group success.**

Build a better connection with anyone by spending a few minutes a day seeking common ground. Focus on what you have in common and bring it out into the open. Work together toward that common goal. The best influencers can find commonalities with almost anyone.

In summary – I believe:

- **Understanding influence strategies is good**
- **Tools (like influence) can be used appropriately**
- **Cooperation can trump division**
- **Remember, I'm pullin' for you. We're all in this together!**

Optimizing the Contribution of Testing to Project Success

Niels Malotaux
N R Malotaux - Consultancy
Bongerdlaan 53
3723 VB Bilthoven
The Netherlands
+31-30-2288868
niels@malotaux.nl
www.malotaux.nl/nrm/English

Abstract

Let's define the Goal of development projects as: Providing the customer with what he needs, at the time he needs it, to be more successful than he was without it, constrained by what we can deliver in a reasonable period of time. Furthermore, let's define a defect as the cause of a problem experienced by the users of our software. If there are no defects, we will have achieved our goal. If there are defects, we failed.

We know all the stories about failed and partly failed projects, only about one third of the projects delivering according to the original goal.

Apparently, despite all the efforts for doing a good job, too many defects are generated by developers, and too many remain undiscovered by testers, causing still too many problems to be experienced by users. It seems that people are taking this state of affairs for granted, accepting it as a nature of software development. A solution is mostly sought in technical means, like process descriptions, metrics and tools. If this really would have helped, it should have shown by now.

Oddly enough, there is a lot of knowledge about how to significantly reduce the generation and proliferation of defects and deliver the right solution quicker. Still, this knowledge is ignored in the practice of many software development organizations. Last year at the PNSQC, I presented a paper: *How Quality is Assured by Evolutionary Methods*, describing practical implementation details of how to organize projects using this knowledge, making the project a success.

In this paper we'll extend these Evolutionary methods to the testing process, in order to optimize the contribution of testing to project success.

Important ingredients for success are: a change in attitude, taking the Goal seriously, which includes working towards defect-free results, focusing on prevention rather than repair, and constantly learning how to do things better.

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 30 projects in 11 different organizations, which led to a wealth of experience in which approaches work better and which work less well.

V1.0 - 20050803

Introduction

We know all the stories about failed and partly failed projects, only about one third of the projects delivering according to the original goal [1].

Apparently, despite all the efforts for doing a good job, too many defects are generated by developers, and too many remain undiscovered by testers, causing still too many problems to be experienced by users. It seems that people are taking this state of affairs for granted, accepting it as a nature of software development. A solution is mostly sought in technical means, like process descriptions, metrics and tools. If this really would have helped, it should have shown by now.

Oddly enough, there is a lot of knowledge about how to significantly reduce the generation and proliferation of defects and deliver the right solution quicker. Still, this knowledge is ignored in the practice of many software development organizations. In papers and in actual projects I've observed that the time spent on testing and repairing (some people call this *debugging*) is quoted as being 30 to 80% of the total project time. That's a large budget and provides excellent room for a lot of savings.

Last year at the PNSQC, I presented a paper: *How Quality is Assured by Evolutionary Methods* [2], describing practical implementation details of how to organize projects using this knowledge, making the project a success. In an earlier booklet: *Evolutionary Project Management Methods* [3], I described issues to be solved with these methods and my first practical experiences with the approach. Tom Gilb published already in 1988 about these methods [4].

In this paper we'll extend the Evo methods to the testing process, in order to optimize the contribution of testing to project success.

Important ingredients for success are: a change in attitude, taking the Goal seriously, which includes working towards defect-free results, focusing on prevention rather than repair, and constantly learning how to do things better.

The Goal

Let's define as the main goal of our software development efforts: *Providing the customer with what he needs, at the time he needs it, to be satisfied, and to be more successful than he was without it*…
If the customer is not satisfied, he may not want to pay for our development efforts. If he is not successful, he *cannot* pay. If he is not more successful than he already was, why should he have invested in our product anyway?
Of course we have to add that what we do in a development project is …*constrained by what the customer can afford and what we mutually beneficially and satisfactorily can deliver in a reasonable period of time*.

Furthermore, let's define a defect as the *cause of a problem experienced by the users of our software*. If there are no defects, we'll have achieved our goal. If there are defects, we failed.

The knowledge

Important ingredients for significantly reducing the generation and proliferation of defects and delivering the right solution quicker are:

- **Clear Goal**: If we have a clear goal for our project, we can focus on achieving that goal. If management does not set the clear goal, we should set the goal ourselves.
- **Prevention attitude**: Preventing defects is more effective and efficient than injecting-finding-fixing, although it needs a specific attitude that usually doesn't come naturally.
- **Continuous Learning**: If we organize projects in very short Plan-Do-Check-Act (PDCA) cycles, constantly selecting only the most important things to work on, we will most quickly learn what the real requirements are and how we can most effectively and efficiently realize these requirements. We spot problems quicker, allowing us more time to do something about them.

Evolutionary Project Management (Evo for short) uses this knowledge to the full, combining Project-, Requirements- and Risk-Management into Result Management. The essence of Evo is actively, deliberately, rapidly and frequently going through the PDCA cycle, for the product, the project *and* the process, constantly reprioritizing the order of what we do based on Return on Investment (ROI), and highest value first. In my experience as project manager and as project coach, I observed that those projects, who seriously apply the Evo approach, are routinely successful on time, or earlier [5].

Evo is not only iterative (using multiple cycles) and incremental (we break the work into small parts), like many similar Agile approaches, but above all Evo is about learning. We proactively anticipate problems before they occur and work to prevent them. We may not be able to prevent all the problems, but if we prevent most of them, we have a lot more time to cope with the few problems that slip through.

Something is not right

Satisfying the customer and making him more successful implies that the software we deliver should show no defects. So, all we have to do is delivering a result with no defects. As long as a lot of software is delivered with defects and late (which I consider a defect as well), apparently something is not right.

Customers are also to blame, because they keep paying when the software is not delivered as agreed. If they would refuse to pay, the problem could have been solved long ago. One problem here is that it often is not obvious what was agreed. However, as this is a *known problem*, there is no excuse if this problem is not solved within the project, well before the end of the project.

The problem with bugs

In a conventional software development process, people develop a lot of software with a lot of defects, which some people call *bugs*, and then enter the *debugging* phase: testers testing the software and developers repairing the *bugs*.
Bugs are so important that they are even counted. We keep a database of the number of bugs we found in previous projects to know how many bugs we should expect in the next project. Software without bugs is even considered suspect. As long as we put bugs in the center of the testing focus, there will be bugs. Bugs are normal. They are needed. What should we do if there were no bugs any more?

This way, we *endorse* the injection of bugs. But, does this have anything to do with our goal: making sure that the customer will not encounter any problem?

Personally, I dislike the word bug. To me, it refers to a little creature creeping into the software, causing trouble beyond our control. In reality, however, people make mistakes and thus cause defects. Using the word *bug*, subconsciously defers responsibility for making the mistake. In order to prevent defects, however, we have to actively take responsibility for our mistakes.

Defects found are symptoms

Many defects are symptoms of deeper lying problems. Defect prevention seeks to find and analyze these problems and doing something more fundamental about them.
Simply repairing the apparent defects has several drawbacks:

- Repair is usually done under pressure, so there is a high risk of imperfect repair, with unexpected side effects.
- Once a bandage has covered up the defect, we think the problem is solved and we easily forget to address the real cause. That's a reason why so many defects are still being repeated.
- Once we find the underlying real cause, of which the defect is just a symptom, we'll probably do a more thorough redesign, making the repair of the apparent defect redundant.

As prevention is better than cure, let's move from *fixation-to-fix* to *attention-to-prevention*.
Many mistakes have a repetitive character, because they are a product of certain behavior or people. If we don't deal with the root causes, we will keep making the same mistakes over and over again. Without feedback, we won't even know. With quick feedback, we can put the repetition immediately to a halt.

Defects typically overlooked

We must not only test whether functions are correctly implemented as documented in the requirements, but also, a level higher, whether the requirements adequately solve the needs of the customer according to the goal. Typical defects that may be overlooked are:

- Functions that won't be used (superfluous requirements, no Return on Investment)
- Nice things (added by programmers, usefulness not checked, not required, not paid for)
- Missing quality levels (should have been in the requirements)
  e.g.: response time, security, maintainability, usability, learnability
- Missing constraints (should have been in the requirements)
- Unnecessary constraints (not required)

Another problem that may negatively affect our goal is that many software projects end at "Hurray, it works!". If our software is supposed to make the customer more successful, our responsibility goes further: we have to make sure that the increase in success is *going to happen*. This awareness will stimulate our understanding of quality requirements like "learnability" and "usability". Without it, these requirements don't have much meaning for development. It's a defect if success is not going to happen.

Is defect free software possible?

Most people think that defect free software is impossible. This is probably caused by lack of understanding about what defect free, or Zero Defects, really means. Think of it as an asymptote (Figure 1). We know that an asymptote never reaches its target. However, if we put the bar at an *acceptable level* of defects, we'll asymptotically approach that level. If we put the bar at zero defects, we can asymptotically approach that level.

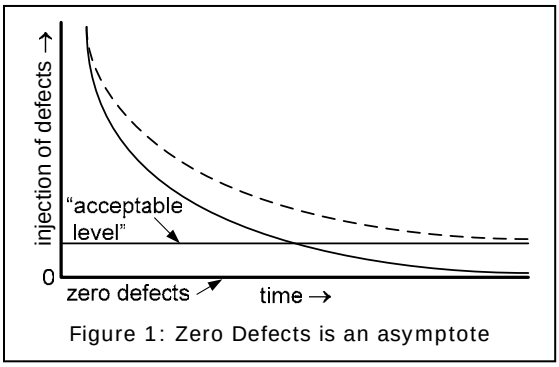


Figure 1: Zero Defects is an asymptote

Philip Crosby writes [6]:

> Conventional wisdom says that error is inevitable. As long as the performance standard requires it, then this self-fulfilling prophecy will come true. Most people will say: People are humans and humans make mistakes. And people do make mistakes, particularly those who do not become upset when they happen. Do people have a built-in defect ratio? Mistakes are caused by two factors: lack of knowledge and lack of attention. Lack of attention is an attitude problem.

When Crosby first started to apply Zero Defects as performance standard in 1961, the error rates dropped 40% almost immediately [6]. In my projects I've observed similar effects.

Experience: *No defects in the first two weeks of use*

A QA person of a large banking and insurance company I met in a SPIN metrics working group told me that they got a new manager who told them that from now on she expected that any software delivered to the (internal) users would run defect free for at least the first two weeks of use. He told me this as if it were a good joke. I replied that I thought he finally got a good manager, setting them a clear requirement: "No defects in the first two weeks of use." Apparently this was a target they had never contemplated before, nor achieved. Now they could focus on how to achieve defect free software, instead of counting function points and defects. Remember that in bookkeeping being one cent off is already a capital offense, so defect free software should be a normal expectation for a bank. Why wouldn't it be for *any* environment?

Zero Defects is a performance standard, set by management. In Evo projects, even if management does not provide us with this standard, we'll assume it as a standard for the project, because we know that it will help us to conclude our project successfully in less time.

Attitude

As long as we are convinced that defect free software is impossible, we will keep producing defects, failing our goal. As long as we are accepting defects, we are endorsing defects. The more we talk about them, the more normal they seem. It's a self-fulfilling prophecy. It will perpetuate the problem. So, let's challenge the defect-cult and do something about it.

From now on, we don't want to make mistakes any more. We get upset if we make one. Feel the failure. If we don't feel failure, we don't learn. Then we work to find a way not to make the mistake again. If a task is finished we don't *hope* it's ok, we don't *think* it's ok, no, we'll be *sure* that there are no defects and we'll be genuinely surprised when there proves to be any defect after all.

In my experience, this attitude prevents half of the defects in the first place. Because we are humans, we can study how we operate psychologically and use this knowledge to our advantage. If we can prevent half of the defects overnight, then we have a lot of time for investing in more prevention, while still being more productive. This attitude is a crucial element of successful projects.

Experience: *No more memory leaks*

My first Evo project was a project where people had been working for months on software for a hand-held terminal. The developers were running in circles, adding functions they couldn't even test, because the software crashed before they arrived at their newly added function. The project was already late and management was planning to kill the project. We got six weeks to save it.

The first goal was to get stable software. After all, adding any function if it crashes within a few minutes of operation is of little use: the product cannot be sold. I told the team to take away all functionality except one very basic function and then to make it stable. The planning was to get it stable in two weeks and only then to add more functionality gradually to get a useful product.

I still had other business to finish, so I returned to the project two weeks later. I asked the team "Is it stable?". The answer was: "We found many memory leaks and solved them. Now it's much *stabler*". And they were already adding new functionality. I said: "Stop adding functionality. I want it stable, not *almost* stable". One week later, all memory leaks were solved and stability was achieved. This was a bit of a weird experience for the team: the software didn't crash any more. Actually, in this system there was not even a need for dynamically allocatable memory and the whole problem could have been avoided. But changing this architectural decision wasn't a viable option at this stage any more.

Now that the system was stable, they started adding more functions. We got another six weeks to complete the product. I made it very clear that I didn't want to see any more memory leaks. Actually that I didn't want to see any defects. The result was that the testers suddenly found hardly any defect any more and from now on could check the correct functioning of the device. At the end of the second phase of six weeks, the project was successfully closed. The product manager was happy with the result.

Conclusion: after I made it clear that I didn't want to see any defects, the team hardly produced any defects. The few defects found were easy to trace and repair. The change of attitude saved a lot of defects and a lot of time. The team could spend most of its time adding new functionality instead of fixing defects. This was Zero Defects at work. Technical knowledge was not the problem to these people: once challenged, they quickly came up with tooling to analyze the problem and solve it. The attitude was what made the difference.

Plan-Do-Check-Act

I assume the Plan-Do-Check-Act (PCDA- or Deming-) cycle [7] is well known (Figure 2). Because it's such a crucial ingredient, I'll shortly reiterate the basic idea:

- We *Plan* what we want to accomplish and how we think to accomplish it best.
- We *Do* according to the plan.
- We *Check* to observe whether the result from the *Do* is according to then *Plan*.
- We *Act* on our findings. If the result was good: what can we do better. If the result was not so good: how can we make it better. *Act* produces a renewed strategy.

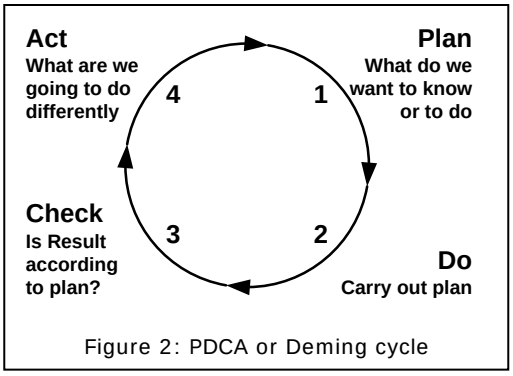


Figure 2: PDCA or Deming cycle

The key-ingredients are: planning before doing, systematically checking and above all *acting*: doing something differently. After all, if you don't do things differently, you shouldn't expect a change in result.

In Evo we constantly go through multiple PDCA cycles, deliberately adapting strategies, in order to learn how to do things better all the time, actively and purposely speeding up the evolution of our knowledge.

As a driver for moving the evolution in the right direction, we use Return on Investment (ROI): the project invests time and other resources and this investment has to be regained in whatever way, otherwise it's just a hobby. So, we'll have to constantly be aware whether all our actions contribute to the value of the result. Anything that does not contribute value, we shouldn't do.

Furthermore, in order to maximize the ROI, we have to do the most important things first. In practice, priorities change dynamically during the course of the project, so we constantly reprioritize, based on what we learnt so far. Every week we ask ourselves: "What are the most important things to do. We shouldn't work on anything less important." Note that priority is molded by many issues: customer issues, project issues, technical issues, people issues, political issues and many other issues.

How about Project Evaluations

Project Evaluations (also called Project Retrospectives, or Post-Mortems - as if all projects die) are based on the PDCA cycle as well. At the end of a project we evaluate what went wrong and what went right.
Doing this only at the end of a project has several drawbacks:

- We tend to forget what went wrong, especially if it was a long time ago.
- We put the results of the evaluation in a write-only memory: do we really remember to check the evaluation report at the very moment we need the analysis in the next project? Note that this is typically one full project duration after the fact.
- The evaluations are of no use for the project just finished and being evaluated.
- Because people feel these drawbacks, they tend to postpone or forget to evaluate. After all, they are already busy with the next project, after the delay of the previous project.

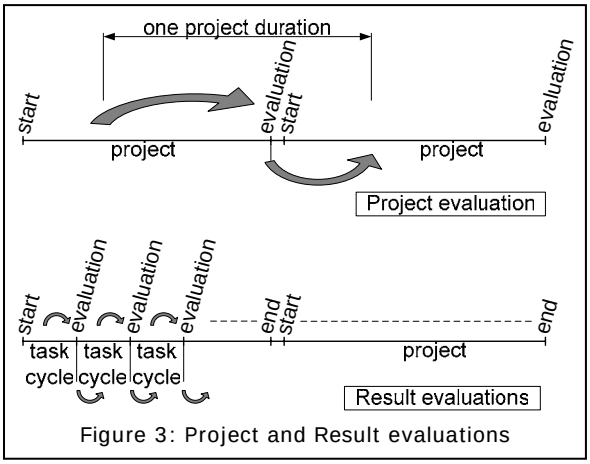


Figure 3: Project and Result evaluations

In short: the principle is good, but the implementation is not tuned to the human time-constant.
In Evo, we evaluate *weekly* (in reality it gradually becomes a way-of-life), using PDCA cycles, and now this starts to bear fruit (Figure 3):

- Not so much happens in one week, so there is not so much to evaluate.
- It's more likely that we remember the issues of the past five days.
- Because we most likely will be working on the same kind of things during the following week, we can immediately use the new strategy, based on our analysis.
- One week later we can check whether our new strategy was better or not, and refine.
- Because we immediately apply the new strategy, it naturally is becoming our new way of working.
- The current project benefits immediately from what we found and improved.

So, evaluations are good, but they must be tuned to the right cycle time to make them really useful. The same applies to testing, as this is also a type of evaluation.

Current Evo Testing

Conventionally, a lot of testing is still executed in Waterfall mode, after the *Code Complete* milestone. I have difficulty understanding this "Code Complete", while apparently the code is not complete, witness the planned "debugging" phase after this milestone. Evo projects do not need a separate debugging phase and hardly need repair after delivery. If code is complete, it is complete. Anything is only ready if it is completely done, *not to worry about it any more*. That includes: no defects. I know we are human and not perfect, but remember the importance of attitude: we *want* to be perfect[1].

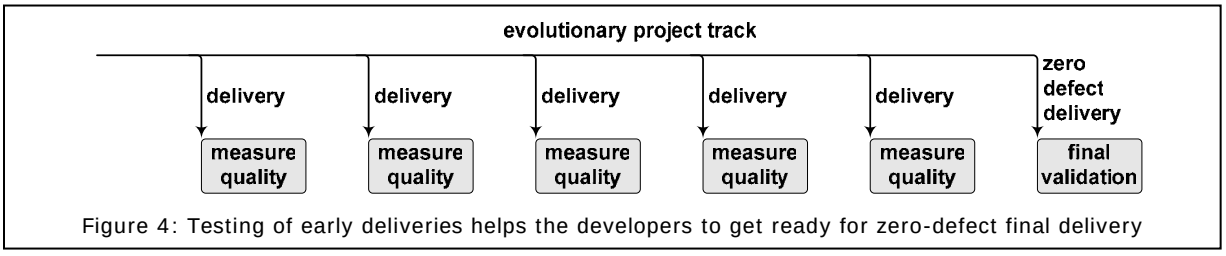


Figure 4: Testing of early deliveries helps the developers to get ready for zero-defect final delivery

Because we regularly deliver results, testers can test these intermediate results (Figure 4). They feed back their findings, which we will use for prevention or optimization. Most issues that are not caught by the testers (I suppose testers are human as well) may be found in subsequent deliveries. This way, most of any undiscovered defects will be caught before the final delivery and, more importantly, be exploited for prevention of further injection of similar defects. Because all people in the project aim for Zero Defects delivery, the developers and testers work together in their quest for perfection[1].

[1] Note that perfection means: *freedom from fault or defect*. It does *not* mean: *gold plating*.

Further improvement

To further improve the results of the projects, we can extend the Evo techniques to the testing process and exploit the PDCA paradigm even further:

- Testers focus on a clear goal. Finding defects is not the goal. After all, we don't want defects. Any defects found are only a means to achieve the real goal: the success of the project.
- Testers will select and use any method appropriate for optimum feedback to development, be it testing, review or inspection, or whatever more they come up with.
- Testers check work in progress *even before* it is delivered, to feedback issues found, allowing the developer to abstain from further producing these issues for the remainder of his work.
  "Can I check some piece of what you are working on now?" "But I'm not yet ready!" "Doesn't matter. Give me what you have. I'll tell you what I find, if I find anything". Testers have a different view, seeing things the developer doesn't see. Developers don't naturally volunteer to have their intermediate work checked. Not because they don't like it to be checked, but because their attention is elsewhere. Testers can help by asking. Initially the developers may seem a little surprised, but this will soon fade.
- Similarly, testers can solve a typical problem with planning reviews and inspections. Developers are not against reviews and inspections, because they very well understand the value. They have trouble, however, planning them in between of their design work, which consumes their attention more. If we include the testers in the process, the testers will recognize when which types of review, inspections or tests are needed and organize these accordingly. This is a natural part of their work helping the developers to minimize rework by minimizing the injection of defects and minimizing the time slipped defects stay in the system.
- In general: organizing testing the Evo way means entangling the testing process more intimately with the development process.

Cycles in Evo

In the Evo development process, we use several learning cycles:

- The TaskCycle [9] is used for organizing the work, optimizing estimation, planning and tracking. We constantly check whether we are doing the right things in the right order to the right level of detail. We optimize the work effectiveness and efficiency. TaskCycles never take more than one week.
- The DeliveryCycle [10] is used for optimizing the requirements and checking the assumptions. We constantly check whether we are moving to the right product results. DeliveryCycles focus the work organized in TaskCycles. DeliveryCycles normally take not more than two weeks.
- TimeLine [11] is used to keep control over the project duration. We optimize the order of DeliveryCycles in such a way that we approach the product result in the shortest time, with as little rework as possible.

During these cycles we are constantly optimizing:

- The product [12]: how to arrive at the best product (according to the goal).
- The project [13]: how to arrive at this product most effectively and efficiently.
- The process [14]: finding ways to do it even better. Learning from other methods and absorbing those methods that work better, shelving those methods that currently work less effectively.

If we do this well, by definition, there is no better way.

Evo cycles for testing

Extending Evo to testing adds cycles for feedback from testing to development, as well as cycles for organizing and optimizing the testing activities themselves.

- Testers organize their work in weekly, or even shorter TaskCycles.
- The DeliveryCycle of the testers is the Test-feedback cycle: in very short cycles testers take intermediate results from developers, check for defects in all varieties and feed back optimizing information to the developers, while the developers are still working on the same results. This way the developers can avoid injecting defects in the remainder of their work, while immediately checking out their prevention ideas in reality.
- The Testers use their own TimeLine, synchronized with the development TimeLine, to control that they plan the right things at the right time, in the right order, to the right level of detail during the course of the project and that they conclude their work in sync with development.

During these cycles the testers are constantly optimizing:

- The product: how to arrive at the most effective product.
  Remember that their product goal is: providing their customer, in this case the developers, with what they need, at the time they need it, to be satisfied, and to be more successful than they were without it.
- The project: how to arrive at this product most effectively and efficiently.
  This is optimizing in which order they should do which activities to arrive most efficiently at their result.
- The process: finding ways to do it better. Learning from other methods and absorbing those methods that work better, shelving those methods that currently work less effectively.

Testers are part of the project and participate in the weekly 3-step procedure [15] using about 20 minutes per step:

1. Individual preparation.
2. 1-to-1's: Modulation with and coaching by Project Management .
3. Team meeting: Synchronization and synergy with the team.

Project Management in step 2 is now any combination, as appropriate, of the following functions:

- The Project Manager or Project Leader, for the project issues.
- The Architect, for the product issues.
- The Test Manager, for the testing issues.

There can be only one captain on the ship, so the final word is to the person who acts as Project Manager, although he should better listen to the advice of the others.

Testers participate in requirements discussions. They communicate with developers in the unplannable time [16], or if more time is needed, they plan tasks for interaction with developers. If the priority of an issue is too high to wait for the next TaskCycle, the interrupt procedure [17] will be used. If something is unclear, an Analysis Task [18] will be planned. The Prevention Potential of issues found is an important factor in the prioritization process.

In the team meeting testers see what the developers will be working on in the coming week and they synchronize with that work. There is no ambiguity any more about which requirements can be tested and to which degree, because the testers follow development, and they design their contribution to assist the project optimally for success.

In Evo Testing, we don't wait until something is thrown at us. We actively take responsibility. Prevention doesn't mean sitting waiting for the developers. It means to decide with the developers how to work towards the defect free result together. Developers doing a small step. Testers checking the result and feeding back any imperfections before more imperfections are generated, closing the very short feedback loop. Developers and testers quickly finding a way of optimizing their cooperation. It's important for the whole team to keep helping each other to remind that we don't want to repair defects, because repair costs more. If there are no defects, we don't have to repair them.

Doesn't this process take a lot of time? No. My experience with many projects shows that it *saves* time, projects successfully finishing well before expected. At the start it takes some more time. The attitude, however, results in less defects and as soon as we focus on prevention rather than continuous injection-finding-fixing, we soon decrease the number of injected defects considerably and we don't waste time on all those defects any more.

Database for Change Requests and Problem Reports

Most projects already use some form of database to collect defects reported (Problem Report/PR: development pays) and proposed changes in requirements (Change Request/CR: customer pays).

With the emphasis shifted from repair to prevention, this database will, for every CR/PR, have to provide additional space for the collection of data to specifically support the prevention process, like:

- Follow-up status.
- When and where found.
- Where caused and root cause
- Where should it have been found earlier
- Prevention plan
- Analysis task defined and put on the Candidate Tasks List [19].
- Prevention task(s) defined and put on the Candidate Tasks List.
- Check lists updated for finding this issue easier, in case prevention doesn't work yet.

Analysis tasks may be needed to sort out the details. The analysis and repair tasks are put on the Candidate Tasks List and will, like all other candidate tasks, be handled when their time has come: if nothing else is more important. Analysis tasks and repair tasks should be separated, because analysis usually has priority over repair. We better first stop the leak, to make sure that not more of the same type of defect is injected.

How about metrics?

In Evo, the time to complete a task is estimated as a TimeBox [20], within which the task will be 100% done. This eliminates the need for tracking considerably. The estimate is used during the execution of the task to make sure that we complete the task on time. We experienced that people can quite well estimate the time needed for tasks, *if* we are really serious about time.
Note that exact task estimates are not required. Planning at least 4 tasks in a week allows some estimates to be a bit optimistic and some to be a bit pessimistic. All we want is that, at the end of the week, people have finished what they promised. As long as the *average* estimation is OK, all tasks can be finished at the end of the week. As soon as people learn not to overrun their (average) estimates any more, there is no need to track or record overrun metrics. The attitude replaces the need for the metric.

In many cases, the deadline of a project is defined by genuine external factors like a finite market-window. Then we have to predict which requirements we can realize before the deadline or "Fatal-Date". Therefore, we still need to estimate the amount of work needed for the various requirements. We use the TimeLine technique to regularly predict what we will have accomplished at the FatalDate *and what not*, and to control that we will have a working product well before that date. Testers use TimeLine to control that they will complete whatever they have to do in the project, in sync with the developers.

Several typical testing metrics become irrelevant when we aim for defect free results, for example:

- *Defects-per-kLoC* or *Defects-per-page*
  Counting defects condones the existence of defects, so there is an important psychological reason to discourage counting them.
- *Incoming defects per month*, found by test, found by users
  Don't count incoming defects. Do something about them. Counting conveys a wrong message. We should better make sure that the user doesn't experience any problem.
- *Defect detection effectiveness* or *Inspection yield* (found by test / (found by test + customer))
  There may be some defects left, because perfection is an asymptote. It's the challenge for testers to find them all. Results in practice are in the range of 30% to 80%. Testers apparently are not perfect either. That's why we must strive towards zero defects *before* final test. Whether that is difficult or not, is beside the point.
- *Cost to find and fix a defect*
  The less defects there are, the higher the cost to find and fix the few defects that slip through from time to time, because we still have to test, to see that the result is OK. This was a bad metric anyway.
- *Closed defects per month* or *Age of open customer found defects*
  Whether and how a defect is closed or not, depends on the prioritizing process. Every week any problems are handled, appropriate tasks are defined and put on the Candidate Tasks List, to be handled when their time has come. It seems that many metrics are there because we don't trust the developers to take appropriate action. In Evo, we do take appropriate action, so we don't need policing metrics.
- *When are we done with testing?*
  Examples from conventional projects: if the number of bugs found per day has declined to a certain level, or if the defect backlog has decreased to zero. In some cases, curve fitting with early numbers of defects found during the debugging phase is used to predict the moment the defect backlog will have decreased to zero. Another technique is to predict the number of defects to be expected from historical data. In Evo projects, the project will be ready at the agreed date, or earlier. That includes testing being done.

Instead of *improving* non-value adding activities, including various types of metrics, it is better to *eliminate* them. In many cases, the attitude, and the assistance of the Evo techniques replace the need for metrics. Other metrics may still be useful, like *Remaining Defects*, as this metric provides information about the effectiveness of the prevention process. Still, even more than in conventional metrics activities, we will be on the alert that whatever we do must contribute value.

If people have trouble deciding what the most important work for the next week is, I usually suggest as a metric: "*The size of the smile on the face of the customer*". If one solution does not get a smile on his face, another solution does cause a smile and a third solution is expected to put a big smile on his face, which solution shall we choose? This proves to be an important Evo metric that helps the team to focus.

Finally

Many software organizations in the world are working the same way, producing defects and then trying to find and fix the defects found, waiting for the customer to experience the reminder. In some cases, the service organization is the profit-generator of the company. And isn't the testing department assuring the quality of our products?

That's what the car and electronics manufacturers thought until the Japanese products proved them wrong. So, eventually the question will be: can we afford it?

Moore's Law is still valid, implying that the complexity of our systems is growing exponentially, and the capacity needed to fill these systems with meaningful software is growing exponentially even faster with it. So, why not better become more productive by not injecting the vast majority of defects. Then we have more time to spend on more challenging activities than finding and fixing defects.

I absolutely don't want to imply that finding and fixing is not challenging. Prevention is just cheaper. And, testers, fear not: even if we start aiming at defect free software, we'll still have to learn a lot from the mistakes we'll still be making.

Dijkstra [8] said:

> It is a usual technique to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

Where we first pursued the very effective way to show the presence of bugs, testing will now have to find a solution for the hopeless inadequacy of showing their absence. That is a challenge as well.

I invite testers from now on to change their focus from finding defects, to working with the developers to minimize the generation of defects in order to satisfy the real goal of software development projects. Experience in many projects shows that this is not an utopia, but that it can readily be achieved, using the Evo techniques described.

References

[1] The Standish Group: *Chaos Report*, 1994, 1996, 1998, 2000, 2002, 2004.
http://www.standishgroup.com/chaos_resources/index.php

[2] N.R. Malotaux: *How Quality is Assured by Evolutionary Methods*, 2004.
PNSQC 2004 Proceedings. Also downloadable as a booklet: http://www.malotaux.nl/nrm/pdf/Booklet2.pdf

[3] N.R. Malotaux: *Evolutionary Project Management Methods*, 2001. http://www.malotaux.nl/nrm/pdf/MxEvo.pdf

[4] T. Gilb: *Principles of Software Engineering Management*, 1988.
Addison-Wesley Pub Co, ISBN: 0201192462.

[5] See cases: http://www.malotaux.nl/nrm/Evo/EvoFCases.htm

[6] P.B. Crosby: *Quality Without Tears*, 1984.
McGraw-Hill, ISBN 0070145113.

[7] W.E. Deming: *Out of the Crisis*, 1986. MIT, ISBN 0911379010.
M. Walton: *Deming Management At Work*, 1990. The Berkley Publishing Group, ISBN 0399516859.

[8] E. Dijkstra: Lecture: *The Humble Programmer*, 1972.
Reprint in *Classics in Software Engineering*. Yourdon Press, 1979, ISBN 0917072146.

[9]	TaskCycle	ref [2] chapter 5.1	ref [3] chapter 3C
[10]	DeliveryCycle	ref [2] chapter 5.1	ref [3] chapter 3C
[11]	TimeLine	ref [2] chapter 5.5 and 6.8	
[12]	Product	ref [2] chapter 4.2	
[13]	Project	ref [2] chapter 4.3	
[14]	Process	ref [2] chapter 4.4	
[15]	3-step procedure	ref [2] chapter 6.9	
[16]	Unplannable time	ref [2] chapter 6.1	
[17]	Interrupt procedure	ref [2] chapter 6.7	
[18]	Analysis task	ref [2] chapter 6.6	ref [3] chapter 8
[19]	Candidate Task List	ref [2] chapter 6.5	ref [3] chapter 8
[20]	TimeBox	ref [2] chapter 6.4	ref [3] chapter 3D

Tuning the Root Cause Analysis Process

Kathryn Y. Kwinn
Hewlett-Packard M.S. 57
3404 E. Harmony Road
Fort Collins, CO 80525
(970) 898-3209
kathy.kwinn@hp.com

Abstract

Root cause analysis (RCA) is a tool found in many quality experts' toolkits. Software teams at Hewlett-Packard have used it for decades, so when we found ourselves with serious patch quality problems several years ago, the HP-UX Patch Program began requiring an RCA for each defective patch. Our expectation was that those who had caused the defects would, by studying their past mistakes, learn how to prevent similar mistakes in the future, and we would therefore see a significant reduction in the number of defective patches.

Unfortunately, we did not see the anticipated decrease in patch defects. As a result, we turned the RCA process on itself and identified changes that we expected would make RCA a more effective tool for preventing patch defects. This paper will discuss what we learned, how we changed our RCA process, and what others can leverage from our experience.

About the Author

Kathy Kwinn joined Hewlett-Packard in 1978 after obtaining a Ph.D. in computer science at Iowa State University. Her projects at HP have included software development, technical documentation, and project and program management. Kathy learned about patches from the customer point of view when she managed the release team in the former Software Engineering Systems Division. She is currently the HP-UX Patch Quality Program Manager.

Background

The definitions and explanations here provide context for the rest of this paper.

Root cause analysis (RCA) is a method of quality improvement based on understanding past mistakes (what, how, why) and preventing their recurrence [1].

HP-UX is Hewlett-Packard's (HP) version of the UNIX® operating system[1]. The code for the HP-UX operating system is developed by teams in several independently managed labs distributed around the world. A central team integrates the code and ships the HP-UX product. In all, roughly 20 labs contribute code to the operating system and to the HP applications that run on it. HP releases a new version of HP-UX every 18 to 24 months, on average.

Upgrading to a new version of any operating system requires considerable customer investment, especially if the customer has a low tolerance for risk. Customers may spend months qualifying the new version and porting their applications in a test environment that closely resembles their production environment. The final move of the new software to their production environment must be timed to minimize disruption of daily business.

A patch is an incremental change to released software, as opposed to a re-release of the entire product. When a patch is needed, the lab that created the code that needs to be changed is responsible for creating and testing the patch. Because patches generally provide much less change than operating system or application upgrades, customers can qualify and begin using patches more quickly.

Customers install HP-UX patches for a variety of reasons:

- to eliminate software defects in their installed operating system or applications,
- to obtain the software to drive a newly-released peripheral,
- to get support for an industry standard that postdates their released software, or
- to take advantage of other software enhancements such as new functionality[2] or performance improvements.

A patch warning is HP's way of notifying customers of a problem contained in or exposed by a patch. The patch warning names the patch and describes the problem that customers may encounter. Whenever a patch warning is issued, HP releases a replacement for the defective patch as soon as possible.

The HP-UX Patch Program is a cross-functional team whose responsibilities include releasing patches to customers, issuing patch warnings, setting and enforcing policies for patches, providing patch development tools for internal use, educating HP support personnel and

[1] UNIX is a registered trademark of The Open Group. The Open Group is the industry consortium that maintains the UNIX standard. For more information, see http://www.unix.org/ and http://www.opengroup.org/.
[2] An example of new functionality introduced through patches is the four-digit date handling that was required for Y2K. All customers needed this enhancement, but most could not afford to upgrade to a new version of the operating system to get it. Patches were the ideal delivery vehicle for this functionality.

customers on the best ways to manage patches, collecting patch metrics, and improving patch quality.

Introduction

Virtually every HP-UX customer installs patches. The customers' experience with the patches HP provides for the HP-UX operating system and applications has a significant impact on their satisfaction with HP-UX software, and therefore with their HP servers and workstations. Problem patches are at best annoying, and at worst have the potential to corrupt customers' data, introduce security vulnerabilities on their systems, or cause schedule delays and lost revenue. In other words, poor patches can negatively affect customer satisfaction.

In the late 1990's, it was clear that HP-UX patches were causing problems for customers. Patches were consistently near the top of the HP-UX support organization's list of issues. The feedback from customers emphasized that too many HP-UX patches were defective and that our competitors' patches had higher quality.

HP has always taken pride in delivering products of exceptionally high quality, so being compared unfavorably to our competitors got our attention. We launched a serious effort to reduce the number of defective patches released to customers. Root cause analysis was the tool we chose to help drive down the number of defective patches. It seemed like a good fit for us because it was an industry standard practice and had been widely used in HP to address other quality problems. We began to require a root cause analysis of each mistake that led to a patch warning. We expected that RCA would lead teams to make changes and that these changes would prevent them from repeating mistakes they had made in previous patches. This would cause a steady decline in the number of new mistakes introduced in patches.

Our Initial RCA Process

There are many ways to conduct a root cause analysis. Variables include the format of the investigation, who participates, when to conduct it, what information to gather, and what to do with the information. When we began requiring root cause analyses for patch mistakes in the late 1990's, we wanted the process to be lightweight in order to gain acceptance by those who would use it. At the same time, we wanted to make sure the analyses would yield useful information and actionable recommendations.

Our intent was to examine each patch problem, identify its root cause, and quickly make one or more changes to prevent that problem from happening again. Many root cause analysis processes use a different approach. They first gather a large body of analysis data and then use a prioritization scheme, such as a Pareto Chart [2], to decide which problems to solve first. This method is valuable when using RCA to evaluate the hundreds or thousands of defects found during the development of a large body of software. However, we felt that our approach was more appropriate for our problem. Customers and management were demanding quick and dramatic improvements in patch quality. We could not afford the time needed to accumulate a

large body of data before beginning to implement improvements. We needed a more nimble RCA process.

We required the people who had created each defective patch to conduct their own RCA. This seemed only natural because these people knew the most about the code they had modified, the documentation they had consulted, the tools and processes they had used, and any special circumstances that might have played a role. These people were best able to suggest changes that would be feasible in their environment. Furthermore, they were more likely to feel ownership for changes they proposed than for changes dictated to them by anyone else.

Information gathered in the RCA report included some basic identifying data as well as the answers to several fundamental questions about what had gone wrong:

- What defective patch or patches did the report describe?
- What was the date of the patch warning?
- Who wrote the report?
- What was the defect in the patch?[3]
- What allowed the defect to go unnoticed until after the patch had reached customers?[4]
- What could the patch creators do differently to prevent the defect from being introduced or released if the patch were being developed today?

We required the authors to submit their RCA to the HP-UX Patch Program within 30 days after the patch warning was issued. There were several reasons for this requirement:

- Having a deadline guaranteed that the RCA would not be postponed indefinitely.
- Conducting the RCA soon after the discovery of the problem in the patch improved the chances that the circumstances surrounding its creation were still somewhat fresh in the minds of the participants.
- Making the results available promptly helped the people developing the replacement patch to avoid making the same mistake.
- Implementing recommended changes as quickly as possible allowed other patch developers to benefit from them sooner.
- Collecting all RCA reports in one location allowed us to monitor completion and gather metrics.

We did not specify exactly how the required information should be gathered. Typically, the participants gathered in a conference room and held a face-to-face meeting to work through our standard template of questions. A project manager or lead engineer often facilitated the discussion. Other methods were possible, such as having one person interview each participant

[3] This was not simply a statement of the symptom reported in the patch warning, such as a core dump or a task failure. Instead, it was identification of the underlying design or code defect (such as an uninitialized variable or a memory leak) or patch construction problem (such as building the patch using the wrong version of a file).

[4] We were looking for answers like, “We failed to review all code changes,” or, “We reviewed all code changes but missed the fact that a loop had a bad termination condition.”

individually or compile written answers to the questions provided by each participant. However, there seemed to be synergy from having everyone together.

A program manager in the HP-UX Patch Program assumed responsibility for receiving all RCA reports, sending reminders for overdue reports, and publishing metrics about patch warnings and their root causes. Metrics compiled each quarter included:

- How many new patches did HP deliver to customers (overall and by lab)?
- How many patches received warnings (overall and by lab)?
- What was the ratio of patches warned to patches created?
- What kinds of mistakes caused the warnings?
- What were the most commonly detected mistakes?

Once the RCAs were submitted to the HP-UX Patch Program, we left it up to the authors to arrange for the implementation of their recommendations. Given our organizational structure, the HP-UX Patch Program did not have the authority to commit resources to most of the proposed changes or to enforce their adoption. We also did not track completion of the recommendations, but trusted that the suggested work would happen promptly.

Initial Results

After a year of requiring a root cause analysis for each patch warning, we saw roughly a 25 percent decline in the number of defective patches. This was good, but not the kind of decrease that we had sought. Furthermore, complaints from customers continued to come in at an unacceptable rate. It was clear that we needed to reduce our patch defect rate even further.

We still had faith in root cause analysis. We had seen it work before, both inside and outside HP. Instead of abandoning RCA, we turned our attention to determining what was wrong with our RCA process. In effect, we did an RCA on our RCA process.

Problems Found and Changes Made

Repeated Mistakes

The statistics we derived from root cause analysis reports on patch warnings clearly showed that our patch developers were repeating their mistakes. This led us to suspect many teams were not implementing the changes suggested in their RCAs. We confirmed our suspicion with a confidential survey of the people who had submitted RCAs. Survey responses revealed that a large percentage of recommendations were never implemented. Others were "in progress" for excessive periods.[5]

[5] Most often, there were resource allocation issues that prevented prompt implementation of the RCA suggestions. All of the labs creating patches were also working on new product development. Diverting resources to implement recommendations from patch warning RCAs could have compromised their committed development schedules. Fixed budgets precluded adding resources, so there seemed to be little choice but to postpone the implementation of the RCA recommendations. In some cases, the authors were familiar with a post mortem process and hadn't

Without betraying the confidentiality of the survey respondents, we reported our survey results to the lab managers. Most took little notice, although their managers were rating them on the quality of their labs' patches. A few saw an opportunity to improve their records with respect to patch quality. They were willing to take a chance that preventing future patch problems would yield savings that would pay for themselves. Some began reviewing all RCAs produced in their labs and approving the recommendations made. This resulted in recommendations that were more realistic. It also made it easier for teams to get the staffing and resources needed to implement their suggested changes. Others insisted that all patch RCA recommendations be implemented before the team that made the mistake released another patch. Since there was pressure from customers to get replacements for defective patches, this virtually guaranteed timely implementation.

By repeating our survey for several quarters, we were able to establish a statistically significant negative correlation between completion of RCA recommendations and future mistakes leading to patch warnings[6]. We publicized this result to the lab managers, whereupon several more immediately adopted the best practices of their peers. We still conduct our survey periodically to continue to remind the labs of the importance of follow-through.

The survey results also prompted us to change our RCA template to make it quite clear that we expected recommendations to be implemented. Instead of asking the RCA team what changes they recommended, we began asking them to enumerate their *committed plans for change*. We also asked them to *supply an owner and an expected completion date* for each committed change. We do not accept RCAs without this information. Some teams have taken this a step farther by tracking their RCA commitments in a defect tracking system.

Narrow Focus

We have found a number of opportunities for improving RCA results by expanding the horizons of the participants.

The teams conducting RCAs tended to take a very narrow view of what problem their recommended changes should address. For the most part, they looked at preventing the mistake that caused the patch warning from happening again in the exact same place in the code. As a result, the most frequently recommended change was the addition of tests, particularly specific tests to trap the particular problem that led to the patch warning. While bolstering the regression suites has been valuable, and in some cases very necessary, it does not address the entire problem. In particular, these specific tests rarely detect the same mistake or variations on it in other places in the code. Furthermore, tests do nothing to prevent the patch developer from making the mistake.

---

appreciated a critical difference between a post mortem and root cause analysis, as done at HP. In a typical post mortem, brainstorming generated many suggestions for improvements. Management later selected a few critical suggestions for implementation. In contrast, the team conducting an RCA was supposed to decide what changes to make and to arrive at a plan for making them.

[6] Simply stated, as completions went up, mistakes went down.

Rewording the questions in our RCA template has helped teams learn to look for more generally applicable solutions. We now ask them to think about what would prevent the given mistake *or a similar mistake* from escaping undetected in the future. We also ask them what could be done at each phase of the patch development lifecycle to prevent or detect the problem. This prompts them to think of solutions other than new tests. Now, when we receive an RCA report suggesting only the addition of a very specific test, we are likely to ask the team to rethink the possibilities and commit to additional changes with broader impact.

Our changes to the RCA template have caused teams to think differently about the possible changes they might implement. Some of the changes they have proposed and are now benefiting from include:

- Additions to checklists used in code reviews. Each addition to a checklist has the potential to improve many future patches.
- More careful selection of the participants in reviews. Most review teams now include a domain expert, and often, an internal customer of the item under review. These participants bring a depth and breadth of knowledge to the review that enables them to identify classes of defects that completely escape the ordinary reviewer.
- Enhancements to automated tools. Tools can often detect problems that even the most diligent reviewers miss[7]. Since our tools are used in the development of revisions and enhancements to the operating system as well as for patches, tool improvements have a big payoff.

Additional evidence of narrow focus was the tendency to propose only solutions that the team itself could implement. This is somewhat natural, since one team cannot tell another one, especially one under different management, what to do. However, sometimes the best solution requires a change from someone else. We addressed this by providing a place in the RCA report to suggest changes to be made by others and to document whether anyone has committed to making them. The HP-UX Patch Program can then exert some additional influence when necessary to arrive at agreements to make the proposed changes.

Changes Resulting from Central Review

Over time, we have seen the value of having one person review all RCA reports. First, the person who reviews all the reports can easily track which ones are overdue and remind the responsible teams to submit them. The reviewer can also evaluate the reports, asking for rework on the ones that are unclear or have missing data or inappropriate recommendations for change. These actions send the message that completing an RCA for a patch warning is not optional and not just a check-off item. We expect a report for each defective patch, and we expect serious thought to go into its preparation. If necessary, we collaborate with the RCA team to arrive at a satisfactory report. Requiring that each RCA meet our standards has helped the labs to understand their mistakes more fully. It has also provided the HP-UX Patch Program with more accurate data.

[7] For example, a code scanner can find "==" where "=" was probably intended.

Central review of the now improved RCAs has enabled us to see the trends that emerge from a body of data. Our metrics showed that, not only were certain mistakes occurring frequently, they were occurring in a variety of different labs. We were then able to work on common solutions to common problems.

For example, we discovered that many labs were releasing patches that did not install and remove correctly on some customer systems, even though the patch developers had tested installation and removal on machines available in their own lab. There was approximately one such problem per month. Digging deeper into the problem, we recognized that there were common reasons for the installation problems:

- The systems available in the labs for installation testing often did not represent a good cross-section of customer systems.
- It was difficult to arrange time on more systems because installation testing usually precludes doing any other work on the system.
- Dedicating additional machines to installation and removal testing in each lab was not cost effective because the machines would sit idle much of the time.

We staffed a team of engineers and equipped them with a variety of hardware and software configurations representative of systems our customers use. We chartered this team with testing installation and removal of all patches on all appropriate test systems and returning the results to the creating team within 24 hours. They have largely automated the testing that they do. Each patch development team is expected to perform one successful installation and removal of their patch using a system in their own lab, then to forward their patch to the central team for comprehensive installation and removal testing. Patches that fail the tests are sent back to the creating team, reworked, and retested before they are released to customers. We have achieved much better test coverage, high utilization of the test systems, and excellent value for the dollar. We have tested more than 3000 patches this way over the past three years, and only one installation problem has escaped.

Not testing the operation of the patch in a variety of configurations was another common source of patch failures. As with installation and removal testing, this was not so much a matter of negligence on the part of the patch developers as a resource problem. Each individual lab simply could not dedicate the amount of equipment or staff necessary to do comprehensive configuration testing. Our solution was to set up another dedicated team, equip it with a variety of test machines, load the machines with same applications our customers' systems use, and develop an automated suite of stress tests. The team tests all core operating system patches in this environment. Again, patches that fail these tests are sent back to the creating team for rework. This testing provides the opportunity to find patch interaction problems and increases the probability that each patch will operate correctly in customer environments.

Putting these dedicated test teams in place meant that more people were handling a patch before it was released. We came to the realization that these people should participate in the root cause analyses for each defective patch they had tested. We now require these teams to evaluate whether the failure was something they might have been able to detect and to make changes where appropriate. For example, in the case of the installation test failure mentioned above, the installation test team identified a gap in their testing and modified their processes to eliminate it.

Other common problems we have been able to identify and address include deficiencies in the code review process. As discussed above, we received one RCA that recommended careful selection of the participants in the code review. We recognized this as a best practice and encouraged it in all the labs. We have also promoted the practice of re-reviewing code once the changes suggested in the review have been made. Additionally, we encouraged the labs to use formal inspections [3], rather than less formal review techniques, when the code changes in a patch involve 200 or more lines or appear risky for a variety of other reasons.

While some mistakes occurred frequently in a variety of labs, we also observed that certain mistakes occurred repeatedly in some labs and not at all in others. Sometimes the absence of a mistake was a matter of having code that was not susceptible to the particular problem. However, in other cases, the apparent immunity was in place because the lab had already found and implemented a solution to the problem. By recognizing these patterns, we have been able to call on the labs to share specific best practices with one another.

Central review also revealed that very few of the defective patches had undergone any kind of beta testing by customers before being released for widespread distribution. We investigated why and found that our process for customer testing was so awkward that customers would rarely agree to evaluate a pre-release patch for us. In response, we simplified our process to make beta testing much easier on the customer. Customers are now more willing to test patches that have passed all of our lab tests for functionality and the central integrity, installation, and stress tests. Labs are free to select which patches customers should test, but they are encouraged to use customer testing for the patches that they believe are riskier than average. Our data show that patches tested by customers are less likely to fail after release than patches that have skipped this optional step. Customer testing appears to be reducing patch warnings by about fifteen percent.

Impact on Patch Quality and Customer Satisfaction

As we improved the questions in our RCA template, got more consistent follow-through on recommendations, and found trends through central review of RCA reports, we saw our patch quality increase at a steady rate. Over three years, we cut the number of defective patches in half.

Although we were proud of our accomplishment, we learned that many of our customers still had not noticed any improvement. Others had, however, so it was important to determine why this was. It turned out that the two groups were using different patch management strategies.

In parallel with the efforts to reduce the number of patch warnings, the HP-UX Patch Program had recommended some new strategies for management of patches on customer systems.[8] Customers who adopted our new patch management strategies found that they rarely had problem patches requiring immediate attention on their systems. They were able to postpone

[8] The details of the old and new strategies are not relevant to the improvements in our RCA process and are omitted in the interest of brevity.

most changes until their next proactive maintenance window. They consequently spent very little time in unscheduled maintenance. Our new strategies, together with the reduction in the number of defective patches, reduced the need for unplanned downtime to a level that these customers found acceptable. On the other hand, the customers who were still using our former patch management strategies found that, even when the work related to defective patches was cut in half, it was still a burden. These customers remained dissatisfied. Understanding this difference caused us to redouble our efforts to get customers to adopt our new patch management strategies.

Even though customers using the new patch management strategies no longer complain about HP-UX patch quality, we have reason to monitor our patch quality metrics and keep trying to reduce the number of defective patches we release. Monitoring allows us to identify emerging problems quickly and react promptly. Continuing to reduce the number of defective patches benefits not only customers, but also HP. In particular, HP can redirect the engineering effort needed to develop and test replacements for defective patches toward adding more functionality to products under development and getting them to market sooner.

Suggestions for Others

Getting Started with RCA

If you are not currently using root cause analysis to reduce your software quality problems, you have little to lose and much to gain by trying it. Potential benefits include not only better software, but also fewer customer complaints, more predictability in your new product development schedules, and lower job stress. You can begin by copying an existing process, or you may wish to design a process of your own. Either way, do some planning before you begin conducting root cause analyses.

Enlist management support for adding RCA to your toolkit. Sell the idea to management if necessary. We were quite fortunate that our upper managers were themselves under pressure to improve patch quality. They wanted us to be successful and were willing to support our efforts. It may be an uphill battle to implement process improvements without the blessing of those with the authority to commit people and resources to tasks. In addition to management support, you will need a change agent who can act on behalf of your management sponsors to influence or require teams to make changes.

Once you have management support and a change agent, determine realistic quality improvement goals, what metrics you will need [4], how you will collect them, and who will analyze them. After you decide on your metrics, you will easily be able to determine what data to gather and what questions to ask in an RCA. Also, decide what training will be necessary for participants, how and when to offer it, and how you will train new participants over time. Training does not need to be extensive. It may be as simple as a short explanatory document, but it should address how and when to conduct RCAs, how to determine the participants, and how the results will help the organization.

Evaluate your RCA process after you have results from a few sessions. Determine whether you are getting the kind of data you expected. Ask the participants to comment on how well the process worked for them. Making necessary adjustment early on will keep you from gathering a large body of unusable data and from prejudicing the participants against the idea of root cause analysis. After a couple of iterations of evaluation and change, your process will stabilize. Once you have had a stable process in place long enough to assess its effectiveness, you should consider tuning it if you are not making adequate progress toward your quality goals.

Tuning an Existing RCA Process

If you are using root cause analysis but are not seeing the quality improvement results you would like, try tuning your RCA process instead of giving up on RCA. Start by doing an RCA to determine your RCA process problems.

As you work on improving your RCA process, you may benefit from some of the lesson we learned. Your success may be enhanced by:

- Having a change agent review all RCA reports. The change agent should know enough about the underlying process to recognize when proposed changes are too superficial or too grandiose and have the courage to send reports back to the authors when their recommendations don't measure up. The change agent should also keep an eye out for opportunities that come from recognizing patterns in the reports.
- Verifying completion of committed changes. This is essential. Improvement comes from implementing change, not simply from identifying potential change.
- Involving the right people in the analysis. Be sure to include representatives from all teams that have dealt with the product or process you are trying to improve. Each such team has a different perspective and therefore different wisdom to contribute. We found that we were involving only some of the right people and were therefore overlooking possible improvements.
- Asking the right questions. Define the metrics you will need [4], and be sure you are collecting the data you will need to produce your metrics. In addition, ask whatever other questions are necessary to stimulate the participants to think about how and why the problem occurred and what can be done about it. Expect that you will need to refine your questions several times before you get what you really need from them.
- Addressing non-technical as well as technical problems. We found that some of our problems were caused by budget constraints, and we had to find ways to get the most quality for our money. Other possible non-technical problems are a need for training, a lack of commitment to quality, or conflicting time demands.
- Solving problems at the optimum level in your organization. You may find that a single engineer makes frequent mistakes with memory management. It is wasteful to retrain everyone because of this; train the person who has the problem. On the other hand, everyone may need training on a new testing tool that you decide to deploy based on your findings.
- Digging deep enough to identify the real root cause of the problem. Common wisdom says that you should ask "Why?" five times. Refuse to accept superficial answers.

As with any process improvement, expect to make some false starts. Use your metrics to help find them. Also, avoid making too many changes at once. It can be confusing to those trying to

execute your RCA process, and it can prevent your metrics from showing you which changes are effective and which are not. Use continuous refinement instead.

Finally, be sure you understand how long it will take to see the results of the changes you make. Be sure to communicate this to your sponsors. You do not want to reject a change before it has had the opportunity to prove itself. In our case, it often takes six months to a year for a patch defect to be detected. We needed to account for this when we began changing our RCA process and not assume failure when we did not see quick changes in our metrics.

Tuning your RCA process can improve the effectiveness of your analyses and yield quality improvements in your software. Give it a try.

References

[1] Rooney, James J. and Vanden Heuvel, Lee N. "Root Cause Analysis for Beginners," *Quality Progress*, July 2004, pp. 45-53.

[2] Simon, Kerri. "Pareto Chart," http://www.isixsigma.com/library/content/c010527a.asp.

[3] Humphrey, Watts S. Managing the Software Process. Reading, Massachusetts: Addison-Wesley Publishing Company, 1989.

[4] Balza, John. "Using Metrics to Change Behavior," PNSQC, 2005.

Focusing on the User Experience: Software Quality is More Than Fixing Bugs

Ryan S. Russell
McAfee
20460 NW Von Neumann Dr.
Beaverton, OR 97006
ryan_russell@mcafee.com

Don Hanson
McAfee
20460 NW Von Neumann Dr.
Beaverton, OR 97006
don_hanson@mcafee.com

ABSTRACT
Software quality is more than fixing errors, documenting code, and testing features. While these are vital for every software project, they are only a piece in the quality puzzle. Software quality is about treating everything from the moment the user opens the box to the time they shutdown the system as one. It is about creating an entire user experience which encompasses more than just the code.

This paper introduces the reader to three major components needed to create a quality user experience. These components are: awareness of quality user experience, building a quality user experience, and testing for a quality user experience. The goal of this paper is to provide a springboard into the world of researching, designing, developing and testing user experiences.

BIOGRAPHIES
Ryan Russell is an interaction designer at McAfee where he is working to establish a user centered design capability within the enterprise division. Prior to McAfee, Ryan worked as an interaction designer for Intel's User Centered Design (UCD) group. While at Intel he led the design effort on the joint Intel / Founder China Child Home Learning PC as well as exploration into health and wellness usage models that included pregnant women, new families, elders, and athletes.

Don Hanson founded his first company and began writing commercial software in the early 90's developing an evolving line of animation plug-in products. He has since worked on enterprise-level commercial software products and was the UI team lead for a mobile wireless navigation startup. Don is the development lead for McAfee's flagship enterprise security management product.

INTRODUCTION

When people discuss software quality, topics such as defect trackers, bugs fixed, build systems, and automated testing are usually at the top of the list. A quick *Google* search on software quality brings up:

- Quality Software – create and run automated test scripts easily.
- The Software Quality Page – a resource for testing, test planning, inspections, metrics and process improvement.
- Quality Software – implement full quality assurance, audits, control quality processes.

Contrary to popular belief, software quality is about more than these topics. Something much greater. Something that, if understood properly, can redefine a software product and possibly the company. The answer is *experience*. People don't buy software, they buy experiences. From *iTunes* to *Tivo*, today user's demand much more than a product with zero defects. They demand an experience that alters their emotional state, something that represents who they are and how they live their life[6, 7].

A quality experience is more than adding another feature or reducing the number of bugs. It is more than a longer help file or free technical support. It is about treating everything from the moment the user opens the box to the time they shutdown the system as a single unified experience. This unified experience encompasses more than just the code, but all of the product pieces that customers interact with.

This paper will introduce three major components needed to create a quality user experience:

- **Awareness of Quality User Experience**
  Be aware that customers experience many different pieces of your product. These pieces combine outside of your corporate infrastructure to become a single experience of your product.
- **Building a Quality User Experience**
  Building a quality experience involves four steps. These steps are: researching users, creating user object models, designing user experiences, and integrating design into the product life cycle.
- **Testing for a Quality User Experience**
  Test the product against the user object models, rather than just the code. Test for the unified user experience rather than each piece separately.

These techniques are not meant to replace current software quality activities but to work in conjunction with them to create a holistic user experience.

AWARENESS OF QUALITY USER EXPERIENCE

The first hurdle to overcome in creating a quality user experience is to understand what a quality user experience is. It is a fully integrated set of parts and services that work together to deliver a single, compelling experience. It is what the user feels and experiences from the moment they open the box or click the download button to the moment they shutdown the software at the end

of the day. A quality experience excites users while building customer loyalty and brand value[6].

There are many pieces that interlock together to form a software experience. They all must work in harmony to provide a quality experience. This is similar to sports teams. If all the teammates don't work together, it is difficult for the team to win and provide a compelling experience for the fans. Unfortunately for the user, a large majority of the software industry is not aware of this. This provides an exciting opportunity for companies to combine these pieces together to build an experience that will delight users and put the competition to shame.

Corporate Website

More and more software is being purchased over the internet. Often, the user's first experience with the brand and products occurs at the corporate web site. It is important for this first experience to be pleasant. If the simple task of purchasing and downloading the software is difficult, a user will be unhappy before they even start using the product.

Out of Box Experience

If the software is to be purchased from a retail store, it is important to create a positive out-of-box experience. Does the packaging attract user's attention and represent the experience waiting to be unlocked inside? When the user gets the product home will they be disappointed to uncover that the package contains a CD in an unprinted white sleeve and a piece of yellow paper stating that if they have *Windows 98* they must download a patch before installing the software? It is important to ask, "what is the experience we are trying to deliver?" and "how do we create an out of box experience that propels the user into it?" The user is not simply purchasing a CD in a box but rather an initial emotional experience that provides a launching pad for achieving goals with your product.

Adobe products such as *Photoshop CS* and *Illustrator CS* are packaged in beautiful boxes that just beg to be purchased and opened. Once opened the user finds everything they need to begin enjoying the experience. This includes is a printed manual and a quick start card, as well as the actual software.

Installer

The installer represents the first interaction with the software. This experience should represent what the product is capable of. If the installation process is difficult and frustrating, what is the user to expect when they actually use the product? An installer should do all the necessary work to set the stage for a quality user experience.

An important part of the installation experience is configuration. Many software products simply install the needed files, place an icon on the desktop and say goodbye. The installer should help the user configure the software so that the first time the interface is visible it is ready to be used.

McAfee's *Protection Pilot* software, a security management application, provides an experience that is tailored for the small business user. The installer collects the necessary environmental information from the user and then configures the application accordingly. When Protection

Pilot is launched for the first time, data is already present and the interface is waiting for the user engage it.

Help Files & Documentation
Help files and documentation are a critical piece in a quality user experience. The user needs information to understand how to use the product to accomplish specific tasks, and help when they get in trouble. Documentation needs to be integrated into the very fibers of the software, not bolted onto the side after the fact. Many of today's products provide links to a help document from the interface. This style of help requires tremendous work on the user's behalf to find the desired information. Instead, help should anticipate problems before they happen and provide information based on how the user is interacting with the product.

Documentation needs to be designed around actual usage of the product in the environment in which it will be used, not using the software in a clean lab. It must address the way the user will truly use the product, not the way an engineer intends it to be used.

A quality help experience should be proactive and assist the user, not get in their way.
Not only does good documentation provide a quality experience for the user but it should help lower support costs.

Technical Support
Technical support helps user's solve their problems which in turn allows them to achieve their goals and have a positive experience with the product. Companies should design their technical support experience to interact with the users in different ways, depending on their situation. For example, if a user is calling because they have a virus on their computer, they are most likely very worried about their data. In this case the first goal of technical support should be to make sure the user's data is safe, then correct the problem with their antivirus software.

Software should work in conjunction with technical support to provide a quality experience. For example, software can collect and submit information on the user's behalf and assist the technical support personal in solving the problem. Remember, the goal of technical support is to help re-enable the usage of the product.

Updates
Updates are yet another piece that must be considered when creating a quality user experience. Updates should enhance or fix a product, and do so in a way that doesn't interrupt or disturb the user experience. There is nothing more frustrating to a user than to have an experience disturbed to process an update. Users want to enjoy the software they purchased. They don't want to be bothered with unzipping files, running update installations, and rebooting their computing device.

User Interface
The user interface is the single most important touch point for a user. It is what the user interacts with to harness the technology underneath. A poor user interface will provide a poor user experience. No matter how advanced the technology, if the customer can't use it, it is of little value.

So how does the interface provide a quality user experience? Consistency is very important. Similar controls within different parts of the product should behave similarly. A consistent visual style (color, typeface, and icons) needs to be used throughout the product. The layout items on the screen should be consistent throughout feature areas of the software. The interface must provide the necessary information so that a user can comprehend how to use it through visual cues, styling, consistency, dynamic help and links to detailed help documents.

The interface should also communicate intended brand values and emotions. Both the interaction and visual design should work together to communicate this. For example, a security application should make a user feel confident, protected, and secure. Therefore information should be easy to understand in a crisis situation and colors like red should be used sparingly for only the most critical items. An instant messaging program should communicate connectedness, friendship, and possibly collaboration.

BUILDING A QUALITY USER EXPERIENCE

Once awareness has been achieved within an organization, the next step is building quality user experiences, not just a pieces of software[7]. This requires adding steps to the classic development process. These steps, if done properly, will yield quality software experiences. Every feature will contribute to meeting the users goals, needs, and desires. The experience will be intuitive and memorable. The integration of user interaction in the technology will be seamless, leaving the users wanting more.

Step 1: Researching Users

A traditional gathering of product requirements often asks customers, “What features can be added?” Unfortunately, users don’t know what they want. A list of features from them is not going to develop a compelling experience. They don’t uncover the insights into user’s behavior and culture. As more and more technology products become commoditized, it becomes increasingly difficult to differentiate a product based on price or features. To become a market leader, deep insights into the user population are required. It is through these insights that truly innovative software experiences can be created.

Ethnographic user research is the key to unlocking these insights. The goal of ethnographic research is to understand the user’s world from their point of view. This requires becoming immersed in the user’s culture, environment, language, and emotions. By seeing the world through the user’s eyes, it becomes possible to develop empathy for them and one can begin to design software that fits seamlessly into the user life.

Erving Goffman states that ethnographic research involves, “subjecting yourself, your own body and your own personality, and your own social situation, to the set of contingencies that play upon a set of individuals, so that you can physically and ecologically penetrate their circle of response to their social situation, or their work situation, or their ethnic situation [3].”

There are numerous techniques for collecting ethnographic user research, including interviews[9], observation, immersion, and cultural probes[4]. These techniques allow teams to uncover an incredibly rich collection of information such as:

- Social behaviors behind teenage communication
- Relationships and interactions between different job roles within an enterprise
- Environment and cultural factors causing low adoption of computers in Asia

The information collected from this research fuels the creation of quality user experiences. User research makes it is possible to know what users desire and the form they prefer the experience packaged in.

Step 2: Creating User Object Models

User research collects tens if not hundreds of pages of data. The next step in creating a quality user experience is to synthesis the research into forms, or user object models, that allow the data to be consumed by others. User object models provide representations of users, their actions, culture, and environment for product development teams (design, engineering, technical publications, marketing, and QA) to rally around.

User object models inform the designer whom to design for. It is impossible to create an interface without knowing the person that will use it and what they are trying to accomplish. User object models settle product requirements debates because all aspects of the software must be traceable to the object models. They also provide the data needed to build a user experience test plan.

Personas, a popular type of user object model, create a representation of the type of user that will use the product. They are derived from data discovered during user research. A persona should define a user's desires, goals, and dreams. Personas are not real people but composite archetypes of actual users[1]. Pictures to visually represent the user, their environment, and the products they use, should also be contained within a persona. This information works together to create holistic portrayal of the user [5].

Another type of user object model is a culture capsule[2]. A culture capsule is a physical re-creation of the user's environment that allows others to experience a pared-down version of that environment. The goal is to take the data uncovered in the research phase and re-create the same environment in which the users live or work. Depending on the project it may be a Chinese apartment or an information technology worker's office. Culture capsules provide inspiration for designers as well as an environment to test designs in. They allow engineers to experience and understand the environment in which their technology will be used. Culture capsules are also helpful in the testing phase, which will be covered in more detail later in the paper.

Storyboards are another useful user object model. They are helpful in communicating user workflows. They create a story that allows designers, engineers, testers, and product managers to visually understand the user in action. They are incredibly helpful in understanding how a user completes a specific task and what conditions they are operating under. Users might complete tasks one way during normal operation and another way during "emergencies."

The user object models created are important for determining the features and behavior of a product, communicating with stakeholders and colleagues, getting teams committed to the design, and measuring the effectiveness of the design. All aspects of a design should be traceable back to a user object model. If a feature can't be traced to a particular model, there is a high likelihood that it is not relevant for the product.

Step 3: Designing User Experiences

Once user object models have been created, the design team (interaction, graphic, product designers, and tech pubs) can use them to design a quality experience. User object models provide the necessary source of information and inspiration needed to create an innovative experience.

As discussed earlier the team must focus on more than just the features if they are to build a quality user experience. The design must encompass all aspects of the product from the out-of-box experience to the way the documentation integrates itself into the user interface[8]. A quality experience is not about individual features but the quality achieved through the sum of the parts.

It is important to take time early in the design phase to narrow the scope of ideas. It is very important to explore and investigate many different conceptual ideas that are inspired by the research. Many times it is the wacky ideas that eventually give birth to the designs that truly resonate with users. It is important to iterate on the concepts, continually refining them until a valid design direction is created. Getting feedback from users during these iterations is crucial. This feedback lets the design team know which ideas should be refined further and which ones should not.

After choosing a conceptual direction, it is time to define the interaction and information design of the product. Object models help designers determine how users will harness the technology behind the interface and ways in which information need to be displayed.

After finalizing the interaction and information design, visual design is then applied. The visual design helps to give the software a soul. It visually expresses the information and controls that users interact with. Visual design displays brand values such as safety and security or freedom and individuality.

Communication with engineering is vital in designing a user experience that is effective and can actually be built. A great design that can't be implemented doesn't help anyone. Remember, all features in the design must be traceable back to the object models. This traceability ensures the design is meeting the target users' needs and will fit into their life.

Step 4: Integrating Design into the Product Life Cycle

Integrating research and design work into the software product life cycle is vital in creating a quality user experience. The research and design work must come in a form that can be easily integrated into the development style (waterfall, iterative, etc) of the organization.

Ideally, integration takes the form of a design model delivered at the time a Product Requirements Document (PRD) is created. The design model walks through the entire product, screen by screen, detailing the interactions, information architecture, and visual arts (color, typography, and iconography).

Some time-to-market windows do not allow all the design work to happen up front. In this case it is important to supply user object models and preliminary design directions for integration into the PRD. By allowing the team to understand users and the high level design direction before implementation begins, they will be able to focus the code towards the correct usages of the product. Designs are then fleshed out in parallel with the engineering effort. It is important to schedule development iterations and design iterations so that the necessary design work is completed before the engineers are required to implement it. To be successful, this style of design and development requires a high level of communication between all team members.

Finding the right balance of providing the user experience design up front versus integrating it throughout the project is similar to the waterfall versus agile development question. It is different for every company and outside the scope of this paper.

Integrating the design requirements into the PRD provides a unified understanding of what is and what is not going to be built. The importance of this cannot be overstated. It helps remove ambiguity that is the bane of successful projects. Design requirements enable management to understand what will be built. Engineers have the information to design software that fulfills the desired user experience. Quality assurance teams have the information required to develop test plans that encompass the complete user experience. Marketing knows exactly what they will be selling.

TESTING FOR A QUALITY USER EXPERIENCE

Traditionally, quality assurance or test teams create test plans that exercise the code in a software product. They use metrics such as the percent of code coverage and the number of defects found. The goal of these tests is to determine if the requested features are functioning properly. These tests are not sufficient for delivering a quality user experience.

User Experience Test Plan

A user experience test plan should be created in order to test the software to see if it delivers the desired experience. This plan should test the software from a holistic point of view and include such items as the out-of-box experience, setup, interface, updating, and technical support.

The user experience test plan should be based on object models. Situations depicted in the object models should be turned into test cases. Using a security administrator for a small manufacturer as an example, information from the object model states that this user is interrupt driven and constantly busy because they are the only security professional in their company. They are constantly overwhelmed by phone calls and emails containing problems to be solved. From this information, a series of tests can be created to test software used by this user. An example test could be having a tester try to complete a task while interrupting them with an urgent task that must be completed first. Can a workflow be easily paused halfway through and resumed at a

future time? If the answer is no, then it is very likely that the user will not enjoy using the product.

Execution of the user experience test plan should take place in an environment that represents where the software will ultimately be used. It is not very valuable to evaluate user experience in an isolated lab environment. If software is to be used by a mobile road warrior, it should be tested in environments such as an automobile, hotel room, airplane, and conference room. If the software is used in a noisy environment, are the sound effects distinctive enough to be heard over all the ambient noise? This type of testing will uncover environmental issues that will impact the quality of the user experience. If business or development reasons do not allow you to test the experience in the actual environment, then using a culture capsule[1] is an acceptable alternative. While it is not the actual environment, it should provide a close representation that should uncover major usage problems.

User experience testing should also include politics and relationships. For example, software that is used interchangeably between parents and children needs to be experience tested for back and forth use by different people:

- Can usage by a child affect how the software will act when the parents come back to it?
- Is it easy for one user to corrupt another user's files?
- Can one user block an update that another user wants?

Only by creating and executing a user experience test plan will a team be able to ship a quality experience. By testing the software in situations and environments that represent the real world, the team will be able to uncover and correct issues that cause a poor user experience. Once the software has passed both traditional QA testing and user experience testing, it is ready to be shipped.

CONCLUSION

To customers, software is much more than the bits on a CD. It is about the experience it provides, the emotions felt, and fulfillment it delivers. As designers, developers, testers, and business people, we must focus on delivering quality experiences to the user. Times have changed. It takes more than just a set of technologically advanced features to be a successful innovator and leader in today's market.

Developing quality user experiences is new to the software community. Don't get frustrated. It takes time to get product teams and companies to shift their thinking to focus on the user experience. Sometimes it may seem like a long road, but it is definitely one worth traveling. By creating quality user experiences, customers will rejoice and the company will win.

ACKNOWLEDGEMENTS

This paper is dedicated to Patrick Y Wang, an amazing software developer and the best friend a guy could ask for. You will live in our hearts forever.

The authors would like to thank McAfee's technical publications team for their help in editing and refining this paper. Additionally we thank McAfee engineering for their support in creating and developing quality user experiences.

REFERENCES

1. Cooper, Alan. The Inmates are Running the Asylum. Macmillan Computer Publishing(1999).
2. Foucault, B. Russell, R. Bell, G. Techniques for Researching and Designing Global Products in an Unstable World: A Case Study. In *Proceedings for CHI '04 extended abstracts on Human factors in computing systems*. (Vienna, Austria, April, 2004). Available at: http://portal.acm.org
3. Goffman, E. Cliffs, N.J. On Fieldwork. *Journal of Contemporary Ethnography*, 18(1989), 125.
4. Gaver, B. Dunne, T. Pacenti, E. Design: Cultural Probes. *Interactions 6*, 1(1999), 21 – 29.
5. Goodwin, Kim. Getting from Research to Personas: Harnessing the Power of Data. Available at: http://www.cooper.com/content/insights/newsletters_personas.asp
6. Mayer, Bill. In Praise of the Purple Cow. *Fast Company*, 67(February, 2003).
7. Nussbaum, Bruce. The Power of Design. *Newsweek*, May 17(2004).
8. Peters, Tom. Re-imagine!. Dorling Kindersley Limited (2003).
9. Spradley, James P. The Ethnographic Interview. Wadsworth Group (1979).

Personal Six Sigma: Adapting Six Sigma to Professional Practice

Richard E. Biehl, CSSBB, CSQE

Data-Oriented Quality Solutions

RBIEHL@DOQS.COM

Author Biography

Rick Biehl is a consultant and trainer with Data-Oriented Quality Solutions in Orlando, Florida. Working for 27 years in information technology, he currently specializes in helping IT organizations integrate Six Sigma and CMMI® practices. He holds a master's degree in education from Walden University, and is certified by the American Society for Quality as a Six Sigma Black Belt and Software Quality Engineer. He currently serves as the Education Chair of the ASQ Human Development & Leadership Division.

Abstract

This paper adapts the tools and thought processes of Six Sigma to the personal practice of software engineering and software quality. Many information technology organizations are adopting Six Sigma practices as part of larger organizational quality initiatives. Software engineers and software quality professionals in these organizations can find it difficult to adapt Six Sigma techniques to their own professional practice because the emphasis of the organizational initiative is on organization-wide and project-level implementation. Individuals working in organizations that are not adopting Six Sigma can feel completely overlooked by the Six Sigma movement. This paper focuses on learning and adapting Six Sigma techniques at a personal level, incorporating these techniques and tools into one's professional practice regardless of whether or not Six Sigma is being implemented at the organizational level.

(This presentation does not presume any particular Six Sigma knowledge or expertise on the part of attendees.)

Introduction

As a quality movement, Six Sigma is about process capability. It emphasizes reducing the variation in a process, and increasing our control over a process, such that we can predict with considerable accuracy exactly how the process will behave. This level of capability can be used to implement improvements in the process where we set targets for future behaviors, and achieve those targets within the levels of quality control that we choose to design into the improvements. This perspective on Six Sigma applies equally well in both individual and organizational settings, and yet the history of the Six Sigma movement is almost exclusively a history of organizational adoption and change. That history, as it applies to individuals, includes obtaining certification as a Green Belt or a Black Belt, always within the context of an organizational Six Sigma program.

This paper discusses how Six Sigma can be adopted and adapted by individuals to personal settings. In particular, it highlights the adaptation of the tools and thought processes of Six Sigma to the personal practice of software engineering and software quality. Many information technology organizations are adopting Six Sigma practices as part of larger organizational quality initiatives. Software engineers and software quality professionals in these organizations can find it difficult to adapt Six Sigma techniques to their own professional practice because the emphasis of the organizational initiative is on organization-wide and project-level implementation. Individuals working in organizations that are not adopting Six Sigma can feel completely overlooked by the Six Sigma movement. You can adopt the Six Sigma thought process to enhance or improve an operational process that you are involved with, regardless of whether or not the organization in which you work has adopted Six Sigma as an improvement model.

This paper focuses on learning and adapting Six Sigma techniques at a personal level, incorporating these techniques and tools into your professional even if you work in an organization that is not adopting Six Sigma. It is beyond the scope of this paper to provide a full coverage of Six Sigma concepts. A brief primer of key Six Sigma concepts is provided in the Appendix, and recommended readings are listed in the Bibliography.

Six Sigma DMAIC

Large organizational initiatives and projects tend to get the most attention in the Six Sigma literature. The intent of most organizations adopting Six Sigma is that the tools and thought processes of Six Sigma will eventually become so embedded throughout the organization that Six Sigma simply becomes the normal way of conducting business for everyone in the organization. Using this perspective, the idea of defining and conducting Six Sigma projects targeted at improving certain processes or products – the current dominant model in the Six Sigma world – can be viewed as an immature application of Six Sigma. As Six Sigma programs mature, the specific improvement project will increasingly give way to continuous improvement of processes and products while they are being used.

Figure 1 – The Six Sigma DMAIC Lifecycle

Whether as a distinct project or an embedded activity, the Define-Measure-Analyze-Improve-Control (DMAIC) lifecycle of Six Sigma (Figure 1) is applicable to a large variety of situations. Its application as an implicit embedded tool is not without precedent. For example, when a project manager or systems engineer tailors a standard organizational lifecycle into a project lifecycle, the tailoring act is an improvement activity preceded by a definition of the project, measurement of risks and opportunities, and an analysis of options for mitigating and managing scope and risk. The resulting tailored project lifecycle is then used to control the effort. The tailoring process is an implicit DMAIC cycle.

The DMAIC thought process can, and should, be used to improve and control virtually any process, whether or not the organization in which that process occurs is aware of its use. When added to existing practices such as project tailoring, additional benefits and improvements will quickly materialize. The benefits are immediate and substantial when introduced into processes that do not already contain any implicit improvement capability. This is the basis for Personal Six Sigma.

Competency Improvement

When personally adopting Six Sigma to a professional practice, an obvious starting point is in the application of Six Sigma thinking, and the DMAIC lifecycle model, to the management of your professional career. The DMAIC lifecycle can be used to understand the elements of a human resource management system that affect your personal career choices and opportunities. (Figure 2)

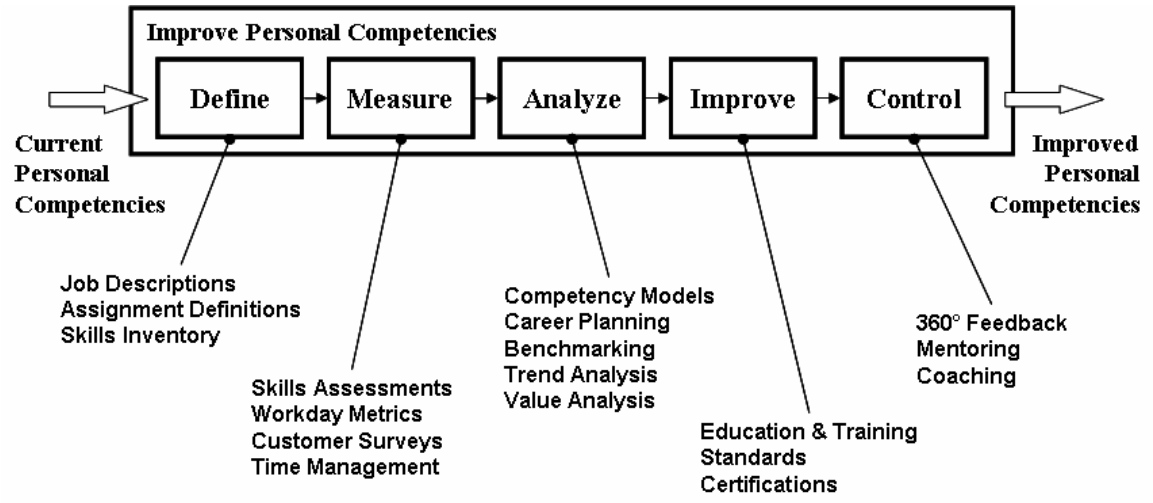


Figure 2 - Personal Six Sigma Process Map

The resulting model involves the process of improving personal competencies, where the key inputs are your current personal competencies, and the key outputs are your improved personal competencies. As a lifecycle, this process can be carried out as often as seems necessary. Initially, an arbitrary plan of conducting such an analysis annually might make sense. As the process matures, the aspects of the DMAIC Control Phase will determine how frequently you need to repeat the lifecycle.

Define Phase

The Define Phase of Personal Six Sigma requires that your job competencies that are to be addressed be clearly and accurately defined so that they can later be measured and improved. The inputs that drive this definition process are job descriptions, assignment descriptions, and a skills inventory. Define Phase activities are the most difficult to complete the first time Personal Six Sigma is put into practice. Subsequent passes through the lifecycle need only update materials created on previous iterations.

Job Descriptions

The first definitional activity requires establishing a detailed definition and model for your job and career path. This involves collecting or creating a job description for your current job, as well as detailing descriptions of any anticipated or desired jobs for the future. If you work in an organization that is good at creating and maintaining accurate and descriptive job descriptions, this activity is much easier than if you work in an organization that only offers vague or superficial job descriptions.

It is critically important that a clear and detailed description of your current and future jobs be defined as the baseline model for the DMAIC improvement cycle. Future job descriptions can be especially challenging because there may be multiple career paths that you consider possibilities, including the potential for future jobs that do not yet exist in the current job market. Your definition need not be clairvoyant to be useful; a best estimate of the future is all that is needed since this DMAIC cycle will be repeated over and over throughout the course of the unfolding of those jobs. Your current view of the competency requirements that will materialize in the future need only be accurate enough to guide effective plans or decisions in the current cycle.

Assignment Descriptions

Job descriptions tend to provide a relatively static definition of your professional responsibilities. An additional layer of definition is provided by systematically defining your actual job assignments. The assignments you carry out in your job provide a more dynamic view of your professional practice. For many, assignments will be dominated by project work. However, even when project work constitutes the bulk of your assignments, there are typically still many aspects of your job that would be omitted if only project work were included. For some professionals, project work may even constitute only a small percentage of their responsibilities.

Where feasible, assignment descriptions should be projected into the future. Being able to forecast the assignments that you plan to undertake in the future can be a key element in career and competency planning. The level of difficulty of these descriptions will depend upon the level of detail and breadth of the job descriptions captured initially. Some job descriptions also describe assignments in a fair amount of detail. However, many do not. Even with detailed job descriptions it can be difficult to get an accurate picture of how a working professional will actually spend her or his days. A complete combination of job and assignment descriptions provides that picture, and to the extent that it projects into the future, will enable more effective measurement activities in the next phase.

Skills Inventory

The third definition activity in the Define Phase involves cataloging an inventory of all of the skills that are implied by the current and future jobs and assignments defined previously. Depending on how they were written, the job and assignment descriptions might already explicitly lay out the skills required. Even so, there are likely to be additional skills that should be included in the skills inventory beyond those explicitly identified. This inventory should be as complete as possible. It is important to capture all of the skills associated with your job, not just those that might be listed in an official job description or assignment.

Measure Phase

The Measure Phase of Personal Six Sigma requires that an accurate and quantified picture of actual job, assignment, and skills performance be collected so that improvement opportunities that are well-grounded in the data can be identified. The inputs that drive this quantification process are skills assessments, workday metrics, customer surveys, and time management data.

Skills Assessment

A critical input to an understanding of professional competencies is an honest and thorough skills assessment. A skills assessment involves reviewing each skill in the skills inventory created in the Define Phase, and assessing whether or not you possess the required levels of skill. This measurement activity depends on establishing a scale for measuring competency levels, and for determining the likelihood that you will need the skills in the future.

For example, you might use a five-point scale, where "0" indicates that you have no knowledge of the skill, "1" indicates a general familiarity with the skills but no current real capability, "2" indicates some knowledge or capability usually as the result of some training, "3" indicates sufficient competency to practice the skill, "4" indicates experienced competency to the point of being able to help others with the skill, and "5" indicates that you have mastered the skill to the point where you can teach, adapt, or extend the skill.

If an item in the skills inventory requires a "4" but your current capability is a "3," then action might be needed to improve the competency. However, if future job assignments might only require the skill at level "3," then the need for immediate capability enhancement might be mitigated. The use of current and future job and assignment descriptions, coupled with the ever-changing profile of job skills across the industry, makes a skills model rather complex. Here in the Measure Phase, the emphasis is on building an accurate skills assessment, not making decisions about improvements. Improvement decisions need to be made based on an entire picture of your capabilities and plans, and will be the subject of the later Analysis Phase.

Workday Metrics

Each combination of job and assignment descriptions will have a unique profile of actual work activities that are conducted on a typical workday. Within this variation, however, it is important to be able to measure the actual activities performed. If you work in an organization that already collects time reporting data, these metrics will be easier to identify and collect. However, since the objective here is competency development, it will be important to collect data beyond traditional time-recording mechanisms. In particular, while job and assignment activities and time should be tracked, so should actual utilization of your competencies in the skills inventory.

Depending upon the specifics of your professional practice, you will count different things. Typical metrics might include timings and quality of requirements, system components, design artifacts, verification or validation items, or any other configuration items touched in the everyday execution of your job. Metrics should include the relative size of each artifact, time or effort expended, defect and issue data, rework time and scale, and skills actually used. Particular note should be made of any skills that you need to immediately improve in order to complete some activity.

Customer Surveys

A critical set of data needed for Personal Six Sigma is customer feedback. A challenge the first time through the cycle is to accurately identify all of the customers of your professional practice. We are often more accustomed to identifying our company's customers, or even the customers of our projects; but the customers of our professional practice are sometimes less visible to us. The customers of interest are the individuals or work groups that directly receive our work products. The list will vary depending on job assignments. For example, as a logical data modeler, your customers might include the requirements analyst for whom a data model is being developed and the data base designer who will eventually use that data model to define a physical data structure. Note that the customer of interest here is the direct and immediate customer.

Once your customers are identified, seek methods or occasions where direct customer feedback can be obtained. While this can be as formal as a feedback survey given to these individuals periodically, it need not be. Effective feedback can be obtained using less formal methods, including simple feedback discussions or a shared coffee break. Many such opportunities might already be available if an effective peer review program is in place in your organization.

Time Management

The fourth category of data to be collected during the Measurement Phase is time data. How do you spend your time? Useful categorizations include value-added versus non-value-added time, as well as time on or off task. Time spent off task might include a variety of meeting types, training, or other administrative activities.

This type of data most closely aligns with data that might already be collected for time reporting in your organization. Depending upon the level of project management maturity in the organization, much of the data required might already be available. However, such data is less likely to be available for non-project work. If sufficiently detailed, this data will identify the aspects of your job assignments that are being complete on-time and on-budget, or late and over-budget.

Don't limit your time data to on-the-job time. Pay particular attention to how you spend time outside of work hours. How much personal time are you spending for learning? How many professional journal articles are you reading per week? How much time are you devoting to professional society memberships? These time investments are strong enablers of career and competency growth that remain hidden if your measurement effort is limited to at-work activities.

Analyze Phase

The Analyze Phase of Personal Six Sigma requires that the metrics collected be reviewed objectively in order to identify trends and opportunities that can be addressed for improvement or growth. The inputs that drive this analysis process are competency models, career planning, benchmarking, trend analysis, and value analysis.

Competency Models

A comparison of your actual job description and assignments (from the Define Phase) to how often you actually use your competencies (from the Measure Phase) can serve as an indication of whether or not your competencies are being developed and used in ways that are consistent with your career plans and goals. If not, then analyzing the gaps between your plan and actual competency use can help identify places where your skills might need to be realigned with real job opportunities, or job opportunities might need to be rethought in light of how your competency development is actually unfolding.

Career Planning

While your organization often has a career planning model available, it is typically limited to career options within the organization. Depending upon your personal goals, you might need to develop a career plan that extents beyond your current organization. If the workday metrics, customer data, and time management data point toward career change opportunities or needs, then potential career improvement might entail a rethinking of the job and assignment descriptions that currently form the basis of this DMAIC cycle. Be particularly alert to new career opportunities that might be emerging that were not available or considered during the Define Phase.

Benchmarking

Envisioning a change is always easier when the target change can be visualized or observed. You should continually look for opportunities to measure or observe other professionals as they progress through similar career paths. Professional publications and conferences are excellent sources of benchmark data; many trade publications provide periodic salary surveys, and many professional societies provide detailed body of knowledge materials to members. All of these sources provide a clear picture of what a "might be" career target can include. This view can be used to help define a working path from your current "as is" model to some future "to be" model.

Trend Analysis

Pay particular attention to trends in the data, even if they are small. Picturing your job five years ago can highlight how much can change over the course of time, and yet those changes rarely stand out during very short-term discussions or observations. The influence of technologies, organizational changes, and skill models are constantly changing the way your organization functions, and the ways you experience being a part of your organization. By noting trends in the data you can spot areas that need improvement long before your supervisors or peers do. Such trends might include the frequency with which your skills are used, or shifts in performance data related to certain types of artifacts meeting specifications. By planning for improvements in these areas, it is possible to prevent anyone else from ever observing slippages in your skills sets or performance.

Value Analysis

A particular form of analysis, perhaps viewable as a subset of trend analysis, is value analysis. What value are you providing to your customers over time? Ideally, there should be significant trends in the conducting of value-added activities that directly impact customers. On-time performance and skills utilization should continuously improve over time. As a Personal Six Sigma goal, these value measures should be improving

notably faster than corresponding value measures for your entire organization. If not, you should work to identify reasons why not, and think about ways to improve such situations.

Improve Phase

The Improve Phase of Personal Six Sigma takes the information gleaned from analysis to directly implement improvements. The inputs that drive this improvement process are education and training, standards, and certification.

Education & Training

The most obvious improvement opportunities will involve forms of education and training. These might vary from very focused training opportunities to improve specific skills, to full academic degree programs. Long-term planning will often involve both extremes. Professional conferences or professional society involvement can also provide for skills enablement or expansion. Simply reading publications from the many professional societies that cover aspects of your job can be an excellent improvement strategy.

Many other educational opportunities are likely to be available all around you in your current position. The many committees, task forces, or special assignments available in your workplace offer excellent opportunities to develop skills that might otherwise not be needed or practiced in the day-to-day activities of your job. The question shifts from whether or not you have the skills to participate, to whether or not participating will help you develop needed or desired skills. Very often these activities present opportunities to develop specific skills or relationships that would rarely be developed through your normal work assignments.

Standards

Standards are an important part of planning improvements. The information technology and systems engineering fields are rich in standards that cover every aspect of IT products, services, and processes. Identifying and adopting standards is the easiest way to improve the direction of your skills and career without needing to reinvent the wheel. Adopting a standard can help even if the organization in which you work is not moving in the direction of that standard. (This paper is an example of such an adoption of Six Sigma if your organization isn't pursuing Six Sigma as an organizational strategy.) Many standards and guidelines are available through ISO, ACM, IEEE, SEI, PMI, and many other related organizations.

Certifications

As an extension of education and training, obtaining professional certification can be an important component of any competency improvement program. The body of knowledge for a certification program serves as a standard for the purposes of defining what needs to be learned. The examination process for the certification serves as an important benchmark for the level of competency that should be developed and how it should be measured. Because most certification programs also require some form of periodic recertification, they also serve as good initial components for the Control Phase.

Control Phase

The Control Phase of Personal Six Sigma works to institutionalize the improvements made, and assure that new cycles of DMAIC activity are driven by the data generated through the improved processes. The inputs that drive this improvement process are 360º feedback, mentoring, and coaching.

360º Feedback

Just as the Measure Phase includes soliciting customer input, the Control Phase also ties actual performance to stakeholder feedback. Data should be continuously collected from customers, suppliers, peers, and managers. Shifts in this feedback over time might represent signals that something has changed in the work environment, or your performance has shifted. In either event, such feedback can serve as an early warning that problems might be occurring that can be addressed through another iteration of this DMAIC lifecycle.

Mentoring

To obtain more future-oriented feedback, it is important to identify one or more mentors, and to enter into mentoring relationships with those individuals. A mentor can often help you identify trends or opportunities that affect your future plans long before the details behind those trends become self-evident. In the most extreme case, such information might trigger a replanning of your job or career options that would necessitate a complete redefinition of future job, assignment, or skill opportunities.

Coaching

Try to identify individuals in your organization who can serve as coaches on a day-to-day basis. Coaching provides an additional control over whether or not things occurring in the environment are being properly interpreted and used to influence plans over time. Again, the emphasis is on using the coaching relationship to identify data or thought shifts that would necessitate a recycling through this DMAIC lifecycle.

Technical Performance

Beyond the improvement of personal competencies associated with job and career planning, Personal Six Sigma also involves the continuous improvement of your technical performance within your job. Six Sigma, as a quality improvement paradigm, can be adapted to any disciplinary setting to make it more domain-specific. The use of Six Sigma within Information Technology presents specific opportunities to tailor Six Sigma thinking to the information technology domain. Within such specialization, IT professionals will put their Personal Six Sigma practices in place. (Figure 3)

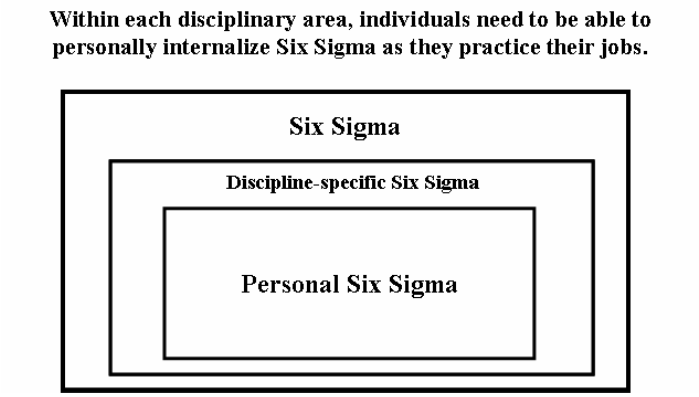


Figure 3 - Personal Six Sigma occurs within discipline-specific contexts

In the context of Personal Six Sigma, there's no presumption that your organization has adopted Six Sigma. The focus here is on personal adoption of these tools and practices in order to become a stronger and more valuable professional. Although the discussion below might make it sound like your organization needs to adopt these tools, that isn't the intention of this discussion. This discussion focuses on you, as an individual, learning and using Six Sigma tools and techniques as part of your job even though your organization hasn't asked you to. The results you achieve might leave those around you wondering how you've become so productive, and why your deliverables seem to so easily address problems and concerns typically seen in the workplace.

As a professional, you can map your own job into the Personal Six Sigma framework in order to decide how and where to incorporate Six Sigma tools and thinking into your everyday practice. You might decide to incorporate Six Sigma into every aspect of your job, or else into only limited areas where the opportunities seem to be greatest, such as system or software development. (Figure 4)

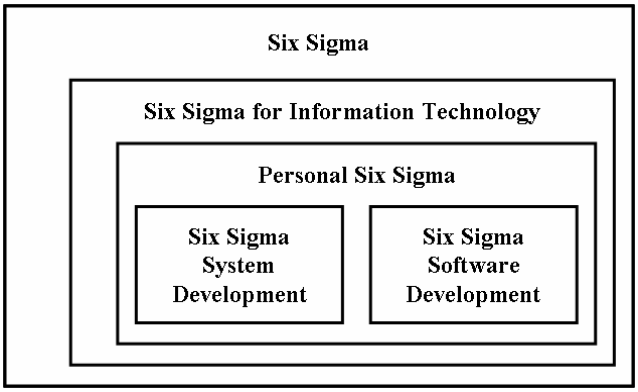


Figure 4 - Personal Six Sigma frameworks for software professionals

Choosing where to apply Six Sigma in your job will be determined by how much time and effort you can put into reading Six Sigma books, browsing the web for useful materials, and practicing your new skills on your projects. Because you are approaching this as a personal goal, you'll also be able to adapt and apply your new skills in setting beyond the workplace, include using these tools in your home and community activities. For this discussion, we're focusing on applying these skills in the workplace.

Within each disciplinary framework, you can adapt Six Sigma to your professional practice in ways that seem most beneficial, blending the specifics of each technical discipline with the available Six Sigma tools and techniques to find a combination that works most effectively for you. Please keep in mind that Six Sigma adds value to any discipline, but can never replace or supersede the specifics of the discipline itself.

Many Six Sigma organizations get themselves into trouble because they overemphasize Six Sigma tools, expecting them to take on a life of their own as replacements for previously used tools. Such organizations are often observed to say that "every project should be a Six Sigma project," implying that every project would begin to use Six Sigma tools as their primary deliverables instead of the traditional domain-specific tools used previously. Instead of advocating such an approach, Personal Six Sigma simply involves gaining an understanding of how each traditional deliverable maps into Six Sigma thinking so that the appropriate heuristics of Six Sigma can be applied to improving any deliverable. (see Figure 5)

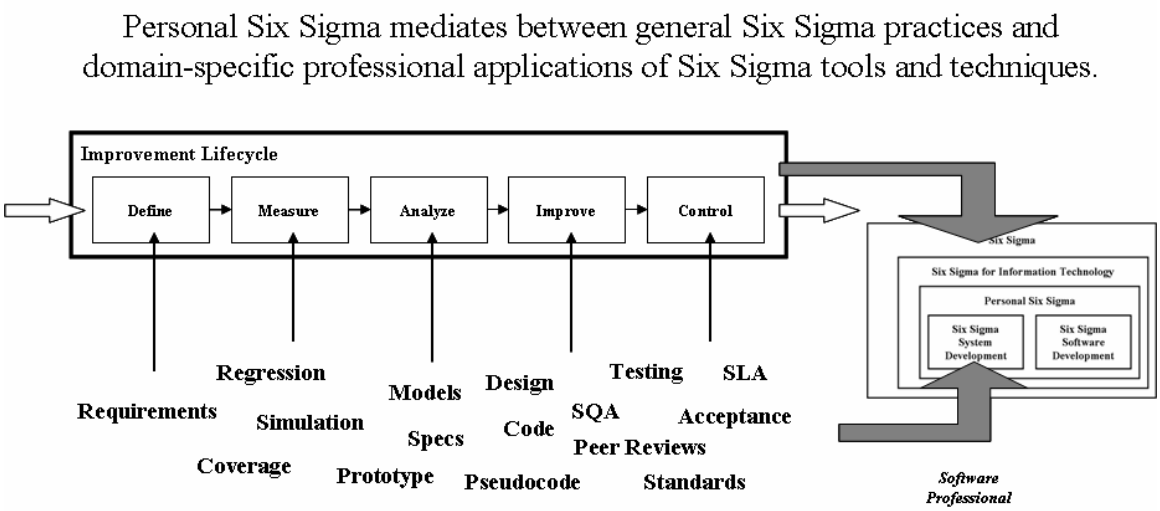


Figure 5 - Relationship between Six Sigma and domain-specific tools in software field

By mapping traditional deliverables against the DMAIC lifecycle, each deliverable can become the focus of a Six Sigma improvement thought process. By understanding where each deliverable fits into the DMAIC thought process, deliverables can be evaluated against a broader array of criteria than available previously. Also, in a full review of the deliverables based on the DMAIC process, it becomes possible to spot weaknesses in the overall IT lifecycle approach.

The synergy of the domain-specific lifecycle and tools with the general DMAIC lifecycle of Six Sigma provides added-value for IT projects and professionals. While the value is maximized in organizations that have adopted Six Sigma, Personal Six Sigma supports the creation of some of that value at the level of each individual contributor. If you are new to Six Sigma, you will find three particular tools most useful:

1. Process Mapping – This toolset helps you understand the requirements for process components and flow, including a characterization of each input and output for each process step. It has several analogs in the traditional IT toolset that can be improved using process mapping analysis. As you advance, you'll also learn a specialized process map that defines top level stakeholders, the Supplier-Input-Process-Output-Customer (SIPOC) tool.
2. Failure Mode & Effect Analysis (FMEA) – This toolset, with its own rich history outside of Six Sigma, helps define and prioritize the aspects of the process maps that require further analysis and the introduction of controls. While an FMEA analysis can look for problems anywhere within the scope of a project, your initial emphasis should be on understanding how process map outputs can fail as a result of problems with the inputs. The correction of these problems is always within your project scope. You'll move on to failure modes in the inputs themselves, but the causes of these failure modes will always be outside of your project scope, and therefore, will be more difficult to address.
3. Process Capability – This toolset provides a mechanism for describing the performance of a process that is independent of what the process actually does or how it is defined. Its results can be understood by anyone who has learned the basic logic of Six Sigma. Although usually treated in the literature as a statistical subject (e.g. calculation of Cp and Cpk ratios from control and specification limits), process capability can be handled simply by sketching a "desired" and "actual" performance curve. A visual comparison of the two curves gives a good picture of the process performance. If the actual performance is wider than desired, the process is not capable. If the actual performance is off-center from the desired target, the process is not capable. Your learning will move on to the more complicated statistical calculations at a later date.

Of these three tools, Process Capability is the key to becoming a Six Sigma thinker. It forces you to define the metric that is being displayed in your performance curves. This critical-to-quality (CTQ) metric is what makes your analysis more quantitative. Your process maps and FMEA analysis then focus on the critical metric. Without the metric, process maps and FMEAs become useful analysis tools, but they aren't Six Sigma. Six Sigma is a quantified discipline that focuses project attention toward key requirement metrics.

Process maps, FMEA, and process capability constitute your Six Sigma starter kit. You will add others as you progress through the available literature, attend workshops or classes, and practice, practice, practice. Don't worry about using the tools incorrectly. Your emphasis should be on using the tools to improve your performance on your projects. Whether or not the tools are used "correctly" is an academic discussion for another time. Using the Six Sigma tools will help you recognize problems in your traditional IT deliverables. Fixing those problems will trigger additional Six Sigma analysis. You should continue to cycle between Six Sigma and traditional analysis until the cycle stops adding value.

Quality Function Deployment

As you advance in your Six Sigma studies and learning, you will continually add tools that can be used to deepen your understanding of business processes, and enhance your ability to use your traditional IT tools. While there are many ways to successfully add Six Sigma tools to your personal toolset, one highly integrated approach is through the use of a core tool in Design for Six Sigma methodologies: Quality Function Deployment (QFD). (Figure 6) [See (Cohen, 1995) for an effective overview of QFD.]

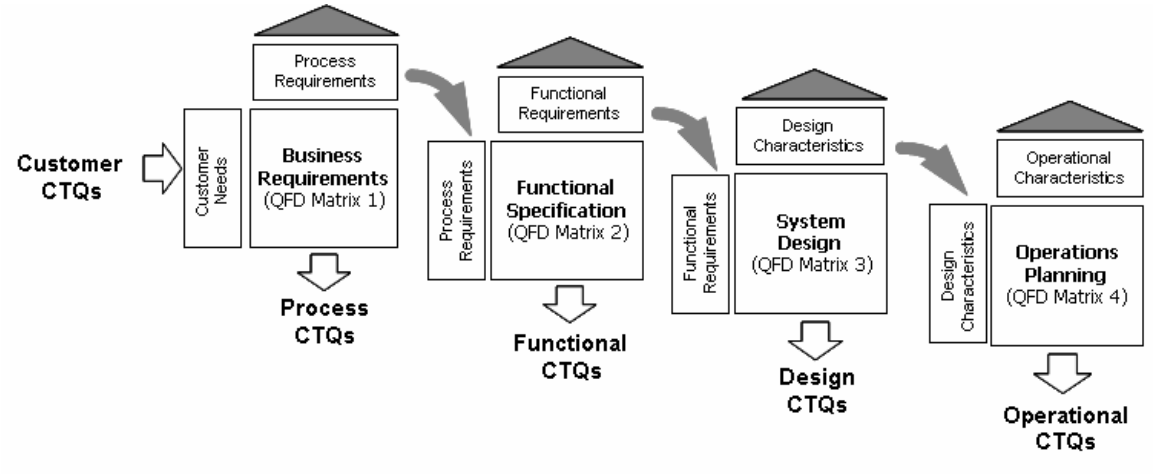


Figure 1 - Four common levels of QFD in IT

Again, note that Personal Six Sigma is not about having your organization adopt and use these tools. It focuses on you developing skills in the use of Six Sigma tools so that you can personally apply them in the workplace. As you read about QFD below, keep in mind that you'll be using it to help improve how you do your job while creating your standard deliverables that you already create on every project. Mapping your current deliverables into the Six Sigma QFD framework will simply help you do a better job in producing those deliverables in the shortest time, at the lowest cost, and with the highest quality levels. Nobody else needs to know that you are using Six Sigma (although you're likely to want to share that fact with anybody who will listen!).

QFD is largely about trying to relate and prioritize different layers of your projects. A four-level QFD can be thought of as encapsulating Requirements, Functional Specifications, System Design, and Production Implementation. Different aspects of the tools your projects are already using will map into these layers. Even if not used as a requirements management tool on a project, QFD offers a structured framework for thinking about how different domain-specific IT tools and deliverables map into the Six Sigma thought process. (see Figure 2)

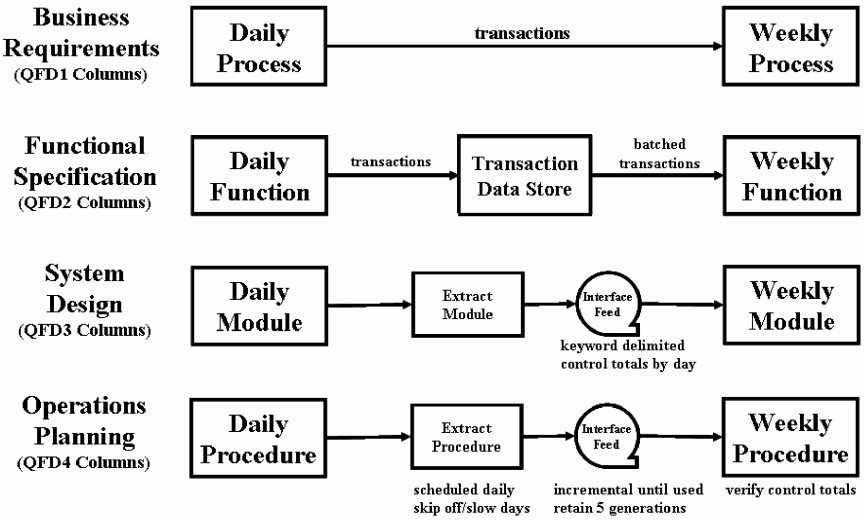


Figure 2 - Example of IT use of four QFD levels

For example, any IT artifact that captures business requirements regarding what processes, inputs, or outputs are needed, including timing and criticality of those inputs and outputs, would correspond to the business requirements in the columns of the first level of QFD matrices (QFD1). Since Six Sigma places the process map at this first QFD level, you'll know to use the verification and validation techniques associated with process maps on the specific requirements deliverables you create.

Many other IT analysis and design tools that you already use can also be mapped into the four-level QFD model. The location to which elements of each tool fit into a QFD matrix will determine which Six Sigma tool heuristics can be used to improve the content of your IT deliverables. Common examples include:

- Use Cases – The actors in your use cases will have customer needs recorded in your QFD1. The actors and use scenarios that you define become a combination of business process columns in QFD1 or functional specification columns in QFD2. Care should be taken to assure that the columns are defined as a hierarchy where standard and exceptional scenarios are included under the use cases that define them. Although actors aren't always thought of as functional components, their columns in QFD2 allow security and authorization rows to be better mapped.
- Logical Data Models (Entity-Relationship/Class Diagrams) – The entities, classes, relationships, attributes, and domains in your data models all become columns in QFD2. It is important to assure that the columns are defined as a hierarchy where attributes and domains are included under the entities or classes that define them. The QFD correlation matrix (e.g., the House of Quality "attic") should be used to record correlations between these data columns and any related functional steps or data structures that impact the data (often using the Create-Read-Update-Destroy designations to indicate the correlation).
- Data Flow Diagrams – The agents, processes, data stores, and junctions in your data flow diagrams also become columns in QFD2. The QFD correlation matrix should tie these functional steps to any associated data columns.
- State Transition Diagrams – The states and transitions in your state transition diagrams also become columns in QFD2. There should be correlation matrix entries to connect each state transition to the functional step responsible for the transition. There should also be correlations defined between the each data state and the associated entities, as well as between each process state and the associated functions.
- Customer Interviews - Customer input belongs in different levels of the QFD depending upon the type of input being collected. Many mistakes are made here, so it is important to pay particular attention to where such data generalizes into your QFD. Not all customer input is a need. If everything the customer says is interpreted as a need statement (row of QFD1), the resulting mess is completely unmanageable, and QFD fails. Customers talk about their needs (rows of QFD1), their processes (columns of QFD1), the functions they are hoping you will build to satisfy those needs (columns of QFD2), some specific design features they are looking for (columns of QFD3), or even what type of training or support they expect (columns of QFD4). The challenge is to put what the customer says into the correct QFD position, and then assure traceability back to the project charter via QFD1. Not everything the customer says will be a new QFD row or column; much of what is said will provide quantification of targets and specification limits for rows and columns that have already been identified in the QFD.

Likewise, deliverables associated with system design (including prototypes, data structures, screens and forms, and program code and scripts) correspond to the columns of QFD3. And finally, deliverables associated with production implementation (including operations plans, training materials, documentation, backup-recovery tools, support scripts and guides, and other operational details) belong in columns of QFD4. Since every row and column of a complete QFD will include measurement criteria for the content of the row or column, building test plans is a natural by-product of using a QFD to integrate the four levels of analysis and design.

Mapping these traditional IT tools that you probably use everyday into the Six Sigma framework defines the points at which Six Sigma will provide immediate value to your personal practice. Learning the basic Six Sigma tools and heuristics will help you improve the actual IT tools and deliverables that you are already using, even if no one ever sees your completed Six Sigma tools. This conceptual mapping provides a mechanism for you to take advantage of your own self-learning of Six Sigma tools and techniques to continuously improve your own technical performance. If eventually coupled with an organizational commitment to Six Sigma, the benefits achieved can be even greater.

SEI Personal Software Process[SM]

Individuals who have had exposure to the SEI Personal Software Process[SM] (PSP) [Humphrey, 1995] will recognize parallels between the PSP model (Figure 8) and the Personal Six Sigma model discussed here. PSP is a process improvement model to be used by an individual practitioner, just as Personal Six Sigma applies to an individual practitioner. The benefits obtained through PSP can be increased if the organization in which the practitioner works also adopts the SEI Team Software Process[SM]. [Humphrey, 2000] While TSP is not a prerequisite to PSP, its use increasingly improves performance. Likewise, Personal Six Sigma improves the performance of the professional who adopts it, and those benefits can be compounded if used in an organization that fully adopts Six Sigma.

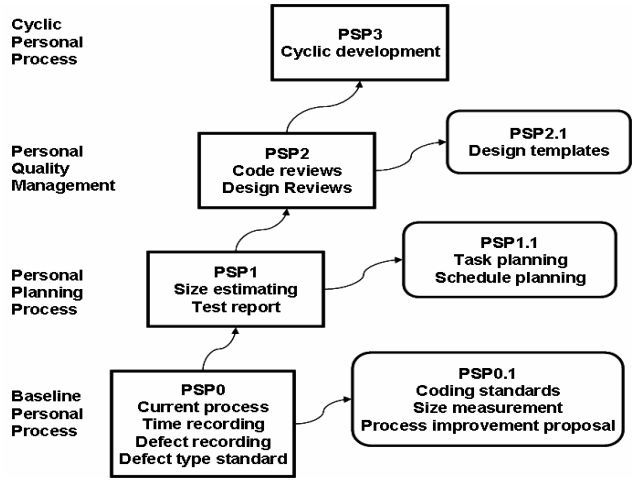


Figure 8 - Personal Software Process [adapted from Humphrey (1995)]

Beyond these parallels, the PSP model offers important insights for practitioners working to adopt Personal Six Sigma. The PSP model is incremental and layered. Beginning with core basic processes at PSP0, the PSP model grows in discrete stages toward cyclic development at PSP3, and encourages practitioners to adapt the model incrementally as confidence and skills develop. The same recommendation can be adapted to the deployment of Personal Six Sigma. You should begin with the basics of Six Sigma – most likely simple process maps, failure mode analysis, and basic process capability measures. In fact, practitioners of PSP will most likely begin their Personal Six Sigma adoption using the procedures and metrics that have been built during the initial implementation of PSP.

Conclusion

Personal Six Sigma offers a framework for you as an individual practitioner to adapt Six Sigma techniques and tools to your personal professional practice. Applications include using Six Sigma to model and improve your personal career path and opportunities, including an improvement cycle that can be repeated over time to help assure that your career is unfolding in the way you desire.

In addition to improving your personal career competencies, major opportunities exist for improving your technical performance in your job through understanding the mapping of Six Sigma processes and techniques against the processes and techniques of your work domain. Whether or not you ever actually create any of the common Six Sigma tools, you'll benefit from adapting Six Sigma thinking to the domain-specific tools you already use. As your knowledge of Six Sigma increases, you will increasingly use Six Sigma tools to supplement your domain-specific tools. The complementary nature of these tools will boost both your productivity and effectiveness. You will probably not spend as much time using Six Sigma tools as you typically spend using your domain-specific IT tools. That path is best reserved for individuals who choose the Six Sigma Black Belt career path. As an IT practitioner, your roots will remain in IT while you develop additional Six Sigma skills to assist you in continuous improvement.

References

Cohen, Lou (1995). *Quality function deployment: How to make QFD work for you*. Reading, MA: Addison-Wesley

Humphrey, Watts S. (1995). *A discipline for software engineering*. Reading, MA: Addison-Wesley.

Humphrey, Watts S. (2000). *Introduction to the Team Software Process*. Reading, MA: Addison-Wesley.

Bibliography

These books provide the basics…

Brue, Greg (2002). *Six Sigma for Managers*. McGraw-Hill.

Chowdhury, Subir (2001). *The Power of Six Sigma*. Dearborn.

Chowdhury, Subir (2002). *Design for Six Sigma: The Revolutionary Process for Achieving Extraordinary Profits*. Dearborn.

Larson, Alan (2003). *Demystifying Six Sigma: A Company-Wide Approach to Continuous Improvement*. AMACON.

These books are more intermediate, and tend to focus on narrower perspectives…

Tennant, Geoff (2001). *Six Sigma: SPC and TQM in Manufacturing and Services*. Gower.

Ehrlich, Betsi Harris (2002). *Transactional Six Sigma and Lean Servicing: Leveraging Manufacturing Concepts to Achieve World-Class Service*. St. Lucie Press, 2002.

Eckes, George (2002). *Six Sigma Team Dynamics: The Elusive Key to Project Success*. John Wiley & Sons.

De Feo, Joseph A.; & William W. Barnard (2004). *Juran Institute's Six Sigma Breakthrough and Beyond: Quality Performance Breakthrough Methods*. McGraw-Hill.

Tennant, Geoff (2002). *Design for Six Sigma: Launching New Products and Services Without Failure*. Gower.

Snee, Ronald S.; & Roger W. Hoerl (2002). *Leading Six Sigma: A Step-by-Step Guide Based on Experience with GE and Other Six Sigma Companies*. Prentice Hall.

George, Michael L. (2002). *Lean Six Sigma: Combining Six Sigma Quality with Lean Speed*. McGraw-Hill.

Appendix: Six Sigma Primer

This Primer provides a very simple introduction to some core concepts of Six Sigma. It is intended for novice readers that have not yet had any Six Sigma training or experience and want a basic background into the core concepts of Six Sigma centering on variation and control. Individuals with Six Sigma project experience will find this material too superficial, and may even spend more time noting the obvious omissions than concentrating on the materials.

Brute Force Quality

Historically, quality improvement was carried out as a management-dictated process of applying brute force effort to particular quality problems. For example, management might set a goal of reducing backorders in an order processing environment by 50%, from a current state of 24% to some target of 12% or less. This goal would drive an effort to attack the problem, making changes throughout the problem area and observing the impact of those changes on the targeted measures. After a time, the backorder rate would be seen to have been lowered to some value at or below the targeted 12%, and the improvement program would be declared a success based on that result. (Figure 9) However, the actual process behavior would still vary considerably.

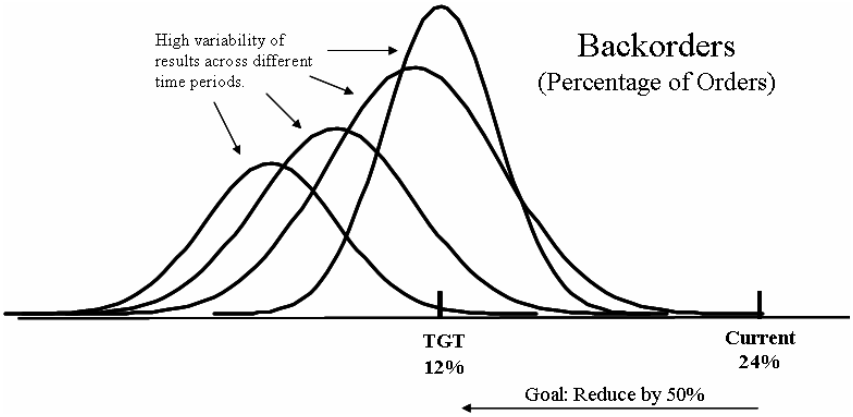


Figure 9 - Wide variability of brute-force outcomes

Problems with the brute force approach were numerous, but centered on the fact that such efforts often focused on incorrect or inappropriate solutions, and the solutions themselves were not usually sustainable. Recognition of these weaknesses caused a shift toward more systematic approaches to quality improvement. Collectively, these approaches came to be known as Total Quality Management (TQM).

Total Quality Management

TQM involved an expanded use of Statistical Process Control (SPC). The effects of SPC could be seen in two key areas: 1) processes were expected to exhibit variation around an average value, but the variation attributable to the process was expected to remain within certain expected ranges (the control limits), and 2) what a customer wanted from a process (the specification limits) wasn't necessarily the same thing as what a process would actually be observed to do. When a process was operating outside of its specification limits, it was said to be producing defectives. When a process was operating outside of its control limits, it was said to be out-of-control.

An out-of-control process is a signal that something is wrong with the underlying process, and that it should be addressed using the methods and tools of TQM. In this way, the SPC analysis tells us both where the problems are (producing defectives outside the specification limits) and whether or not we could cost-effectively fix them (out-of-control process behaviors indicating special causes that can be identified and corrected).

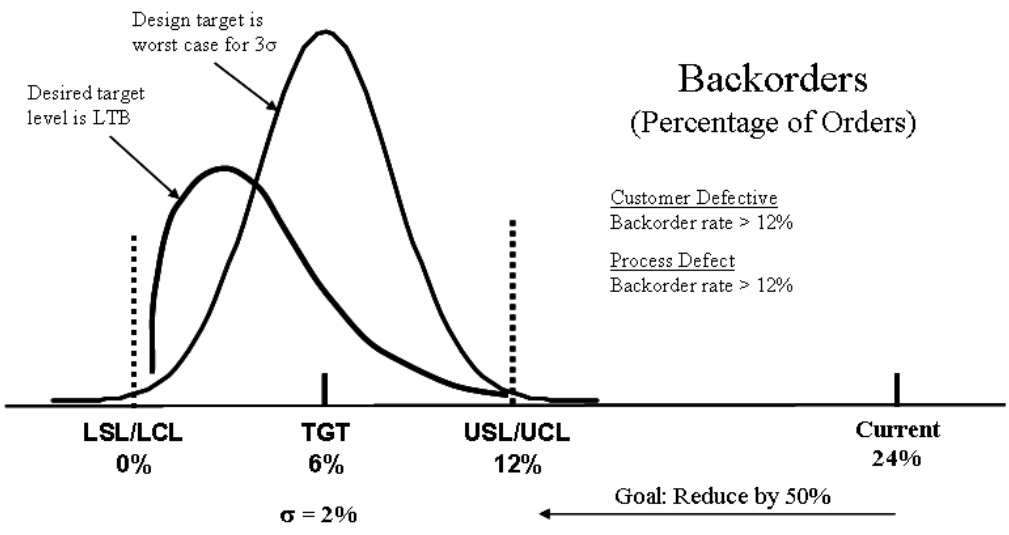


Figure 10 - TQM controlled processes to operate within specifications

Figure 10 illustrates the backorder problem using basic SPC thinking. The original target becomes the upper specification limit (USL) of the desired new process. The objective of the design will be to build a process that doesn't result in a backorder rate higher than this value, making the design target upper control limit (UCL) also 12%. Presumably the backorder rate should be reduced as much as possible (the lower-the-better, or LTB), and so the lower specification limit (LSL) and lower control limit (LCL) are both set to 0%. The target value for the process redesign is typically the mid-point between the two specification limits, or 6%. The new process is intended to deliver a backorder rate of 6%, with little enough fluctuation that any variation within three standard deviations (or 3σ) from the mean will still be within the 12% USL. The resulting process will exhibit a 3σ quality level.

Six Sigma

What makes the newer Six Sigma movement different from TQM is its emphasis on raising the bar on quality. The processes designed in TQM initiatives became very sensitive to 3σ control exceptions in SPC, with on-going improvement occurring incrementally at these margins. The Six Sigma movement uses all of the tools and techniques of these TQM initiatives, and adds an emphasis on long-term process variability and shift. Processes that were in-control in the short-term (typically operating within 3σ of their mean), would typically appear out-of-control in the long-term as greater variability was seen in human factors and error, equipment wear-and-tear, and gradual deterioration of process conditions.

With this increased variability included, TQM programs failed to deliver adequate quality, even at short-term 3σ levels. The short-term desired defect rate of less than 1% for 3σ processes could be seen to rise above 5% as a result of the long-term shifting of the process. With expectations expanded to 6σ quality, new processes could be defined that provided acceptable levels of quality while taking into account the implications of long-term process shift.

SPC is still used to monitor and evaluate process performance at 3σ levels. However, the identified exceptions beyond the 3σ control limits are now occurring well within the 6σ specification limits. In TQM, process defects and customer defectives were typically both defined at 3σ, and so process improvement to bring the process back into control was required while also dealing with customer defectives outside the specifications. Six Sigma separates the discussion of process defects (outside 3σ) from the recognition of customer defectives (outside 6σ) to allow processes and systems to self-correct and adjust to results in the 3σ to 6σ range.

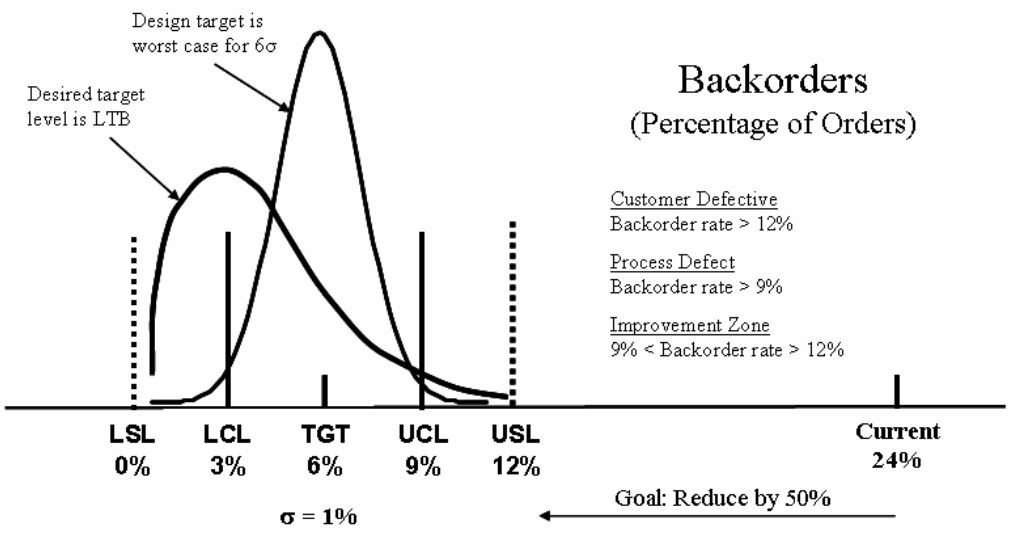


Figure 11 - Six Sigma brings even exceptions within specification

Figure 11 illustrates the 6σ viewpoint using the backorder rate example. The specification limits do not change because they represent what the customer wants, which doesn't change based on how quality is being measured; but, the control limits do change. Design target SPC control limits are still 3σ above and below the target, although the specification limits are now 12σ apart in this new Six Sigma view. This means that the revised UCL is now 9%, or the mid-point between the target value of 6% and the USL of 12%. There is now an improvement zone available between the UCL and the USL. Values above the control limit are process defects that SPC tells us can be economically corrected. If they can be corrected before they rise to the USL, the customer need never see a defective.

Implications for Practitioners

As processes are redesigned to align with Six Sigma thinking, process engineers have an opportunity to implement controls that take advantage of the improvement zone between 3σ and 6σ process performance. By building critical customer metrics directly into processes they can be made self-correcting by enabling specific actions to be taken when process defects are seen in the improvement zone. These actions need not always involve sophisticated technical solutions to be beneficial. Controls can be as simple as an e-mail notifying support personal of defects above the 3σ level, or a periodic report highlighting activity in the 3σ to 6σ zone. The point isn't to build processes without defects, but to build business solutions that can be kept from producing defectives in spite of their defects. That is the essence and opportunity of Six Sigma: making processes and systems increasingly self-aware and self-correcting.

UPGRADING YOUR TOOLBOX:

Adapting and Adopting Tools & Practices

October 10-12, 2005
Oregon Convention Center – Portland OR

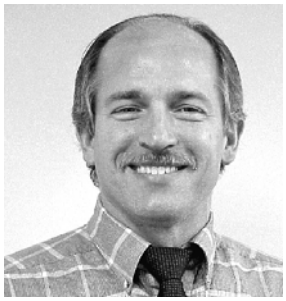
PRESENTATION BY

Joe Towns and Cordell Vail
Washington School Information Processing Cooperative

Stress, Load, Volume, Performance, Benchmark and Baseline Testing at a price you can afford!

WSIPC™
Technology for Educational Communities

THE AUTHORS



Joe Towns brings to the presentation a developer's perspective. He has eighteen years experience as a software developer, systems engineer, and supervisor.
jtowns@wsipc.org – www.wsipc.org

Cordell Vail brings to the presentation a test engineer's perspective. He is a Certified Software Test Engineer and Certified School Business Specialist with nine years experience in manual and automated testing. Cordell has made several presentations on "Improving Testing Processes" at both local and national conferences.
cvail@wsipc.org - www.vcaa.com

INTRODUCTION

This presentation has two purposes. The first purpose is to give you a knowledge base to help you learn how to select the correct testing tool for your testing environment. The second purpose of this presentation is to help you actually find an inexpensive testing tool that will do the job for your specific situation. This presentation is not intended to teach you how to do the testing.

OVERVIEW

At the Washington School Information Processing Cooperative (WSIPC) we are in the process of deploying a web based, teacher desktop application in a complex multi-component, integrated Progress database environment. This product had never been released into production. We needed to know what the capacity of our servers would be in relation to handling the load. We wanted to know what the user response time would be if a great number of teachers all logged in at the same time. We also wanted to know what effect that would have on the web server.

We had experience in selecting an automated testing tool, and had dealt with the leading testing tool vendors in the past to select that testing tool, however, we had not done any performance or load testing in that environment before. The concern we had was that we did not have a tested and proven testing tool that would allow us to test this application or web server for things like user response time, maximum server load, or database locking. We were not even sure what else we would need to test to make sure the application would stand up to the tremendous potential load that could be placed on it the first day.

Any time you are using an application over the Internet there is a possibility of 1000s of users logging in and hitting the server at the same time. This would almost certainly be the scenario in our case. We were introducing the application to large school districts comprised of numerous schools with scores of teachers in each. Each of these teachers, as well as administrators, could potentially log on and take attendance or run administrative reports via the web during the first day the application was available.

When the management asked us to do this testing, we did not have a testing environment set up to perform this load testing. We were starting from square one.

Our requirement then was to be able to test load-balanced web servers to see how many school teachers could be logged on at the same time, all doing different tasks. In some of our school districts there could be as many as 2,000 teachers logged on at the same time, submitting requests to the servers. We needed to be able to tell the school districts and data centers approximately how many servers they would need available, at any one time at the district level, to handle their load, and still have a user response time of less than two seconds.

KNOW WHERE TO START

Our experience told us that we could not find the right testing tool unless we knew the end goal of our testing. We knew that we would have to understand the testing environment. We knew we would have to organize a testing team that understood the terms of the testing. And we knew we would not know if the testing results were accurate unless we had a baseline for test results comparison. So our quest to find a testing tool started by organizing the testing process.

BEGINNING WITH THE END IN MIND

In order to be able to find the correct testing tool, our organization had to know the full scope of the task at hand. It has been our experience that one of the critical mistakes that many organizations make is to begin testing without a test plan. There is a common joke among testers that goes like this. The manager says, “Here, start testing this and I will go work on a test plan while you are testing”. Sadly, that is all too true and often the test plan is never even created. Stephen Covey gave good advice that applies to all testing, “Begin with the end in mind”.

Among other things, we knew that a test plan would help everyone on the testing team to know what the objective was, what was to be tested, what was not to be tested, what resources were available, and the test results to be reported. Here is an outline of the simple test plan we created:

- Scope
- Resources
- Timeline
- Risk Analysis
- Deliverables
- Glossary

We wanted the test plan to be as simple as possible. The kind of test plan selected, of course, depends on the complexity of the project and especially the degree of risk associated with the project if it fails. We have provided, to the attendees of our presentation, a CD with a copy of the final test plan we used in this project. The CD also has other example test plan templates of differing complexities, for you to use as a resource in your test planning as you start the task of finding the right testing tool for your situation. The test plan could be a simple check list, a basic test plan for short projects, a full range test plan for projects that will last several weeks, or a very complex and comprehensive test plan for enterprise testing projects that will last several months, have high risk and involve multiple departments within your organization.

UNIFORMITY

It is very common to find testing groups using some of the common testing terms as synonyms. This can be very confusing in an organization if you have the managers thinking that Load Testing is finding out how many users can be on the system at one time and the testers are calling that type of testing, Performance Testing. Therefore we propose that, at the outset, you include as a part of your test plan, a glossary of terms so everyone in your organization is defining the testing terms in the same way. It is not so important what definition you give to the testing terms as it is that everyone in your organization define and understand the terms in the same way.

This is an important consideration in finding the right test tool so you do not buy a testing tool that the vendor says will do one thing and then find out it actually does something else. We point this out because we actually had that experience. After working with a vendor who said the tool did “LOAD TESTING” we discovered that our definition of “LOAD TESTING” was different than the vendor’s and the tool did not do what we wanted to do at all.

Due to our experience, we suggest that you define at least the following list of terms listed below. These are not proposed as the industry standard definitions. They are just examples of how your glossary might look in your test plan. If you do not agree with these definitions, then change them to definitions that your testing team and management all agree upon. When you talk with vendors make sure that they define these terms the same way you do.

Stress Testing: Tests the server. Peak volume over a short span of time.
Load Testing: Tests the database. Largest load the database can handle at one time.
Volume Testing: Tests the server and the database at the same time - heavy volumes of data over time (combining Stress Testing and Load Testing over time).
Performance Testing: Tests user response and processing time.
Benchmark Testing: Compares your testing standards to the same testing standards in other similar organizations in the industry.
Baseline Testing: Setting testing standards to be used as a starting point for subsequent comparisons within your own organization.

Naturally there are many other terms that can, and may, need to be defined in your environment. The important point here is to make sure that the glossary in your test plan keeps all of the members of the testing team on the same vocabulary page so you can use the uniformity to help you in finding the right testing tool for the job.

UNDERSTAND OR IDENTIFY THE PURPOSE OR TARGET OF THE TEST

Again, the purpose of this presentation is not to teach you how to do the testing but to help you in knowing how to select the right testing tool for the job at hand. Every testing situation will be different. There will be different reasons why you are doing your testing and different things you will monitor based on those variables. Some of the things that you may consider testing are:

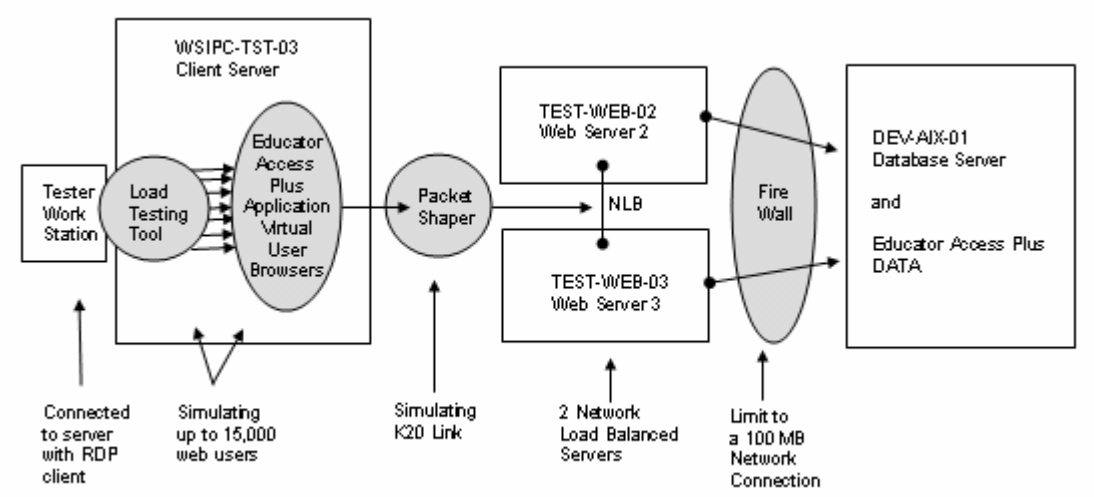
Bandwidth	**Multithreading**
Concurrent users	**Disk capacity**
Multiple platforms	**Faults**
Multiple browsers	**Memory**
Users per server	**User response**

KNOW THE TESTING ENVIRONMENT

You will also need to consider the environment necessary to execute the testing. The test plan preparation will help you to determine this. This information is critical in finding the right testing tool, because when you look for an inexpensive tool, you will be able to determine whether it has enough horsepower to test in your environment and give reliable results. Therefore, you need to consider adding to the test plan things like:

Number of client workstations	**Connectivity to servers**
Database availability	**Production copy of application**
Application and web servers	**Bandwidth and LAN**
Test tools to monitor results	**Reporting requirements and capabilities**

You may want to create some kind of flowchart to outline what your test environment is helping you to know and what testing tool or tools you will need to do the testing. Here is an example of a flowchart of the testing environment we set up at WSIPC to do this testing:



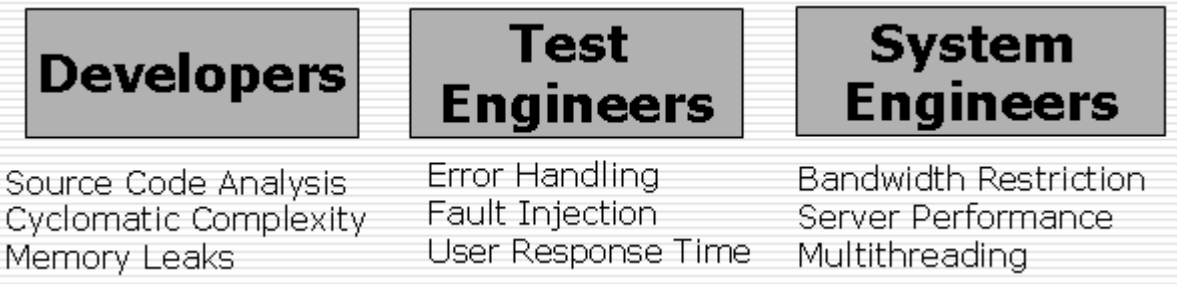
(Notice in this diagram we called the testing tool a "LOAD TESTING TOOL" and in this case we used WAPT with 2000 virtual users)

ONLY BUY WHAT YOU NEED

To find an economical tool you must know what your testing needs are. Each tool is very different. You are not going to pay $200 for a tool and then get a tool that will do what an $85,000 tool will do. So you must consider what your need is in testing and then find a tool that will fill that need.

CONSIDER TEAM COMPOSITION

The most effective testing team will be composed of individuals who will be able to address the scope, needs, risks, execution and monitoring of the test plan.

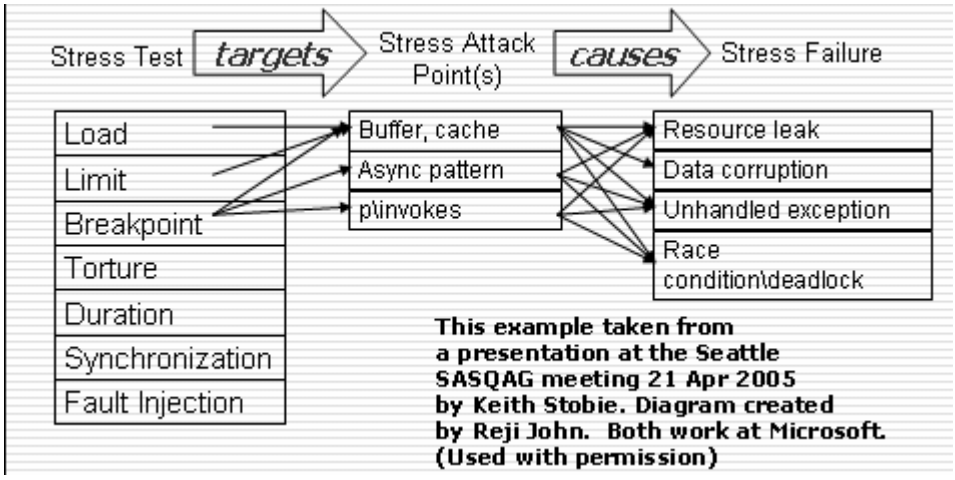


The members of your testing team will be able to help determine which tests need to be done and therefore the testing tool or tools that will be required to accomplish the testing in an acceptable manner.

KNOW HOW YOU ARE GOING TO TEST IT

Having picked a specific target of testing, for example bandwidth, user response time, and server performance, you will be in a position to look for a tool that will perform those tasks. You do not need a tool that will do all things for all testing tasks. There are tools on the market that will do that. Normally, they cost thousands of dollars. We wanted to find inexpensive tools that would perform only the tasks that we need done in our testing. Again, you may want to create a diagram to illustrate what your testing methodology will be.

Here is an example of a testing strategy used by Microsoft where they determined the types of testing that they would do and then what the testing tool would test.



TRUSTING THE TEST RESULTS

One of the main issues we have discovered in this type of testing is to know if the test results are valid when you have no live production data for comparison. The key element in the validity of test results is having something to compare your test results to. If you have a new web application and it has not been released to production, with a new testing tool, how will you know that your test results are realistic? This is especially true if you are using a new, inexpensive tool, for which there is little support and no known test results available from other testers.

Unless you have been around open pit mining, you likely would have no idea how big this equipment is. Testing tool results can be very similar to this example. It is very easy to use some of these inexpensive tools, but because it is not so easy to know if your test results are valid, you need to have something to compare the test results to.

POSSIBLE SOLUTIONS

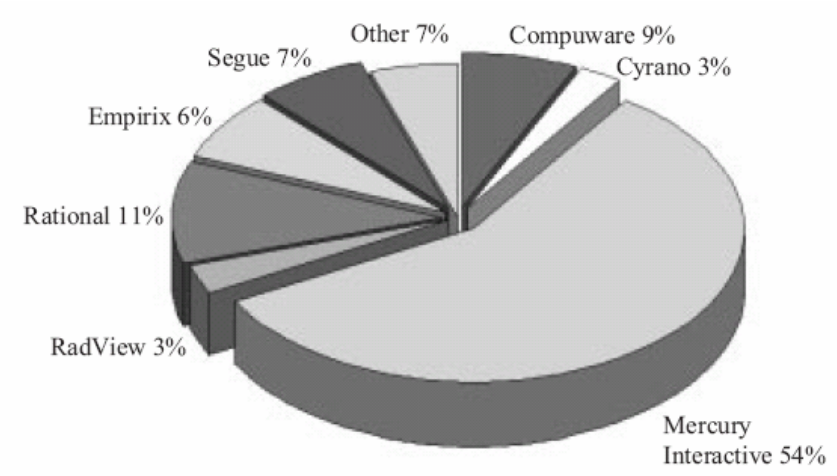
One possible solution in using a new testing tool would be to test it against an application where you already are able to measure the test results by monitoring the servers or with test results from previous testing. This test results comparison could come from your own baseline testing on a previous or known application where you already know what the results should be. You could also compare your results to a bench mark test from some other company doing similar tests on similar applications with the same tool.

Sometimes when you have a new application and a new tool, you will not be able to validate the test results until the application is actually released into production. Then, if your test results match the actual production results, you have established a baseline with that tool for further testing of other applications in the future.

Another possible solution, if you are using a new tool on a new application, would be to test the new application with more than one of these inexpensive tools. Next compare the results. That is one of the main points of this presentation. With inexpensive tools, using more than one tool is a feasible thing to do.

CAN WE TRUST MARKET SHARE?

In finding the actual tool, it was logical to first go to the marketing people and find out what others were doing to solve similar situations in testing. We found that 93% of the market share for this kind of testing was controlled by seven large vendors.



Source: http://www.mercury.com/us/pdf/company/newport_load2000.pdf

AFFORDABLE TESTING TOOLS

The first thing that became apparent to us was that the prices being charged to the 93% of the market share being controlled by the seven large venders was very high. That was more than we wanted to pay, thousands of dollars more. Therefore, we decided to take a look at that other 7% of the market to see what kind of tools that represented and what kind of pricing would be offered.

TESTING TOOL COMPARISON LINKS

To do the tool evaluation we first went to many sources on the Internet including vendor web pages to gather information about every tool available on the market. You will find a listing of all of those vendors on the presentation CD handout that you will receive in the class. There are over 200 vendors in that list.

Here are some of those sources we used to find those vendors and make our testing tool comparison list:
http://www.vcaa.com/testengineer/links.htm
http://www.testingfaqs.org/t-load.html
http://hammerhead.sourceforge.net/
http://opensourcetesting.org/performance.php
http://www.grove.co.uk/Tool_Information/Choosing_Tools.html
http://www.softwareqatest.com/qatweb1.html#LOAD
http://www.sqa-test.com/toolpage.html
http://www.webservices.org/index.php/ws/content/view/full/102
http://opensourcetesting.org/performance.php
http://sourceforge.net/projects/dieseltest/

Using those sources, we found Internet links to all of the seven vendors that represent the 93% of the market share (each of these vendors charge more than $5,000 and some over $100,000):

www-306.ibm.com/software/awdtools/tester/performance/index.html
www.segue.com/products/load-stress-performance-testing/index.asp
www.mercury.com/us/products/performance-center/loadrunner/
www.radview.com/products/WebLOAD.asp
www.quotium.com/qpro_overview_load_testing.html
www.empirix.com/default.asp?action=article&ID=418
www.compuware.com/products/qacenter/performance.htm

We also found links to many smaller vendors that were more reasonably priced. Here are the ones that we evaluated and will demo in the presentation:

TestMaker – PushToTest: **FREE**
http://www.pushtotest.com/Downloads/

WAST – Microsoft: **FREE**
http://www.microsoft.com/downloads/details.aspx?FamilyID=E2C0585A-062A-439E-A67D-75A89AA36495&displaylang=en

LoadTester – AppPerfect: **FREE**
http://www.appperfect.com/products/devsuite/lt.html

Site Tester 1 – Pilot: **$29**
http://www.pilotltd.com/eng/index.html

Portent Supreme – Loadtesting.com: **$279**
www.loadtesting.com

WAPT - Logasoft: **$299**
http://www.loadtestingtool.com

Webserver Stress Tool 7 – Paessler: **$625**
www.paessler.com

HOLODECK - SISE: **$1,500**
http://www.sisecure.com/holodeck/learn.shtml

NOTE: Holodeck is a fault injection tool - not a normal virtual user testing tool

CONTINUED RESEARCH

We would like to ask you to join us in this testing tool evaluation project. We have included on the CD handout several hundred testing tool web pages that we located. We have not evaluated all of them personally. We have just provided you with our research results hoping to save you the time of having to spend the same weeks and weeks of work that we spent looking for these testing tool links. That way we hope you will be able to make your own evaluations of the ones that we did not evaluate.

In addition to this conference documentation, all of our testing results are also posted on our testing web page at http://www.vcaa.com/testengineer/toolcomparison.htm . We will continue to post our testing results there as we do further testing of this type.

If you have tested a testing tool in this category of testing and would be willing to send us your evaluation to post on the web page it would be welcomed. The more evaluated tools we have, the easier it will be to find an inexpensive tool that fits our needs. All evaluations received will be posted to our web page with proper credits given to you and the vendor. Also, if you would like to have contact information such as your email address, web page or other contact information posted there with your evaluation for networking purposes, we will do that as well.

Joe Towns jtowns@wsipc.org - Cordell Vail cvail@wsipc.org http://www.vcaa.com

Creating a Performance Testing Team

Without Experience

By Ellen P Ghiselli, PMP

Pacific Gas & Electric Company
Information Systems and Technology Services/Application Services
Architecture and Engineering Services

May 2005

Author:

Ellen Ghiselli has been with PG&E's ISTS department for 11 years. Her current position is that of Project Manager. Ellen began her professional career as a credentialed teacher in secondary education in California; teaching at the Jr., Sr. and Junior College levels. She received her MA in Instructional Design in 1991. Since joining PG&E in 1992, she has worked in the following capacities: Instructional Designer for Nuclear Operator Training at Diablo Canyon Nuclear Power Plant (DCPP), Technical Writer for PG&E's Mainframe Computer Operations Group at DCPP, Computer Programmer/Analyst supporting various DCPP computer applications, developed the Windows '95 self-paced training for all of PG&E (as well as DCPP), Web Developer, and most recently creating and managing a performance testing team to support the new n-tiered Customer Information System at PG&E.

Abstract:

This paper presents a case study of the author's adventure into load/performance testing of a new infrastructure system across a WAN in order to characterize system load and anticipate the network upgrades needed to carry the new network load. Additionally, presented is a brief overview of how the group has evolved the science and art of load/performance testing for the CIS (Customer Information Systems) department and how it plans to apply a new architecture to further leverage the most recent version of their tool, Segue's SilkPerformer.

Acknowledgements to the following for their input and use of data: Min Tan, Segue Software, Inc.; Rob Batey, Bellcamp Corporation; Michael Hansen, Nora Khachaturova and Loren Lemons, PG&E.

Copyrights:

Introduction

In 2000, PG&E entered the final stages of their multi-year "Integrated Customer Care Solutions" project. The main component of this project was the acquisition of an "off the shelf" software package from SPL Worldgroup, Inc., *CISP*lus™. *CISP*lus has evolved from CorDaptix™ (CDx) to SPL Customer Care and Billing™ (SPL CC&B). For the purposes of this paper, it will be referred to as the CIS system.

The role of my group in the CIS department is that of Technical Architecture. The group specifies the infrastructure in order to support applications/systems. Over the course of a few months, in conjunction with the vendor, the architecture for the new CIS system developed into the following:

- PC client with Microsoft Internet Explorer (and the PG&E standard desktop image) (2000-4000 anticipated concurrent users)
- 2 – Intel/BigIP-based load balancers (in failover configuration) connected to:
- 2 – SUN 450, LDAP authentication servers
- 8 – SUN 420, WebLogic™ servers; each with 2 instances of WLS (16 total instances of WLS) connected via JOLT™ to:
- 2 – SUN E10k's and eventually migrated to 2 v880's, Tuxedo™ transaction servers connected via DB2Connect™ to:
- 1 – IBM z900 mainframe server running a DB2™ database of 8 million customer accounts

The above architecture supports the Online Transaction Processing portion of the application/system. The batch portion runs on 3 SUN v890's. An Enterprise Application Integration (EAI) layer allows other Utility Operations applications to interface real-time with the CIS database.

The architecture was frozen, the task of searching, sampling, demonstrating and getting buy-in for an automated load-testing tool fell to me.

Go Find a Tool

In October of 2000, we started hunting for a tool. System testing was new to me and I needed to find a way to educate myself. So, what does a newbie do to educate themselves? What I did was go out and visit QAForums (http://www.qaforums.com) Performance Testing forum, which isn't tool specific. I used this to familiarize myself with Performance testing in general and also how different tools are implemented for different reasons amongst the various enterprises. The following specialty discussions were very helpful for my purposes:

- Network and Distributed Testing
- Requirements Design

QAForums is an excellent tool for jumping into the deep end, when you don't have the time or the money to go to formal training. It seemed to me that I got the benefit of others working out their issues in the discussion forums. I found this very helpful in that it gives a frame of reference for possible problems, issues, etc that you should anticipate in load and performance testing.

The first challenge: Concurrently, I was trying out a tool on Microsoft's (WebTool) website. It did not fit as it was difficult to configure and run in the mini test environment and the documentation was limited. I concluded that I must find:

- a tool that was robust
- a tool that provides ample initial training,
- a tool that had good documentation and technical support.

Using the Microsoft tool and other free tools along with monitoring QAForums discussions helped me narrow down the list of possibilities to two: Mercury's LoadRunner and Segue's SilkPerformer.

Based on the industry; LoadRunner and SilkPerformer were market leaders. My inexperience told me I needed to go off the shelf, especially since PG&E's CIS had no load-testing expertise. Another driver was my being told to come up with recommendations, consensus and a purchase order by the end of the year! I contacted the two vendors and arranged for both web cast and onsite demonstrations to key stakeholders on the CIS project.

Another internal PG&E software-testing group was already licensing the LoadRunner tool. I interviewed the group's lead and found they were using the tool in a simple way. The other group had one controller and a minimal license for 20 virtual users. Their methodology was not going to work for CIS. When I asked Mercury to come in and demo the product in some other ways, they were unable to accommodate my timeline. I eliminated them based on that criterion.

Segue put on several demonstrations:

- They presented to me.
- They presented to the two mainframe systems programmers targeted to participate in the load-testing
- They presented to the Technical Architecture group.

Segue focused their demos on our needs. Everyone particularly liked how the tool showed various views of the load: user view, transaction view and server view (showing WebLogic, Tuxedo and DB2).

Part of the business requirements for the CIS system was:

- That it needed to be able to handle up to 4000 concurrent users in an emergency.
- This requirement lead to the acquisition of a 2000 virtual user license, three controller/agents and four Dell fully ramped up servers to serve as the SilkPerformer agents.
- The agent servers have 2 GB ram in each. The purpose of the servers is to facilitate load/performance testing.
- The agent servers are located in PG&E's backup data center in the basement of the company headquarters in San Francisco.
- The "production" system was in place in the data center located about 50 miles east in Fairfield, California.

The testing environment was running in parallel with the legacy system. The SilkPerformer agents were connected to production via a very fat network pipe (OC3 -155.52 Mbps).

Learn to Use the Tool

In the spring 2001, PG&E entered into a financial crisis, triggered by record prices for energy that had to be purchased outside of PG&E. A system engineer and I were assigned to become a team . To save money, when PG&E went into bankruptcy, PG&E only allowed the system engineer to go to Segue for one week of SilkPerformer training, a course now known as "Modeling and Implementing Load Tests".

The idea was to send one person to the training, who then returned and mentored others (me). The engineer said he found the class useful. The problem was that he is really not a software programmer or engineer. He was a system engineer, actually monitoring and analyzing system behavior (a small subset of the system). His skill sets were in monitoring the infrastructure-- not devising load. The engineer did his best to apply the techniques of record and playback to the scripts we were trying to develop based on what he had learned and practiced in class.

The problem was that the transactions were fake and the record and playback wasn't working correctly. Our progress consisted of:

- working our way through the playback,
- correcting error after error on the recompile and playback.

First Lesson: *While training is valuable for becoming familiar with a tool, it isn't enough to make you an expert. After the first test, it was clear we needed experts in order to properly characterize load on the system.*

Over the course of the next 6 months, I tried to figure out:

- What transactions should we should try to emulate?
- How do we record a transaction?
- How do we play back a transaction?
- How do we load a transaction?

The legacy 3270 direct connect mainframe system (home grown and fondly referred to as TP – for Transaction Processor), did not have online transactions. The clunky, internally developed Customer Care Management System (CCMS) did not interface directly with the legacy system either. We placed much of the data in a temporary area on the mainframe and it was eventually batch processed 36 hours later.

The new n-tiered CIS system introduced many new business processes to PG&E. Unable to find what transactions we needed to be done, I needed find out what I did not know. Other than payments, I guessed at these high-volume transactions:

- customer contact (a customer has called about their account),
- scheduling a field visit to a home or something similar,
- start services or stop services

No one I contacted agreed on any of the procedures for any of the above transactions in which I was interested.

Eventually, four transactions were in our load library. A system integrator, PwC, came onto the project and we started the drive to start running tests. The engineer and I ran the scripts and they were a failure! We could not get more than about 36 concurrent users and there were many errors. It was clear to me that we were not ready because these first scripts:

- Employed record and playback only (very low level first generation type automation)
- Employed no programming around data extracts or XML request/response bodies
- Employed no modularity

The second challenge: In my opinion, our testing was of no use at that point to the project. However, it did tell me I had three hurdles to overcome:

- I had to figure out what transactions represented the bulk of PG&E's daily business
- I had to determine how many users conduct each type of transaction
- I had to find test engineers to quickly understand the business case and translate it into good, working load-testing scripts.

Develop Use Cases and Determine Load

I strongly believed (and still do) in having use cases that are going to look like real transactions, otherwise why bother? By August of 2001, I finally came upon a person who could provide me with what I needed. This Subject Matter Expert (SME) worked in the Call Centers for a number of years and was now assigned to the part of PG&E responsible for developing all the online business transaction procedures. These procedures were targeted to be trained to all CIS users.

I worked with management to get this SME assigned to me. Working closely with my SME, I asked the following initial questions to get me started:

1. What and where were the largest Customer Information user groups in PG&E?
    - Call Center Operations (500+ concurrent users)
    - Stockton Credit Collections and Records (500+ concurrent users), consisting of 2 sub groups:
        - Credit and Collections
        - Billing Errors
    - Local Offices (84 offices through out California, from Eureka to Taft) (450+ concurrent users) and Payment Processing Center in Sacramento (50+), all taking online payments and handling a subset of the same transactions as Call Centers and Credit and Collections.
2. What types of transactions were typical of the each of these groups?
3. How many of these transactions were they doing in a typical day? Week? Month?

The SME did not have any notion of transaction volumes. She pointed me to supervisors and managers in the three client groups. I found that in PG&E, there is a big difference between:

1. Normal day processing
2. Bad weather (storm) day processing

I also learned that part of PG&E's reporting responsibilities to the California Public Utilities Commission (CPUC) was (and still is) for:

1. Types of calls were taken by Call Centers each month
    - average handling times
    - categorize the calls by types
    - the volumes of each category of call received each month
    - volumes of calls handled by CSR's,
    - volume for dropped calls
    - volume handed off automatically to an Interactive Voice Response Unit (IVR)
2. Received payments for each month and where they were received (PG&E, third party, etc.)
3. The types of payments received each month (cash, check, credit card, phone, online, etc.)

Determining Average Daily Load

The SME pointed me to key management persons in the three client groups. They were asked the following questions:

- Which transactions were performed most often (daily, weekly, monthly)?
- Who performed those transactions?
- More importantly how many people would be staffed on a "normal" day?

The information I received back was invaluable and gave me great insight into the core customer care business. In the case of the Call Centers and Payments group, I received current copies of the data sent to PUC for reporting purposes. This helped me to translate to the business processes that were being created in the new CIS system to handle the call types (transactions ensuing from calls) and payments.

The third challenge: Internally, I ran into a long standing issue of the lack of detail around the work done by a certain group. How would I determine transaction volumes with the lack of information coming from this group? The information forwarded was that a certain percentage of the Customer Service Representatives (CSR's) worked certain types of Help Tickets (a workflow type system) in the legacy system. When asked for details around specific transactions (Create a Bill online, or Cancel a Bill/Rebill), no answer at all was given! I located a SME familiar with the types of transactions, along with the average number of Help Tickets they worked and we came up with a formula to create a metric around transaction volumes for this group. I referred to it as "fuzzy" metrics. However, it worked and after go-live was determined to actually be quite close to "normal" load volume.

I was given staffing numbers for a normal 8-5 day:

- Staffing Call Centers (250 CSR's – note that Call Centers run 24 x7, but heaviest staffing is during the day)
- Staffing Credit & Collections (150 CSR's)
- Staffing Billing Errors (350 CSR's)
- Staffing Local Office CSR's and Payment Processing CSR's (344 CSR's statewide)

Based on the data concerning types of transactions and who was handling those transactions, a formula was created to determine the metric for *average daily load.* The formula is different for each of the groups, because there were differing inputs from each of the user groups. There were several formulas used to derive the *average daily load* for each user type.

A matrix was created that detailed (for a start) the 20 online transactions to create in our tool, SilkPerformer, along with targeted hourly transaction rates. The base 20 quickly grew to 23:

- We needed to leverage some XAI interface load (XML application interface) for direct interfaces
- We needed to create a "login" transaction to simulate users logging into the system constantly throughout the day
- We needed to differentiate between 2 high volume searches.

Determine Average Storm Load

As above, I contacted the management and supervisors in the three large client groups were contacted and asked:

- Was there a difference in types of transactions conducted during Storms? If so, which transactions got done? Which did not? Was it driven by customers calling in?
- Who performed those transactions, or more importantly how many people would be doing transactions during a storm?

The fourth challenge: It turned out that no differentiation was made for storm vs. normal transactions (which transactions were dropped or emphasized by the customer?), especially in the Call Centers. The good news, the Call Center kept transaction volume statistics for storm seasons, the last *bad* storm being 1995-1996, as well as statistics for January and February, 2001.

In 2001, it turned out that PG&E's financial crisis translated into the highest monthly volume of calls since tracking of calls had begun! The total number of calls for one month was 2.3 million compared to the average 1.3 million calls. The driver for these calls was that PG&E was in the middle of an energy shortage, as well as, on the verge of bankruptcy. This generated a record number of calls for several one-month periods and seemed ideal to model the high water mark for load-testing. Technical Architecture determined that mimicking the peaks and valleys of load during normal and storms days would take too much time to analyze and implement in the short schedule timeframe we had to create load.

Based on the data from the call volumes, which it turned out was the only data available for load differentiation, a formula was derived for each of the 3 client groups to determine the metric for storm loads.

The storm metric trade-off: Based on all the information and data, we made the decision to substitute the metric for storm loads. Instead, we created several load profiles for testing:

- 100% load (using average daily load metric)
- 200% load (referred to as "storm" load – 2x's 100% load)
- 300% load (3x's 100% load)
- 400% load (4x's 100% load)

We incorporated the storm metric for each transaction and each user group was incorporated into the matrix developed. However, the only metric actually used was for the *average daily load*.

Create the Team and Develop the Library for Execution

Concurrent with the determination of high volume transactions and developing a user mix/load profile (of sorts), it became clear that two inexperienced people were not going to be able to create the scripts in a timely enough manner. I then hired three experienced QA software engineers, one of whom was from Segue and was steeped in the tool. This was a coup. All three engineers were experienced with SilkPerformer and SilkTest. The engineer from Segue allowed us to hit the ground running. I gave them the desktop procedures, lined them up with the business analyst, who would do the transaction online while we recorded. From there, we programmed the request and response data.

The fifth challenge: In the process of the programming and unit testing, we learned that the new CIS system sent a whole lot of data back and forth that was never seen on the screen, in XML format. This cached data might not even be used by the application during the particular transaction. However, if the client did not send it back in the proper XML format, the transaction errored out and was unable to complete. Much time was taken during each code drop to:

- rerecord the transaction
- compare the request/response form posts
- determine where changes were
- reprogram the affected script

For the next 11 months, each code drop resulted in rerecording and reprogramming of 50% of the load scripts.

By October, 2001 the performance testing team was created and producing load scripts. At that time, a network planner colleague approached me with his problem. He was one of 2 that had been assigned to this project to determine if the network would need any upgrades in order to support the new application (moving from a SNA network to 100% TC/PIP based network).

The sixth challenge: He did not know how he was going to figure that out. I told him about SilkPerformer; how it worked and its ability to have agents geographically dispersed, yet controlled by the test engineers in San Francisco. The network engineer enthusiastically went back to his management to request 22 PC's. This was approved and we sat down and analyzed where these agents should be installed. We wanted to have a representative group, from large and small local offices, call centers, Stockton Credit and Collections, and WAN nodes throughout the system.

Performance Engineering Testing

In December, 2001 we were set to do Performance Engineering Testing (PET). The purpose of PET was to characterize load on the system and flush out issues. We ran a series of three tests during this period. Each test consisted of attempts to do the following:

- 50% load
- 100% load
- 200% load
- 400% load
- 100% load plus running batch billing

PET Issues as a result of testing:

1. Only 12 of 20 scripts (transactions) were able to run because:
   - Tuxedo services were not completing
   - The converted database was too problematic for update of some transactions to run under any kind of load.
   - 12 Incident Reports (IR) associated with the non-functioning scripts were filed with the vendor.
   - 10 Incident Reports associated with actual performance issues filed with the vendor.

Second Lesson: *Track the Incident Reports (IR's) and keep on top of the vendor. This is especially important when you are a third party testing team (as is done inside PG&E). There is an additional few layers between the testing team and the development team and the challenge is to turn IR's around FAST for retesting.*

2. The testing revealed the following script development problems:
   - We needed to assign individual IP addresses to each virtual user in SilkPerformer (known as IP spoofing) so that WLS would be properly loaded.
   - Only one HTTP request with all the users multiplexing on that one request was getting generated. So we created vLANS on each agent and assigned individual IPs on each vLAN, to which SilkPerformer would dynamically assign the IP address to each user.
   - XML parsing errors. Data being sent back through the infrastructure needed <CDATA> tags from the SilkPerformer tool.
3. At login, 1.4 mb of *uncompressed* data was returned to each user who logged into the CIS system. **This had broad network implications.**
4. The mainframe was constrained because the CPU would hit 100% utilization with online utilization alone.
   - Login particularly was putting excess load on the mainframe
   - There was some evidence on DB2 that that the increase in login transaction time might be caused by contention on a shared resource, perhaps associated with an increase in wait time on a lock.
5. During running a particular batch job alone (no concurrent online running), mainframe I/O was identified as a bottleneck.
6. Certain Tuxedo services either:
   - exceeded the blocking time out period (300 seconds),
   - showed long service compilations
   - just plain increased under load (going from 100% to 200%).

However, the resource depletion seen at this time was attributed to depletion or contention in the DB2 database by the BEA Tuxedo expert.

PET Lessons Learned:

1. The identified load issues were a major contributor to the go live date being moved out from June 2002 to November 2002.
2. Operational Readiness Testing (ORT) planning used the outputs and issues from the PET load-testing as an input.
3. The vendor was going to have to make some serious revisions to the application, particularly around how the SQL was executed in Tuxedo and DB2.
4. The vendor did not have a performance-testing environment even approaching the scale of PG&E's production environment, so we became the load/performance QA for the vendor.
5. None of the above issues would have been made visible if we had not created a transaction and load profile. I considered this the first major success for the SilkPerformer team.

The SilkPerformer team became a key player in the monthly Operational Readiness Tests (ORT's) planned for 2002. The purpose of the ORT's was to:

- facilitate legacy database conversion scripts and routines (from hierarchical to relational. 3-4 day process by the end of 2002)
- facilitate installation of the database (8+ million converted accounts)

- facilitate installation of the application on all tiers
- facilitate configuration of the infrastructure on all tiers
- facilitate V & V (verification and validation of configuration, based on changes from previous months testing and IR's generated) of infrastructure
- facilitate "turning on" the application on all tiers
- facilitate running online load with the 24 hour clock (all the background processes and direct/indirect interfaces available)

The seventh challenge: the SilkPerformer team was re-recording and analyzing where the application had changed in order to keep going with the testing. There were enough changes in the application to render at least 50% of the scripts broken at the beginning of each ORT cycle. We usually were given 2-3 days to turn this around. Because of the parsing of the XML request/response bodies, it was quite a challenge. Several of the scripts were very complicated and the CIS system sent *huge* XML bodies back and forth. We routinely worked 12 -18 hours a day either in recording/programming or testing. The application finally stabilized in October 2002.

As of 2005, the team has created and leveraged recording rules within the SilkPerformer tool that facilitates the programming/parsing of the XML request/response bodies. This reduces programming time by upwards of between 60% - 80%.

The eighth challenge: As the Scripting team (as we came to be called) started working the ORT's, the system integrator, PwC approached us to see if we would be able to include the Call Center Management System (CCMS2) in our load library. So, by the end of January 2001, we also began to work on including ten of the high volume Call Center Management application online transactions, most were queries or just plain displaying html information pages, but three of the transactions were update transactions. They were extensions of three of our existing CIS scripts.

The third lesson: Of the above three scripts identified, the script/transaction "Schedule a Routine Field Activity" became a very key transaction. This transaction

- began inside CIS,
- then traversed across the EAI layer
- logged into the separate call center system
- looked at available dates/times in the Field Activity System (FAS)
- selected a date/time
- sent data back across all the FAS and EAI layers to update the CIS database.

This script became very key to CIS load, EAI load, call center application load, but also to overall failover testing. Failover testing was very complicated because there were multiple layers in which to fail servers: CIS, EAI, FAS, Call Center Management application. But this script worked like a champ.

Load Customization

About four months into the ORT cycles, we were confronted with a complication around the load being generated via SilkPerformer. We did not have think times incorporated into the scripts. Why you ask? Because we had no way of knowing what they would look like. The Technical Architecture team decided that representing the actual transaction in terms of think time was not important for load and performance. However, once we started getting more and more scripts fixed (via vendor bug fixes) and added to the mix, we observed an unusual phenomenon.

Once users logged on and achieved "steady state" we observed porpoising on the tuxedo layer. The user requests would "burst" out of WebLogic to Tuxedo in large groups and then no work would take place for several seconds, and another burst would happen. We could not find any functions in SilkPerformer at the time that would control this.

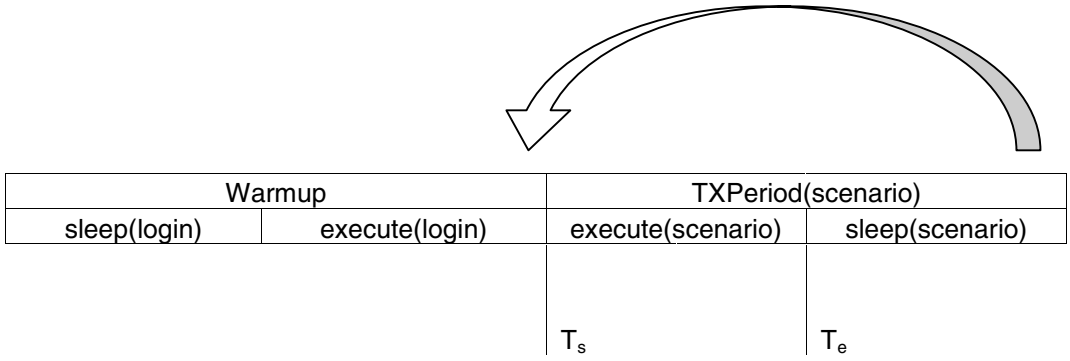
The three scripters, myself and the Risk Manager spent time devising a custom algorithm. This algorithm is still used today, although it has been refined at an even more granular level. And instead of it being hard coded into the script, it has become a separate function that is called from within the scripts.

How the algorithm works:

1. During each iteration, within each script, the dynamic execution time of the corresponding transaction is calculated.
2. The wait time before the next transaction is computed using a predetermined transaction period time minus a uniformly distributed random variable value that is between 90% and 110% of the above dynamic execution time. The reason for randomizing this dynamic execution time is to eliminate the pulsing (porpoising) behavior the script traffic produced when coming into the system.

The algorithm worked. We eliminated the porpoising and saw true steady state. And the algorithm allowed us to target an hourly transaction rate so that it equaled the daily average "normal" load as determined in the matrix.

Custom Wait Time Algorithm



sleep(scenario) = TXPeriod(scenario) - (.9 + .2*Rnd())*(Te - Ts)

sleep(login) = TXPeriod(scenario) * Rnd()

where: Rnd() = random number between 0 and 1, flat distribution

The algorithm was tweaked to simulate abnormal or storm loads. In the 200% load profile, the wait time before the next transaction was computed in the same way as the first, and then it was further divided by two. Another common variation of this 100% load profile configuration was a 300% load profile. In the

300% load profile, the wait time before the next transaction was computed in the same way first, and then it was further divided by three. One more common variation of this 100% load profile configuration was the 50% load profile. In 50% load profile, the wait time before the next transaction was also computed in the same way first, and then it was further multiplied by two.

Many of the transactions were extremely complex. Also, transactions executed very differently from one account to another. What became clear to the SilkPerformer (Scripting) team and Technical Architecture, was that in the new system, if an account had a lot of specific data attached to it (compared to a smaller account) it would take longer to process and longer to commit in the database because it was updating all data real-time on the account. This wasn't readily evident as we transitioned to a web architected system that builds its pages dynamically. This was vastly different from the legacy CIS, which was all direct connect, SNA, batch processed. Users were able to commit instantly, but not really. It was a very hard concept for some of the users and for the business to understand, accept and get behind. The business had specified that they wanted sub-second page to page and commitment response time. Because the page to page and commitment realities were not properly communicated and demonstrated to the management of the user community, it proved to be a political hot potato.

Transition from Migration to Operation and Maintenance

November and December of 2002 were focused on the migration and go live activities. The scripting team role was to qualify the system at 2000 users on the night the system was to be brought up. The gotcha was, it was a false qualification, since no update transactions (which of course were the highest volume transactions) could be performed on live customer data. However, we did qualify the system.

It was during this period that a new use for SilkPerformer was determined in CIS. As the go live approached, a serious issue around 200,000+ incorrectly converted accounts arose. They now had the wrong Field Order activities associated with them. This amounted to a lot of potential problems as it could inadvertently lead to either the electricity or gas or both being shut off.

What the scripting team did was work with the Technical Analyst to determine the best way to correct the accounts online. Why online? Why not create a Data Correction Routine and go directly against the database? All the system algorithms and validation routines needed to be brought into play.

The catch is, manually correcting it would take way too long and be way too costly for human resources to do the long, complex online transactions. So, we proposed SilkPerformer, because it can do the work at a minimum of 25 times faster than a CSR. In most cases, even faster. We came up with a series of two scripts to complete the data correction. It ran the night all the call centers were coming up live in the switch from the legacy CIS to new CIS. We were up all night running the script, looking at the data. Since we were not interested in loading the system, just correcting data, we put in wait times between transactions.

All in all it, took eight hours to do the work. Had it been done manually, it would have taken days, even weeks. My team had had provided a solution to a potentially embarrassing situation for PG&E.

That was the beginning of our transition into Data Correction.

Beginning in July 2004, PG&E built a full scale test environment (DR/FSTE) to duplicate production. This environment doubles as the Disaster Recovery business resumption environment. I ran the FSTE, as Test Coordinator until March 2005. In the FSTE, the SilkPerformer team refined its algorithms and re-architected the current script library as well as, to add 10 XAI (XML application interface) scripts to the mix. The XAI interface has more and more importance in CIS, as that interface allows PG&E customers to do limited online transactions via the internet.

The SilkPerformer scripts are used for testing infrastructure (when new hardware is being introduced into production), patch upgrades, system application upgrades as well as being the central portion of the quarterly System Integration tests. In the CIS system, it is the online portion that truly integrates the whole system.

In 2005 and 2006, SilkPerformer (Performance Engineering team) is going to play a key role in the CIS application major upgrade. The client interface has switched from .jsp's to XSLT. The upgraded system is leveraging Web Services. In the fall of 2005, the team will remotely work with an IBM test lab on the east coast to test load on a proof of concept CIS on UNIX running Oracle RAC.

As for the tool and our algorithm, now that CIS is live, abundant server log statistics are available to refine load profile and make sure our tests reflect actual production volume. Because SilkPerformer interacts directly with the web server layer, server logs from the WebLogic servers are the main source to calibrate the load profile.

The throughput metric *requests per hour* is obtained for several subsystems of WebLogic servers from CIS. Those subsystems are:

- Login
- Customer Information (CI)
- Field Orders (FO)
- Payment/EFT (PY)
- Billing (BI)
- XAI
- Meter Read (MR)

Also, we obtain the total *requests per hour* from all CIS subsystems. These statistics are used to further augment the load profile, to reflect the overall behavior of CIS.

Lessons Learned

What would I do differently? At this point, I'm not sure. In hindsight, it's amazing that I insisted that we model the business transactions as closely as possible to the real process (UML modeling). Also, it was extremely fortunate that we gained the Segue engineer. He has since returned to Segue; but we hired a different Segue engineer to join the Performance Engineering team!

Knowing what I know now, I would spend the first 2 to 3 weeks focusing on developing and documenting programming conventions and leveraging more of the tool's functions so that the scripts would be more

modular. But, I was new to load-testing and we were extremely short on time, being driven very hard to produce. All in all, I found the whole experience to be one of the most positive in my professional career.

In conclusion, to prepare for load-testing a 3rd party, n-tiered, web-based, enterprise application; you should do the following to mitigate problems:

- Learn load/performance testing principles.
- Leverage known QA sites i.e. www.QAForums.com.
- For 3rd party applications/systems, leverage an off the shelf product, that has robust technical support and can handle large numbers of concurrent users.
- Create a team by hiring the best knowledge and experience that you can afford. Use this knowledge and expertise to enhance your own skill sets. Make sure that you can match skill sets to tasks.
- When faced with no information about types of activities or transactions, go looking inside the business for internal Subject Matter Experts. Don't let manager's tell you that it's not important. **It's Important!**
- Be agile and ready to evolve. When in the throws of implementation, look ahead to how you can evolve yourself and your team and leverage its expertise or add to its expertise. *Nothing stays static*! Especially in a transactional environment.
- Be prepared to mentor the team in its transition points to create and document an architecture, update and document its programming conventions, and adapt better recording and programming techniques.
- Make sure your documentation is *empirically* derived. This way, when you are confronted with naysayers or technicians who want to blame your tool; you can just push back and ask them to empirically prove you and the tool inaccurate.

References

1) Microsoft Web Application Stress tool http://www.microsoft.com/technet/archive/itsolutions/ecommerce/maintain/optimize/d5wast_2.mspx
2) QA Forums - http://www.qaforums.com
3) Segue Software – http://www.segue.com
4) Mercury Interactive - http://www.mercury.com/us/?siteselect=us
5) Software Test and Performance magazine - http://www.stpmag.com/

TECHNIQUES TO KICK-START AND REV UP YOUR QA PROGRAM

By: Stacy Tjossem, IBM Business Consulting Services

Stacy.Tjossem@us.ibm.com

Bio: Stacy Tjossem is a Senior Business Consultant with IBM and has more than 20 years experience using information technology to solve business problems. Her customers have included Federal and State Governments, and commercial customers in the manufacturing, financial services and information technology industries. She has led efforts to develop and implement enterprise technology standards, methodologies and processes; managed successful reengineering initiatives to improve organizational productivity; and implemented organizational changes related to technology modernization and CMMI initiatives.

Abstract: Establishing meaningful and well respected QA processes, and overcoming resistance from Managers and Co-workers alike, is a frequent challenge faced by many QA Teams. Managers may not feel they are getting their moneys worth out of "process improvement" efforts and Co-workers may feel that QA processes will only add to being over-worked. Drawing upon lessons learned I would like to share specific techniques that have proven successful. These techniques can be readily adapted and implemented regardless of project or company size.

First published in the Pacific Northwest Software Quality Conference Proceedings, 2005.

TECHNIQUES TO KICK-START AND REV UP YOUR QA PROGRAM

How do you get the most from your QA staff and process improvement activities? You begin with good program leadership that communicates priorities and goals, and actively participates in defining and implementing quality programs. Asking a quality assurance organization to establish, implement, or maintain an on-going quality program without providing management direction is the same as asking developers to code without telling them what they are building.

Too frequently, senior managers leave decisions regarding which quality programs to implement, and how they shall be implemented, to the QA staff. Rarely do you find a senior manager who sets qualitative or quantitative expectations (and has the data to understand what is realistic) prior to implementing process improvement or quality program initiatives. Lacking guidance from senior management, the QA staff will most often attempt to implement an industry model, or attempt to create a unique model when this situation occurs, it can result in losses for the organization.

INVOLVE SENIOR MANAGEMENT IN CHOOSING A MODEL

When QA staff opts to use a specific model without management's participation in the selection or tailoring process, progress towards achieving benefits can be hampered due to lack of support or contradictory direction from management's day-to-day decision. Any model, when it has not been tailored to align with organizational goals and direction, can result in an implementation that pulls people in multiple directions. Add to that conflicting management directions, and you have erosion of workplace productivity. Only by tailoring the model to a unique business case and aligning it with goals and priorities does it become useful to management; and to achieve this, it is necessary to have their understanding and participation. In this context, management participation means understanding the tenets of the model selected, validating that tailoring of the model is in line with business goals, energizing the implementation, and managing based on the results.

Management must communicate priorities, such as what is more important: schedule, cost, consistency, defect ratio or some other factor. This acts as a facilitator to the QA organization to better prepare or recommend an appropriate model or quality framework, and to initiate tailoring activities that align with what is considered to be most important. When management communicates priorities such as "fast and cheap" it is problematic, unrealistic, and signals a lack of appreciation for the efforts of those working to improve return on investment. The same priorities can be expressed as "reduce cycle time by x days and costs by y%." This requires management to understand what is realistic, but accomplishes the establishment of clear, specific targets for the QA group and others to focus on achieving.

In some fields, there are business benefits to be gained for being fully compliant with an industry model. If your organization pursues, or is planning on pursuing, Federal, State and Local, or large clients, or if your key competitors are compliant, then it is worth it to attain the status of compliance. In the current market place, competition has increased and off-shore

developers offer economical alternatives. Model compliance can be a market differentiator. If there is no clear business mandate to become compliant with an industry model then consider implementing process improvements strategically and carefully.

Each leading model can result in improvements to productivity, predictability of cost schedule outcomes, or overall decrease in business risks, when implemented. Let your position in the market, business goals and strategy be your guide. On the flip side, not all organizations will benefit from implementing all aspects of a particular model; and it's foolish to think of compliance as a silver bullet.

HIRE A QUALITY ASSURANCE MANAGER WITH THE RIGHT CREDENTIALS

Staffing is another major area where senior management often misaligns their actions in contrast with prioritized organizational goals. Organizational priorities and goals should be used to determine the required credentials of the QA Manager.

Frequently, an individual who has a strong background in testing is promoted to be the new QA Manager from within the organization. This individual may be able to identify immediate improvements, leverage his/her knowledge of the organization, and make recommendations to improve requirements and test planning—all factors that could help an organization deliver more reliable products. But, will this same person be able to identify and make improvements to cost/schedule controls or implement non-testing based industry models?

Selecting a QA Manager who has experience in many disciplines, or those disciplines under consideration for implementation (e.g., project management, requirements management, configuration management, and quality assurance, risk management), will help facilitate organizational change; *if* an organization is implementing a multi-discipline model like the Software Engineering Institute's (SEI) Capability Maturity Model Integrated (CMMI) or International Organization for Standardization/International Electrotechnical Commission (ISO/IEC). I have found that the wider the range of process changes being implemented, the more important the QA Managers individual experience becomes. The more quickly a QA Manager is able to establish a common understanding of the issues and challenges related to a specific area, with the practitioners who perform them, the faster changes can be implemented. Yet, there are always exceptions, based on individual personalities.

Which approach is better, the individual with a deep test background or the individual with a broader base of experience? The answer depends on the decision as to whether to implement an industry model or not; or to focus on a distinct quality attribute. Regardless of the selected candidate's credentials, publicly supporting and acknowledging that the individual is in place to deliver management goals will make his/her job easier and will increase the chances that organizational goals will be met.

BEGIN BY CREATING A BASELINE

Before embracing all aspects of a particular model, I recommend that the Quality Assurance Manager's first task be to create a baseline against the target model. The baseline

may be established for all or part of the organization, and should look for the most rigorous level of compliance with the model selected. That way, the baseline serves two purposes: first, it provides a realistic view of where your organization is as compared to a fixed reference point. Second, it provides quantifiable opportunities for process improvement to choose from.

In developing the baseline it is important to keep the findings based on what the data shows, rather than relying on a general impression from the individuals involved. The opinions of the individuals involved should be considered, but not form the entire basis of the findings. Very frequently, when assessing an organization for the first time, findings are based predominately on how the individuals that perform a function feel about it. The result is that capabilities become over-stated. This occurs for many reasons, but among them are pride, fear, and ignorance of the evaluation criteria behind the model.

PRESENT "NEUTRAL" BASELINE FINDINGS

The QA Manager typically presents the results of the baseline in the form of a gap analysis, depicting strengths and weaknesses against the selected model. In doing so, the opportunities for improvements can be revealed in a non-confrontational manner. The model defines the level of "goodness" desired. The data tells if the "goodness" has been achieved or not achieved. Senior management should then determine which gaps are worth closing, based on the gap analysis and organizational priorities, not the Quality Assurance Manager.

My experience has been that when the management team sees the baseline results is the point when most model implementation projects are killed. I believe that this is partly because most organizations do not perform as well as they think they will. When developing the findings I am sensitive to depicting an overly-negative picture of where the project is relative to the baseline and the project's goals. I find it is better to use data displays that communicate how close a program is to achieving results, or how good they are, rather than how far they need to go. By stressing what is done right it establishes three things:

- It provides validation to management and coworkers that some things are already in place and are working well. They already know this to be true on an intuitive level and it communicates to them you won't be wasting time fixing something that isn't broken. (Strengths)

- It communicates that everyone involved has done a good job so far and that they individually recognize that there is always room for improvement. (Opportunities for Improvement)

- Most important, it communicates that with additional effort, there is an additional success story to tell. (Weaknesses. Gaps that need to be closed to achieve compliance.)

MANAGE EXPECTATIONS

Let's assume for a moment that you are a Quality Assurance Manager either *in situ* or have just been hired for that new job. With luck, the senior manager is savvy and has all the aforementioned priorities and expectations in mind—you've been hired for all the right reasons

and you're the one for the job. If you aren't that lucky, it may be up to you to clarify what it is that you are vaguely tasked to do and make sure that you and your manager are in agreement.

First and most important thing to know is: Are you the QA Manager or a Test Manager? Senior managers frequently rely on the testing function as their only means of quality assurance and make testing "responsible for quality". But how can you expect a testing team to instill "quality" at the end of a life-cycle? You can counter this by ensuring that there are two distinct organizations, even if that means designating one person, part time, to be responsible for QA.

Because the functions of testing and QA are inherently different, it doesn't hurt to review the expectations for the testing organization as compared to a QA organization, and make sure that your management understands how the two are inherently different.

Here are some thoughts that might help you to frame expectations:

- Testing can identify defects for removal and returns products for rework
- Testing validates that the product meets stated requirements
- Testing provides empirical data on defects detected, defects removed, and additional information about the source of defects.
- Quality Assurance gives an independent appraisal of how the overall organization is implementing agreed to processes
- Quality Assurance helps identify and facilitate improvements to processes
- Quality Assurance facilitates the collection of empirical data, on a broad range of indicators, upon which informed decisions can be made
- Neither Testing nor Quality Assurance is responsible for "achieving a high quality" product. That is the responsibility of the entire team.
- Quality can never be achieved by hiring good developers alone, or be instilled in a product by testing.

DO A REALITY CHECK

Before staffing up and laying siege to the defects, waste, and inefficiencies that are waiting for you, conduct a reality check.

- Do you have management's commitment?
- Has the message been communicated to the team?

- Do you have a clear agreement with your management as to what is important to them and what part they have to play?

- Do you have agreement with your management on what the strategy for going forward is?

- Are there sufficient resources to carry out the strategy in the time frame expected?

- Do you have the trained resources to do the job? If no, is access to training available?

A 'no' answer to anyone of these questions should cause you to go back to your management and turn a no, into a yes.

LAUNCHING THE QA PROGRAM ON A POSITIVE NOTE

I can't tell you how many times I see a quality assurance campaign kick-off with a slogan—as if that's all it takes. How many organizations pronounce: "quality is our number one priority!"? And of course, you have to have a sign. . .My experience has been that most people will immediately call to mind a Dilbert cartoon when this approach is used. Instead, plan on using real data, drawn from authoritative sources to support your position and build a business case for change. The data should be meaningful to managers and engineers alike.

When you don't have internal data to draw upon, use authoritative sources such as Gartner, SEI, or other accepted authority in support of your assumptions. For example, you might propose that by adopting a common methodology across divisions would lead to company savings of 20%. You might state that this approach would improve communications, employee mobility, reduced training, etc. However, what is most likely to sell your management, your team, and others, is citing an authoritative source, such as Gartner, that says the same thing.

If you don't have access to subscription services such as Gartner or IEEE, you can use case studies posted as part of conference proceedings (e.g., International Conference on Software Engineering (ICSE), IEEE International Symposium on Reliability Engineering (IISRE), Practical Software Quality and Testing (PSQT) or from well known organizations. There are several organizations that provide timely and supportive data, including the following:

- The American Society for Quality (http://www.asq.org) provides information tailored to key industries
- The Software Engineering Institute (http://www.sei.cmu.edu) provides assistance with processes and the Capability Maturity Model/Integrated
- The Project Management Institute (http://www.pmi.org) provides information about project management, project planning and project tracking and control
- The International Function Point User's Group (http://www.ifpug.org) for assistance with software sizing

There are of course many others, as well as accepted published references.

Conference Presenters often include specific details and quantify the benefits realized from an adopted approach—and there are typically very few that present negative results. When case studies are used, beware of using data drawn from organizations with no direct parallels to your own. For example, if your shop develops and maintains financial applications, parallels drawn to missile launch routines will probably be dismissed. Also keep in mind that quoted return on investment numbers may have been calculated using very different formulas, and may depend on some rather generous assumptions. Rarely are such estimates created by unbiased third parties.

SHOW – DON'T TELL RESULTS

The QA organization is often seen as the purveyor of bad news and negative findings. When the QA organization is mistrusted it frequently results in withholding of needed information and/or information being supplied from the organization that is incorrect. A technique that I have found helps avoid this from occurring is by focusing heavily on information design, and letting the data speak for itself.

Consider the following when designing QA data displays:

- Is performance reporting clear, concise, complete, unambiguous, and easy to understand?
- Has data been analyzed and used to depict multiple dimensions (e.g., time, cause and effect)?
- Does the data tell the story:
  - This is where we are, this is where we are going
  - This is the cause, this is the effect
  - This is the answer, as compared to what?
- Will the viewer be able to take in all the information provided in one display?
- Can false conclusion be drawn from the data?
- Has the organizational structure constrained the information display and prevented it from being more meaningful?

If uncertain where to begin, and what data to present, a good starting point is the basic performance measures from the SEI CMM. These measures cover planned versus actual performance for cost and schedule; provide visibility into requirements volatility and configuration changes; and provides performance indicators that the QA effort is being carried out. A good caveat to keep in mind is that QA measures must not be used as measures of an individual's performance.

A risk of this approach is that in the typical shop, "stuff" happens. Unfortunately, in most cases it happens with little or no warning. If you continue to re-evaluate and improve your measures you want to be sensitive to, and avoid charts that don't alert you to an upcoming problem. Usually it takes time as well as trial and error to develop and implement measures that provide forewarning. Until that happens, be prepared to have Managers making decisions based

on how they feel things are going rather then relying on empirical data and prepare yourself with answers to tough questions when the data displays have failed to forewarn of problems.

CULTIVATE DIVERSE VIEWS

Let's assume that your program is up and running smoothly. Opportunities for improvement will present themselves. However, not all opportunities will come from the data or from problems encountered. You need to establish an active outreach program for all practitioners involved in the process in order to get their ideas. Keep in mind that not everyone will want to contribute. Many people find it annoying, or will feel that they shouldn't need to defend their ideas if challenged, or will be unhappy if their ideas are not used. Others will not be prompted to participate unless there are incentives and rewards that appeal to them as individuals.

The standard approach in many organizations is to establish Process Action Teams (PATs) to develop solutions and document processes. This enables those closest to performing a process to have a say in how it is to be performed. But do take a step further. Take all those that use the output, and ask them "what would make the product better?" The answer may surprise you!

IN CONCLUSION

There is no single formula for success implementing quality assurance programs. I believe it primarily entails successfully developing wide respect for the QA discipline and overcoming resistance to change. In this paper I have attempted to present some proven solutions for individuals tasked with implementing QA programs. In summary, I share the following thoughts on what it takes to "rev up" your QA program and achieve the maximum value for the effort.

It takes:

- Communicating a clear message of the value and support for QA by obtaining Senior Management involvement and commitment to the approach.

- Providing credible knowledge and skills to lead the effort by ensuring that the QA Manager and staff have the right skills to do the job.

- Identifying a clear target and reporting progress on achievements by creating a baseline and measuring and reporting progress against it.

- Managing expectations by revealing what is and what isn't possible.

- Monitoring and reporting performance indicators which inform, but don't criticize. Achieved by developing data displays that reveal truths, including complexities, and do not distort findings.

The Gap Between Business and Code[1]

Brian Marick
marick@exampler.com

This paper argues that the continuing problems we have with requirements elicitation and transmission are not a result of poor skills or sloppy people. It's rather that the entire idea is based on dubious assumptions. A different assumption is introduced, namely that effectiveness can only be obtained by parties having iterative conversations over time, and that communication is meant more to provoke right action than to transmit understanding. Based on examples of this assumption in practice, suggestions for software development are sketched.

Biographical sketch

Brian Marick was a programmer, tester, and line manager for ten years, then a pure testing consultant for ten more. Since 2001, he's been heavily involved in the Agile methods. He was one of the authors of the Manifesto for Agile Software Development <www.agilemanifesto.org> and is past chair of the board of the Agile Alliance nonprofit. At this point, it's probably fairer to call him an Agile consultant and advocate than a testing consultant, though his Agile work is heavily informed by the lessons of testing. His testing work can be found at <www.testing.com>, and his agile work at <www.exampler.com> and <www.testing.com/cgi-bin/blog>. Many of his writings make it into the pages of Better Software Magazine, of which he is a technical editor.

[1] Because of poor time management on my part, this is not the final version of this paper. You can find that at <http://www.testing.com/writings.html>. Sorry about that. Based on past experience, I can say that you might find that version radically different from this one.

> "... some feel they should not be given a headache when trying to understand meaning. Why? Why on earth should it be simple to explain how people create exquisite, infinite variations of meaning from elaborate squeaking and grunting rituals?"
>
> - Nathan Stormer (impersonal communication[2])

As far as I know, no alchemist ever succeeded in turning base metals into gold. Why not? It wasn't for lack of trying: many people tried hard for many years, not without progress. Perhaps they were just using the wrong techniques – many of them thought so and devoted their lives to improvement and discovery. Perhaps they didn't take it seriously enough: it was widely thought that the purification of gold had to go hand-in-hand with the purification of the alchemist's soul.

Or maybe they were working from some bad assumptions. That's what I think.

We've been trying for many years to elicit and transmit requirements reliably. The results have been disappointing, though not without progress. Why? Maybe we don't know the right techniques - certainly we have people devoted to discovering new ones. And maybe we're not pure enough. Seriously: if the business people weren't so focused on what they see on the screen, and the programmers weren't so obsessed with the code behind it, maybe they would participate long enough and intently enough to get the requirements right.

Or maybe we've been working from some bad assumptions. That's what I think.

The way we think about requirements is shaped by two metaphors: the **conduit metaphor** and the **map metaphor**. If those metaphors are bad ones, no amount of skilled elicitation will produce golden requirements. Maybe we should give up and try something else. This paper is about the problems with those metaphors and the alternatives suggested by abandoning them.

Two Metaphors

The map metaphor is all about correspondence. Things in our heads – or words in our language – correspond to things outside in the world. You are always able to think the thought "chair" or say the word "chair", point at something in the world, and say whether that thing is a chair.

If you believe in this metaphor, the requirements problem amounts to building a map of the world inside which the program must perform well. This map allows one to mechanically solve any problem the program will encounter: that is, the solution can be computed using *only* the map and a fixed set of rules for working with it. Once given this map, the programmer's job is to translate it into some form a dumb binary computer can

[2] He wrote this in a comment to a blog posting, so it's not a personal communication. I did get permission to quote him.

compute with (given its fixed set of rules). This conversion does not change the map: it involves adding implementation details, not changing the nature of the program's world.

The conduit metaphor is about communication. It says that communication between two people consists of converting the map in someone's head into a message – likened to a physical object – which is shot over to the other person, who decodes it into an equivalent map. The conduit metaphor is deeply embedded into our offhand language, as is exhaustingly demonstrated by Reddy (1979), who provides many examples like this:

- You've got to get this idea *across* to her.
- I *gave* you that idea.
- It's hard to *put* this idea *into words*.
- Something was *lost* in translation.
- The *thought's buried* in that convoluted prose.
- I remember a book with that idea *in it*.
- That talk *went* right *over my head*.

This folk wisdom about communication affects our actions. When we have a hard time communicating, we use the metaphor to reason about the problem we're having and seek solutions (Lakoff and Johnson 2003). The requirements process demonstrates that. We seek ever better ways to make a written requirements document a faithful encoding of a map, so that any arbitrary qualified person will be able to decode it to the same result.

A different metaphor

I happen to believe those metaphors are incorrect. Philosophers have spent better than two thousand years trying to make the map metaphor work well enough to survive the implications of quite straightforward statements (Rorty 1981):

- "To what thing in the world does the word 'unicorn' correspond?" We talk about things that don't exist.
- "I now pronounce you man and wife." Some statements aren't representations *of* the world; instead, they're actions *in* the world (Austin 1975).
- "That depends on what the meaning of 'is' is." All words admit of multiple interpretations. Given enough work, some would say that *any* word can be made to have an indeterminate meaning (Culler 1983).
- "Why is a bean-bag chair a chair while a three-legged stool isn't?" Categories do not have clear boundaries (Lakoff 1990).

Of these, the last two are most relevant to our work. Lakoff shows that many categories we use have fuzzy boundaries. In particular, they have *central and peripheral examples*. When I say the word "chair", you probably don't envision a beanbag chair; instead, you picture a more typical chair: seat, four legs, back. Other kinds of chairs are less "chair-ey" than that.

As another example, Lakoff asks us to consider the word "bachelor" and asks, "Is the pope a bachelor?" Your answer is probably something like "well, yes, technically, I suppose the pope is a bachelor" – but he's clearly not as *good* a bachelor as a 25-year old subscriber to *Maxim* magazine living in the youth-oriented center of some city. The

simple definition of "bachelor" – an unmarried male – is surrounded by a penumbra of preferred connotations. The pope isn't a bachelor because there's no reasonable prospect of him marrying, and the canonical bachelor (in my culture, at least) is someone who's *not yet* married – someone who will, more likely than not, marry before too long. For that reason, it's a little bit odd to call an unmarried gay man a bachelor.

Or is it? It may be odd for me, in a country (the USA) where gay marriage is intensely controversial and far from the law of the land. Is it as odd in Canada, the Netherlands, and Belgium, which allow same-sex marriage? There, the standard image of the young male bachelor sowing his wild oats before settling down to a committed legal relationship might well become independent of sexual orientation.

Because of this, you and I might have very different maps of the world that we both encode in the word "bachelor". Our problem is not just that "it depends on what the meaning of 'is' is" – that single words can have multiple meanings, and that the cannily malicious speaker can mislead us by allowing us to assume one meaning while pointing to an alternate meaning to defend against the charge of perjury. It's that the multiple meanings of words are not pin-down-able. There are, in principle, at least as many meanings for a word as contexts people operate in.

It gets worse: Every word's dictionary definition is made using *other* words, so that – if we believe in the map metaphor – to really understand the definition of "bachelor", we'd have to understand the definition of every word used in "bachelor's" definition, and every word in *their* definitions, and so on. That multiplies the number of possible (mis)interpretations. Worse: the process doesn't bottom out: eventually, some definition of "is" will use the word "is", and we're caught in a circularity. On the old *Star Trek*, the computer would now burst into flame.

We don't burst into flame because, I believe, the map metaphor is only a tool. For certain problems, communication and social action are convenient if we all pretend that our words really do map into categories in the world. It's like using Newtonian mechanics when calculating slow-speed motion: it's not *true*, but it works well enough until you get into the hard (relativistic) cases. Situations of great precision – like causing a dumb binary computer to make judgments humans think are sensible, even though it's not naturally good at categories without hard margins – require us to use different tools.

So what's a better metaphor for communication? One, which I heard from Richard P. Gabriel, is A POEM IS A PROGRAM. That is, a poem is some text that combines with its input (the mind, memory, and experiences of the reader) to produce a result (an understanding, an emotion, a mental map). The good poet writes poems that produce intended (or surprising-but-good) effects in the intended audience. The effects are more predictable when the audience belongs to the same *interpretive community* (Fish 1980), people who have been trained to react to certain texts in certain ways. That's the sense in which art can be seen as a conversation among those people – both artists and critics – so deeply involved in the creation of the art that they teach each other, through example or commentary (Bloom 1997, Gombrich 1995).

I'd like to extend Gabriel's metaphor, which still seems to me to bow too much toward the map and the conduit. My extension is to take up the notion of certain cyberneticians (Pickering, in press) that the brain is not about thinking in the sense of representing the world, but is rather a *performative* organ. Ross Ashby (1948) puts it well:

> ...to the biologist the brain is not a thinking machine, it is an acting machine; it gets information and then it does something about it.

So I'm agnostic about whether a poem (or a requirements document) is a program: I don't know what happens after it enters the brain. What it does may sometimes be well analogized by the running of a program – that is, thinking that way might allow us to make better predictions about what will happen when someone reads a requirements document – but sometimes it might not. Rather than taking one approach to the limit, we'll need what James Bach (1999) calls "diverse half measures": a number of measures, each inadequate on its own, that in combination do better than any one of them possibly could.

In the absence of a single theory about what the brain is doing, let's concentrate on what it does things with: the inputs. Those can be divided into three kinds: statements (verbal or written), actions (smacking the programmer upside the head), and the setting (passive inputs gathered by the receiver, not projected by the sender).

I'm going to skip talking about setting, despite its importance. Drug users can elaborately arrange their environment (Leary et. al. 1963, Zinberg 1986) to maximize the effects they experience. The same person, taking the same drug in the same dosage, might experience very different effects in one setting than another. I expect that's relevant to debates such as offices vs. cubicles vs. bullpens, but I don't have anything useful to say about it.

I will also combine statements and actions. At least some statements *are* actions. Consider "I now pronounce you man and wife" above or "I christen thee *The Luisitania*". They are what Austin (1975) calls *performatives*. Derrida thinks all statements can be treated as performatives (Nilges, in press). I'm not sure of his argument, but that seems reasonable to me. Consider the statement by a person of influence that "there are no good chairs here". That's a statement about the world, but it's also likely a "speech act" that will cause someone with lower status to fetch a chair. Furthermore, it causes that person to make a specific interpretation of "good chair" – not as a universal, but as a set of objects that will cause the high status person pleasure when it's brought to her. A hearer is always actively (if implicitly) interpreting statements in terms of what the consequences of particular interpretations will be, which – given that we're primarily social animals – means that we wonder what *they'll* do should we do *X* in response to message *Y*.[3]

If all statements are actions, it doesn't make much sense to treat them separately. That allows us include physical actions as a part of speech, which seems more abstract. The written statement "the input field must accept all unicode characters" has a different effect than do the same words spoken and accompanied with a vigorous pointing gesture.

[3] Dennett (1992) proposes that consciousness was created through this mechanism. As social beings, we must become adept at making good – reasonably accurate, easy to use – models of other people's behavior. Consciousness is what results when that ability is turned on its possessor. (Greg Egan's novel *Diaspora* contains a narration of an individual's creation of consciousness by this means.)

Learning from the bottom up

Here's an example of causing the right behavior without any obvious use of a map or a conduit.

> My wife Dawn teaches veterinary students how to cure cows. Each sick cow is assigned a student and each day that student has to decide—among other things—whether the cow is bright or dull. Students usually err on the side of bright. It's Dawn's job to correct them. She and a student will stand near a misclassified cow, and Dawn will say, "This cow is dull. See—it's not cleaning its nostrils."[4]
>
> From this conversation, the student will create a rule, a little bit of a map of the world:
>
> > (1) If a cow looks bright, but it's not cleaning its nostrils, it's dull.
>
> That rule might work for a while, but eventually Dawn and the student will be standing beside a cow, and Dawn will say, "That cow is dull." The student will say, "But it's cleaning its nostrils!" To which, Dawn will reply, "But its ears are droopy."
>
> Now the student adds a new rule:
>
> > (2) Droopy ears mean dull, and perky ears mean bright.
>
> But those two rules don't work either. It might take a while for the student to be able to reliably judge between bright and dull. What's interesting is that, when she does, *she's lost the rules*. She can't articulate any complete set of rules that define bright or dull. In fact, the notion of decision rules seems somewhat beside the point. Cows simply *are* either bright or dull, the way the student herself is either alert or sleepy, or the way a joke is either funny or lame. Any explanation of *how* she knows seems contrived and after the fact. It's as if the student's perceptual – not conceptual – world has expanded.

Several interesting things are happening here.

1. This is an example of what Lave and Wenger (1991) call *legitimate peripheral participation*. What the students are doing is legitimate because what they do *matters*; this is a real cow that will either go home or to the cooler. It's not a classroom exercise.

   The students are peripheral in that they begin by having unsupervised responsibility for very little. (As my first boss put it, "we're going to put you somewhere where you can't do much damage.") As they gain experience, their decisions become more central to the success or failure of the case.

   The students participate in a larger group activity, where they have frequent opportunities to talk with both experts and learners like themselves.

2. The process is highly iterative. Early in our courtship, I realized that it was no wonder that Dawn was so much more competent at her job than I was at mine: in

[4] Trust me. You don't want to know how cows clean their nostrils.

the time I spent working toward a single product release, she would see hundreds of cases through from start to finish. The feedback on her work was immensely faster than on mine. Further, the example shows that a student gets expert, useful feedback not just at the end of the case, but continually throughout it.

3. This learning is driven by examples. We can say the students generalize from them (while remaining agnostic about whether the generalizations are expressible in any language). That's different from the "top-down" approach in which students are taught general rules and then how to apply them to particular cases. The bottom-up approach seems highly risky: how can you be sure students trained at Illinois will make the same judgments as students from Colorado State? You can't, I suppose: but, nevertheless, they do.
4. Learning is an explicit goal of all involved. The students know that every case is about learning veterinary medicine, not merely about curing one cow. That affects the way Dawn speaks to a student when making a correction. It also causes the professors to organize explicit sessions devoted to review and extension of learning ("rounds").

I hasten to say that not everything is learned this way. For example one *can* give rules that distinguish between the diagnostic categories "alert" and "depressed". At some point, physicians recognized that it made practical sense to distinguish those states – they generalized from examples – and were able to make those states visible even to amateurs. Would that everything were so simple. Certainly much software is not.

Implications

The job of software development is to produce a product that people will pay money for, plus an organization that will be funded to build the next product. I'm assuming that it is impossible to communicate what the successful product will be, even were people forbidden to change their minds. What's needed is an iterative approach that allows frequent correction:

> "I think this is a satisfactory product."
>
> "Yes, so far as it goes, but it also needs *X*."
>
> "It now has *X*, so it's a satisfactory product."
>
> "No, that's not really an *X*. To make it an *X*, do this."
>
> "Now it's an *X*!"
>
> "It's a good enough one for now. Now it needs a *Y*."

Just as Dawn doesn't evaluate her student's preparation to evaluate a case, but rather the evaluation itself, a project team's output should be an actual working product at frequent intervals. It must be evaluated by the same person or people who will evaluate the final product.[5] This desire for iteration, unsurprisingly, fits with the Agile methods. (Unsurprisingly, because I'm an advocate for those methods.)

[5] The more people who have influence over whether the product was worth the money, the harder that is. You quickly run into problems because you have to select mouthpieces

Legitimate peripheral participation has apprentices learn by starting with activities that are simultaneously simple and low risk. As they learn, the risk and difficulty grow together. I don't know, but I suppose that the complexity factor is about easing learning while the risk factor is about protecting the master's livelihood from the apprentice.

In Agile software projects, the emphasis is often on making the early iterations the most valuable. Each successive iteration should provide less value. When the next iteration would not be worth the cost, you stop the project. That's a risk-reduction strategy: it minimizes the risk that the project sponsor will get impatient and disrupt or cancel the work.

The lesson of legitimate peripheral participation is that the early iterations should also be easy. In a project that's starting from scratch, it seems fairly easy to put the valuable features in early. It's more difficult in legacy code, which suggests that the project must balance ease and risk. The project should also keep in mind that it's explicitly a learning project, that one output of each iteration should be the team's increased ability to make decisions pleasing to their "professor". It appears that Agile practice is learning how to strike a balance among many factors when scheduling; see, for example, the discussion in Cohn (in press).

The need to produce greater capacity from each iteration produces a question: is it better for schedules to be broad or narrow? Suppose a product has three constituencies. Which leads to most efficient learning of the business domain: satisfying each constituency in each iteration, thus getting knowledge of all parts of the domain, but knowledge of any given part at a slow rate? Or concentrating first on one domain, thus learning that part of the domain well but the others little or not at all? I don't know, so I change the subject.

In our field, we underrate examples. We – especially those of us with programming roots – love abstractions. Like the stereotypical mathematics text, with its repetitive sequence of definition-theorem-proof, we have a tendency to assume that those who come after us needn't follow the laborious process of coming to the right definition.

I suggest that people explaining business domains reduce the number of speeches that begin "A bond is a…" and increase the number that begin "Suppose you have $5,000 and you want to buy the simplest kind of bond. You *could*…" My preference for examples is to begin with step-by-step descriptions akin to use cases (Cockburn 2000), though I generally prefer more "implementation detail" than is considered wise in use cases. While I recognize that implementation detail that comes too early can lead to design decisions made thoughtlessly, there are counterbalancing advantages to detail:

- It's not uncommon for a completely arbitrary choice to reveal something about the business domain, especially when people choose outlandish, fun examples (Buwalda 2004). As a simple example, someone thinking of an unusual example for a ZIP code field would surely use Canadian postal codes (which, unlike US codes, contain letters, and are used by people who get annoyed – albeit politely – when programs assume all the world's the US) or countries that have no postal codes (a problem if it's a required field).

---

for a large population. Nevertheless, the team will learn what's wanted faster and better if their finished work is evaluated more frequently.

- Anyone who's read about the handling of a sports car and then given one of those cars a test drive should know that you learn more – or perhaps just differently – from a test drive. That's especially true if you and the magazine reviewer don't already share common interpretations of words. There's a difference between talking *about* something (at a remove) and using it. The use of specific detail makes it easier for people to imagine use.

These sort of step-by-step examples are often used as tests, sometimes tests written before the code, in test-driven design fashion (Beck 2002, Astels 2003). I've come to believe that those kinds of tests are often a bad practice. What seems to work better is to use the collection of examples in what feels like a "boiling down" process, so that the resulting tests contain only those facts necessary for the most concise examples possible. (See Mugridge and Cunningham 2005.) Such tests are an interesting combination of abstraction and specificity: unlike normal requirements, they contain exact details, but only the details necessary for understanding.

When such examples are constructed in conversation (as they should be, so that people can ask questions), the examples themselves are likely the only thing that need be written down (on a whiteboard). When people refer to them later, it's likely they'll need a summary – probably abstract – of what the examples are all about. Those summaries might well begin "a bond is…" In that sense, I have a motto: *conventional requirements are merely commentaries upon examples*.

Ubiquitous languages as creoles

Another story, again based on personal communication:

> Ward Cunningham's team was working on a bond trading application called WyCash. It was to have two advantages over its competition. First, it would be more pleasant to work with. Second, users would be able to generate reports on a position (a collection of holdings) as of any date.
>
> As the team worked on features requiring them to track financial positions over time, some code got messier and messier and harder to work with. The team was taking longer to produce features, and they created more bugs.
>
> Much of the problem was due to a particular method (chunk of code) that was large and opaque.At some point, Ward's team made a concerted effort to simplify it by turning it into a *method object*. A method object is one that responds to a single command: "do whatever it is that you do".[6] A new kind of object has to have a name; they picked Advancer, because it came from the method that advanced positions. For technical reasons that don't concern us here, method objects are useful when modifying overly complex code. They're often an intermediate step - you convert a bad method into a method object, clean it up by splitting it into smaller methods, then move those to the classes where they really belong. This team, however, left the method object in the program. The reasons

[6] You can find more about method objects in Fowler's *Refactoring* (1999), the canonical text on how to make code better without changing its behavior. There is a whole craft around refactoring. *Refactoring* talks of that, as do Wake's *Refactoring Workbook* (2003) and Kerievsky's *Refactoring to Patterns* (2004).

are lost. Perhaps, as is often the case, the right way to split it up wasn't apparent. Perhaps they already knew it was useful.

Because it *was* useful. As the project dealt with the normal stream of changes, the programmers found they could get an intellectual grip on them by thinking about how to change existing Advancers or create new ones. They wrote better code faster. It seemed as if Advancers must correspond to something in the business world (must map to something "out there"). So the programmers asked the experts what Advancers were "really" called – but the experts didn't have a name. "Advancer" wasn't an idea that bond traders had. So the programmers kept the name and continued to figure out what it meant by seeing how it participated in program changes.

For example, the program calculated tax reports. What the government wanted was described in terms of positions and portfolios, so the calculations were implemented by Position and Portfolio objects. But there were always nagging bugs. Some time after Advancers came on the scene, the team realized they were the right place for the calculation: it happened that Advancers contained exactly the information needed in their instance variables. Switching to Advancers made tax reports tractable. Another gain in the team's capability, and a further understanding of what Advancers were about.

It was only in later years that Cunningham realized why tax calculations had been so troublesome. The government and traders had different interests. The traders cared most about their positions, whereas the government cared most about how traders came to have them. It's the latter idea, one that the experts did not know how to express, that Advancers capture. And once it's captured, the complexities of tax calculations collapse into (relative) simplicity. But at no point in the story did the programmers specifically set out to invent something new in the language of bond trading. They were only trying to generate the required reports while obeying rules of code cleanliness.

This story reminds me of two related ideas from science studies. The first is what Star and Griesemer (1989) call *boundary objects*. They have several important properties:

- If *x* is a boundary object, people from different communities of practice can use it as what Chrisman (1999) calls a *common point of reference* for conversations. They can all agree they're talking about *x*.
- But the different people are not actually talking about the same thing. They attach *different meanings* to *x*. An Advancer to a bond expert is a novel way of talking about the history of positions. To a programmer, it's a chunk of code that allows certain tasks to be done in certain ways.
- People use boundary objects as a *means of coordination and alignment* (Fischer and Reaves 1995). Advancers are a way for business people to tell programmers what to do with increased confidence that they'll be pleased by the results. They allow different people to *satisfy different concerns simultaneously*.
- Despite different interpretations, boundary objects serve as a *means of translation*. If it becomes important that a programmer understand more about bond trading, the business person can use Advancers to create telling examples.

- Boundary objects are *working arrangements*, adjusted as needed. They are not imposed by one community, nor by appeal to outside standards (Bowker and Star 1999).

Star and Greisemer are using physical objects as an analogy. Galison (1997) uses language. In his study of how experimental and theoretical physicists work together, he describes them creating what he calls "creoles" by analogy to the trading languages developed at shared boundaries of cultures.

Galison adds, I think, an extension to the semi-common project practice of creating what Evans (2003) calls a *ubiquitous language*. Such a language is composed of nouns and verbs spoken by the project team and also found in the text of the program (as class and method names). Those nouns and verbs allow the same coordination as boundary objects do.

Galison steers us away from thinking of the ubiquitous language as being discovered in the business domain. Instead, it's mutually created, over time – just as Advancers were. And, like Advancers, the words can come from either domain – the business domain or the programming domain. What matters is successful coordination and effective learning.

In a way, creoles take us full circle. No matter what really happens within brains, the ubiquitous language lets everyone involved in a project make a shared (enough) map of the world. It's not transmitted down a conduit from person to person – it's created word by word, example by example, as people converse and correct.

References

Ashby, Ross (1948). <<<missing>>>. Quoted in Pickering (in press).

Astels, David. (2003) *Test Driven Development: A Practical Guide*.

Austin, J.L. (1975). *How to Do Things With Words*. (2/e)

Bach, James. (1999) "Heuristic Risk-Based Testing", *Software Testing and Quality Engineering*, November.

Beck, Kent. (2002) *Test-Driven Development: By Example.*

Bloom, Harold. (1997) *The Anxiety of Influence: A Theory of* Poetry.

Bowker, G., and S.L. Star. (1999) *Sorting Things Out: Classification and its Consequences*

Buwalda, Hans. (2004) "Soap Opera Testing", *Better Software*, February.

Chrisman, Nicholas. (1999) "Trading Zones or Boundary Objects: Understanding Incomplete Translations of Technical Expertise", 4S San Diego, 1999.http://faculty.washington.edu/chrisman/Present/4S99.pdf

Cockburn, Alistair (2000) *Writing Effective Use Cases*.

Cohn, Michael. (in press) *Agile Estimating and Planning*.

Culler, Jonathan. (1983) *On Deconstruction: Theory and Criticism after Structuralism.*

Daniel Dennett. (1992) *Consciousness Explained.*

Evans, Eric. (2003) *Domain-Driven Design: Tackling Complexity in the Heart of Software*

Fish, Stanley. (1980) *Is There a Text in this Class?: The Authority of Interpretive Communities*.

Fischer, G., and B.N. Reeves. (1995) "Creating Success Models of Cooperative Problem Solving", in Baecker et. al. (eds), *Readings in Human-Computer Interaction: Toward the Year 2000*.

Fowler, Martin. (1999) *Refactoring: Improving the Design of Existing Code*

Galison, Peter Louis. (1997) *Image and Logic: a Material Culture of Microphysics*

Gombrich, E. H. (1995) *The Story of Art*. (16/e)

Kerievsky, Joshua. (2004) *Refactoring to Patterns*.

Lakoff, George (1990) *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind.*

— and Mark Johnson. (2003) *Metaphors We Live By.* (2/e)

Lave, Jean and Etienne Wenger. (1991) *Situated Learning: Legitimate Peripheral Participation.*

Leary, T., G. Litwin, and R. Metzner. (1963) "Reactions to psilocybin administered in a supportive environment." *Journal of Nervous and Mental Disease*, 137.

Mugridge, Rick and Ward Cunningham. (2005) *Fit for Developing Software: Framework for Integrated Tests*.

Nilges, Edward. (in press) "Deconstruction for Programmers". To be published in the 2005 *OOPSLA Companion.*

Pickering, Andrew (in press). *The Cybernetic Brain in Britain*.

Reddy, M.J. (1979) "The conduit metaphor – a case of frame conflict in our language about language." In A. Ortony (ed.), *Metaphor and Thought*.

Rorty, Richard. (1981) *Philosophy and the Mirror of Nature*.

Star, S.L., and J.R. Griesemer. (1989) "Institutional Ecology, 'Translations', and Boundary Objects: Amateurs and Professionals in Berkeley's Museum ofVertebrate Zoology 1907-39", Social Studies of Science, Vol. 19.

Wake, William C. (2003) *Refactoring Workbook.*

Zinberg, Norman. (1986) *Drug, Set, and Setting.*

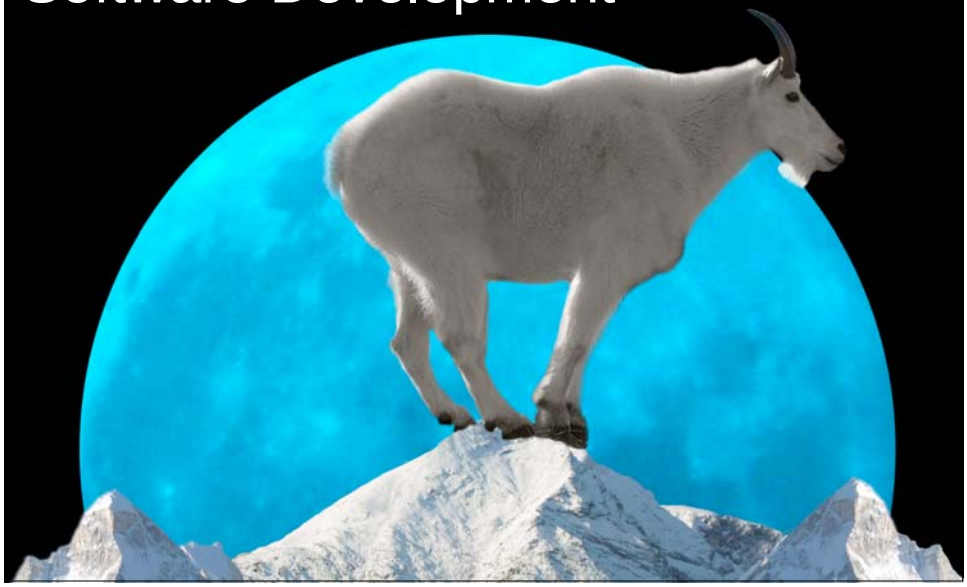
User Stories For Agile Software Requirements

Mike Cohn
Mountain Goat Software, LLC

The technique of expressing requirements as user stories is one of the most broadly applicable techniques introduced by Extreme Programming. User stories are an effective approach on all time constrained projects, not just those using XP. In this talk, we will look at how to identify and write good user stories. The talk will describe the six attributes all good stories must exhibit and present thirteen guidelines for writing better stories. We will explore how user role modeling can help when gathering a project's initial stories.

Mike Cohn is the founder of Mountain Goat Software, a process and project management consultancy and training firm. He is the author of User Stories Applied for Agile Software Development and Agile Estimating and Planning, as well as books on Java and C++ programming. With more than 20 years of experience, Mike has previously been a technology executive in companies of various sizes, from startup to Fortune 40. A frequent magazine contributor and conference speaker, Mike is a founding member of the AgileAlliance, and serves on its board of directors.

User Stories for Agile Software Development

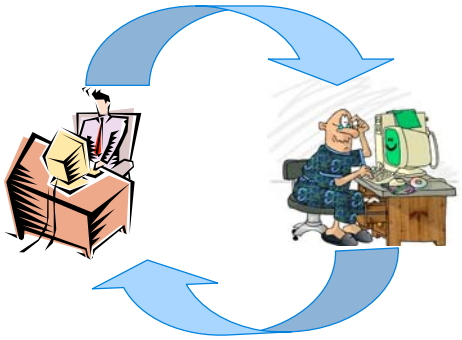


Mike Cohn—background

- Programming for 20 years
- Author of
  - *User Stories Applied*
  - *Agile Estimating and Planning*
  - Java, C++, database programming books
- Founding member and director of the Agile Alliance and the Scrum Alliance
- Founder of Mountain Goat Software
  - Process and project management consulting and training

What problem do stories address?

- Software requirements is a communication problem
- Those who want software must communicate with those who will build it



Balance is critical

- If either side dominates, the business loses
- If the business side dominates…
  - …functionality and dates are mandated with little regard for reality or whether the developers understand the requirements
- If the developers dominate…
  - …technical jargon replaces the language of the business and developers lose the opportunity to learn from listening

Resource allocation

- We need a way of working together so that resource allocation becomes a shared problem
- Project fails when the problem of resource allocation falls too far to one side

Responsibility for resource allocation

If developers shoulder the responsibility…

- May trade quality for additional features
- May only partially implement a feature
- May solely make decisions that should involve the business side

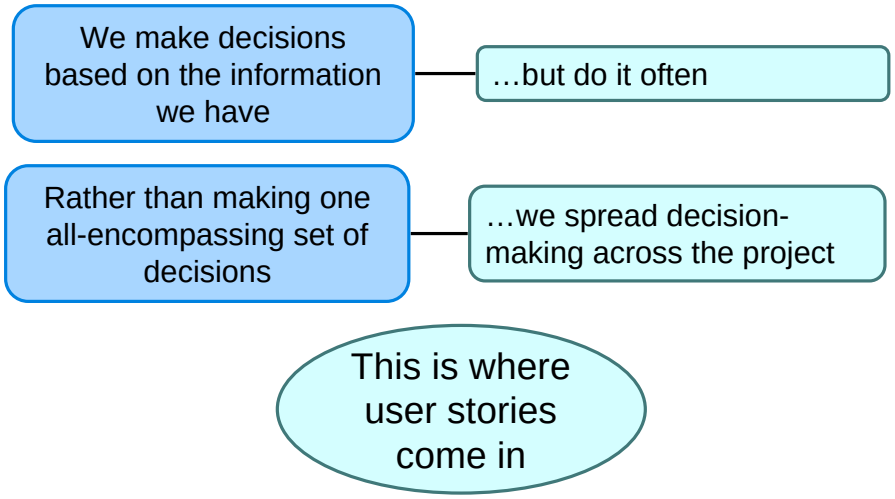
If the business shoulders the responsibility…

- Lengthy upfront requirements negotiation and signoff
- Features are progressively dropped as the deadline nears

Imperfect schedules

- We cannot perfectly predict a software schedule
  - As users see the software, they come up with new ideas
  - Too many intangibles
  - Developers have a notoriously hard time estimating
- If we can't perfectly predict a schedule, we can't perfectly say what will be delivered

So what do we do?



Today's agenda

- ❑ What stories are
- ❑ User role modeling
- ❑ Story writing
- ❑ INVEST in good stories
- ❑ Why user stories?

Ron Jeffries' Three Cs

Card

- Stories are traditionally written on note cards.
- Cards may be annotated with estimates, notes, etc.

Conversation

- Details behind the story come out during conversation with customer

Confirmation

- Acceptance tests confirm the story was coded correctly

Samples – Travel reservation system

As a user, I can reserve a hotel room.

As a vacation planner, I can see photos of the hotels.

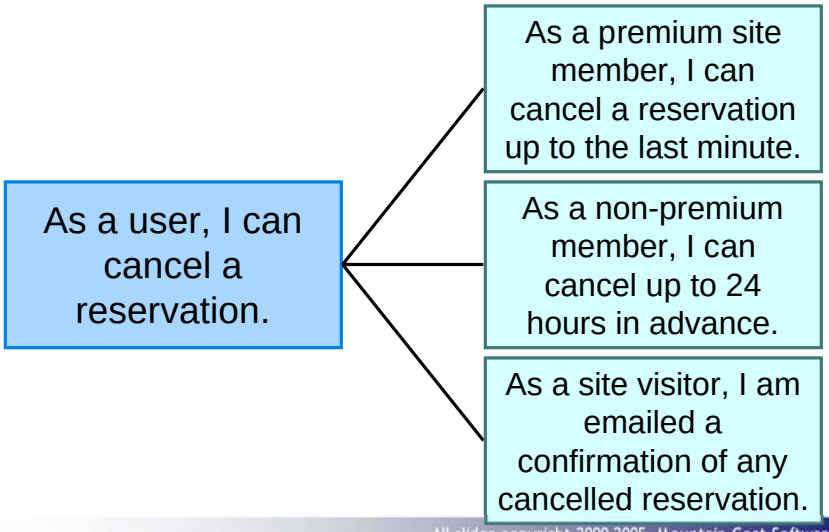
As a user, I can cancel a reservation.

As a user, I can restrict searches so I only see hotels with available rooms.

Where are the details?

- As a user, I can cancel a reservation.
  - Does the user get a full or partial refund?
    - Is the refund to her credit card or is it site credit?
  - How far ahead must the reservation be cancelled?
    - Is that the same for all hotels?
    - For all site visitors? Can frequent travelers cancel later?
  - Is a confirmation provided to the user?
    - How?

Details added in smaller “sub-stories”



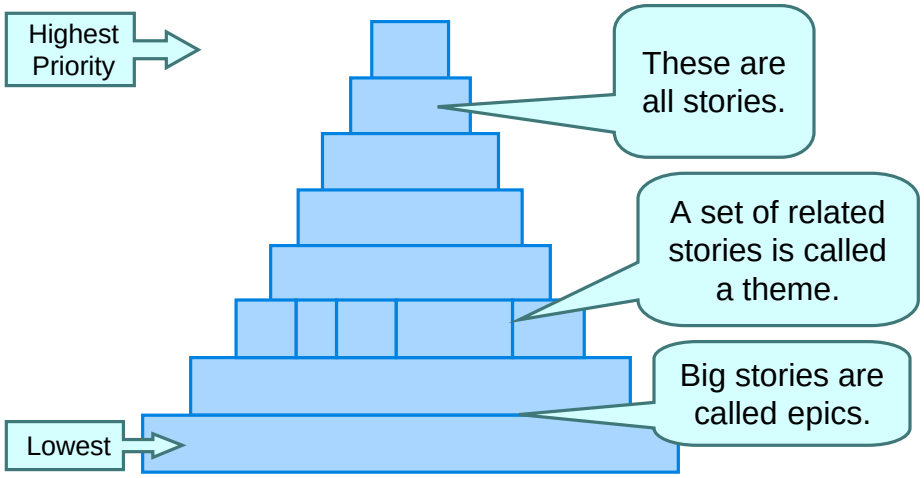
Details added as tests

- High level tests are added to the story
  - Can be used to express additional details and expectations

As a user, I can cancel a reservation.

- Verify that a premium member can cancel the same day without a fee.
- Verify that a non-premium member is charged 10% for a same-day cancellation.
- Verify that an email confirmation is sent.
- Verify that the hotel is notified of any cancellation.

Prioritized requirements list (PRL)



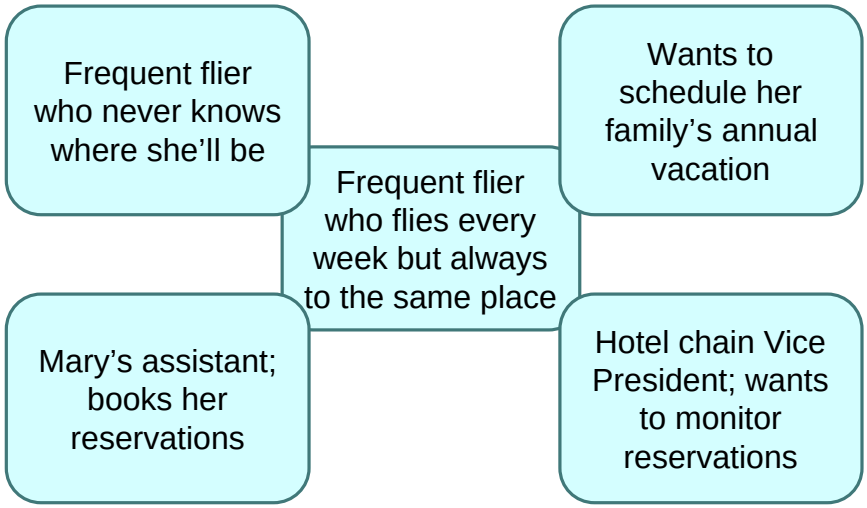
Today's agenda

- ❑ What stories are
- ❑ User role modeling
- ❑ Story writing
- ❑ INVEST in good stories
- ❑ Why user stories?

“The User”

- Many projects mistakenly assume there’s only one user:
  - “The user”
- Write all stories from one user’s perspective
- Assume all users have the same goals
- Leads to missing stories

Travel Site—Who’s the user?

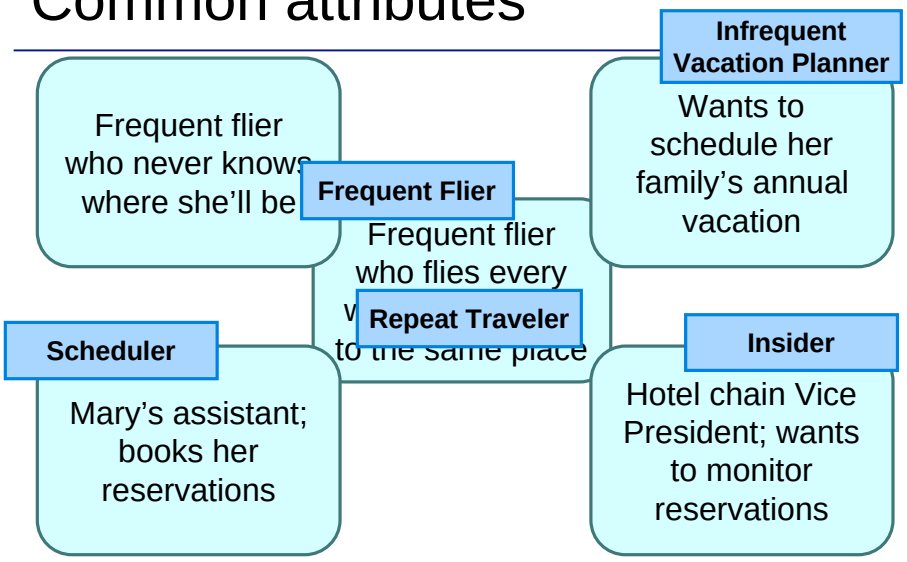


User roles

- Broaden the scope from looking at one user
- Allows users to vary by
  - What they use the software for
  - How they use the software
  - Background
  - Familiarity with the software / computers
- Used extensively in usage-centered design
- Definition
  - A user role is a collection of defining attributes that characterize a population of users and their intended interactions with the system.

Source: *Software for Use* by Constantine and Lockwood (1999).

Common attributes



Advantages of using roles

Users become tangible	Start thinking of software as solving needs of real people.
Avoid saying “the user”	Instead we talk about “a frequent flier” or “a repeat traveler”
Incorporate roles into stories	“As a <role>, I want <story> so that <benefit>.”

Today’s agenda

- ❑ What stories are
- ❑ User role modeling
- ❑ Story writing
- ❑ INVEST in good stories
- ❑ Why user stories?

Questioning the users

"Would you like it in a browser?"

"Of course, now that you mention it!"

- A problem
  - The question is closed
    - {Yes | No}

We can do better

"What would you think of having this app in a browser rather than as a native Windows application, even if it means reduced performance, a poorer overall user experience, and less interactivity?"

- It's open
  - Full range of answers
- But it has too much context

The best way to ask

"What would you be willing to give up in order to have it in a browser?"

- We want to ask questions that are
  - Open-ended
  - Context-free

It's my problem, I know the solution

- Having a problem does not uniquely qualify you to solve it
- "It hurts when I go like this…"

We need to stop asking users

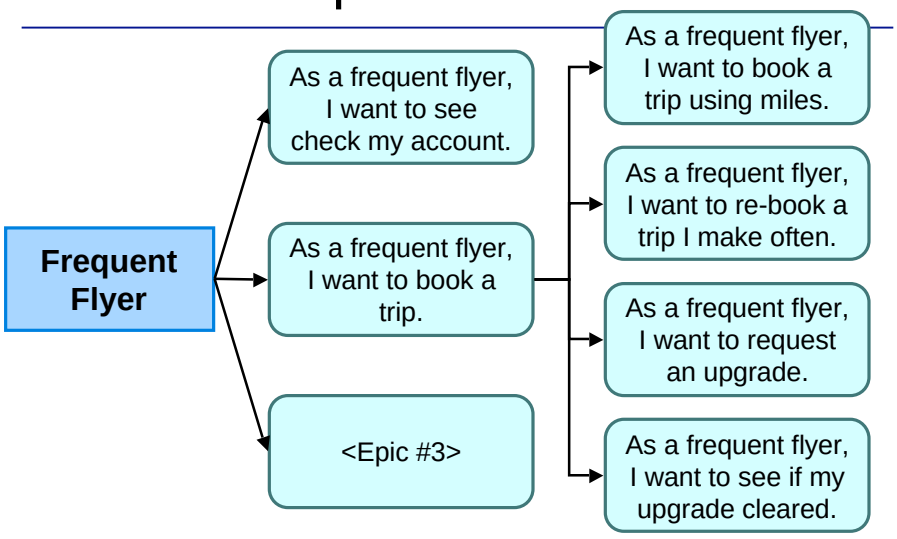
- Since users don't know how to solve their problems, we need to stop ***asking***
- We need to ***involve*** them instead

Empirical design	Designers of the new system make decisions by studying prospective users in typical situations
Participatory design	The users of the system become part of the team designing the behavior of the system

Story-writing workshops

- Includes developers, users, customer, analysts, designers, everyone
- Goal is to brainstorm stories for the system
  - Some will be "implementation ready"
  - Others can be "epics"

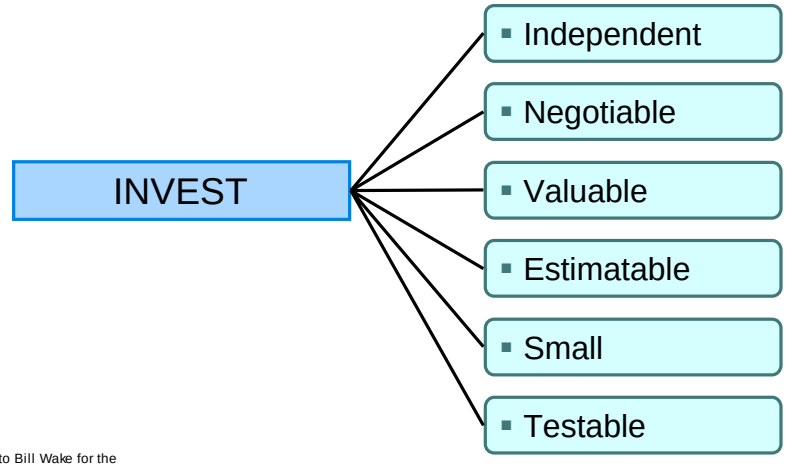
Start with epics and iterate



Today's agenda

- ❑ What stories are
- ❑ User role modeling
- ❑ Story writing
- ❑ INVEST in good stories
- ❑ Why user stories?

What makes a good story?



Thanks to Bill Wake for the acronym. See www.xp123.com.

INVESTing in good stories

- **I**ndependent
  - Dependenices lead to problems estimating and prioritizing
  - Can ideally select a story to work on without pulling in 18 other stories
- **N**egotiable
  - Stories are not contracts
  - Leave or imply some flexibility
- **V**aluable
  - To users or customers, not developers
    - Rewrite developer stories to reflect value to users or customers

INVESTing in good stories

- **E**stimatable
  - Because plans are based on user stories, we need to be able to estimate them
- **S**mall
  - Complex stories are intrinsically large
  - Compound stories are multiple stories in one
- **T**estable
  - Stories need to be testable

Small

- Large stories (epics) are
  - hard to estimate
  - hard to plan
    - They don't fit well into single iterations
- Compound story
  - An epic that comprises multiple shorter stories
- Complex story
  - A story that is inherently large and cannot easily be disaggregated into constituent stories

Compound stories

- Often hide a great number of assumptions

As a user, I can post my resume.

- A resume includes separate sections for education, prior jobs, salary history, publications, etc.
- Users can mark resumes as inactive
- Users can have multiple resumes
- Users can edit resumes
- Users can delete resumes

Splitting a compound story

Split along operational boundaries (CRUD)

- As a user, I can create resumes, which include education, prior jobs, salary history, publications, presentations, community service, and an objective.
- As a user, I can edit a resume.
- As a user, I can delete a resume.
- As a user, I can have multiple resumes.
- As a user, I can activate and inactivate resumes.

Splitting a compound story, cont.

Split along data boundaries

- As a user, I can add and edit educational information on a resume.
- As a user, I can add and edit prior jobs on a resume.
- As a user, I can add and edit salary history on a resume.
- As a user, I can delete a resume.
- As a user, I can have multiple resumes.
- As a user, I can activate and inactivate resumes.

Other ways to split large stories

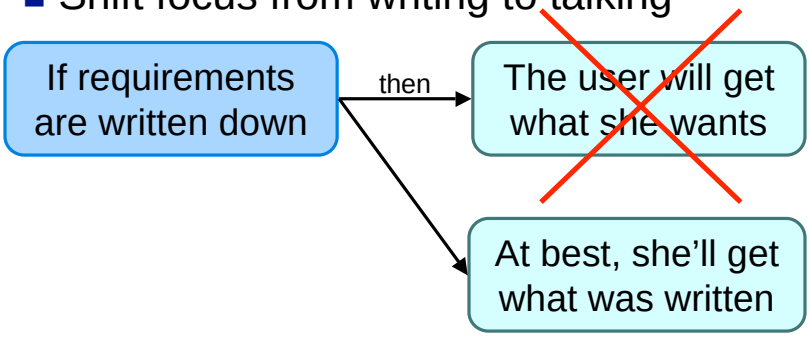
- Remove cross-cutting concerns
- Don't meet performance targets
- Avoid splitting stories into tasks
- Avoid the temptation of related changes

Today's agenda

- ❑ What stories are
- ❑ User role modeling
- ❑ Story writing
- ❑ INVEST in good stories
- ❑ Why user stories?

So, why user stories?

- Shift focus from writing to talking



- "You built what I asked for, but it's not what I need."

Words are imprecise

Entrée comes with soup or salad and bread.

- (Soup or Salad) and Bread
- (Soup) or (Salad and Bread)

Actual examples

The user can enter a name. It can be 127 characters.

- Must the user enter a name?
- Can it be other than 127 chars?

The system should prominently display a warning message whenever the user enters invalid data.

- What does *should* mean?
- What does *prominently display* mean?
- Is *invalid data* defined elsewhere?

Another real example

"I handed in a script last year and the studio didn't change one word."

"The word they didn't change was on page 87."

~Steve Martin

Additional reasons

- Stories are comprehensible
  - Developers and customers understand them
  - People are better able to remember events if they are organized into stories[†]
- Stories are the right size for planning
- Support and encourage iterative development
  - Can easily start with epics and disaggregate closer to development time

[†]Bower, Black, and Turner. 1979. *Scripts in Memory for Text.*

Yet more reasons

- Stories support opportunistic development
  - We design solutions by moving opportunistically between top-down and bottom-up approaches†
- Stories support participatory design
  - Participatory design
    - The users of the system become part of the team designing the behavior of the system
  - Empirical design
    - Designers of the new system make decisions by studying prospective users in typical situations

†Guindon. 1990. *Designing the Design Process.*

Most importantly…

Don't forget the purpose

- The story text we write on cards is less important than the conversations we have.
- "Stories represent requirements, they do not document them."†

†Rachel Davies, "The Power of Stories," XP 2001.

Mike Cohn contact information

- mike@mountaingoatsoftware.com
- (303) 810–2190 (mobile)
- (720) 890–6110 (office)

- www.mountaingoatsoftware.com

Mountain Goat SOFTWARE

The Professional Tester's Toolbox

Julie Fleischer
Intel Corporation
Julie.N.Fleischer@Intel.com

Those who have been in the software testing field a while definitely consider software testing a profession with a skill set to master and refine. However, they have probably also encountered those new to the field or new to interacting with software testers that do not realize the amount of training and experience that professional testers have acquired. This paper will discuss those testing tools that professional testers have placed in their toolbox and will give the reader information on the profession of being a tester.

Biography

Julie Fleischer is a software engineer at Intel Corporation, and she has held a variety of software engineering roles, including test engineer, QA lead, and, recently, program manager for validation efforts. Throughout these roles, she has received as well as provided direction, training, and guidance on QA processes and procedures. This paper is a result of her work with her memorable QA teammates.

Introduction

Those who have been testers for a while understand the excitement of the profession. They may have immediately connected with the skeptical mindset required or been attracted to the broad range of business and technical expertise they would acquire over the course of a career, or they may enjoy the challenge of becoming a skillful negotiator that consistently champions doing the right things. They most likely have been educated by talented and motivated professionals that are also convinced that testing is a profession requiring a broad range of technical and communication skills.

However, they have probably also encountered the reverse view of the profession. They may have spoken with managers who believed that any developer could test, but only the really talented and motivated testers could develop. They may have watched their testing peers become frustrated at *having* to do testing when the "real work" was being done by the developers. Or, they may have watched their testing peers become disillusioned with the finesse required to influence the quality of a product.

But, chances are they have acquired a set of skills that enable them to be successful in spite of these naysayers. They most likely have a toolbox full of tools that they can pull out to accomplish their testing goals and to show others that testing is a challenging and exciting profession. They know how to test, are familiar with the software lifecycle, can easily communicate with and influence stakeholders, have a detective mindset, and have a passion for their job. These are the tools of a professional tester, and this paper will give the details on these tools for any tester that wants to know how to make testing a career-long profession.

The Toolbox

Tool #1: Knowledge of how to thoroughly test a product

A common misconception of the uninitiated is to define testing too narrowly. New testers might create and execute a test script in a few hours, and think they have completed their "testing" effort. An initial test suite may contain plenty of functional test cases of typical behavior, but ignore error conditions. However, professional testers have a broad knowledge of how to test and use this knowledge to create thorough test suites.

Professional testers have taken the time to hone their testing knowledge. They know the different types of testing, including the basics of functional, performance, and stress testing as well as other the less-discussed test types, such as acceptance testing or ad-hoc testing. They also know how to plan testing to manage risks.

This tool consists of the following skills:

Comprehension of the different types of testing is an overlooked skill. Many people think of functional testing (testing the behavior of the product under normal conditions) first when they think of testing. This makes sense since most customer requirements tend to be towards functionality, and the correct functionality needs to be present. However, it also ignores many other aspects of a product.

Many people also think of performance and stress testing (testing to determine the optimal conditions and limits of the system with respect to things like memory, CPU, or bandwidth) if they are testing a product with critical performance requirements (ex. a highly available web server). However, performance and stress testing can apply to all products to some degree. Performance and stress testing also tend to require quite a bit of training and experience to perform correctly. Some professional testers have spent parts of their career investigating the tools and procedures needed to do these things well, such as memory leak detection tools, commands to overload the CPU, and others.

Testing error conditions and negative test cases are also often overlooked. Professional testers are good at thinking up potential error scenarios and negative test cases.

More than that, there are other types of testing that have harmful effects if neglected. Install/uninstall procedures, product documentation, integration of the product with the targeted hardware, OS, BIOS, or software, and the backward compatibility of the product are just some.

Knowledge of how and when to create basic acceptance tests can save testers time. Most testers have experienced the frustration of setting up a test environment for a full test pass and discovering halfway through that key functionality is broken and the full test pass cannot be performed. Basic acceptance tests are designed to prevent this waste of time. These are tests of the basic functionality of the product. In general, these tests must pass before testing will begin, or, at the least, testers must be able to run through these, document all issues, and make a judgment call as to whether it would be worthwhile to continue testing.

Knowing how to successfully use a regression suite is also a valuable skill. Bugs tend to recur, and customers (not to mention management) tend to have less patience for issues they have seen before. Some teams use their basic acceptance suite as a regression suite to ensure previously existing functionality isn't broken in a new release. Some teams also add test cases that verify bug fixes of severe defects into their regression suite. For example, if the system hung when the tester tried to put it into suspend mode while playing an audio file, the tester would create a regression test case which includes the steps 1) play an audio file, 2) attempt to put the system into suspend mode, and 3) attempt to return to normal mode.

The discipline to test towards requirements is a skill that enables testers to ensure that their work is accurately representing all of the customers' needs. Professional testers find a way to ensure they are testing towards formal requirements, using traceability tools if needed. When requirements are not available, professional testers seek out requirements information, such as related product documentation (design documents, product specifications, marketing documents, and other written collateral), key player input (marketing, developers, management, and customers), and also examine the behavior of the product to "reverse engineer" requirements.

However, professional testers also take the responsibility to ***test towards more than the requirements***. Professional testers use their knowledge of the various aspects of a quality product (correct functionality, graceful error handling, performance, and the other items described in the testing types) and ensure that all applicable aspects are tested. When possible, they will also request changes to the requirements to reflect the aspects that their test cases covered that the requirements documentation did not.

Testing towards more than the test plan is also a great habit of professional testers. Professional testers don't just stop after all tests have been run. They take the time to investigate things that "don't seem right," and they also investigate areas where they feel bugs may exist. Some teams have even recognized the benefit of this type of testing and formalized it into an ad-hoc testing phase.

The ability to automate is also an important skill for most testers to have. Most professional testers have software development experience and treat their automation efforts as software development projects. They also understand when to automate and what to automate, knowing that it's not always desirable to automate everything as soon as possible. Testers who make automation the ultimate goal of testing, thinking that testing is not an efficient use of their time unless it's automated testing, can often lose sight of the other types of testing, most notably ad-hoc testing. In addition, testers that only run automated tests can lose the ability to investigate curious issues seen during testing.

Viewing test planning as managing risks is also a skill that professional testers use to effectively plan and communicate testing efforts. Testers cannot test everything, so they need a well-defined plan that explains how they will ensure that the best results for the customer. Testing efforts are planned around factors such as feature importance to the customer and perceived stability of the code (based on factors

such as the complexity of the design, the ability of the developer, the amount of reused code, and the past stability of the feature).

Tool #2: A detective attitude

An outsider who sees only defect reports may think that testers just document the obvious and miss the real forensics work that professional testers perform.

Professional testers have a detective mindset focused on revealing the critical bugs in a system before the customer does. They are inherently suspicious and investigate any out of the ordinary behavior to see if a bug is lurking underneath. They have ***strong problem solving skills*** to uncover the root cause of the bugs they find. They have great perseverance to keep digging to find the cause of a bug or to run the same test suite again on a new release to see if anything has broken.

Professional testers also ***use past experiences to help guide future plans.*** A tester's past experience with the project, the team, or similar projects and teams can serve as a great basis for planning the next testing effort. Risk analysis can take into account previously complex or buggy features. Testing processes can also be improved based on past experiences.

In addition, professional testers tend to be "Jacks of all trades, master of none" since their job requires them to know enough about all aspects of the system to understand their functionality, but doesn't allow them the time to dig in too much depth on any one part. However, professional testers will have strong ***domain knowledge and understanding of the customer perspective*** to be able to accurately create tests for the software.

Tool #3: Familiarity with and ability to work within the software lifecycle

When testers first start out, the full software lifecycle can seem daunting and sometimes irrelevant to the testing task at hand. However, thinking this way is a mistake. If a tester thinks their only job is to write tests and run them when the product is finally released, they will miss the benefits of understanding the big picture in which software is developed and how quality can apply to all parts.

Professional testers understand the software lifecycle as it spans from market idea to planning, then to development, and finally to maintenance, and they understand how they can make their testing life easier by promoting good processes starting early in the lifecycle. Some key lifecycle areas that professional testers know to take advantage of are listed below.

Professional testers are ***plugged in to the requirements definition process.*** Professional testers push for involvement in requirements definition as early in the cycle as possible because they know that clear, concise, complete, and *testable* requirements will make their life much easier. They will be able to create test cases faster both because the requirements will be easier to understand and test against, but also because testers can trust that the requirements are more thorough and accurate and so testers do not need to spend as much additional time coming up with test cases beyond the requirements. Professional testers also know that the benefits go beyond just themselves. Having accurate requirements early in the development cycle has been shown to reduce some development time as well [2].

In addition, ***getting plugged in to the design process*** can give testers great benefit. Performing peer reviews of design documents helps testers to drive quality into the process early and also gives them a good understanding of the internals of the system, which assists in creating test cases. Having developers peer review testing collateral helps testers get input to ensure they are testing the right things and also helps establish the relationship between testers and developers.

Professional testers also ***understand the benefit of attaching quality criteria to project milestones*** to help track and ensure the quality of the product. Exit criteria, or release criteria, can be attached to each milestone to set up the criteria that must be met before the milestone can officially be released. These criteria can cover everything from defect counts to test pass rates to successful resolution of legal issues.

Agreeing on these in advance reduces the risk of bad decisions being made at release time when everyone is desperate to ship the product.

Testers who understand the software lifecycle also understand the ***best time to perform different types of testing***. For example, unit testing can be done by developers as soon as they have compiling code. Functional tests can be written as soon as functional requirements exist, and they can be run as soon as code implementing the functionality exists. Stress and performance test suites can be started early, but are typically best run later in the product when more functionality is available and developers' efforts can be spent fixing performance and stress issues. Testers who understand the software lifecycle also know to begin thinking about things like legal issues early to prevent being surprised at release time.

In addition, testers who understand the software lifecycle understand the importance of ***finding defects early in the lifecycle***. Defects that are found early on have less risk of causing a schedule slip than those found at the last minute. Along those lines, verifying defects soon after they have been fixed ensures that any problems or miscommunications are caught and resolved quickly.

Tool #4: Ability to communicate

For a tester, it's not enough to be technically sharp, or logically correct, or just openly honest. A professional tester needs to be able to effectively communicate all of their results to others.

Testers' effectiveness relies on their ability to communicate with all stakeholders, such as managers, developers, other testers, and marketing.

Communication with managers

Communication with managers requires that testers understand how to display testing data in a format that can quickly be absorbed and acted upon. Thus, testers need to understand the ***basics of metrics gathering*** so they can understand which data to collect, when, and how to present it. New testers that fall into the trap of reusing metrics from past projects or from more senior testers can run the risk of producing data that isn't used and of missing critical quality issues due to monitoring the wrong information.

Professional testers work backwards from software quality milestones and ask themselves questions, such as:

- *What goals do I need to achieve?* For example, maybe a tester has a goal that functional test cases should be created for all features by the beta milestone.
- *What do I need to track to ensure I achieve my goals?* In the example above, tracking the number of functional tests created against the goal can highlight progress.
- *What goals does the project need to achieve?* For example, maybe all show-stopper bugs need to be fixed or have workarounds by the beta milestone.
- *What information will management want to see to show progress towards project goals?* In the example above, showing the total number of open show-stoppers and how many have workarounds on a weekly basis can help management make release decisions.

The goals of the test team and project can relate to test pass statistics, defect data, code coverage, project tracking, continuous improvement, process checking for effectiveness or compliance, or other quality criteria.

Testers also need to ***write defect reports such that they that can be correctly triaged*** by management. Professional testers learn to describe the business impact quickly and accurately in defect reports, and also use a structure that is easy to navigate so that management can easily find key sections, like the business impact, the expected results, the steps to reproduce, and others.

Communication with engineers

Professional testers also have a good working relationship with developers and can communicate test and defect information to developers and gain design and development insight from them.

In addition to writing defect reports management can disposition, professional testers also ***write defect reports that are easy for developers to attempt to fix***.

Developers often ask for additional information to help reproduce or find the root cause of a bug (like a certain log file or error message). Professional testers respond quickly to these requests to demonstrate commitment to getting the defect fixed and respect for the developer. They also take the opportunity to amplify this goodwill by remembering what the developer has asked for in the past and supplying that information on all future applicable defect reports.

Professional testers also spend time obtaining and writing the steps to reproduce the issue. This can often be the most tedious part of defect writing; however, the time spent can mean the difference between a developer being able to fix a defect and a developer returning the defect as invalid.

Sometimes it's not clear what is or isn't a bug, and at these times professional testers work with developers to use their domain knowledge in understanding system behavior. In projects without formal requirements, starting a dialogue with the developer is often the only way to understand the correct system behavior.

Communication with others

It's not just managers and developers that testers interact with. They also need to be able to work with other testers to share knowledge and solve problems together. In addition, they need to be able to work with the customer (or marketing as the liaison) to understand customer requirements and habits and incorporate those into the test plans.

Communication to influence

Testers also need to be able to influence others when necessary to correctly disposition bugs, or to be motivated to fix a defect, or to stop the release of a product until critical defects have been fixed. Professional testers are able to influence others by using good negotiation tactics.

Professional testers ***see themselves as a member of the team***, and they frame their arguments based on what's best for the team, giving them the energy to overcome disagreements and believe in what is right because the team has a mutual goal.

They also ***use data-based arguments.*** They use metrics and data, when possible, to convey information to others and show why their position should be noticed. For example, instead of writing "the application loads slowly" in a defect report, a professional tester would write "the application takes an average of 30 seconds to fully appear." Also, when writing defect reports, professional testers learn what product behavior is truly a bug by reading and referencing requirements documentation, and also use their discussions with developers, managers, marketing, and customers to support their position on a defect.

More than that, professional testers are also ***comfortable with the fact that testers generally influence without authority.*** Testers are placed in a difficult position: They need to be responsible for influencing the quality of the project, but are not given the authority to enforce changes. Professional testers pick their battles based on importance and what they are likely to be able to actually change. Professional testers take the attitude that they are responsible for highlighting information on the quality of the product, but they aren't solely responsible for the quality of the product. Thus, they focus their energy on enabling others to understand the data they have collected, but not always influencing others to take a given course of action based on the data.

Tool #5: A passion for quality assurance

To those who mistakenly think of testing as the boring, repetitive work of junior engineers, the idea that anyone can be excited by it is shocking. However, since professional testers have mastered the above four tools, this tool tends to come naturally.

The hallmark of a professional tester is that the professional tester enjoys testing and enjoys quality assurance. While this may not be for everyone, many new testers actually don't realize the potential of the profession. The list below gives some basic reasons why the testing profession is an exciting one.

Professional testers add value to the organization. Although the tester's value can't be measured as concretely as seeing a piece of software run, testers add great value to the organization by ensuring that the software that ships will delight the customers. Testers give the customer a voice. And, testers give their company the information they need to make good decisions for the company and the customer.

Professional testers get to focus on organizational impact. Not only to testers add value, but their job also involves a focus on the bigger picture. Testers need to think about more than just the product they are testing. They need to think about the customer's needs and the organization's goals and how their results can best serve them. Incidentally, this holistic approach that testers need to take is also a good preparation for those who want to go in to management, and it can be beneficial preparation for testers with management aspirations.

The people make it worth it. The professional testing field is full of neat people. The professionals that regularly present keynotes and other talks at PNSQC as well as other conferences are a motley assortment of highly interesting people: professional consultants, lawyers, psychologists, testers, managers, and others.

Conclusion

Professional testers have a toolbox full of tools, such as knowing how to test and knowing software lifecycles, the ability to communicate at all levels, and the moxie to be skillful detectives that love what they do. These tools ensure that they see themselves and that others perceive them as professionals doing a challenging and rewarding job. Sharing these tools with new testers can help ensure the testing profession always has new recruits, and sharing them with software peers can help ensure the profession gets the recognition it deserves. Testers that use these tools start behaving more like professionals, and so others treat them as professionals, and the whole team benefits! Every professional tester should consider sharpening and honing the tools in their toolbox.

Acknowledgements

This paper was greatly improved by the input of two outstanding reviewers: Rick Anderson and Debra Schratz. Their wisdom and experience enhanced the quality and content of this paper in many ways. Thanks!

Bibliography

[1] Anderson, Michael J.. "Case Studies in Adopting Software Quality Release Criteria." 2003. Pacific Northwest Software Quality Conference Proceedings. 18 May 2005. <http://www.pnsqc.org/proceedings/index.php>

[2] Blackburn, Scudder, et al. "Improving Speed and Productivity of Software Development: A Global Survey of Software Developers." 1996.

[3] Goldberg, Eli. "Bug Writing Guidelines." 18 May 2005. <http://landfill.bugzilla.org/bugzilla-tip/page.cgi?id=bug-writing.html>

[4] Kaner, Cem. "Effective bug reporting." [SLIDES] (Tutorial session) 15th International Software Quality Conference (Quality Week), San Francisco, CA, September, 2002. 18 May 2005. <http://kaner.com/pdfs/BugAdvocacy.pdf>

[5] Lynch, Rusty, Glass, Stephanie, et al. "OSDL Carrier Grade Linux Validation Framework." 2002. The Open Source Development Laboratory. 18 May 2005. <http://www.osdl.org/docs/cgl_validation_framework_specification___10_doc.doc>

[6] Marick, Brian. "Classic Testing Mistakes." 1997. Testing Foundations. 18 May 2005. <http://www.testing.com/writings/classic/mistakes.pdf>

"A toolkit for predicting and managing software defects - before a single line of code is written."

By: Ann Marie Neufelder, SoftRel, amneufelder@softrel.com

Abstract

Since the late 1960's there has been a variety of methods available for predicting software defects. Unfortunately, nearly all of these methods are unusable until system testing has started. So, they have limited benefit to a software engineer since the code has already been written by the time the prediction method can be used.

There have been a few methods for predicting defects before testing but these have been largely based on one parameter such as the Software Engineering Institute Capability Maturity Model (SEI CMM) level [1,6]. The problem with one parameter models is that they don't facilitate tradeoffs since the input won't realistically change during a project.

In 1993, the author developed a multi-parameter method to predict defects long before the testing begins. This method has been updated with new software technology and data from a variety of industries and application types every 12-18 months. This method is easy to use but also powerful for planning maintenance and warranty cost as well as project cost and schedule.

This paper will discuss how a software engineering organization can predict defects and manage them before ever writing a single line of code. This kind of toolkit gives a software engineering group the ability to decide and plan:

- How many defects will escape to the customer with the currently planned practices
- The additional practices (if any) that can to be employed to reach some warranty objective in ranked order
  - Smallest cost to implement when implementing the first time
  - Smallest time to implement when implementing the first time
  - Highest probability of reducing escaped defects
  - Highest predicted volume of reduced escaped defects
- The ability to benchmark the results against similar application types and industries

This toolkit includes:

- A survey and scoring mechanism that is used to predict escaped defects
- The relative cost to implement, time to implement, correlation to escaped defects and predicted volume of reduced defects for every practice in the survey
- The confidence of the predicted defects
- The average predicted defect density per industry/application type for purposes of benchmarking

Biographical Sketch

Ann Marie Neufelder has been in software engineering as either a software engineer or software manager since graduating from Georgia Tech in 1983. In the last 2 decades, Ann Marie has also developed numerous software reliability metrics for the purpose of reducing fielded software defects with the least cost and schedule time. From 1983 to 1991 Ann Marie worked as a software engineer developing electronic warfare systems and commercial financial systems. Since 1992 Ann Marie has evaluated the processes, design, code, methods and product characteristics of more than 90 organizations in the aerospace, defense, medical devices, and healthcare industries. Ann Marie has been teaching System Software Reliability with the Reliability Analysis Center since 1991. In 1993 Ann Marie published "Ensuring Software Reliability" with Marcel Dekker, Inc. She has also published numerous papers for IEEE, ASQC, and NAECON and has a patent for a software invention to predict the time to Rastor Image Process (RIP).

Introduction to the Goals and Tasks

Figure 1 illustrates the author's goals for inventing this predictive model. The three goals that people generally want to achieve are to 1) improve the software by reducing the defects, 2) staff effectively to avoid a late delivery/warranty cost overrun and 3) quantify software and system reliability. These three goals are layered from top to bottom on Figure 1. Figure 1 also shows the 4 basic steps that are executed for any or each of these goals 1) Data collection 2) Prediction 3) Management and Planning and 4) Execution. These steps are layered from left to right on Figure 1.

This model can and is used to achieve all three goals and more importantly it can achieve the three goals independently of each other. Table 1 identifies the steps necessary for each goal. This paper will focus primarily on the tasks necessary to improve the software and staff effectively. Each of the three goals has several phases including data collection, prediction, management and planning and finally execution.

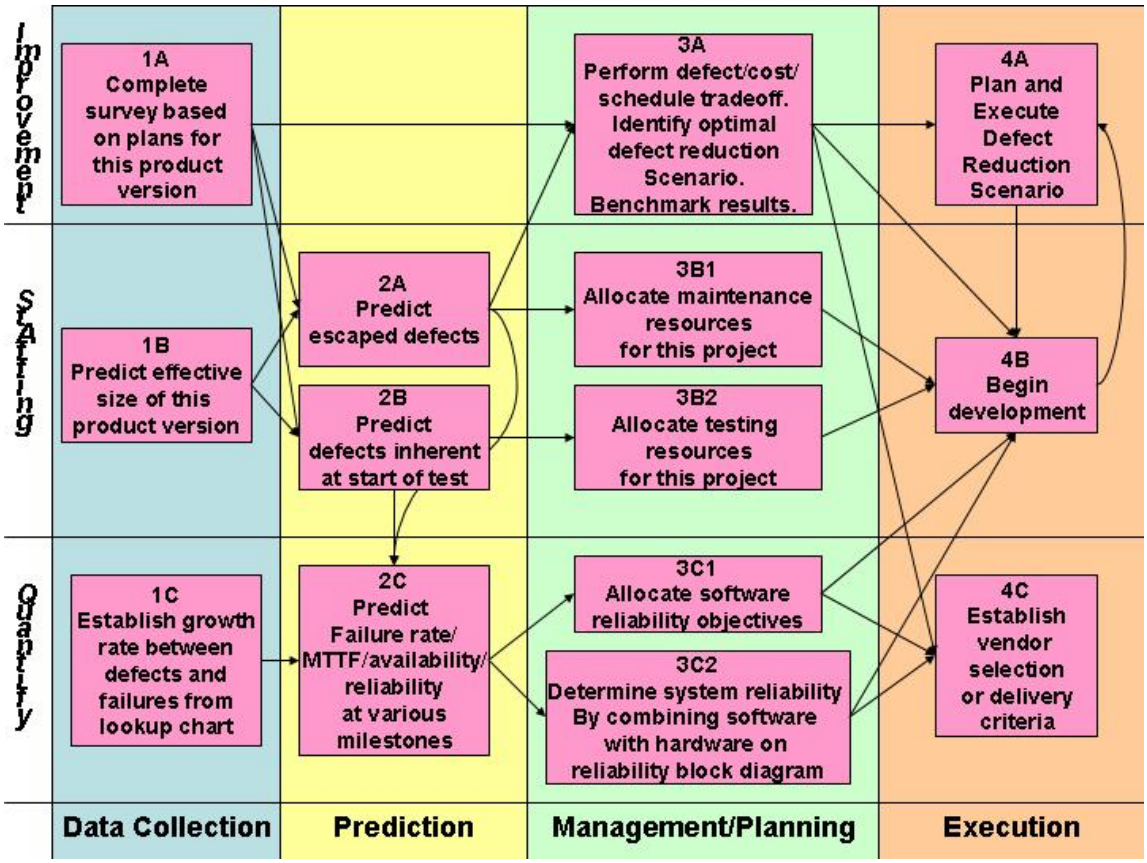


Figure 1 Software Prediction Goals and Tasks

The survey was developed based on a decade long data mining/statistical modeling project in which the author collected vast amounts of data from real software projects from defense, aerospace, electronics, medical, healthcare, semiconductor and other industries. The software application types include firmware, client server, database applications, web based applications, robotics, artificial intelligence and mathematical systems. The staffing size of the individual projects included in my database range from 2 to 120 people. The company sizes ranged from small (< 1000 people) to very large. The projects also spanned nearly every SEI CMM maturity level. A plethora of characteristics were correlated to the *actual* number of defects found during testing and after testing. Statistical modeling was then used to develop a formula to map these characteristics to defect density. The white paper for this work discusses how the survey evolved and how the scoring mechanism and final predictions were mathematically derived. [1]

Table 1 - Tasks required for each goal

Goal	Tasks required
Improvement goals	
Reduce defects that are visible to your customer(s) by a specific predefined percentage	1A, 3A, 4A (Step 2A is optional)
Assist a vendor to reduce defects that are delivered to you from a vendor	1A, 3A, 4C (Step 2A is option). All steps are performed on vendor version.
Staffing goals	
Predict defects that will be found by customer so as to plan for warranty and maintenance	1A, 1B, 2A, 3B1
Predict defects that will be found during testing so as to plan resources required to find those defects	1A, 1B, 2B, 3B2
Quantify reliability goals	
Identify the system reliability for hardware and software (usually to provide to one or more of your customers)	1A, 1B, 1C, 2A, 2C, 3C2
Establish vendor selection or delivery criteria for vendors supplying only software	Defect driven - 1A, 3A, 4C OR Reliability driven -1A, 1B, 1C, 2C, 3C1, 4C Performed on each vendor version one at a time
Establish vendor selection or delivery criteria for vendors supplying hardware and software	1A, 1B, 1C, 2A, 2C, 3C2, 4C performed on each vendor system one vendor/system at a time

Task 1A - Complete survey based on plans for this product version

The survey should be completed by the appropriate subject matter experts for *one software version at one software organization*. Ideally, it should be completed prior to development for this version. The survey is completed based on what is *planned* for this project and what has been *executed by this software organization in the immediate past*.

The result of the survey is one of 5 defect percentiles: 90, 67, 50, 33 or 10. Since the measure being predicted is defects, a predicted percentile of 10 is more desirable than a predicted percentile of 90. The defect percentiles are associated with the respective defect density percentiles of the many software projects in my defect density database. So, a defect percentile of 10% means that a project exhibited a fielded defect density that is bigger than only 10% of the other projects in the database. A defect percentile of 90% means that the exhibited defect density for that project was higher than 90% of the projects in my database. Figure 2 shows some of the qualitative characteristics that distinguish the 5 percentile groups.

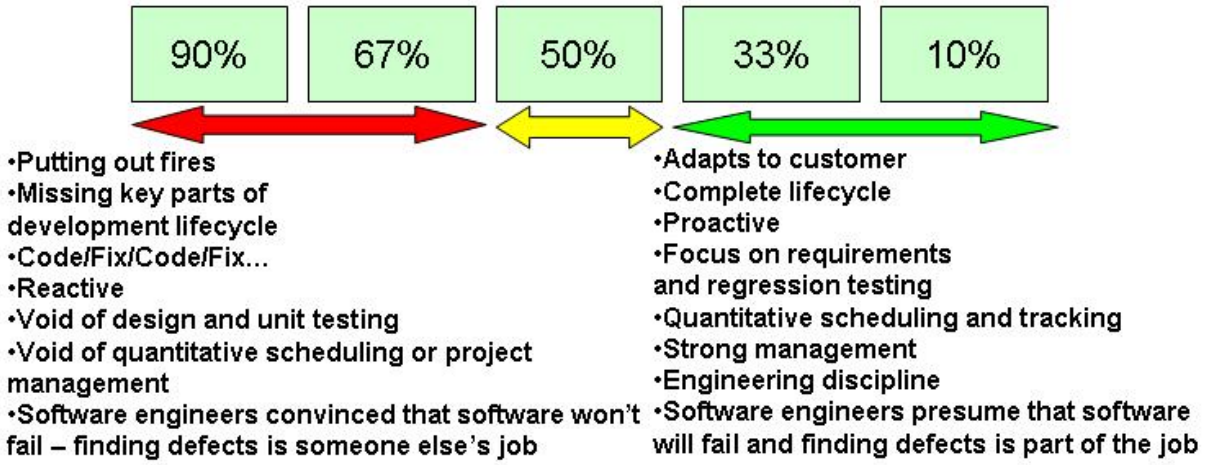


Figure 2 Qualitative descriptions of the 5 percentiles

Step 1 Answer the survey. For questions 3-119 answer either Yes or No. For questions 1 and 2 enter applicable value. If a question is not applicable, answer no. See the glossary at the end of the paper for the definitions of particular items. *Ignore the cost, time, and index columns as these columns will be used in step 3A.*
Step 2 Review the prerequisite column. If you answered yes for a particular question but did not answer yes for ALL associate prerequisite questions, you must change that particular yes to no. *Do not skip this step. If you ignore the prerequisites, the prediction result will not be accurate.*
Step 3 Accumulate 1 point for each yes answer and add results of questions 1 and 2.

Table 2 - The survey[1]

Question	Prerequisite	Cost	Time	Index
1. Assessed SEI CMMI maturity level (Enter 1 to 5)	A formal assessment if > 1	3	3	3
2. The most calendar time is typically spent in 3) requirements, 2) design, 1) code or 0) test?		1	2	239
3. Upper management reviews software engineering status		1	3	72
4. There is a defined team structure between software requirements analysts, designers, coders, testers and management		1	2	75
5. Software engineering is operated similarly to any other engineering discipline		1	3	37
6. System testers will not code on this project (even if they can)	Dedicated testers	2	2	4
7. Software QA is separate from software engineering (not the same people)		2	2	37
8. At least one person on the software team has end application domain knowledge and is available to the rest of the team		3	3	22
9. There is a software manager that does not code (but can)	A software manager	2	1	78
10. Turnover rate < 20 % per year for software and test team		3	3	19
11. Percentage of testers to total software team is > 20 and < 40%		2	2	4
12. Subcontractors are chosen for their skill set when that skill set is not present in house. This team also avoids reinventing the wheel by employing subcontractors/vendors for more general purpose code that has already been invented or developed.		2	2	29
13. Geographically intact software team		2	1	33
14. This team avoids making enhancements after coding phase has ended (for this version, spiral or increment)		1	3	85
15. This team avoids making enhancements after systems testing phase has ended (for this version, spiral or increment)		1	3	83
16. We have a formal means to filter, assign priority/schedule customer requests as opposed to shoot from the hip commitments		1	3	114
17. Interruptions to software engineers are allowed only when needed for clarifications for the work at hand		1	3	108
18. Average software engineer has a reasonably quiet workspace (a cubicle as opposed to bullpen)		1	2	163

[1] Copyright Ann Marie Neufelder DBA SoftRel, 2004

Question	Prerequisite	Cost	Time	Index
19. Current ISO registration that covers software organization (last 2 years)	A current certificate	3	3	43
20. Requirements phase activities are explicitly on the schedule		1	2	73
21. Procedures/templates for requirements process are defined		1	1	132
22. Voice of customer (either internal or external) is involved in requirements/analysis		1	1	80
23. System requirements translated to software Requirements before design or coding		1	1	92
24. Risks related to the requirements phase are identified and managed before designing or coding		1	1	113
25. Formal requirements review prior to design or code		1	1	426
26. Requirements are prototyped by any means other than words		2	1	145
27. System testers on the project during requirements phase		1	2	15
28. System test plan started during requirements phase		1	1	282
29. There are requirements tools and they are used		3	2	13
30. Design phase activities are explicitly on the schedule		1	1	195
31. Procedures/templates for how to design exist and are used		1	1	203
32. Requirements are explicitly traced to design in writing		2	1	122
33. Top level design is done (architecture, behavior, states)		2	1	41
34. Detailed design is done (pseudo code, module design)		2	1	35
35. Risks related to the design phase are identified and managed before coding		1	1	234
36. Formal design reviews conducted prior to coding	30,31	1	2	77
37. Design is prototyped via any means other than words	30	2	1	124
38. System testers are on the project no later than design phase	6,30	1	1	193
39. Design tools exist and are used	30	3	1	16
40. Procedures for how to code exist and are used		1	1	170
41. Requirements are explicitly traced to code in writing	20,21	2	2	61
42. There are written standards for exception handling	40	1	1	244
43. Debuggers are used during coding		1	1	61
44. There are coding tools other than compiler/debugger and they are used (i.e. static code analyzers)		1	1	190
45. System test plan started no later than coding phase	6	1	1	344
46. Code reviews done consistently based on a specific criteria list	40	1	2	34
47. Unit testing activities are explicitly on the schedule		1	2	110
48. There are procedures for how to unit test and they are used	47	1	1	292
49. Requirements are mapped to unit tests explicitly in writing	47,48,20	2	2	53
50. Risks identified during unit testing are managed prior to systems testing	47,48	1	1	348
51. Functionality as per the requirements is explicitly unit tested	47,48	1	1	418
52. Algorithms are explicitly unit tested	47,48	1	1	276
53. Exception handling is explicitly unit tested	47,48	1	2	167
54. System test plan is started no later than unit testing phase	6	1	2	97

Question	Prerequisite	Cost	Time	Index
55. There are tools for unit testing and they are used	47	1	1	438
56. Unit tests are reviewed by non-peer subject matter expert		2	2	28
57. System test is explicitly on the schedule	47,48	1	1	122
58. There are procedures/templates for how to system test and they are used	57	2	1	38
59. Requirements are explicitly mapped to system tests	20,21,57,58	2	2	42
60. Risks identified during system testing are managed before delivery	57	1	1	206
61. The functionality that represents the critical path is system tested	57,58	1	1	1
62. System level exception handling is system tested	57,58	2	2	30
63. Behavior and state models are system tested	33,57,58	1	1	121
64. Installation (i.e. CD, cab files, etc.) is system tested	57,58	1	1	89
65. There are test beds (Test data in which the correct result is known)	57,58	2	2	78
66. There are performance tests if applicable	57,58	1	1	226
67. There are security tests if applicable	57,58	2	1	104
68. There are multi-user tests if applicable	57,58	1	1	139
69. The user documentation (manual/release notes) is explicitly tested (not just reviewed)	57,58	1	1	164
70. There are system testing tools and they are used	57,58	3	1	3
71. There are simulators and they are used	57,58	3	1	29
72. Regression test activities are explicitly on the schedule	57	1	1	308
73. There are procedures for how to regression test and they are used	57,72	2	1	107
74. All defect fixes since last baseline tested during regression test	57,72,73,86	1	2	105
75. All upgrades since last baseline tested during regression test	57,72,73	1	1	278
76. All other modifications since last baseline tested during regression test	57,72,73	1	1	265
77. Test areas that have high user impact are run during regression test	57,72,73	1	2	76
78. Test areas that have high development risks are run during regression test	57,72,73,86	1	2	95
79. There are procedures for defect fixing and they are used		1	1	53
80. Changed units retested during defect fixing	79	2	2	57
81. Changed paths retested during defect fixing	79,55	2	2	49
82. Source control is maintained during defect fixing	79	1	1	133
83. New features retested during defect fixing	79	1	1	197
84. Defect tracking system used for defect fixing	86	1	1	266
85. Any utilized global variables retested during defect fixing	79	1	2	63
86. There is a Software Defect Tracking System (SDTS)		1	1	142
87. There are procedures for how to use the SDTS	86	1	1	15
88. SDTS is used by all software engineers	86	1	2	76
89. The SDTS is automated (not on paper only)	86	1	1	168
90. System testers use SDTS	86,6,57	1	2	119

Question	Prerequisite	Cost	Time	Index
91. Customer has access to SDTS (access may be by phone or other indirect method)	86	1	2	18
92. Field engineers use SDTS	86	1	1	4
93. There is physical 2 way link from SDTS to Configuration Management (CM) system. All changes to source are recorded and all changes in the CM system are linked to the source.	86,89,110,111	3	1	36
94. SDTS is on an internet or intranet	86,89	3	1	24
95. There are QA procedures and they are used	7	1	2	96
96. There are delivery procedures and they are used		1	1	143
97. There are Project Management tools	102	1	2	45
98. The software is Object Oriented (OO)	99	1	3	16
99. If the software is OO then every software engineer knows the 4 things that make OO different		2	3	0.3
100. There are peer reviews during every phase		1	1	191
101. There are product metrics such as size, complexity		2	4	12
102. Time for detailed scheduling is allowed prior to committing to a schedule		1	1	65
103. Actual versus estimated progress tracked		1	2	107
104. There are defect metrics		1	2	99
105. There is a reuse library that is accessible		2	1	51
106. Root cause analysis is done regularly on software defects (no less than once every major release)		1	2	53
107. Proactive changes made as result of root cause analysis		1	3	29
108. Programmers workbench (The target HW or fragment)		3	2	8
109. Life Cycle Model is either spiral or incremental		1	3	73
110. Configuration management (CM) exists		1	2	26
111. CM procedures exist and are used		1	1	14
112. The CM procedures are used by all software engineers		1	1	161
113. CM system allows a stop shipment (i.e. via allowed or active part numbers) for discontinued or unsupported versions		1	1	33
114. CM system used for requirements and design		1	1	52
115. Number of bug fix only releases per year averages <= 4		1	3	111
116. This team avoids concurrent releases		1	3	34
117. The oldest part of the software is less than 5 years old		2	3	52
118. Are there between 1 and 5 subcontractors?		1	1	186
119. % of code that is NOT cloned is > 60%		2	3	12

Step 4 Review Table 3. Find the row that encompasses your score and select the possible percentile results. For example, if the score is 61, the 67, 50, 33 percentiles are possible, but not the 90 or 10. The shaded rows indicate that a tiebreaker is needed to determine the final percentile.

Step 5 If there is more than 1 possible percentile group then perform the indicated tiebreakers in Table 4. If there is a three way tie, start from the tiebreakers for the highest percentile groups (90, 67, etc…) and eliminate until one percentile is left. For example, if your score is 61, review the tiebreaker for 67 and 50 first. If the result is 67 then you stop and 67 percent is your answer. Otherwise if the tiebreaker result is 50 then continue to the 50/33 tiebreaker. Do not use the tiebreakers unless there is a tie between the 2 percentiles in that tiebreaker and all higher percentiles have been eliminated. Now that you know the

predicted escape defect density and percentile for this project, you can use this percentile as inputs for Tasks 2A and/or Task 3A.

Table 3 Select the possible percentile(s)[2]

	Possibility of this percentile				
Your survey score	**90**	**67**	**50**	**33**	**10**
<17	Yes	No	No	No	No
>=17 <=24	Yes	Yes	No	No	No
>24 <=39	No	Yes	No	No	No
> 40 <= 60	No	Yes	Yes	No	No
>60 <= 67	No	Yes	Yes	Yes	No
>67<= 69	No	No	Yes	Yes	No
> 69 <= 84	No	No	Yes	Yes	Yes
> 84	No	No	No	Yes	Yes

Table 4 Percentile Tiebreakers

Between these percentile groups	Criteria for determining percentile - you need only 1 of these for lower percentile presuming all higher percentiles have been eliminated
10% and 33%	19, 53
33% and 50%	93, 12
50% and 67%	15, 17, 45, 56, 81, Question 2 - Longest phase of lifecycle is not testing
67% and 90%	21, 71, 59, 88, 83, 94, 65, 101

Step 6 To benchmark your survey score against similar types of systems or software, review figures 3 and 4. These figures show the average score and standard deviation of the scores filtered by industry or the type of software application. Note that the standard deviations for the scores for educational and medical/healthcare, client server and web application software are not available.

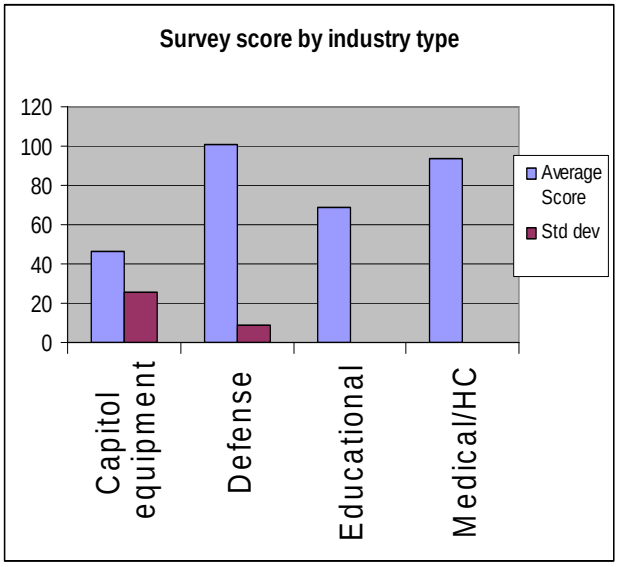


Figure 3 Survey Score by Industry

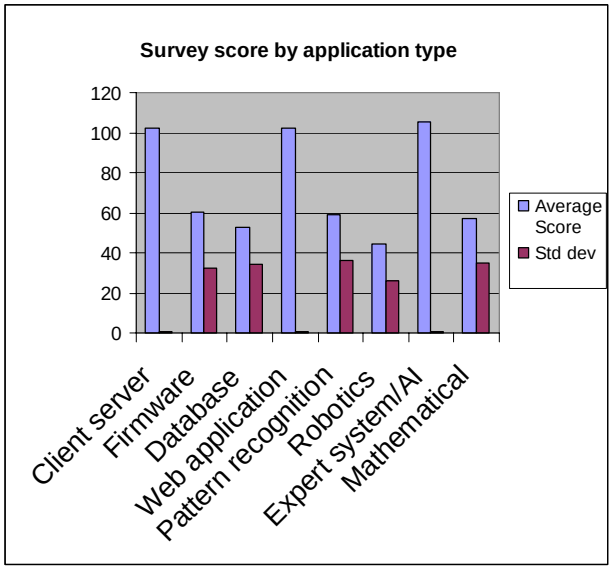


Figure 4 Survey Score by Application Type

[2] Copyright Ann Marie Neufelder DBA SoftRel, 2004

Task 2A - Predict Escaped Defects
The escaped defects are those defects that are not found prior to a customer delivery. Escaped defect density is the defect density at the point in which all development and testing has finished and the software is now being distributed to at least one external customer.

Step 1 Predict escaped defect density in terms of escaped defects per 1000 effective executable lines of normalized code. In task 1B, the normalization process is summarized. Substitute the variable x below with the percentile computed as a fraction. For example, if the resulting percentile is 67 then substitute .67 in the below formula.

Predicted escaped defect density in normalized KSLOC[3]: KSLOC = $(1.5776*(x)^2) - (0.1022*x) + 0.0651$

Step 2 Establish the 95% upper and lower confidence bounds on your prediction by selecting the resulting percentile from Table 5. So, if your predicted percentile is 50, then add .073 to your prediction from step 6 to establish an upper bound on defect density. Then subtract .073 to establish a lower bound on defect density.

Table 5 Confidence bounds on defect density predictions

Percentile predicted	**10**	**33**	**50**	**67**	**90**
Add/subtract this value to prediction for 95% confidence bound	0.015	0.037	0.073	0.113	0.132

Step 3 Multiply the predicted defect density and each of the upper and lower bounds by the effective normalized size computed for this version of software. See task 1B for instructions on how to compute effective normalized size. The result is the predicted number of defects that may become visible to your customer/end user and an upper and lower 95% confidence bound on that prediction. Now that you have predicted the number of defects that will be visible to your customers, you can proceed to task 3B1 to determine warranty staffing, or you can proceed to task 3A to determine how to cost effectively reduce those defects.

Task 3A - Perform defect cost/schedule tradeoff

Table 6 - How the percentile groups compare with respect to defects and on time delivery

Percentile predicted	**90**	**67**	**50**	**33**	**10**
Average normalized defect density by percentile	1.232	0.718	0.370	0.198	0.075

The percentile predicted in step 1A is the key to performing cost and schedule tradeoffs. Table 6 shows the average defect density for each of the 5 percentile groups. *Notice that the average defect density is approximately cut in half between each percentile group.* This information will be used very shortly to determine cost/time/defect tradeoffs. Refer back to Table 2 in Task 1A and note the 3 columns to the far right of this table - cost, time and index. These three columns are further defined below in table 7.

Step 1 - Identify all yes answers from the originally completed survey in table 2 and accumulate the cost column for only the "yes" answers. This is your current cost baseline. If you wish to measure the impact on calendar time, you can also accumulate all of the time columns for every yes answer to establish a time baseline.

Step 2 - Use the below Table 8 combined with the cost and time columns in the survey in Table 2 to find the cheapest, fastest way to get to the next percentile and reduce escaped defects. Identify the "no" answers and evaluate which have the biggest index and can be met given the prerequisites required. Do not try to transition more than one percentile at a time. There are usually many scenarios that mathematically achieve the desired percentile.

[3] Copyright Ann Marie Neufelder DBA SoftRel, 2004

For each scenario identified execute steps 3 and 4 below.

Step 3 - Accumulate the cost of all of the selected characteristics for your defect reduction scenario. Divide this by your current cost baseline. This is the cost increase percentage required to make this tradeoff. Multiply this by the total cost in dollars for the last 4 quarters or most recent project to get an estimate of the total cost increase in terms of dollars. Note that this cost increase is only to implement the characteristic for the very first time. Repeat this step using the time baseline to compute the percentage increase in schedule time required for the most recent project.

Table 7 Definition of time, cost and index

	Cost	**Time**	**Index**
Measure	Relative cost of implementing this characteristic for the first time if it has not been implemented before	Relative amount of calendar time (not effort) required to implement this characteristic for the first time if it has not been implemented before	Correlation * impact ---------- Cost * Time Correlation - linear relationship between characteristic and defects Impact - volume of reduced defects when characteristic is present
Range of values	1- can be bought without a PO 2 - requires a PO 3 - requires engineering management approval 4 - requires executive approval	1 - can be implemented immediately 2 - can be implemented before the next project 3 - can be implemented on a near future project 4 - several years or projects from now	0 to 800 - The higher the index, the more optimized this characteristic is for defect reduction, cost and time

Step 4 - Now refer to the "reduction" column in table 8 and the row with your current percentile. This is the percentage of defects reduced by transitioning to this new percentile.

A. If you know the total tangible and intangible cost of maintenance and warranty then multiply it by this percentage to yield the total cost reduction in dollars
B. Otherwise, assume that the cost of fixing one escaped defect is 10 times the cost of fixing one defect found during testing. Estimate the cost of fixing one testing defect by dividing the total resources expended during a recent testing cycle by the total number of defects corrected during that testing cycle. Multiply that number by 10. Multiply that by the difference in total escapes between the current plan and the proposed scenario to yield the total cost reduction.

Table 8 Cost/Time effectiveness suggestions for transitioning to the next percentile

%	**Suggested techniques for fastest /cheapest way to transition to next percentile**	**Defect Reduction**
90	You will need either a minimum of 17 points with one tiebreaker or 25 points without a tiebreaker to reach the 67 percentile. Consider these characteristics which were prerequisites for the most number of others: 20, 21, 30, 31, 47, 48, 57, 58, 79, 86, 87.	42%
67	You will need either a minimum of 41 points with one tiebreaker or 68 points without a tiebreaker to reach the 50 percentile. Consider implementing the tasks that allow you to have a complete lifecycle and not just coding, testing and fixing.	48%
50	You will need either 61 points with one tiebreaker or 85 points without a tiebreaker to reach the 33 percentile. The 33 and 10 percentile groups on average are strong at mapping the requirements to the test plans and in having system testers on the project long before testing starts.	46%
33	You will need at least 70 points to reach the 10 percentile. It is not possible to reach it without one of these tiebreaker questions - 19 and 53. Question 53 (formal unit testing) is generally significantly cheaper/faster than question 19 (current ISO registration). Focus on changing culture to removing defects versus proving that none exist. The 10 percentile group on average is strong in design and unit testing.	62%

Step 5 - Select the scenario with the smallest cost and/or time increase from step 3 compared to the cost reduction from step 4.

Task 4A Establish a plan for reducing defects and execute it

The purpose of this paper was to identify a toolkit for establishing the defect reduction scenarios that optimize cost and time. Once you have done this, establishing the plan and executing it is largely dependent on the scenario chosen and extends beyond the scope of this paper.

Task 1B Predict Effective Size

Size prediction is not novel. There are multiple methods for predicting the size of code before it is ever developed [7]. Once the size has been predicted using one of these methods, you will need to normalize it to be compatible with the prediction model in this paper. This normalization step consists of converting the size predictions of each language into one normalized language which is assembler. This is done because different languages have different densities which have nothing to do with the inherent quality of the language and everything to do with how much object code is generated from each line. The steps are as follows:

For each unique language implemented for this version of this project:
Step 1 Predict the number of NEW executable non-blank non-comment statements in that language
Step 2 Predict the number of modified executable non-blank non-comment statements in that language
Step 3 Predict the number of reused executable non-blank non-comment statements in that language
Step 4 Multiple the result of step 3 by 10% to account for the fact that the reused code has been tested previously
Step 5 Add the results of steps 1, 2 and 4
Step 6 Multiply the result of step by the appropriate code expansion from the below table
Repeat for every language in this version and accumulate the results of each step 6.

Table 9 Converting to normalized KSLOC [2]

Language	KSLOC of assembler for every KSLOC of this language
2nd generation (C,Fortran)	3
Object oriented C++, Java, Ada 9x, VB	6

Task 2B Predict defect density at start of testing

The author is currently developing an independent formula for predicting defects specifically at this milestone. The research compiled to date shows that on average there are 15 times more defects during system testing than after delivery. So, you can derive the predicted number of defects to be found during testing by multiplying the predicted escaped defects by 15.

Task 3B1 Predict maintenance resources

One brute force method for staffing maintenance is to presume that all predicted escapes will occur uniformly every month before the next major release. So, if the releases are 5 months apart and 100 escapes are predicted, one simple way to plan resources uniformly over the 5 months so as to fix 20 defects per month. Typically, however, the escapes aren't detected uniformly but rather in a decreasing number from month to month. An exponential model can be used instead [3].

Number of defects identified in month i = N_{delp} ((exp(- Q_{del}/TF)*(i-1)) - (exp(-Q_{del}/TF)*i)

N_{delp} = predicted delivered defects - (See task 2A)
Q_{del} = predicted growth rate after delivery - (See task 1C)
TF = number of months that defects will be observed for this version (This can be approximated by the number of months until the next major release)

i = 1..TF - this provides an array of predicted defects for every month of support

Task 3B2 Predict testing resources

Once you have predicted the number of defects that will be inherent at the start of testing (N_{0p}), you can predict the staffing required to find and remove some percentage of them.

Step 1. Determine the initial failure rate (λ_{0p}) during testing for the last project developed and tested by this team. This is computed as 1 divided by the number of hours it took for the first defect to be found during a system level test.

Step 2. Now multiply the result of step 1 by .05. This will tell us the failure rate at the end of testing (λ_f), if 95% of the defects are found (and removed) by testing. Note that you can set this percentage higher or lower as needed.

Step 3. Testing time required to find 95% of the defects = $(N_{0p}/\lambda_{0p} * LN(\lambda_{0p} / \lambda_f)$ [4]
This result will be in the same unit of measure that you used in steps 1 and 2.

Task 1C Establish growth rate between defects and failures

The software growth rate is what determines how fast escaped defects will become observed by the end users. This growth rate is essential for predicting failure rate, MTTF, availability and reliability. The growth rate experienced during testing may/will be different than the growth rate experienced in the field. The table below shows some typical growth rates by industry or by application type. It is beyond the scope of this paper to present the mathematics behind the growth rate so references [1,5] are provided.

Table 10 Average software growth rates

Industry/Application type	Growth rate during testing	Growth rate after delivery
Capitol equipment	3.66	3.77
Defense	4.26	6.74
Educational	2.90	n/a
Medical/Healthcare	1.10	n/a
Client server	5.71	4.50
Firmware	3.02	3.39
Database	4.19	3.36
Web application	5.71	4.38
Pattern recognition	3.72	4.29
Robotics	3.05	3.07
Expert system/AI	2.08	n/a
Mathematical	3.64	4.29

Tasks 2C, 3C1 and 3C2

The methods for predicting failure rate, MTTF, availability and reliability for software are largely the same as for electronics once you have a prediction for the escaped defects. The scope of this paper does not allow for a detailed toolkit for this. However, you can refer to www.softrel.com/publicat.htm for a complete set of instructions [6].

Task 4C Establish Vendor Selection or Delivery Criteria

This entire toolkit can be administered to vendors delivering software to your company. The survey can be used to evaluate competing vendors as well as plan for warranty issues that will result from the vendor supplied software. When administering the survey to a vendor, ensure that physical evidence of every answer is provided with the survey to ensure accuracy.

Conclusions

I would like to conclude by answering some frequently asked questions pertaining to this toolkit.

Table 11 Frequently Asked Questions

FAQ	Answer
How were the survey, scores, formulas and percentiles determined?	Via years and years and years of statistical analysis of a large amount of real data collected from many real companies. You can find the complete results of the statistical modeling process in this white paper [1].
How do the 5 percentiles relate to the SEI CMM or SEI CMMi model levels?	The 5 percentiles predicted from this model have absolutely no relationship with the SEI CMM or SEI CMMi levels. DO not presume that an SEI CMM level of 5 corresponds to the 10% as it does not!
How long does it take to complete the survey?	On average, it takes about 3 hours.
Many of the survey questions appear to be related to SEI CMMi model. Can't I just use that to predict?	1. The survey contained in this paper measures more than the process. 2. Many of the characteristics in this survey are implied by not guaranteed from the SEI CMMi assessments. Since many companies do not have an SEI CMMi level above 1, using only the SEI CMMi level to predict defects is not nearly as accurate as measuring specific characteristics. 3. Most importantly, I have several companies with an unrated or SEI CMMi level 1 assessment that have predictions in the 10 and 33 percentiles.
How accurate is the prediction?	I track the predicted versus actual on an ongoing basis. Most importantly, I use the demonstrated accuracy to make further refinements to the model to keep it up to date with current technology and applications. You can find out the accuracy at any given time by contacting the author. As a note, the accuracy is greatly improved when the surveyor collects physical evidence for every yes answer.

Glossary

Upper management review - The layer of management that is above the engineering management level and above the project management level.

Engineering discipline - this means that the software engineers report to an engineering manager and adhere to the same basic engineering procedures as the other disciplines such as electronics, mechanical engineering, etc. The software engineering procedures may be part of the overall engineering procedures or they may be a tailored set of procedures that mirror the engineering procedures for hardware.

System testing and testers - System testing is testing done by people who are not software engineers and includes any test other than unit or white box testing. Systems testers are people who are not developing the software and are testing the software without the source code or debugger.

Software QA - in this survey it means an organization that has responsibility for ensuring conformance of standards and processes executed during the project. The software QA persons may or may not perform software testing.

Formal reviews - Whenever the word formal is used in the survey in Table 2, it means that the review is not among peers, there is a set of signatures required at the end of the review and the review is a milestone. A formal design review does not imply Fagan inspections. It is simply a design review that meets the above criteria.

Non peer subject matter expert - This person is usually either a domain expert working with the team who has technical authority or a supervisor/manager who has subject matter expertise.

Test beds - This is a set of correct answers that is not prone to changes very often and is used usually to regression test software. Test beds can be in the form of any media.

Programmers workbench - if the target hardware is not a personal computer or a relatively easy to acquire hardware platform, a workbench is provided that has the basic hardware features. Usually it is a skeleton or fragment of the target hardware if the target hardware is very large or expensive.

Cloned code - The opposite of object oriented code is cloned code. Cloned code is when the same code is copy and pasted many times and subtle changes are made to that code as opposed to putting that code in a class.

Critical path - The critical path is the core functionality without any exception handling

Voice of Customer - The customer or subset of customers or an entity that represents the customer is involved in determining the priorities and the features as well as making usability related decisions.

References

[1] The Naked Truth about Software Engineering, Neufelder AM, 2004, 110 pages. You can acquire this document at www.softrel.com.

[2][6] The code expansion ratios were derived from Table 7-9 of this reference. The methods for predicting availability and reliability can be found in section 4 of this document. System Software Reliability Assurance Guidebook, P Lakey, A Neufelder, 1995, produced for Rome Laboratories.

[3][5] This exponential model is not novel to software prediction. One of the many references for the formula can be found in Science Applications International Corporation & Research Triangle Institute, Software Reliability Measurement and Testing Guidebook, Final Technical Report, Contract F30602-86-C-0269, Rome Air Development Center, Griffiss Air Force Base, New York, January 1992. This document also shows some average software growth rates for aerospace and defense applications.

[4] This is an exponential extrapolation which is discussed in "Software Reliability Measurement, Prediction, Application"; Musa, Iannino, Okumuto, McGraw-Hill, 1987.

[7] A summary of the methods available for predicting size can be found at http://sern.ucalgary.ca/~hhorrian/seng621/sizeEstimationWebdoc.html.

Affordable Homegrown Test Automation

Alan White
Alan.White@NielsenMedia.com
Nielsen Media Research

Abstract
Even the most mature form of automated testing can result in implementation failure or lack of commitment for long term utilization if software quality assurance professionals find it difficult to set up, maintain, or produce test assets. A low cost, easy-to-use way of avoiding these pitfalls can be achieved by utilizing a common software package already available in most business' software portfolios. A simple to develop, user friendly product that reduces errors and increases productivity in test asset creation can ensure continued success in your test automation efforts while keeping costs affordable.

In this session, you will

- understand the concepts behind one of the most recent and more successful test automation frameworks to date,
- find out which widely used program can be used to quickly produce a visually cohesive, wizard based GUI tool for creating test assets, and
- learn how to quickly and intuitively generate and store test assets with relative ease while reducing error prone scripts

Autobiography
I have more than 7 years of applied experience in information technology. I have been a Programmer, Software Developer and Test Engineer. Tasked to spearhead an automated test effort for one of Nielsen Media Research's most critical applications I discovered that to retain interest and commitment from the user community that it was necessary to employ a product that is intuitive, visually binding, and easy to use and maintain.

1 Introduction

Companies are continuously looking for ways to improve the software quality in their product offerings and are open minded to adopting a test automation solution that is proven to produce favorable results. Some companies are finding that employing the use of an automation framework that is reusable, portable, and easy to maintain is possible and can provide a healthy return on investment (ROI) but they sometimes lack the education and available resources to institute such a framework.

2 Problem

Traditional test automation techniques have been known to be maintenance intensive, fragile in nature, require technical capabilities of those working with test automation scripts, and typically allowed for only static data to be captured in the test scripts. Additionally, the scripts recorded in test automation programs were usually tightly coupled to that program and to the application under test (AUT). There were usually little reuse capabilities therefore productivity gains have been known to be minimal when compared to that of manual testing over time.

3 Solution

Our approach to test automation breaks work down in an intuitive, structured, and maintainable approach allowing all members of a testing team to focus on what they do best. A test engineer (usually a test professional with a programming background) can dedicate his/her efforts to coding what is needed to interact with the AUT. The functional tester, subject matter expert (SME), or business analyst (BA) can dedicate his/her efforts to the creation and maintenance of test assets.

Tools can play a vital role in software testing. The right tools can offer productivity gains with improved quality. This paper describes the design and construction of an in-house built automated test creator tool that supports record and playback test automation programs like Rational® Robot. With the following goals in mind, we embarked on an effort to develop a tool that could be constructed in a reasonable amount of time and offer us the flexibility to enhance the tool to meet our internal demands.

- Provide a loose coupling of the application under test (AUT), the test automation program, and the creation of test scripts.
- Offer the ability to create automated test scripts that can be read and understood as manual test scripts.
- Abstract the complexity of maintaining test scripts from the non-technical user community.

The proposed solution is designed and constructed using a “keyword-driven” or the “table-driven” framework to test automation. The test creator tool is the means with which a user interacts with the aforementioned test automation framework to produce test assets in the form of test steps, test cases, and test suites. It is a productivity tool that is application and software development platform independent.

4 Costs and Benefits

By adopting this approach, there are initial costs. Individual results may vary. It took six months to produce the first release of this tool, which comprised the following: familiarized myself with the keyword-driven test automation, learned the Rational® Robot test automation program, developed the processing code, constructed the test creator tool, and provided a graphical user interface (GUI) map for a 75 screen (less message boxes) application with over 1200 components. During that time and among other activities, a training manual and labs for the tool was created, formal training for more than 20 prospective users was conducted, over 150 test assets were constructed, and this presentation was prepared.

Using the proposed design, you too can create an application and platform independent tool that compliments industry test automation programs such as Rational® Robot, Mercury QuickTest Professional™, and Segue SilkTest® and can store and maintain the GUI Map for Java, PowerBuilder,

HTML, .Net and Visual Basic development platforms. Costs can be recovered by the tool's flexibility, maintainability, reuse capabilities, portability, dynamic data capabilities, and by the fact that you may already have a rapid application development environment already installed at your enterprise.

In addition to the cost benefits that can be realized by the test creator tool, other benefits may be realized by adopting a keyword-driven approach to test automation including but not limited to the following:

- Reduced test execution time
- More efficient than manual testing
- Greater test coverage
- Long term cost savings over manual testing
- More manageable with known execution time
- Can run scheduled and unattended
- Facilitates continuous integration development
- Lower maintenance effort
- Quicker response to application changes
- Facilitates the use of dynamic data

To date, several project teams at Nielsen Media Research are utilizing the keyword-driven test automation framework and the proposed design to the framework. It is currently being used for HTML, Java, and PowerBuilder applications. One project group has adopted the strategy of rolling out the test creator tool to its development community as well as its software quality assurance (SQA) team and potentially will offer it to their user acceptance test (UAT) group. The developers are creating test suites for unit test plans after each build and prior to submitting to the SQA team. They also plan to build suites to regression test an in-house application prior to the department's rollout of Windows XP. It is also anticipated that the department may pilot the tool for use in test-driven development.

5 Division of Effort

To make this approach successful, it is recommended that a test automation team have a dedicated test engineer. BA, SME, and/or SQA professionals can comprise the user community. The test engineer should ideally dedicate 100% of his/her effort to the construction of software and the GUI map; their time requirement may be reduced during the ensuing maintenance phase. The user community should dedicate the necessary time for initial training on the test automation framework and the test creator tool, and for creating and maintaining test assets. The following highlights the test engineer's focus on constructing the test creator tool using the keyword-driven framework and the proposed design.

6 Framework Overview

The test automation framework which we incorporated into our test creator tool design consists of the following fundamental elements:

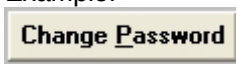
- GUI Maps
- Test Assets
    - Test Steps
    - Test Cases
    - Test Suites
- AUT Invocation Code
    - Main Loop
    - Function Libraries

6.1 GUI Maps

GUI Maps are tables that translate meaningful names (human readable) to technical names (machine readable) for all screens and all components on each screen. Typically, the name that was given during development of each component in the application is the technical name the test automation program

uses to recognize the component. Depending on the development platform, the title bar caption or the name property may be used to identify a screen. Those responsible for the GUI map may use a label or caption on or next to a component to map that component to its technical name.

Example:



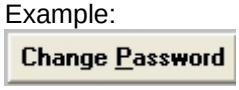
During application development, the programmer may have named this component *cmdChgPsswrd.*

A test engineer may choose to map the caption, *Change Password*, to this component so that the test creator tool users are better able to determine what component to interact with.

Change Password would be the meaningful name given to the technical name, *cmdChgPsswrd*.

Each screen in an application has one or more components. Every component is assigned a control so that the test automation program is able to determine in which way it must interact with the AUT.

Example:



Change Password is the component.
The control for this component is a **PushButton**.

Example of a *Login Screen* GUI Map:

Screen Map

Name=applogindlg

Component Control	Component Map Component Map Argument
Cancel PushButton	Name=cb_cancel
Change Password PushButton	Name=cb_change_password
OK PushButton	Name=cb_ok
Password TextBox	Name=dw_login;\;Name=password
Status Bar StatusBar	Name=st_messageline;State=Disabled
User ID TextBox	Name=dw_login;\;Name=userid

6.2 Test Assets

6.2.1 Test Steps

Test Steps are low level tables that link components to screens and actions to controls. Default parameters can be optionally set for test steps. Screen and component names come from the GUI map. The default parameters represent data normally used for input or verification and can be overridden by all test cases that use any of the test steps containing them. The test steps interact directly with the AUT to perform specific functions including navigation and the manipulation and verification of data.

Test Asset: Step_LoginToMSM

Screen	Component	Control	Action	Parameter
Login	User ID	TextBox	InputValue	
Login	Password	TextBox	InputValue	
Login	OK	PushButton	Click	

6.2.2 Test Cases

Test Cases are higher level tables that combine a sequential list of test steps to perform specific test operations beginning and ending at the AUT central point. Test cases allow for assigning new or overriding existing values to parameters declared in test steps. This feature leads to high reuse. Once built, the test cases resemble manual test scripts.

Test Asset: Case_ChangeDateForSpecBAandVerifyStatusText

Screen	Component	Action	Parameter	Process
Launch Panel Parent	Launch Panel	Click		☐
Launch Panel	Spec BA List	DblClick		☑
Spec BA List	Custom Criterion	SelectValue	Folder ID	☑
Spec BA List	Custom Criterion Value	InputValue	03304426	☑
Spec BA List	Refresh	Click		☑
Spec BA List	Member	SelectValue	No	☑
Spec BA List	Horizontal Scroll	ScrollPageRight		☑
Spec BA List	Contact Performed Date	InputValue	040705{TAB}	☑
Spec BA List	Status Bar	VerifyValue	Please enter a valid Contact Date	☑
Spec BA List	Menu File->Exit	Click		☑
Dialog - Save Spec BA List	Window	VerifyExistence		☑
Dialog - Save Spec BA List	Yes	Click		☐
Dialog - Save Spec BA List	No	Click		☑
Dialog - Save Spec BA List	Cancel	Click		☐

6.2.3 Test Suites

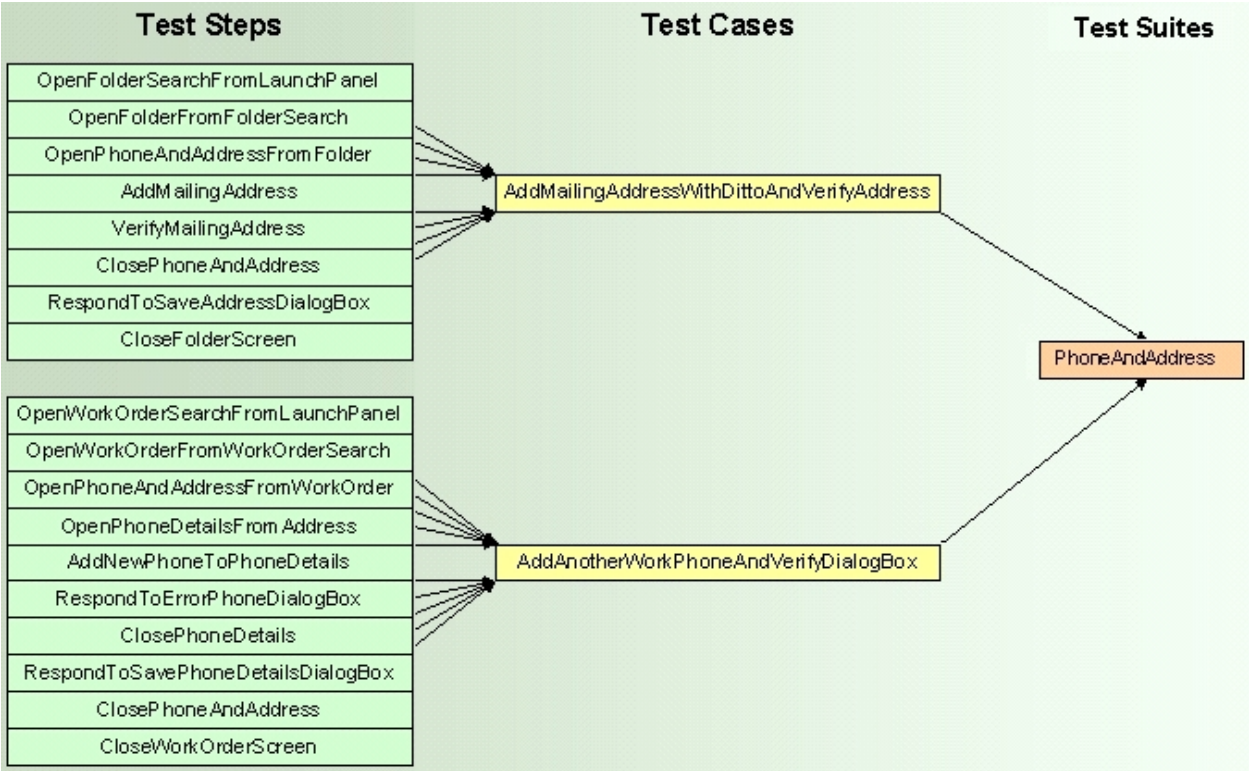
Test Suites are the highest level tables that specify which test cases to be executed. Different test suites can be defined to perform on different modules or business components of an application.

Test Asset: Suite_PhoneAndAddress

Test Case Name	Process
Step_cmdLaunchApplication	☑
Step_cmdLoginToMSM	☑
Case_ChangeCellPhoneProviderAndVerify	☑
Case_RemoveCellPhoneProvidersAndVerifyAvailability	☑
Case_AddNielsenPhoneAndVerifyFieldProperties	☑
Case_ChangeTypeOnPhoneDetailsAndVerifyFieldProperties	☑
Case_AddMailingAddressWithDittoButtonAndVerify	☑
Case_AddAnotherWorkPhoneAndVerifyDialogBox	☑
Step_cmdExitApplication	☐

6.2.4 Dependencies

Test assets are linked and dependent upon one another. The following diagram illustrates how test steps are the reusable building blocks to test cases. Test steps should be broken down by their function: steps for navigating, steps for changing data, and steps for verifying values and properties. Likewise, test cases are essential elements that must be constructed in order to create suites that call on them.



6.3 AUT Invocation Code

Each of the previously detailed elements (GUI maps and test assets) of the automation framework are entered, stored, and maintained in the test creator tool. The final element, the code that is used to execute automated tests scripts against an AUT, is developed in the record and playback test automation program and is the backbone of this design. The code interacts with the screens and components of the AUT and reports the results of such interactions to a log for review. The code can be written to input or select values, verify values or properties, interact with (e.g. click) and verify the existence of screens and components. The code uses simple decision constructs and can easily be rewritten should it be necessary to change to another vendor's record and playback program.

6.3.1 Main Loop

The Main Loop is a script that loops through each instruction presented by test assets. A case statement, which lists the standard controls (e.g. textbox.) and custom controls (e.g. ActiveX) used in various software development platforms, is used to process each instruction by the control it contains. The case statement ensures that each instruction being processed is branched into its appropriate function. When the function returns a result to the Main Loop, a pass or fail message is recorded to the log.

```
Open sFilePath for input as #1

Do While Not Eof(1)

    Input #1, sTestCase, vProcess

    If vProcess = "1" Then

    SQAConsoleWrite "*** START TEST CASE *** " & sTestCase
    SQALogMessage sqaPass, "< Start Test" & sTestCase, "Begin " & sTestCase

    'Begin test case(s)
    Open sTestCase & ".csv" for input as #2
    Do While Not Eof(2)

        Input #2, vProcess, vScreen, vComponent, vControl, vAction, vParameter, _
                  vScreenMap, vScreenMapArg, vComponentMap, vComponentMapArg, _
                  vAuxComponent, vAuxComponentArg, vOnPass, vOnFail

        If vProcess = "1" Then

            Select Case Ucase(vControl)

                Case "COMBOBOX"
                    Result = fComboBox (vScreenMap, vScreenMapArg, vComponentMap, _
                             vComponentMapArg, vAuxComponent, vAuxComponentArg, _
                             vAction, vParameter)
                    If Result <> sqaSuccess Then
                        SQALogMessage sqaFail, "  Screen: " & vScreen & ", & _
                            Component: " & vComponent & ", Action: " & vAction & _
                            ", Parameter: " & vParameter, vAction & " Failure"
                    Else
                        SQALogMessage sqaPass, "  Screen: " & vScreen & ", & _
                            Component: " & vComponent & ", Action: " & vAction & _
                            ", Parameter: " & vParameter, vAction & " Success"
                    End If
                Case "LISTBOX"
                    Result = fListBox (vScreenMap, vScreenMapArg, vComponentMap, _
                             vComponentMapArg, vAuxComponent, vAuxComponentArg, _
                             vAction, vParameter)
                    If Result <> sqaSuccess Then
```

6.3.2 Function Libraries

Function Libraries contain the functions created for each control. Each function can determine the necessary action to perform on the control via a case statement. The test automation program uses each control's function, called by the Main Loop, to determine how to interact with the application under test. Done properly, code written for an application can be reused for other applications having the same software development platform. This is due to the fact that no business logic is programmed in the code; rather, only the interactions between the test automation program and the controls themselves are coded.

```
Function fComboBox (vScreenMap, vScreenMapArg, vComponentMap, vComponentMapArg, _
                    vAuxComponent, vAuxComponentArg, vAction, vParameter) as Integer

Dim Result as Variant
Dim ExpectedValue as String
Dim ActualValue as String
Dim Property as Variant

    'First, set the context to the window of the component
    Window SetContext, vScreenMap, vScreenMapArg

    'Determine the action to be performed on the control
    Select Case Ucase (vAction)

        Case "SELECTVALUE"

            'The control itself is clicked and recognized.
            DataWindow Click, vComponentMap, vComponentMapArg
            'Then a common required activation is called.
            Window SetContext, "DropDownDW", "Activate=0"
            'Finally, the value chosen in the ComboBox is selected.
            DataWindow Click, vAuxComponent, vAuxComponentArg & vParameter

        Case "VERIFYVALUE"

            'The control itself is clicked and recognized.
            DataWindow Click, vComponentMap, vComponentMapArg
            'Then, the actual value stored in the combobox is retrieved
            Result = SQAGetProperty(vComponentMap "Value", ActualValue)

            'Verify if the Actual Value is what is expected
            If ActualValue = ExpectedValue Then
                'Validation Succeeds
                fComboBox = 0
            Else
                'Validation Fails
                fComboBox = 1
            End If

        Case "VERIFYPROPERTY"
```

7 Rapid Application Development Tool

Most business' software portfolios include a program that is ubiquitous in the corporate world but sometimes not utilized to the extent for which it was developed. Microsoft Access™ is a tremendous resource for building self contained applications yet often overlooked as a serious product with which to develop software. Other development languages and databases can be used to develop a similar product but MS Access™ is usually already available and encapsulates most everything needed to develop the test creator tool.

The development environment in MS Access™ is wizard based and is programmable using Visual Basic for Applications (VBA) and Structured Query Language (SQL), relatively easy languages in which to learn and develop software. MS Access™ is a relational database that in addition to providing the means for building and storing tables and queries also provides the capabilities for creating and storing forms, reports, macros, code modules, and web pages, thereby simplifying the construction and deployment of software applications such as the test creator tool. For more information on MS Access product features see http://office.microsoft.com/access.

When tasked to develop a product using a keyword-driven approach to test automation, the choices were: 1.) Use the available open source project, http://sourceforge.net/projects/safsdev, which uses linked spreadsheets, 2.) Purchase a product that may or may not suit our business requirements, or 3.) Develop the product in-house using an industry known framework ensuring that the features most important to us were developed while keeping costs minimal. The choice was simple for us as a skilled development resource was available and the framework itself is comprehensive and uncomplicated, lending itself to a simple design concept which performs complex tasks,.

8 Proposed Design

8.1 Database

8.1.1 Tables

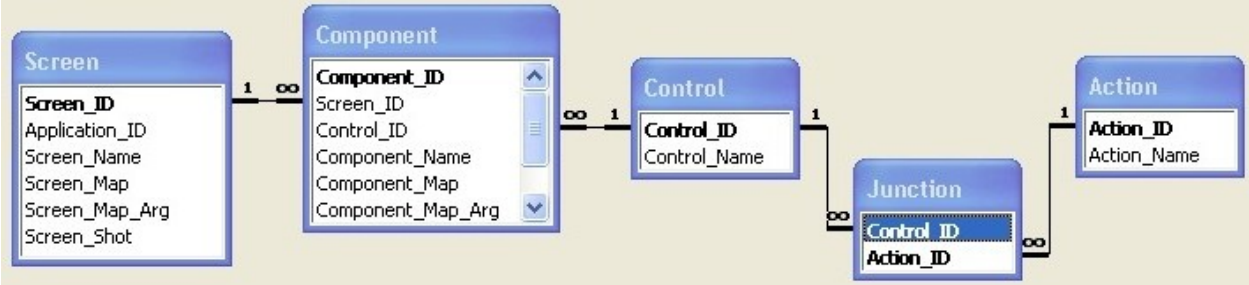
First, create the following tables used for supporting GUI maps and test assets:

- Screen
- Component
- Control
- Action
- (Junction) For many-to-many relationship

8.1.2 Relationships

Next, develop the following table relationships:

- One-to-many relationship between Screen and Component
- One-to-many relationship between Control and Component
- Many-to-many relationship between Control and Action



8.1.3 Referential Integrity

Finally, enforce referential integrity (optional)

- Cascading Updates
- Cascading Deletes

By incorporating this basic table design, the construction of forms, queries and reports is simplified. The test engineer is responsible for building the various forms that the user community will use to create test assets. The data relationships established promote a more user-friendly, less error-prone, and visually

cohesive product. These relationships allow for combo boxes to be built off of one another as in the Test Step Creation Form example.

8.2 GUI Map Forms

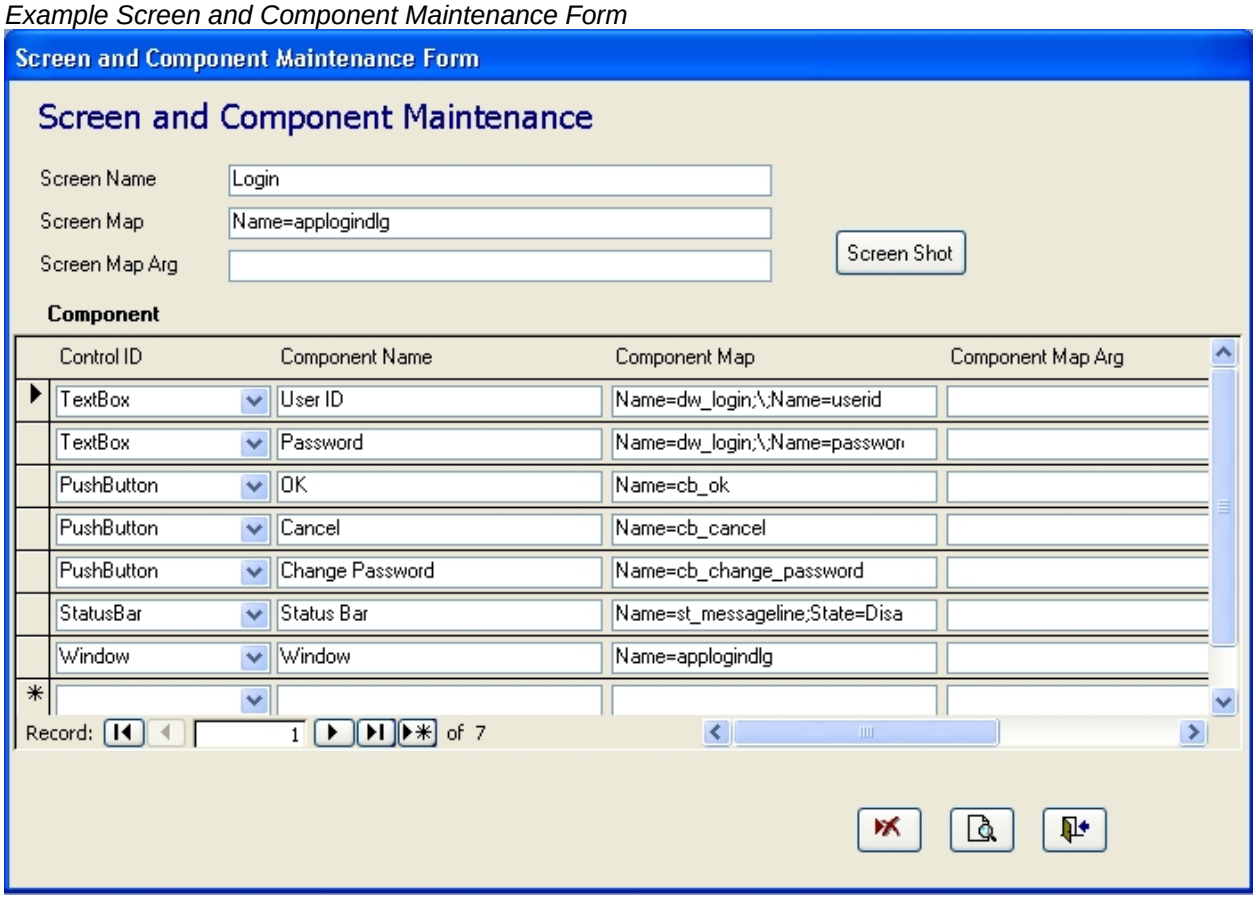
8.2.1 Screen and Component Maintenance

The test engineer is also responsible for mapping an application's GUI. The one-to-many relationship between screen and components allows for a screen and all of its components to be GUI mapped on the same form. It also allows for components on different screens to share the same name without any additional provisions. Control assignments can be made to components on this form as well.

Optionally, an image of each screen can be stored which can then be used when creating or editing test steps to help the user discern which components to use for interacting with the AUT. Linking the screen prints to the original image files can be beneficial in that when changes are made to a screen's design, the image can be updated and the link will reflect the changes automatically.

Screens and components must be entered into the GUI map prior to the construction of test assets that need them. GUI maps and test assets can be constructed iteratively. In test-driven development, the screen and component names are required to build test assets but the component map is not.

Example Screen and Component Maintenance Form

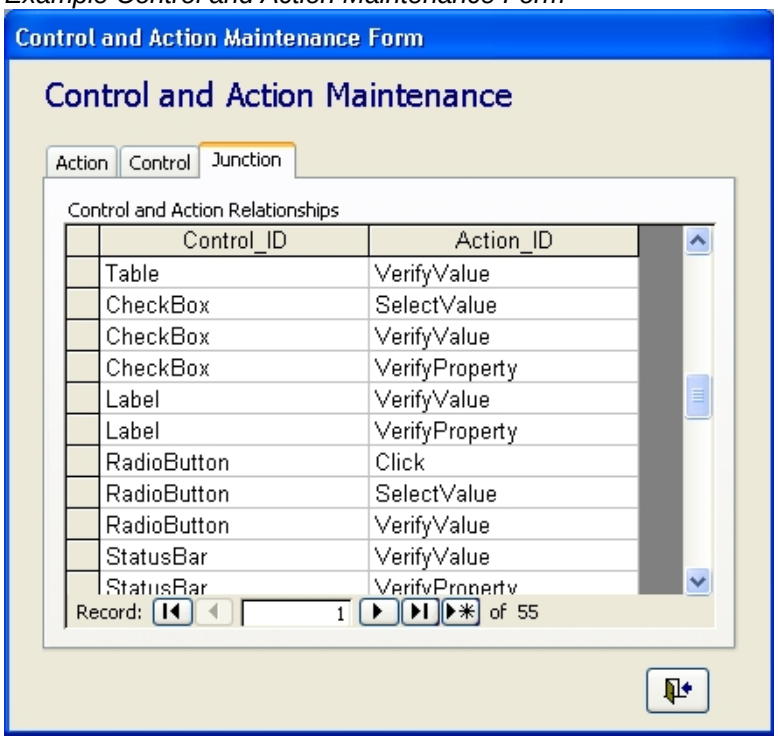


8.2.2 Control and Action Maintenance

The relationships established in the table design facilitate the use and convenience gained by linking controls to actions. A separate maintenance form should be constructed and used to enter the various controls, their actions and the many-to-many relationship between controls and their actions. Controls should be entered on this form prior to making assignments to components on the Screen and Component Maintenance Form.

Also, the actions entered here for each control will prove beneficial when creating test steps as will be illustrated in the Test Step Creation Form. These actions will later dictate how the test automation program interacts with the AUT.

Example Control and Action Maintenance Form



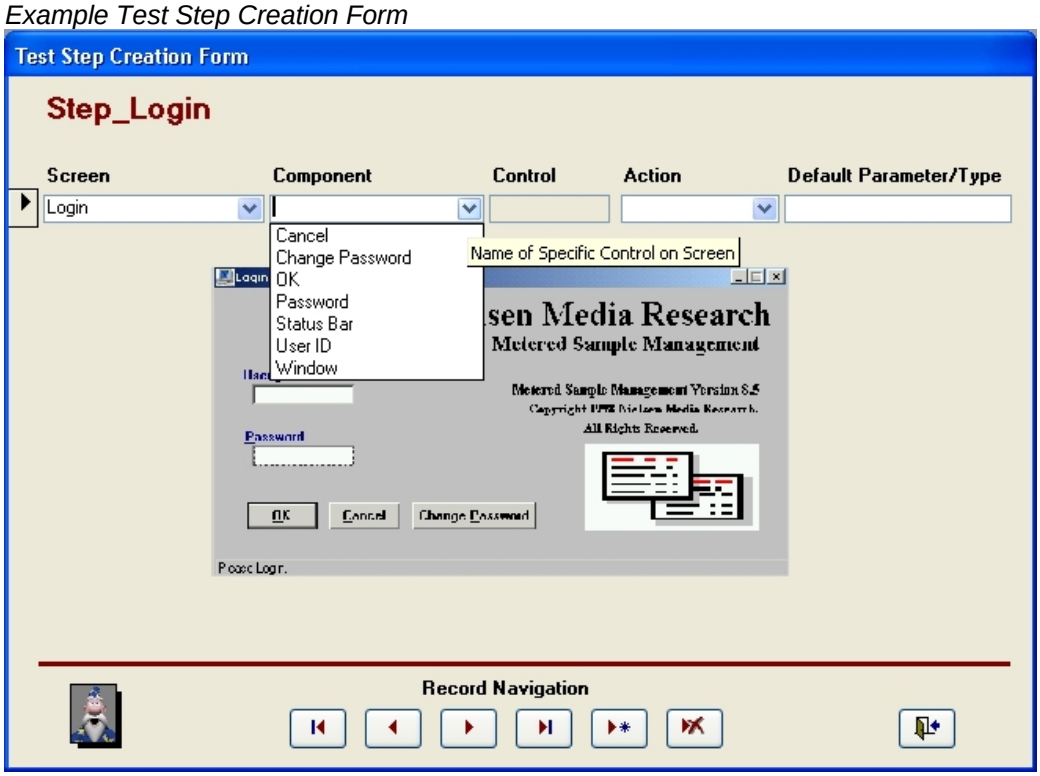
8.3 Test Asset Forms

Once all forms for creating the various test assets have been created, the user community can begin focusing their efforts on creating test steps, test cases, and test suites. The functional tester should not have to know anything about the details of a GUI map or the code. They should be able to intuitively create test scripts using English language constructs and process flow. The following form descriptions and behaviors assume that the test engineer constructs forms similar to the examples provided.

8.3.1 Test Step Creation

The Test Step Creation Form can be programmed to behave intelligently due to the previous design work of the test engineer. On this form, screen names are listed in a combo box. When a screen name is selected, an image of that screen appears. Also, the component combo box dynamically changes and presents only those components related to the chosen screen. The same feature is applied for actions. Actions are linked to controls, therefore, when a component is selected (as stated earlier, each is assigned a control), only the actions belonging to that component appear. The persons responsible for generating test assets can optionally provide a default parameter for those test steps that require them.

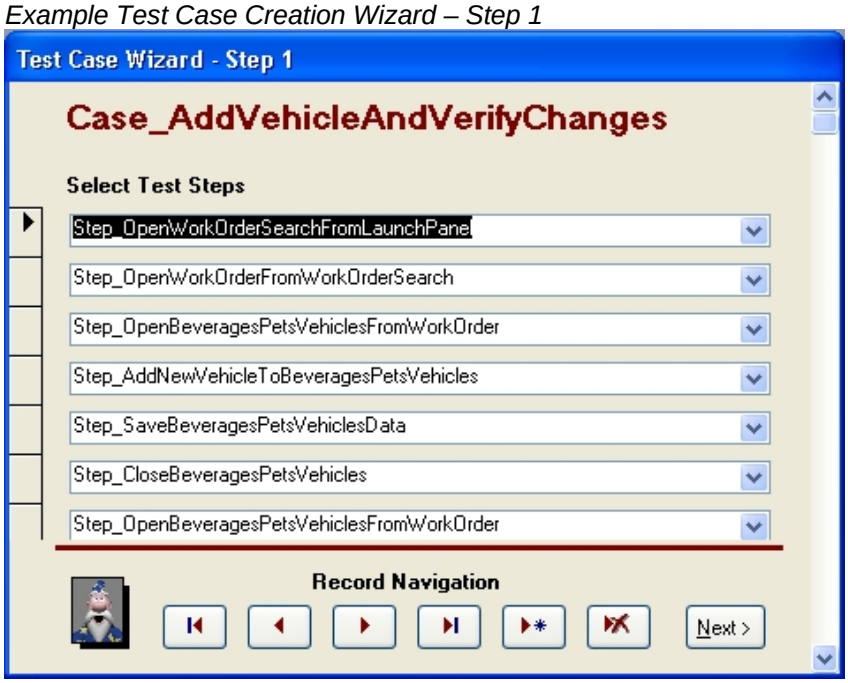
Example Test Step Creation Form



8.3.2 Test Case Creation

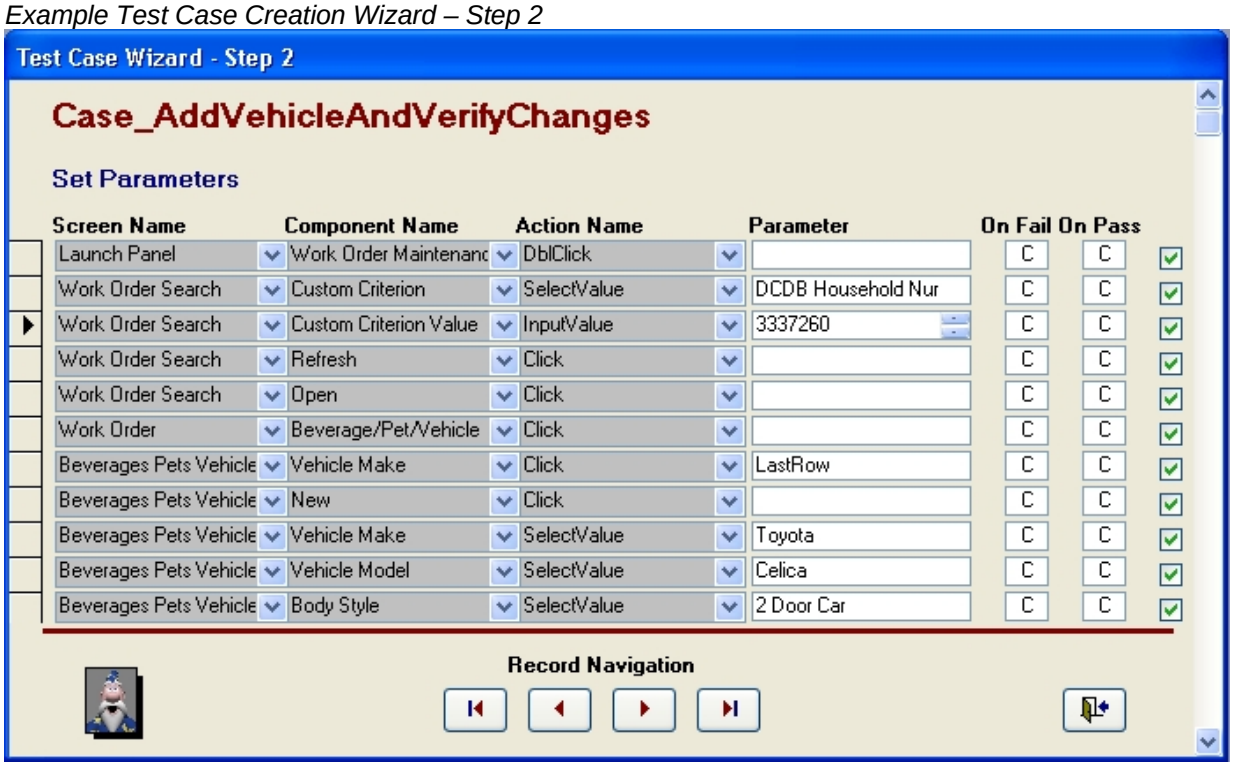
On the Test Case Creation Wizard, test steps are listed in a series of combo boxes. First, a user can select all the necessary test steps to be used in the test case. After all test steps have been selected, the user can advance the wizard to see each line item for each test step in the test case. This is where the default parameters created at the test step level can be overridden. Additionally, a checkbox can be unselected if a particular test step is not to be processed in the current execution of the test case.

Example Test Case Creation Wizard – Step 1



Notice how similar the following screen is to a manual test script.

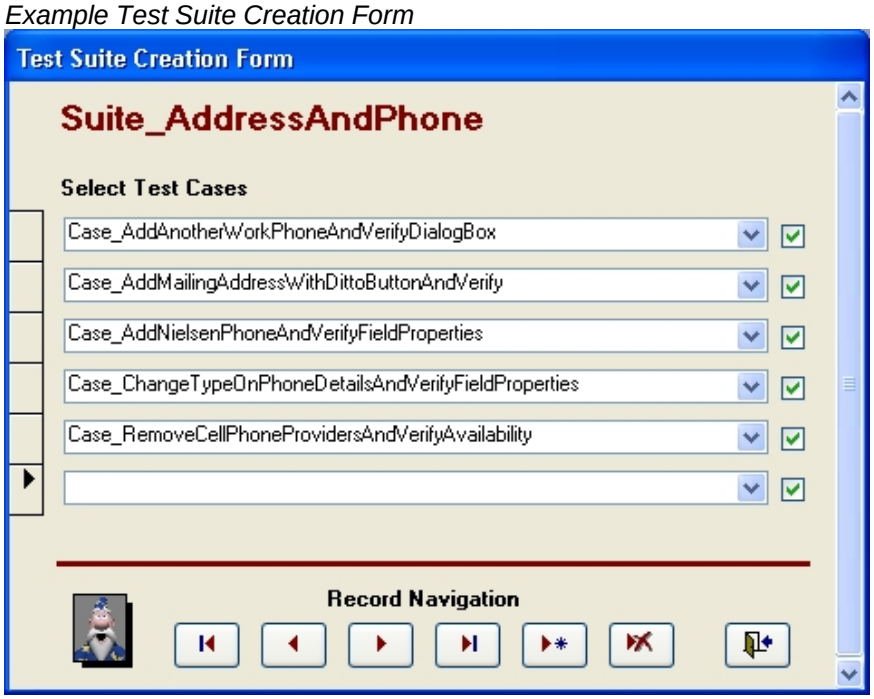
Example Test Case Creation Wizard – Step 2



8.3.3 Test Suite Creation

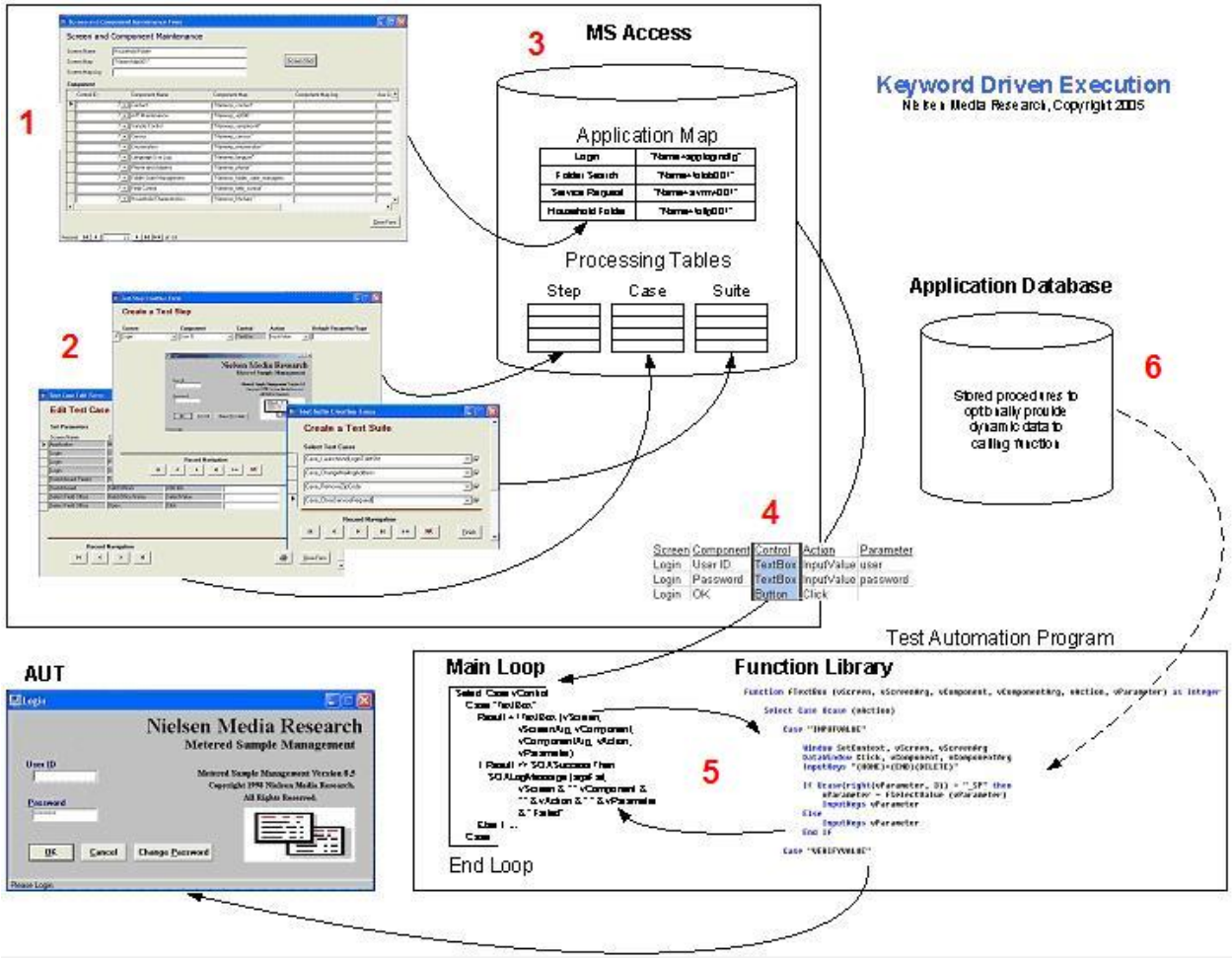
The Test Suite Creation Form can be easy to use if constructed properly. In this example, a user simply chooses the test cases to be executed in a given test suite. Optionally, a user can uncheck any test case that they do not wish to be executed for the current test execution.

Example Test Suite Creation Form



9 Putting it All Together

The following diagram demonstrates how all of the pieces work in concert. After the test creator tool has been developed, 1.) The application's GUI must be mapped. 2.) Test assets can then be created by the user community. 3.) All of this data can be stored in an MS Access™ database. 4.) The GUI map data and test asset data can be joined and published to a LAN as a comma separated file to be processed by the Main Loop and the function libraries. 5.) The Main Loop and function libraries coded in the test automation program work directly with the AUT based on the controls and the actions provided in each test step. 6.) Optionally, stored procedures can be invoked by controls in a function library when they are used in place of hard coded parameters declared in the test steps and test cases. This allows for the dynamic creation and use of data against the AUT.



10 Summary

We have described how a “keyword-driven” or the “table-driven” test automation approach can greatly enhance your overall quality assurance methodology. Test automation can be used as a supplement to your regression testing effort. Not all test cases qualify as candidates for test automation. There may already be a software package, MS Access™, in your company that is ready and available to assist in the creation of a test creator tool that costs no more than the effort put into developing the tool using the design provided here. After the tool has been developed, it can be easy to create test steps, test cases, and test suites to be used with a test automation program for interaction with the AUT.

11 Additional Reading

- Nagle, C. “Test Automation Frameworks” White Paper http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm
- Hayes, L. “Implementing a Test Automation Framework” Tutorial presented at STAR EAST 2005 Conference, Orlando, FL, May 16, 2005.
- Automated Testing Specialists, *Methodologies for Automated Testing* http://www.sqa-test.com/method.html
- Zambelich, K. *Totally Data-Driven Automated Testing* 1998 http://www.sqa-test.com/w_paper1.html
- LogiGear Whitepaper Series, *Achieving the Full Potential of Test Automation* October 25, 2004
- Mosley, D. & Posey, B. *Just Enough Software Test Automation* New Jersey: Prentice Hall PTR, 2002.
- Buwalda, H. & Janssen D. *Integrated Test Design and Automation* Great Britain: Addison-Wesley, 2002

Title: An Outlining Program For Testers To Perform Test Analysis and Assist in Generating Test Plans and Documentation

James T. Heuring
Micro Encoder, Inc.
11533 NE 118th Street
Building M, Suite #200
Kirkland, WA 98034

JTHDEV@AOL.COM

Keywords
Outline Explorer, Testing Methodology, CSV, Stream-based, Object-Oriented Interpreter, C++.

About the Company
Micro Encoder Inc., a subsidiary of Mitutoyo Corporation of Japan, is a state of the art research and development facility for computer aided measurement technologies. The software developed here, QVPAK®, uses the advanced sensor hardware also developed here to perform measurements used to improve the manufacturing processes.

About the Author
Jim Heuring has worked as a software engineer since 1985. This experience includes the development and testing of software for medical devices, avionics and software tools. Other experience includes over 3 years as a tester at Microsoft on Exchange (the mail software) and 9 months at Microsoft as a developer on the data access team where he wrote the data access install verification tool called “Component Checker”. Jim is currently a Senior Software Test Engineer at Micro Encoder.

Key Definition: **Variance Table** is a table of possible choices you can make in the testing process. For example, say you are testing a program that can save data in the following different formats (Plain Text, CSV, XML). You might call the table "Output Formats" as shown below:

Output Formats
Plain Text
CSV
XML

BACKGROUND:
For the last 2 years I have been using a rigorous process of generating test specifications and test cases from product requirements. In brief, the process has the following steps

1) Variance Tables are generated from the product requirements
2) The Variance Tables are read into a spreadsheet program.

3) A test analysis is performed using Variance Tables. The output of the analysis are spreadsheets that are generated by combining the Variance Tables.
4) The Variance Tables, test analysis and combined Variance Tables are inserted into the test specification and comprise the bulk of the specification.
5) The Variance Tables and combined Variance Tables are your test cases.

This process was previously done by hand and certain aspects of it were very time consuming and tedious. Therefore, I have developed a tool to automate certain aspects of the process. This tool is called the "**Outline Explorer.**"

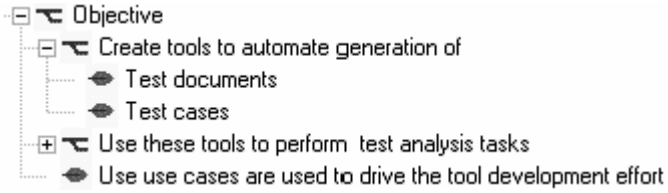
Two other criteria of the tool that were of primary importance to me:

1) The tool can be used without interrupting your thought process. For example, you can use the tool to take notes in a meeting without having to grab the mouse to access a menu items, buttons, etc.
2) Text and hierarchal information can be easily imported and exported.

How to get the most value out of reading this paper:

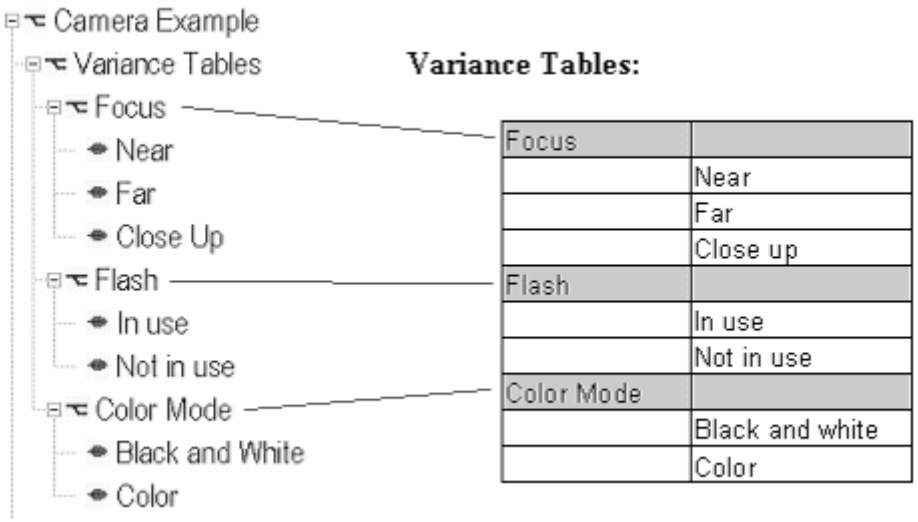
1. The process discussed is conceptually simple and well founded.
2. The Outline Explorer can be used free of charge. The location where it can be obtained from will be announced at the conference and also available on the PNSQC web site. In addition, you may Email me directly (see top of paper) for more information.
3. The Outline Explorer may be used for any task that can benefit by representing data in a hierarchal form. Some useful tasks for this tool:
    a. Brain storming
    b. Writing outlines for papers
    c. Organizing complex thoughts
    d. Chronicling bugs with complex histories
    e. Organizing your life
4. There will be instructional video demos of how to use the tool.
5. Source code for the Outline Explorer may be used freely; however, you must obtain a license to develop against the Flex Grid control. More information on the VSFlexGrid is available on http://www.componentone.com/.

Abstract:



Overview of the testing process using a simple example:
Consider the following example. You are assigned to test a digital camera. One of the first things you want to do is to generate a list of things to test. You might want to consider testing the following features: Focus, Flash, and Color Mode.

Generate Variance tables
The features (Focus, Flash, and Color Mode) are elaborated in "Variance Tables" in the following diagram:



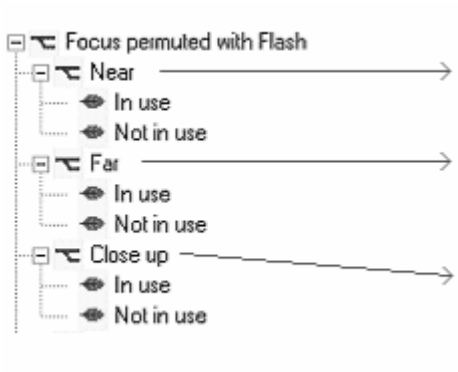
Variance Tables:

Focus	
	Near
	Far
	Close up
Flash	
	In use
	Not in use
Color Mode	
	Black and white
	Color

The Variance Tables are included into the Test Specification.

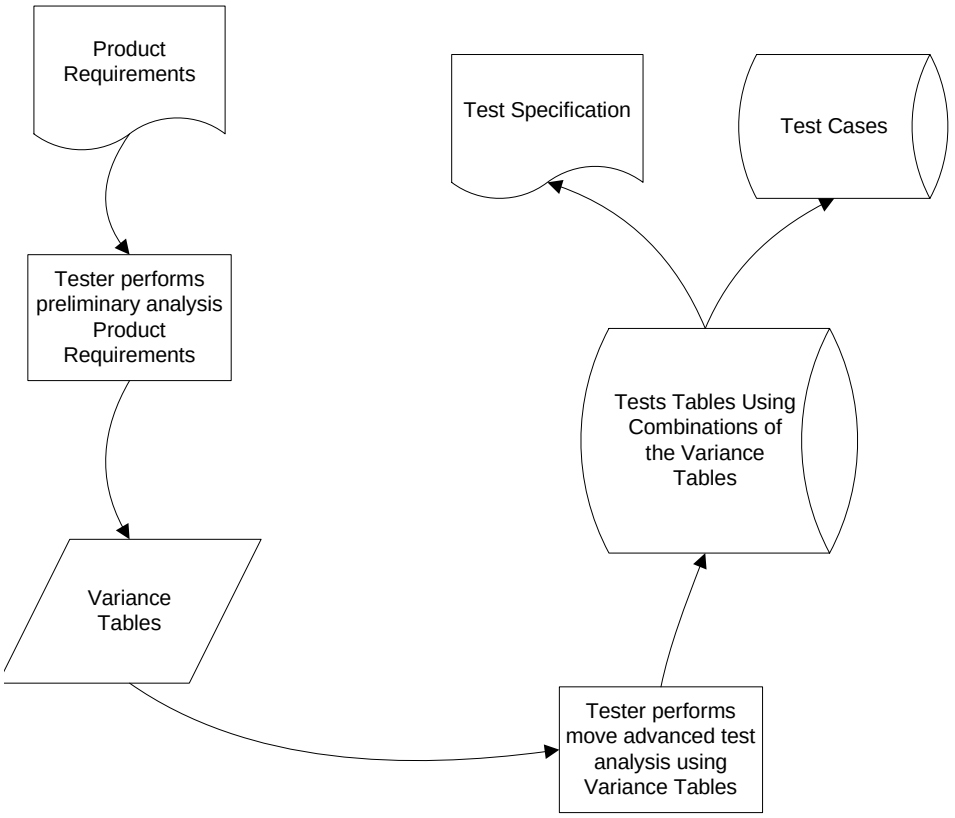
Combine Variance Tables to perform test analysis and create test cases

The Variance Tables are combined to create more elaborate tests. In the case of testing the camera we might want to test all combinations of Focus and Flash. The combinations are shown in the diagram below and adjacent table.



Focus	Flash	Pass/Fail
Near		
	In use	Pass
	Not in use	Pass
Far		
	In use	Pass
	Not in use	Fail
Close up		
	In use	Pass
	Not in use	Pass

Overview of the testing process: without tools



Comma **S**eparated **V**alue (**CSV**) data files can be read by many common Spreadsheet program.

In the diagram above the following tables are stored as **C**omma **S**eparated **V**alue (**CSV**) files:

Variance Tables
Tests Tables Using Combinations of the Variance Tables
Test Cases

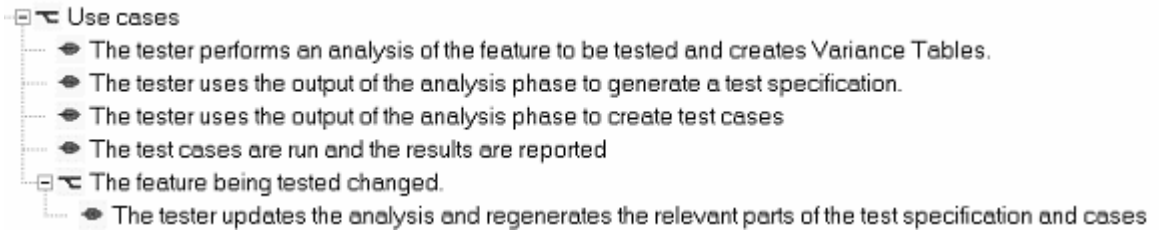
Benefits of the process described above are:

- The tables (spread sheets) present the information in a clean and concise visual way. This makes it easier to review the document.
- The process itself is well defined and guarantees that all test specs have the same look and feel.
- The variation tables represent a clear breakdown of the most basic items being tested. This makes it easier to do the coverage analysis.
- These Variance Tables can be reused in the creation of other test specifications.
- More complex test scenarios can be described as combinations of Variance Tables. This makes the more complex test scenarios easier to understand and review.
- Tools can be applied at several points in process to speed up the delivery of test documents.

Use Cases:

The tool development effort was driven by the use cases. The use cases allowed tool and process development to focus clearly on the intent of the tools rather than the feature set. This approach greatly increased the productivity of the tool development effort.

The use cases for the process and test development effort are listed in the tree diagram below:



How the process works with the use of tools

Below is a summary of the process that will be repeated and fleshed out through the following examples:

1) From the requirements create Variance Tables
2) Use the Variance Tables to perform a test analysis
3) Insert the Variance Tables and test analysis into a Test Spec

Create Variance Tables

The tester creates a Variance Table from the requirements.

Suppose you are given a requirements statement for the camera and it reads as follows:

Requirements Document:

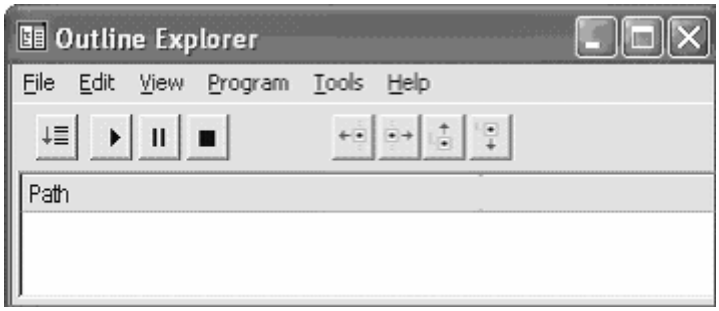
Requirements for the (DCAM 1000) digital camera
Feature 1:
Focus
The Focus feature will have the following options: (Near, Far, Close Up)

Feature 2:
Flash
The Flash feature will have the following options: (In use, Not in use)

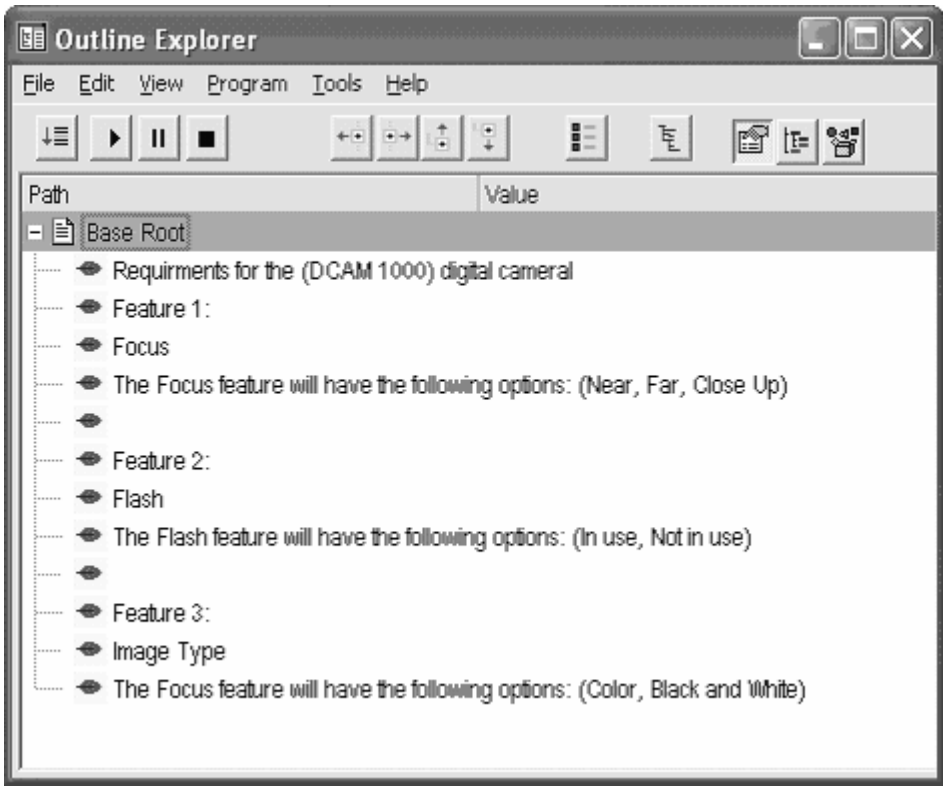
Feature 3:
Image Type
The Focus feature will have the following options: (Color, Black and White)

First: copy the contents of the requirements document into the copy buffer.

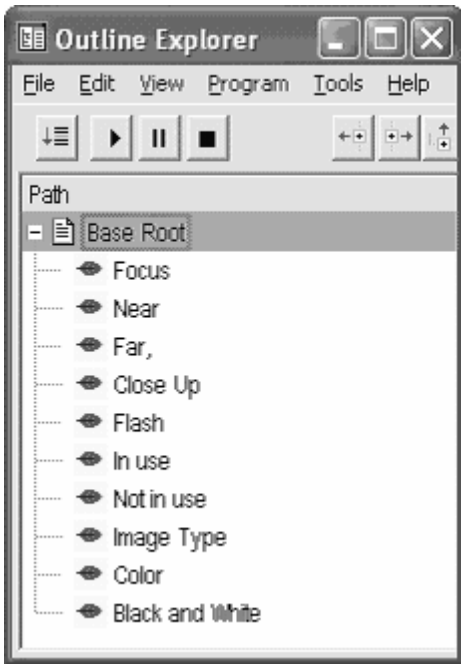
Second: Run the Outline Explorer (shown below)



Third: Paste the requirements you have copied into the Outline Explorer. The pasted text is shown in the following image.



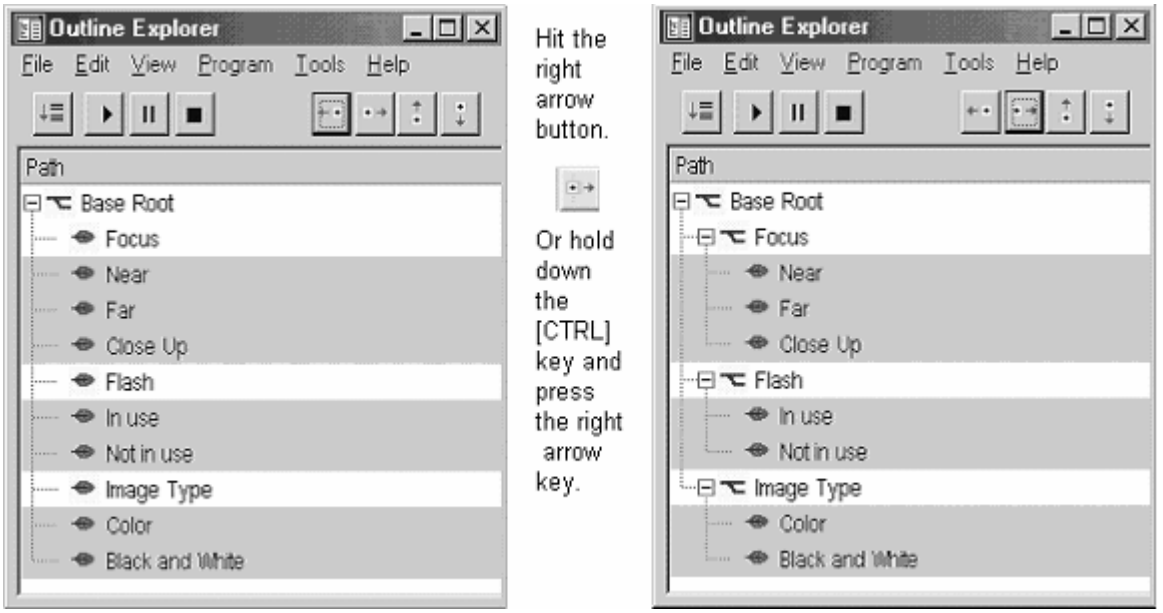
Fourth: Edit the requirements text until you have the Variance Tables. You can use the Outline Explorer like a normal text editor with the exception that you can move the lines of text around in a hierarchal form. In this example, I deleted out the bulk of the text with the exception of the feature names and options as shown in the image below:



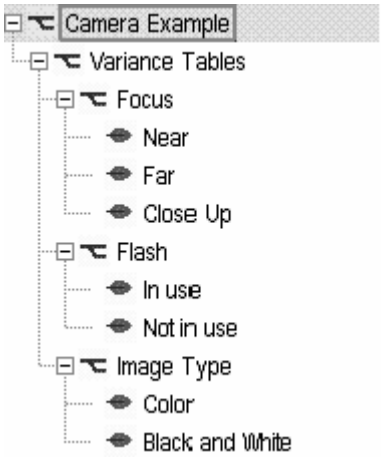
Next you can use the arrow keys in the tool bar () to rearrange the remaining data as shown below

Select all the entries you wish to make leaf nodes of as shown in light blue below.

Next press the right arrow button () or press [ctrl-right arrow] for the mouse free operation to create the hierarchal form as shown below:



Add some additional nodes (Camera Example, Variance Tales) and shuffle things around a bit Eventually you have populated all of your Variance Tables as shown below.



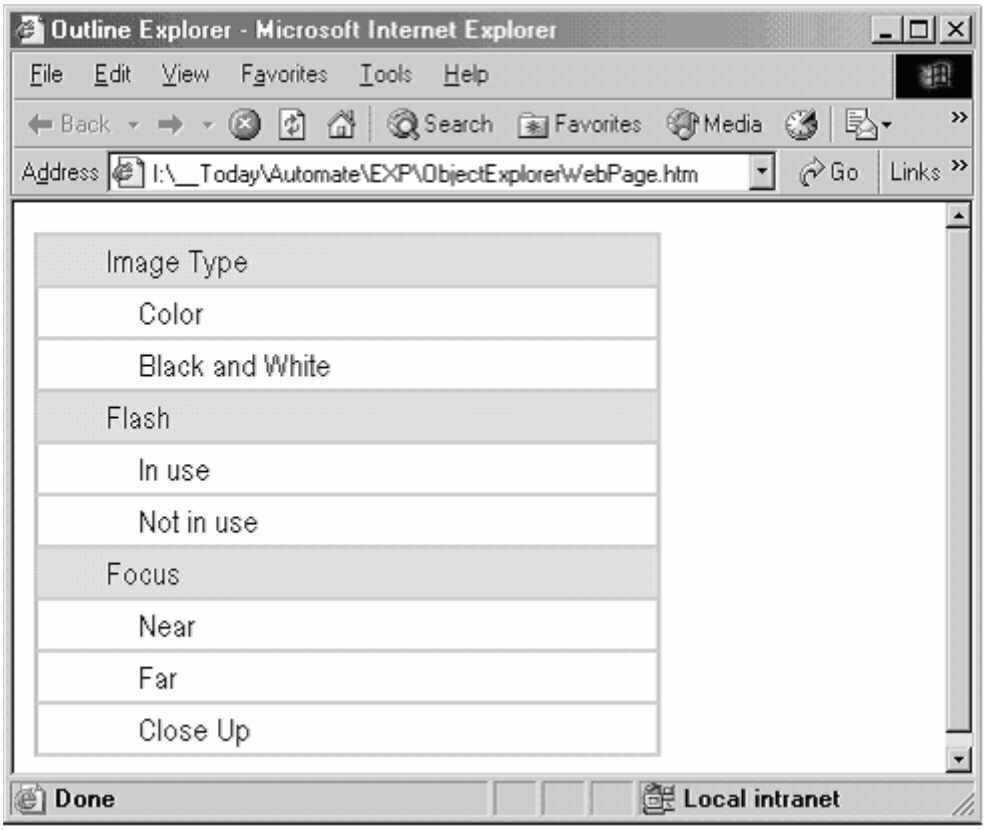
Fifth, Put the tables you have created into the test specification:

Steps
1 Select the items you want to create tables for
2 From the context menu select:

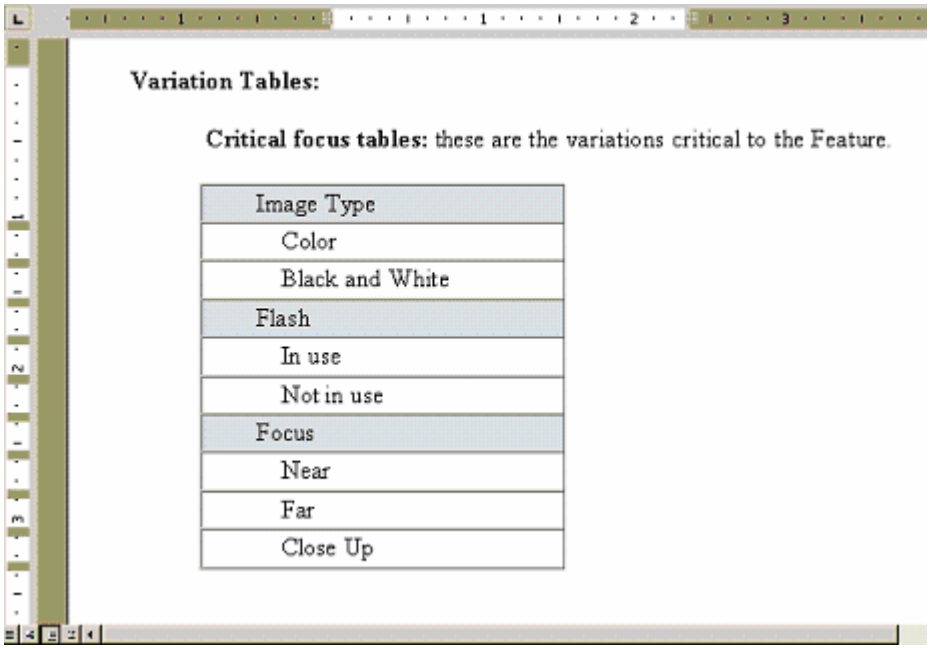
"HTML: Plain text: Selected item and all items below"

Note: you can color the headings using context menu operations.

3 You now have a web page show below.



4 Copy the items from the web page and paste them into a word processor document (shown below).



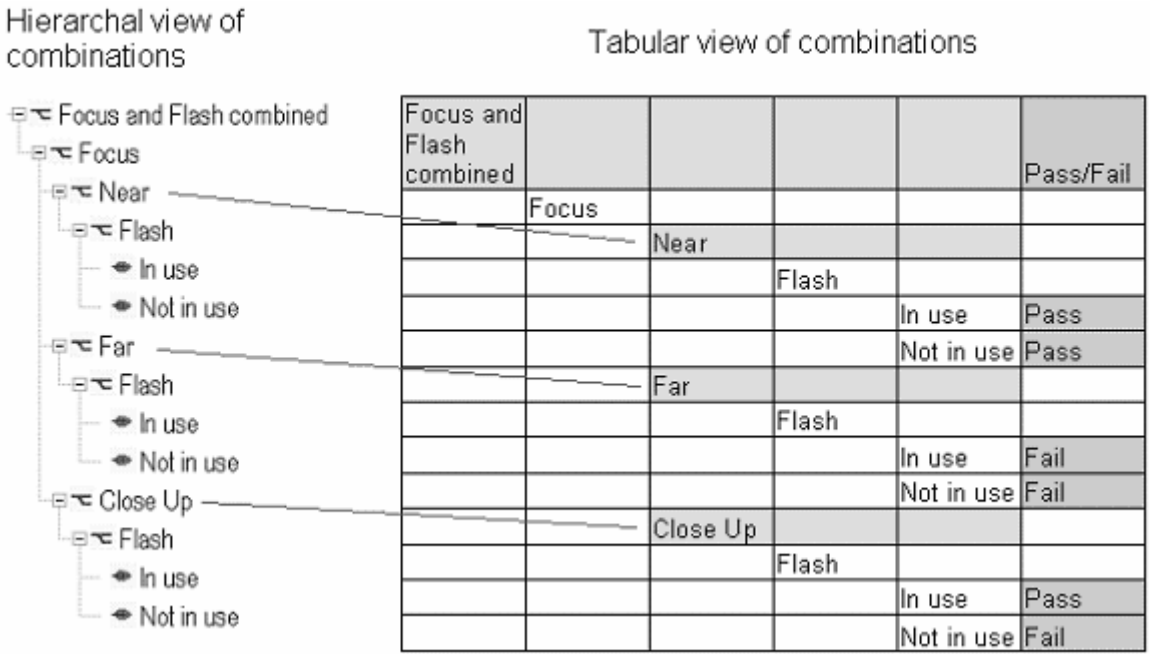
Variation Tables:

Critical focus tables: these are the variations critical to the Feature.

Image Type
Color
Black and White
Flash
In use
Not in use
Focus
Near
Far
Close Up

Create more advanced tests using the Variance tables

Now that you have created the Variance Tables you want to create more advanced tests using them. One way to do this is to copy and paste nodes. The result of this process is shown in the image below on the left hand side. A table is generated from the hierarchal view as show in the right hand side of the image below:



Hierarchal view of combinations

- Focus and Flash combined
  - Focus
    - Near
      - Flash
        - In use
        - Not in use
    - Far
      - Flash
        - In use
        - Not in use
    - Close Up
      - Flash
        - In use
        - Not in use

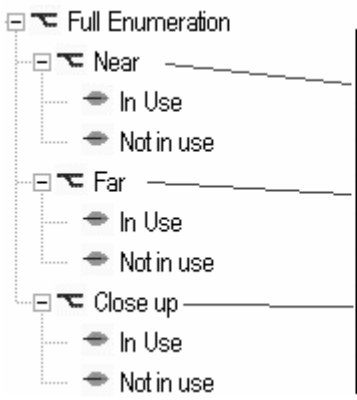
Tabular view of combinations

Focus and Flash combined					Pass/Fail
	Focus				
		Near			
			Flash		
				In use	Pass
				Not in use	Pass
		Far			
			Flash		
				In use	Fail
				Not in use	Fail
		Close Up			
			Flash		
				In use	Pass
				Not in use	Fail

Software to programmatically create permuted tables
Another way to create combined Variance Tables is by using software that creates combinations of test cases based on the entries of two or more tables. The software that combines tables is called the Permuter.

Permuter functionality is available via the UI or though the underlying interpreter, which is discussed further on.

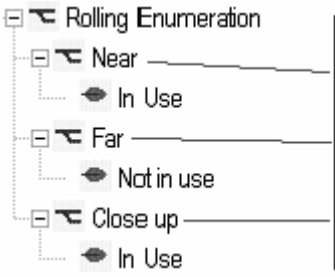
Different kinds of Permutations
A Permuter that uses all possible combinations of table entries is called **Full Enumeration Permuter** or **Full Permuter**. From the example above, when we fully permute Flash and Focus we get the following set of permutations:



Focus	Flash	Pass/Fail
Near		
	In use	
	Not in use	
Far		
	In use	
	Not in use	
Close up		
	In use	
	Not in use	

Sometimes we want a Permuter that only enumerates the full number of values in the first table (Focus) and selects values iteratively from the second table (Flash).

In this case we have been discussing above we end up with the following hierarchy and table:



Focus	Flash	Pass/Fail
Near		
	In use	
Far		
	Not in use	
Close up		
	In use	

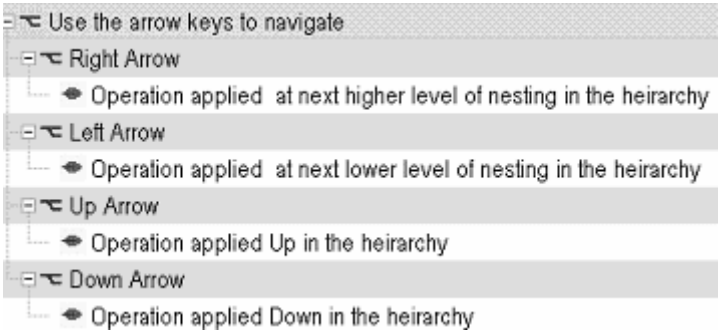
This kind of permutation is called a **Rolling Enumeration Permuter** or Rolling **Permuter** because the values in the second table (Flash) are enumerated only one at a time for each of the values in the first (Focus) table. When the end the end of the second table (Flash) is reached the software rolls back to the beginning of the table to select the next entry.

In either case, a tool is run to generate the individual test cases. The leaf nodes of the hierarchy are individual test cases that are converted into a CSV format.

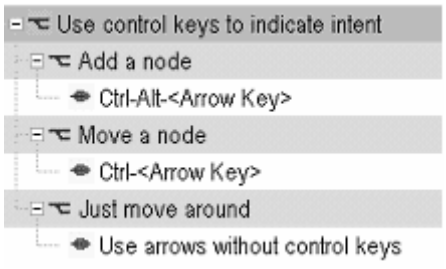
Use a Rolling Permuter to increase the range of your testing
You can add additional Variance Table values to your testing and not increase the number of test cases.

Getting around in the UI:

Use the arrow keys on the keyboard to specify the direction.

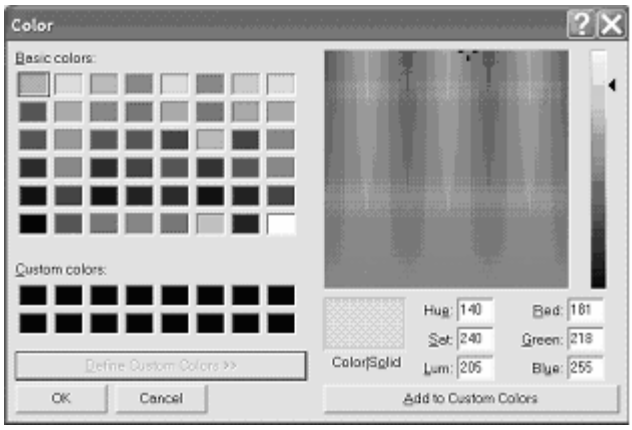


Use the [Alt] and [Ctrl] keys to indicate the intent of your operation.



Use Color to add meaning to the hierarchy:

In the images in the section " **Getting around in the UI** " we use color focus the eye on items of importance. You can change the color of selected items using the standard window control shown below:

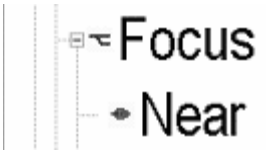


Color can be used to indicate:

* Items in a document that need special attention.
* Items that you are currently focusing attention on.

Change the font size

Use the CTRL key in conjunction with the mouse wheel to enlarge:



or shrink the font:

The image above might seem a little extreme; however, it does allow you to get a general sense of the structure of what you are working on.

Broader uses of the Outline Explorer

Because of the power of controlling hierarchal information I have applied this tool to:

* Communicating weekly status to my supervisor
* Capturing complex information pertaining to bug investigations
* Planning guitar/music practice sessions

Other uses for web pages created from the Outline Explorer

The web pages can be generated in Plain Text, CSV, XML.
The web pages can be used to save the state of your work for you to review your work over time.
The web page allows you to select and multi select easily. You can also use the web pages distribute the information.

The Outline Explorer has a built in object oriented interpreter

The object-oriented interpreter has a set of APIs for creating permuted tables. The interpreter is a separate application called RegistryChecker.EXE.
RegistryChecker is program that reads in a stream of object-based instructions from a text file and performs the indicated operations.

The following is a set of instructions fed to the interpreter to create a permutation:

```
Converter.InputFile_FirstTable  = c:\First_Table.TXT
Converter.InputFile_SecondTable = c:\Second_Table.TXT
Converter.OutputFile            = c:\Permutation_Result.CSV

Converter.Generate_Full_Permutation_Table

' or

Converter.Generate_Rolling_Permutation_Table
```

To run these set of instructions against the interpreter copy the instructions above into a text file, say "Text_Permutation.TXT".

Next, create a batch file with the following entry:

```
RegistryChecker.exe "/s:Text_Permutation.txt"
```

CONCLUSIONS

Lessons learned:

- Rolling Permuters allow you to extend the range of you testing without increasing the number of test cases you run.
- Use cases provide a powerful way to focus process and tool development effort
- CSV files were more useful than XML/XSD files. XML files have restrictions on the characters that you can enter. They are harder to read and tweak.

Outline Explorer anatomy:
The Outline Explorer is written C++. The WTL (Windows Template Library) is a lightweight Template Library used as the framework to simplify the Windows development. ATL (Advance Template Library) is used for general operations. Active X is used for the controls.

Clear, concise and measurable requirements –

Are they possible?

Debra Schratz

This paper demonstrates methods used at Intel to write clear, concise, and measurable requirements. Requirements engineering, specifically writing “good” requirements is really hard to do! While working with hundreds of product development teams at Intel, we have developed a keyword-driven syntax, adapted from Tom Gilb’s Planning Language (Planguage) which allows teams to write requirements that can be re-used over and over for subsequent projects. This paper will explain how requirements authors at Intel define “requirements engineering”, expose the power of using a keyword driven syntax through discussion of key lessons learned, and provide the necessary skills to effectively use this methodology.

Author’s Bio:

Debra Schratz has 8 years experience in quality engineering. Debra currently works as a Program Manager in the Platform Quality Methods group, part of the Corporate Quality Network at Intel Corporation. Her role is to help drive good requirements engineering practices into Intel’s Mobility product development teams. Prior to her work in quality, Debra spent 8 years managing an IT department responsible for 500+ node network for ADC Telecommunications. Debra is a member of Portland, Oregon’s Rose City Software Process Improvement Network (SPIN) steering committee, coordinating the monthly speakers and communicating meeting information. In addition, Debra serves as Vice President of the Pacific Northwest Quality Conference (PNSQC) held in Portland every fall. She holds a Bachelor of Arts degree in Management with an emphasis on Industrial Relations.

Introduction

"I held my **entire program up for 4+ weeks** due to unclear, unwritten requirements. Took some heat for that in the beginning, but the deep dive requirements effort is highlighting a Silicon spin we didn't know about, standards that we don't support, other post-launch requirements nobody considered…all of this causing us and management to question the viability of the product.

By the way, this is all stuff we wouldn't have realized until it smacked us in the face 6 months from now. **Spending a month now prevented us from spending millions** before a conscious decision."

The above quote is from a Project Manager from Intel Communications Group (ICG) who attended the "Writing Good Requirements" workshop developed at Intel, and then went back to his group and applied the concepts of requirements engineering to his project.

In 2004, ICG embarked on a requirements engineering initiative to improve the quality of the requirements written by their silicon product teams. This paper explains how requirements authors at Intel define "requirements engineering", and how we have tapped into the power of using a keyword driven syntax to help requirements authors write clear, concise and measurable requirements. Many key lessons learned from ICG will be shared to provide you and your team with the necessary skills to take this methodology back to your organization to begin using it effectively right away.

What is a requirement?

Let's begin with defining the word "Requirement". The Institute of Electrical & Electronic Engineers (IEEE) definition [1] is:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
3. A documented representation of a condition or capability as in items 1 or 2.

The IEEE definition emphasizes requirements come from user needs, contracts, standards, and other imposed conditions. IEEE acknowledges that requirements exist **even if they are not documented –** an important note.

Many definitions exist for the word "requirement". A basic Google search returned 16 different definitions. One reference provided a link to a glossary of requirements engineering terms housed on Karl Wiegers' website. [2] Karl states:

> **"Requirement:** A statement of a customer need or objective, or of a condition or capability that a product must possess to satisfy such a need or objective. A property that a product must have to provide value to a stakeholder."

Within Intel, we have studied the industry definitions and have defined "requirement" as a statement of any one of the following:

1. What a system must do (a functional requirement)
2. How well the system must do what it does (a non-functional requirement i.e. quality or performance requirement)

3. A known resource or design limitation (a constraint, boundary condition, or budget)

Purpose of Requirements

What is the value of teams spending time to document the project's requirements? We define the purpose simply as:

> Requirements establish a clear and common understanding of what the product must accomplish.

The term *clear* means that all requirements statements are unambiguous, complete, consistent and concise. This clarity allows the stakeholders to share a *common understanding* of what the team needs to accomplish.

We demonstrate to teams how requirements are the basis on which products are built (including documentation, support mechanisms, end-user training, etc.). An architect from ICG stated during a discussion on why we specify requirements: "If you don't spend time to document the requirements, chaos will occur among the downstream stakeholders who must take the requirements and go do something with them!"

Define Requirements Engineering

Once a team shares a "clear and common understanding" of the concept of "requirements", we take a broader view and discuss: What is "Requirements Engineering"? Again a definition from Karl Wiegers [2]:

> **"Requirements Engineering:** The domain that encompasses all project life cycle activities associated with understanding a product's necessary capabilities and attributes. Includes requirements development and requirements management. A sub discipline of system engineering and software engineering."

At Intel, we define "Requirements Engineering" as:

> The systematic and repeatable use of techniques for discovering, documenting, and maintaining a set of requirements for a platform, product or service.

Typically, Requirements Engineering is broken down into five major activities:

Elicitation

Elicitation is the process used to draw out requirements from stakeholders. Techniques include interviews, focus groups, surveys, participant observation, reviewing competing products, and requirements workshops.

"Elicitation" is not to be confused with "solicitation", which is usually defined as an oral or written persuasion to join in some endeavor or buy some product or service.

Analysis & Validation

Analysis and validation activities involve spending time to assess, negotiate and confirm with key stakeholders the right requirements have been obtained during the elicitation activities. Analysis and validation activities are closely connected and many times are done simultaneously with elicitation.

Specification

This is the work of actually writing down the requirements! Organizations without an understanding of the discipline of requirements engineering often think this is where the requirements process begins. Specification is the process of documenting the requirements – elicited from and validated by the stakeholders - in a form that can be shared and managed easily. This process is very iterative in nature. At Intel, we *strongly recommend* the use of Planguage to specify requirements invaluable to create measurable, concise requirements and design statements.

Verification

Getting the requirements right – we ask the team, "Are the requirements specified correctly?" Note that this differs from saying *we've written the right requirements down*. Verification is the process of reviewing the requirements document to determine if the requirements are clear, concise, feasible, unambiguous, and measurable. This activity is focused on assessing the requirements quality level.

Requirements Management

Requirements management includes all the activities to maintain the accuracy of the requirements during the entire life of the project. The emphasis is on controlling the changes to the requirements to help manage scope creep and to ensure the product is built as specified.

Requirements engineering has been practiced for decades, in many industries to provide a framework for successful product development. These methods can be used for all levels of planning, communicating project deliverables, as well as for the specification of more detailed requirements.

Types of Requirements

Before we can dive into writing better requirements, we need to describe in more detail the types of requirements as well as the importance of each type. Routinely, requirements are broken down into three categories:

- Functional
- Non-Functional
- Constraints

Functional Requirements

A functional requirement is defined as "What the product must do". A functional requirement states the required functionality the product shall have. Functional requirements are important because they are typically binary; they either function as specified or they don't.

Within Intel, we train requirements authors to break out a functional requirement into the following simple syntax:

Trigger/Actor/Action/Condition

Example: When an Order is shipped, the system shall create an invoice unless the Order Terms are "Prepaid".

Applying the simple syntax, let's break out this functional requirement so it is easier to see the requirement:

Trigger: When an order is shipped

Actor: the system

Action: create an invoice

Condition: unless the Order Terms are "Prepaid".

When teaching this technique, we first focus on explaining the Actor is the "who", which is frequently the system being developed and the Action is the "what" we want the system to do. Both must be present for a functional requirement. When the actor is implicit, the resulting requirement is passive and ambiguous. For example, if a requirement included the phrase "…invoices are produced…" it would be unclear whether the system, the user, or some combination of the two produces the invoices.

The Trigger is the statement of "When" the requirement happens, and the Condition is a statement of exceptions to that rule. Both Trigger and Condition are optional and aren't always needed. Missing triggers is a common problem, so we spend time asking questions such as "When do you want the system to display the cost of the total order? If the answer is "all the time", then the trigger is most likely not needed. We then discuss "Are there any instances when you don't want the system to display the total cost?" If no, then the Condition is not needed either.

Some requirements authors prefer to combine the Trigger and Condition. This is acceptable. Placing them together is fine so long as the resulting requirement is easy to understand. Be careful when combining time-based triggers (when, before, etc.) with logic-based conditions (unless, except) since this can make the requirement ambiguous.

Functional requirements are typically the easiest for stakeholders and project team members to define, and for test organizations to validate. Either a requested feature is present, or it is not. However, a feature may be present, but still not meet the stakeholder's needs, because related qualitative requirements were never captured along with the requirement for the feature itself. Non-functional requirements allow us to capture those "fuzzier", non-binary requirements.

Non-functional Requirements

Non-functional requirements describe everything *except* functionality. These requirements express the qualitative attributes of the system. A few examples:

- Reliability
- Maintainability
- Usability
- Security
- Performance
- Legal
- User Interface
- Documentation

Later in this paper we will discuss how Planguage [3] provides a standard format and vocabulary for each non-functional statement to make even qualitative statements clear, concise and measurable.

Constraints

Constraints are intentionally imposed restrictions. They are items that restrict the product's design and/or development options. Constraints can come from a variety of sources, such as schedule deadlines, headcount or financial resources, technical limitations, and expectations for return on investment. These are items you simply must live with. For example: "The system will operate using [XYZ] 8.0 operating system."

Methodology

Planguage is a keyword-driven syntax created by Tom Gilb. At Intel, Planguage is strongly recommended as the foundation for both functional and non-functional requirements and is now used by most product development groups. Planguage provides a standard format and vocabulary for each requirement statement. This helps prevent omissions (it's easy to spot a blank line after a Planguage keyword), and promotes requirements reuse.

Teaching requirements authors to use Planguage as a basis for their requirements writing has been a fun experience at Intel. We have engaged with hundreds of product teams and trained over 5,000 people in the past 5 years.

Instructing this technique begins by defining what a functional requirement is and a review of the simple syntax: (Trigger/Actor/Action/Condition). Planguage is then introduced as a list of basic key words and definitions as outlined in Figure 1.

Key Word	Definition
Tag	A unique, persistent identifier
Source	Origin of the requirement (Group, Person, Standard, etc.)
Requirement	The text that details the requirement. (We recommend using the simple syntax Trigger/Actor/Action/Condition)
Priority	A statement of priority and claim on resources
Key Stakeholders	Parties materially affected by the requirement
Rationale	The reasoning that justifies the requirement
Assumptions	All assumptions or assertions that could cause problems if untrue now or later
Risks	Anything that could cause malfunction, delay, or other negative impacts on expected results
Defined	The definition of a term (We recommend a project or organization-level glossary)

Figure 1

We then introduce a few symbols that assist authors when writing requirements:

- Fuzzy concepts requiring more details are identified: <*fuzzy concept*>.
  - This allows authors to identify any concept or word that doesn't have a clear definition to revisit and refine later.
  - For example, "Average time for <novices> to become <proficient> at a defined set of tasks."
- Qualifiers are used to modify other keywords: [*when, which,* …].
  - Qualifiers are very powerful, and often under-used by newcomers to Planguage. Qualifiers add specificity and clarity to Planguage statements, and permit parameterization of statements.
  - For example, "Priority: [*Phase1*] High, if not implemented in Phase 1 will reduce program ROI by 25%."
- A collection of objects is used to show more than one item: {*item1, item2,* …}.
  - For example, "The system shall support high definition Ethernet connectivity." {*Customer1, Customer2, and Customer3*}.
- The source for information. We use an arrow to indicate where the information came from: Statement ← Source.
  - For example, "Calculations will be captured during UI testing". ← S. Smith End-User Design Engineer.

Functional Requirement

Teams in training begin by writing functional requirements with their instructor/coach. Below is an example of a functional requirement with the minimum set of key words:

Tag	Order.CreateInvoice
Source	Jay Smith, Marketing Manager
Requirement	When an Order is shipped, the system shall create an invoice unless the Order Terms are "Prepaid".
Priority	High. If not implemented, it will cause business process reengineering and reduce program Return On Investment (ROI) by $400K per year
Rationale	Task automation decreases error rate, reduces effort per order. Meets corporate business principle for accounts receivable.

Most teams find they want additional and sometimes system, or project-specific key words to help provide clarity. One team within ICG felt it was important to extend Planguage key words to help capture more information on the key stakeholders (to identify who would be unhappy if a specific requirement was not implemented), the market segment for which the product was targeted, the customers asking for the requirement, and what the competition was doing in regard to that requirement.

Example below:

Tag	Gigabit.Ethernet
Source	Sandra Johnson, Product Marketing Engineer
Requirement	The system shall support extended Gigabit Ethernet connectivity.
Priority	High, if not implemented will lose 45% of projected revenue.
Rationale	Improved performance and support 1600 Mbps.
Key Stakeholder	L. Smith, Architecture Engineering
Market Segments	Modular Router LAN Access Security Networking Gateways
Customers	Customer1, Customer2, and Customer3
Competition	Most competitors are lagging right now or provide extended connectivity in their high-end products. Implementing this feature provides a competitive advantage resulting in over $1.5M in revenue.

Functional requirements written using Planguage are more concise because we only use the key words that add value. Also, Planguage provides better clarity around what the product must do because the information is easy to find and to understand.

Non-functional Requirements

After the requirements authors have experienced how easy it is to write functional requirements using Planguage, we switch gears and talk about the non-functional requirements. Non-functional requirements are more difficult for most requirements authors to specify. Before teams learn Planguage, the non-functional requirements sections are either left blank (hoping someone else will write them) or the non-functional requirements captured are not measurable.

When we write a non-functional requirement, we specify a few unique items not found in a functional requirement. *Scale*, *Meter*, and the measurement values (*Minimum, Target, Outstanding and Wish*) are generally stated only for these qualitative statements. See Figure 2 for the key words and definitions tailored for use at Intel.

Key Word	Definition
Ambition	A short description of the goal
Scale	The scale of measure used to quantify the requirement
Meter	The process or device used to establish location on a scale
Minimum	The minimum level required to avoid political, financial or other type of failure
Target	The level at which good success can be claimed
Outstanding	A stretch goal if everything goes perfectly
Wish	A desired level of achievement that may not be attainable through available means

Past	An expression of previous results for comparison
Trend	A historical range or extrapolation of data
Record	The best known achievement

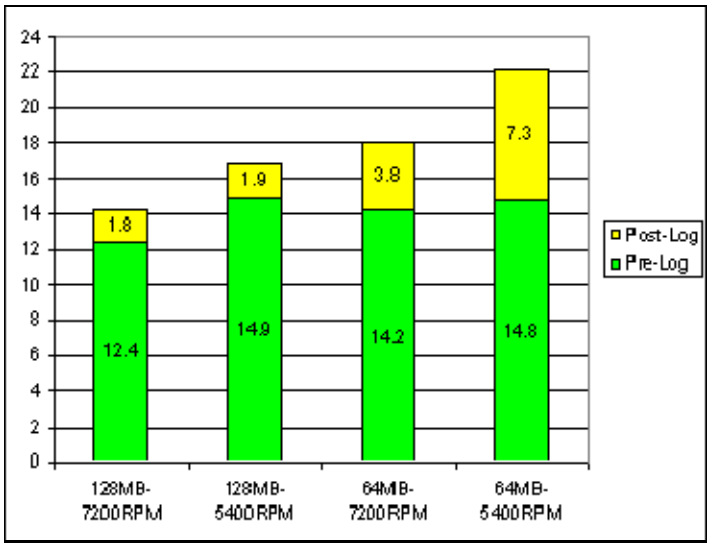
Figure 2

As with functional requirements, teams find the appropriate mix of Planguage keywords for their particular circumstance. All teams are coached to only use the keywords that add value to the requirement. When writing non-functional requirements we recommend at minimum, teams use *Ambition*, *Scale, Meter, Minimum, Target*, and *Outstanding* (and *Past*, for a follow-on product or release). As mentioned earlier, customization is encouraged, as long as the definitions for each keyword are clearly understood by the team members.

When training new requirements authors, we spend time explaining how to re-write a poorly written non-functional requirement such as "The system must be fast" into a clear, concise, measurable statement. Notice the formatting of all the Targets (*Outstanding, Target, and Minimum*) onto one line to make it easier to understand and save space:

Tag	Throughput.Performance		
Source	Company 1, S. Smythe, Product Marketing Manager Company 2, J. Johnson, Planning Manager Company 3, P. Anderson, Architecture		
Priority	High, if not implemented will reduce projected revenue by 66.8%, equating to over $5.8M dollars.		
Rationale	In order to compete with 10 GbE based solutions available today and compete in the market for the next 36 months, we must deliver the same level of throughput performance but at a much lower price.		
Ambition	Deliver a 10 GbE product with the best throughput performance.		
Scale	Gbps		
Meter	The performance measurement configuration must have Penguin CPU, Whaleback chipset and run in Porpoise Gen 2x4 slot. The testing traffic pattern will have standard Ethernet1500B and Input size of 128KB.		
Targets	**Outstanding:** 7.0 Gbps	**Target:** 6.5 Gbps	**Minimum:** 5.0 Gbps
Risks	CPU processing power bandwidth will potentially limit the performance when more than six 10 GbE ports are functioning.		
Key Stakeholders	Company 1, J. Martinez, Embedded Engineering Company 2, L. Block, Planning and Architecture Company 3, P. Anderson, Architecture		

Below is another example of a well written, clear, concise, and measurable non-functional requirement, notice the use of an embedded chart:

Tag	Windows*.BootTime
Ambition	Minimize the time needed before a user is able to use the system when it transitions from mechanical off to working state.
Rationale	While the number of system cold starts should be reduced by many features, boot time continues to be an expressed user frustration.
Priority	High. End users are concerned with boot time.
Scale	Elapsed time from depressing the power button to display of desktop icons ← Windows* Platform Design Note "Fast System Startup for PCs Running Windows XP*"
Meter	Hand timing of boot sequence for a representative configuration under the following conditions: 1. Boot trace disabled. 2. Windows XP* single partitioned hard drive 3. No network cables connected.
Note	Timing must exclude any latency introduced while the tester selects an account at the XP account screen.
Minimum	Under 20 seconds
Target	Under 15 seconds
Outstanding	Under 10 seconds
Note	High-end PCs with high capacity and multi-platter hard drives. Far Eastern language operating system may take longer.
Past	[2005, Company1, 533MHz, Windows XP* Home edition, US version]  ← Windows* Platform Design Note "Fast System Startup for PCs Running Windows XP*"
Record	N/A
Trend	[2000->XP] Approximately 8-10x faster <- Windows* Platform Design Note

	"Fast System Startup for PCs Running Windows XP*"
Owner	B. Smith
Source	User Centered Design Group, B. Smith
References	Windows* Platform Design Note "Fast System Startup for PCs Running Windows XP*" Advanced Configuration and Power Interface (ACPI) specification, available at http://www.acpi.info/default.htm
Key Stakeholders	Company 1, S. Chase, Design Engineering Company 2, R. Elbert, Windows* Architecture Team Lead Company 3, B. Smith, User Centered Design Group
Rev., date, author	Rev.0.1, 7/25/03, A. Jones – original Rev. 0.2, 10/7/03, E. Simmons – added details and edited language of the original.

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

As you can see from the examples above, Planguage allows requirements authors to take a lot of information and condense it to create a clear, concise and measurable statement.

- **Clear:** Information about the requirement is easy to find and understand.
- **Concise:** Only use the Planguage key words that add value, no more! All rationale, examples, and other supporting data are separated from the requirement statement. The requirement is expressed using the simplest grammar and as few words as possible.
- **Measurable:** The *Scale* and *Meter* provide information of what and how the requirement is to be measured. The Intel customized key words *Minimum*, *Target*, *Outstanding* and *Wish* identify the target values to measure success.

Key Lessons Learned

In late November, 2004, we began collecting data from the requirements authors to measure the effectiveness of this training. The objective of the requirements engineering initiative was to focus on training, mentoring and reviewing requirements to enhance ICG's ability to gather, write and manage requirements.

So how did we do?

In total, we engaged with 26 teams who were in the requirements phase of their respective projects. All the requirements teams attended a Writing Good Requirements full-day workshop. We conducted a retrospective (Lessons Learned) survey with the teams in November 2004 asking the teams what went well. Feedback indicated that the focus on requirements resulted in:

- A more complete set of requirements documents
- Increased requirements re-use
- Reduced ambiguity which resulted in clearer, more precise understanding of the requirements
- Better priority management
- Easier decision making

One question was asked of every team: **"What went well while writing your requirements using Planguage?"** The top three answers:

- **Timing of the requirements writing workshop was excellent**
  - We engaged with the teams as soon as the requirements authors were identified. Many times the requirements workshop piggybacked on their first "kick off" face to face meeting.
  - Many of the teams had not yet begun to write their detailed requirements, so they were hungry for a process. They liked using Planguage to ensure they had all the information needed to build and test their products.
    - Teams that were farther into writing their requirements when they were intercepted by the Writing Good Requirements training were more reluctant to go back and re-write their requirements using Planguage. For these teams we compromised and encouraged them to only go back and apply Planguage syntax to requirements that were new or risky.

- **Intact teams made learning and using Planguage easier**
  - The workshops were delivered to the whole team at once, which allowed the requirements authors, owners, and contributors to be in one room together.
  - Participants left the training with a common syntax and vocabulary.
  - As a team, they decided on which Planguage key-words were going to be used.
    - Many teams extended Planguage as needed with new keywords. We spent time to help define new keywords carefully so they would be used consistently.

- **Using their requirements document rather than a "Mocked up" example made it real for them**
  - Requirements authors left the writing workshop feeling they didn't spend a day away from their project.
    - We focused the training on writing actual requirements for their project using the Planguage templates, adding or removing keywords if necessary.
    - Many of the teams found they were able to re-use a lot of their non-functional requirements for subsequent projects. The time needed to write their next requirements specification was considerably less, because all they had to do was change the targets and update any information within the key words.

When we asked the teams, **"As a result of the requirements writing workshop, what are you going to do differently?"** here are some of the responses:

- "I need to record the *SOURCE* for all requirements so the team can go back and ask better questions." (Product Marketing Engineer)
- "I now understand and will use the Good Requirements Checklist to ensure our requirements have all the attributes - concise, unambiguous, and measurable." (Architect)
- "Will try the new writing techniques (Planguage) to help me write measurable requirements." (Engineering Manager)
- "Capture feedback from requirements team earlier into our document so we have more time to react to their feedback." (Product Marketing Engineer)
- "My team will be increasing the level of effort put into creating a clear, concise, and measurable document because we now understand how important they are to the success of the project." (Project Manager)

Conclusions

Specifying requirements so they are clear, concise and measurable is possible, and even enjoyable! Working with product development teams at Intel, we have adopted a tailored version of Planguage that provides many benefits when writing functional and non-functional requirements:

- **Decreased time to market:**

  Using Planguage to specify requirements results in decreased time to market by:

  - Clearly defining what the product must accomplish (functional and non-functional requirements)
  - Clearly communicating what will be produced
  - Validating correctness prior to additional work

  This method of specification resulted in less re-work and fewer mistakes in product development.

- **Improved customer satisfaction**

  Capturing the Voice of the Customer (VOC) improves customer satisfaction. Planguage provides the basis for developing products that meet the customer's needs and expectations by documenting rationale, assumptions, and priority with the customer in mind.

- **Reduced risks**

  One not-so-obvious benefit of using Planguage to specify requirements has been better planning and risk identification. Clear, concise and measurable requirements statements defined earlier in the project result in products being identified more easily, reducing financial risk and allowing valuable resources to be focused on viable products.

We help requirement teams at Intel write clear, concise, and measurable requirements using an informal, but structured keyword-driven syntax, Planguage. The benefits of using Planguage include decreased time to market, improved customer satisfaction and reduced risks.

If YOU want clear, concise and measurable requirements, try using Planguage on your next project!

Acknowledgements

I would like to acknowledge two co-workers, Erik Simmons and Sarah Gregory. They both spent numerous hours sharing their wisdom gained while teaching and using Planguage. They truly enhanced the quality of this paper. Without their help this paper would not have been published. Many thanks!

References:

[1] IEEE Standard Glossary (1997)

[2] Karl Wiegers' website http://www.processimpact.com

[3] *Competitive Engineering*, Tom Gilb, Elsevier 2005

Improving Requirements Traceability Effectiveness

Les Grove, Perry Hunter, Doug Reynolds
Tektronix, Inc.
P.0. Box 500 M/S 39-548
Beaverton, OR 97077

les.grove@tek.com; perry.w.hunter@tek.com; douglas.f.reynolds@tek.com

Abstract

Requirements traceability is an essential component of an effective requirements management process. Having well written requirements is not enough if they can not be used and effectively managed through the project lifecycle. Requirements traceability and the use of traceability matrices are one way to follow requirements forward and backward from a customer needs assessment through engineering and design.

In this paper and presentation we will describe different levels of requirements traceability and the problems, hurdles, and experiences of trying to perform requirements traceability by hand through the use of word processors and spreadsheets. We will provide a list of the minimal requirements to perform requirements traceability and we will highlight the benefits of performing requirements traceability.

Requirements traceability will then be presented in the context of requirements engineering tools and how these tools can provide more in-depth answers to questions along with being integrated with other tools. A summary will be provided on how requirements traceability and requirements engineering tools can provide the ability to trace, integrate, and share project information. A description of how these tools can also help identify areas of concern that can lead to root cause analysis and training will be provided.

Biographies

Les Grove is a software engineer at Tektronix, Inc., in Beaverton, Oregon. Les has worked at Tektronix for nine years and has 20 years of experience in software development, testing, and process improvement. He holds a Bachelors of Science in Computer Science from California Polytechnic State University, San Luis Obispo and has a Masters degree in Software Engineering from the University of Oregon through the Oregon Masters of Software Engineering program.

Perry Hunter is a Software Quality Engineer at Tektronix, Inc., in Beaverton Oregon. Perry has worked at Tektronix for four years, and has a total of almost 20 years of experience in software development, software quality and project management. He holds a Bachelors of Science in Oceanography from Humboldt State University (California).

Doug Reynolds is a Software Quality Engineer at Tektronix Inc., in Beaverton Oregon. Doug has worked for Tektronix for the past 13 years and holds a Master of Computer Science and Engineering from Oregon Health & Science University (OHSU) and a Bachelors of Science in Electrical Engineering Technology from Oregon Institute of Technology.

Introduction

Over the past several years, projects at Tektronix have attempted to trace all software requirements to the corresponding software test cases. Some of these projects realized the quality benefits from these traceability matrices and reduced effort needed for follow-on products, but the maintenance effort for the matrices turned out to be a full time job.

Additionally, a Tektronix committee to improve requirements practices set goals of developing requirements traceability matrices that capture relationships from original customer need, to product requirements, to engineering requirements, to downstream work products and test cases. A far reaching set of goals to improve how requirements were managed at Tektronix were established:

- Capture the bi-directional relationships between requirements and work products
- Identify inconsistencies between the requirements and work products, making corrections when inconsistencies are discovered
- Manage all changes to the requirements and downstream changes to work products

Creating a comprehensive requirements traceability solution was a key part of attaining these goals and it was clear that achieving this type of traceability would be extremely difficult with our current tool set of word processors and spreadsheet applications. The committee also knew that selling these more complex traceability goals would be difficult as the effort would be highly administrative in nature and not the kind of work that software engineers prefer to do. A tool was needed.

Based on these goals, the committee began a search for a tool to help manage requirements better, with specific capabilities around traceability. After finding and evaluating 18 tools on the market, three were found to meet all of the committee's requirements. Those three tools were brought on-site for a detailed evaluation and eventual selection to be included in an enterprise-wide solution. Today, the tool is being rolled out throughout the company.

This paper communicates what we have found to be the best methods to perform requirements traceability and the benefits of using a tool to perform requirements traceability.

1. Essentials of Traceability

Through the years we performed some requirements traceability to help us reduce the amount of rework, identify the impacts of changing requirements, and to demonstrate coverage. In addition, what has been voluntary in the past is now mandated that through our development policy: "Every project shall demonstrate traceability between its requirements and its final work products". The experience that we have obtained through performing requirements traceability has enabled us to identify what we think is essential requirements practices that must be in place in order to achieve traceability.

Each written requirement must be an independent statement that can stand on its own *and* must have a persistent label or identifier. Once each requirement has met this criteria, links can be created between each requirement statement and other work products (customer needs sources, other requirements specifications, design components, test plans, test cases, etc.). Of course, any linked work products also need to be uniquely identifiable. From a requirements traceability standpoint, it's also very helpful that the specification itself is well-written and organized.

An example of a fictitious car radio will be used to demonstrate the essentials of requirements traceability (see Table 1). Each requirement has a unique persistent label which can be used for

traceability. Also, each requirement is written so it can stand on its own, not requiring more detail to be understood out of the context of it's preceding requirements.

Label	Requirement
CR 1	*The car radio On/Off button shall not function when the car is turned off.*
CR 2	*The car radio shall provide up to and including six pre-selected channel settings.*
CR 3	*The car radio shall permit manual volume adjustment between 0 and 95 dBA (inclusive) in 1 dBA increments.*
CR 4	*The clock settings (hours and minutes) on the car radio shall be adjustable whether the car radio is on or off.*

Table 1 – Car Radio Requirements

[1]For more information about writing good requirements see section 3 of the paper "Control Your Requirements Before They Control You", PNSQC, 2003

In the earliest stages of implementing Requirements Traceability, requirements were defined in simple MS-Word documents, identified by section heading numbers and traced to test cases via an Excel spreadsheet. Here is an example requirement (somewhat edited):

> 4.1.2.1 Operational Requirements
>
> The following existing pre-defined standard rates will be available as an enumerated selection from both the GUI and PI for the 80XYZ module:
>
> - OC3/STM1 155.52 Mb/s
> - OC12/STM4 622.08 Mb/s
> - FibreChannel: 1.063 Gb/s
> - Gigabit Ethernet: 1.250 Gb/s
>
> In addition, the following rates must be supported for *Option NDA*, with an enumerated selection from both the GUI and PI:
>
> - OC192/STM64 9.953 Gb/s
> - 10Gbase-W 9.953 Gb/s
> - 10Gbase-R 10.31 Gb/s
> - 10G FibreChannel 10.51 Gb/s

From a developer's point of view, it gives clear and concise information about a series of capabilities the software needs to provide. However, when a matrix was created to establish linkage to test cases, the following problems became obvious:

1) There are really many requirements in this single entry which might have to be implemented by several different people. How can progress on 4.1.2.1 be tracked? Who is responsible for implementing this?
2) A single test case cannot cover all these elements, as failure of any single element would fail the entire test case. Are all these items essential? What if we failed to implement only one, would we ship the product?

3) It is not granular – changes to a single element in the list might be missed by those who need the information. What if we decided the FibreChannel value should really be 1.0625 gb/s – will the developers catch that without having to have a specific discussion? Will the testers catch the change and update the test case?

4) The document does not use unique identifiers for each requirement but instead relies on auto numbering. When someone inserts a new section, the sections below are all re-enumerated. Because of this, changes are difficult to track, linkage to specific test cases is all but impossible to manage. What if developer X was assigned to implement this requirement, and later it changed to be 4.2.2.1 – would they be able to follow that change and not miss adding the new feature that disrupted the enumeration scheme?

As it turned out, all of these pitfalls were realized at one point or another in the project. The team's initial enthusiasm for implementing traceability was diminished very quickly and it was perceived as another layer of bureaucracy they had to fight to get their jobs done.

At a basic level – it worked. If the team was willing to exercise discipline, avoid breaking the linkages, analyze all of the changes instead of just the ones they thought they were responsible for, the effort did yield the ability to identify things not implemented, work needing to be done and provide evidence that test coverage was adequate. It's better than nothing, but it's a very labor-intensive and frustrating thing to do in a project with more than a few people and requirements that change more than a very few times.

2. Benefits of Requirements Traceability

Performing requirements traceability provides several benefits during project development: project members can more fully assess the impact of change, gain confidence in meeting project objectives, improve accountability to stakeholders, track progress, and reduce costs. Through measuring implemented and verified requirements, a project leader can help alleviate the problem of projects being perpetually "90% done".

It is a given, in our world, that there will be changes to customer needs and a project's requirements. The ability to assess the impacts of these changes can be very powerful to project teams as the changes to downstream work products can be analyzed and understood. Additionally, the effects of modifying or removing requirements on upstream requirements can be evaluated. This can lead to further analysis and understanding of the effects of postponing requirements to meet schedule or removing a requirement due to technical challenges, etc. With this technique we can analyze each requirement individually seeing the far reaching effects of each change before committing to making any changes.

As a product driven company, Tektronix not only needs to ensure that each product meets customer needs, but also to deliver the product to the market in a timely manner. Requirements traceability helps insure that both of these project objectives are met. Traceability not only shows how the requirements are being satisfied, it also can help identify requirements with no customer need (gold plating), or determining if the system is adequately tested.

When each requirement is tracked, traceability can be used to determine the percentage of the product that is operational. Through missing links we can identify the work products that have not been accounted for, as well as determine what percentage of requirements are successfully verified. These are just some of the measurements that we can easily make using traceability.

3. Basic Requirements Traceability

A simple definition of traceability is the definition provided by Wiegers[2] which states that traceability is a logical link between one system element and another. Traceability was instituted by Abbott Laboratories in 1987 where they coined the phrase, "You can't manage what you can't trace."[5] These elements can be use cases, requirements, design components, code modules, test cases, or any other work product. As there are dependencies between different sets of elements, traceability provides a mechanism to find and follow those dependencies as a project progresses. For now, we will focus on creating this linkage between requirements and a specific set of downstream work products – something we call *basic requirements traceability* (see Figure 1).

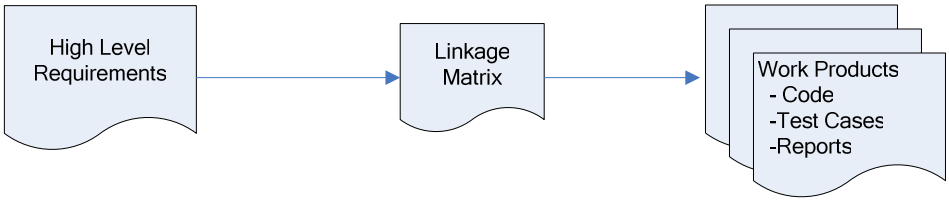


Figure 1, Basic Requirements Traceability

An example of basic requirements traceability is shown in Figure 2 providing a cross-reference between the requirements and their associated test cases – this should be a minimum for any project with more than one requirement. Creating a traceability matrix can be a tedious job, but we have found the benefits outweigh the risks of finding a missing or untested requirement late in the project. Building and maintaining a traceability matrix should be part of regular, productive work, rather than a task imposed near the end of a project or program in order to satisfy a quality certification audit. Other matrices can be created as appropriate: use cases to requirements, to design components, to code modules, to test plans, to documentation, etc. Unfortunately, linkages requiring two matrices to determine the dependencies are only implicit and are hard to follow.

By analyzing the requirements traceability in Figure 2 it can be easy to pick out the following three issues:

1) Car Requirement CR 3 is not being tested.
2) Test Case #2 may not be needed since the requirement CR 4 is already being tested by Test Case #1.
3) Test Case #3 may not be needed since it does not cover any requirements.

Test Cases / Requirements	Test Case #1	Test Case #2	Test Case #3	Test Case #4
CR 1	✓			✓
CR 2				✓
CR 3				
CR 4	✓	✓		

Figure 2, Example Traceability Matrix

There are shortcomings to basic requirements traceability:

- Matrices are typically only 2-dimensional – requiring many matrices to be fully linked
- Relationships can be many-to-many – difficult to see in two dimensions
- Matrices become very large with too many empty cells

These are complicated by the fact that requirements change and that it's difficult to understand the full impact of a change using simple traceability matrices. It's also difficult for downstream users to be aware of all changes as changes to requirements are not signified in the matrices, requiring a good deal of word-of-mouth or continuous email communications. It also takes a great deal of time to maintain one matrix, let alone several, and it's very easy to make mistakes.

The full impact of a changed requirement is often not immediately understood. With traceability, a changed requirement can at least be traced through its links to provide a context for the change. For example, in Figure 2 if requirement CR 4 needed to change, we would know that test cases 1 and 2 may also need to change. Without requirements traceability it can be hard to keep up with changes as changes to affected work products would need to be somehow communicated. In the ideal situation, once requirements are baselined there are no changes; but in reality requirements are continuously being refined.

In the past, we learned that even though team members were interested in requirements traceability, only a few actually went through the effort. These team members usually only linked the work products that they are interested in and did not take on the responsibility of linking all the project work products. Therefore, only two or three work products were ever linked and rarely were linkages complete. An example would be a software engineer who links the functional requirements specification to the product requirements or a software quality engineer who links the software test cases to the software requirements. Because only a few work products are actually linked it is hard to know all the effects of the upstream and downstream changes. Work products from one stage were typically not synchronized to the next stage. The context of a lower-level requirement (such as a test case) as it related to a customer need was missing.

Requirements traceability techniques without dedicated requirements engineering tools are typically only two dimensional. These types of traceability matrices have a many-to-many relationship and can be very sparse. To demonstrate the how the size of a multi-dimensional traceability matrix can become unmanageable on one project we had over 250 customer needs, 2800 software requirements, and 900 test cases. This would have provided for a traceability matrix of over 630 Million cells!

Performing requirements traceability without tools has proven to cause additional problems:

- It is difficult, and for anything other than a very small project, almost impossible to keep up with changes (this just can not be stressed enough for those who have not been through it).
- It allows for more frequent and more expensive mistakes and provides for missed links (during the porting of our software requirements and software test cases into a tool we found that we were missing greater than 5% of the links to requirements!). This means that we missed testing greater than 5% of the requirements.

Without tools we started out by writing requirements that were grouped in paragraphs. The requirements were not as granular or persistent as the document changed. As the requirements document changed so did any traceability links that we were using (i.e. section numbers, paragraph numbers etc). We then created Microsoft Word templates for developing our requirements specifications. Though this had its advantages (as we wrote standalone requirements that used unique and persistent labels using VBA macros) it also had its own set of problems.

Requirements traceability using hyperlinked documents (like Microsoft Word):

Hyperlinked documents have been used to perform requirements traceability. We have tried using hyperlinked documents but gave up because of negative side effects. Two side effects of using hyper-linked documents are:

- Requirements traceability and analysis requires that you physically traverse the links
- Working with living documents quickly causes hyperlinks to become broken or deleted. The only way to verify that there are no broken links is to traverse the work products.

Requirements traceability using auto-numbering in documents (like Microsoft Word):

Word processors have been used for the development of requirements documents. They provide some benefits like formatting, spell checking, etc. For a period of time we used the 'auto numbering' feature of word processors as a unique requirement number. The problem that we found that was over time as requirements were added and deleted these numbers would change and we would lose our traceability links between work products.

Requirements traceability using a spreadsheet:

Spreadsheets have been used for a long time to perform requirements traceability. We used spreadsheets at one time to help maintain our basic requirements traceability. The problems with using spreadsheets are:

- As the number of requirements increases both axes become very large. It's very easy to run out of columns as the limit is usually 256.
- Relationships tend to be a sparse leaving a large number of empty cells.
- Hard to work through multiple layers of traceability
- Traceability information is often separated from the details of the requirements

From these lessons learned over the years we have become better at writing requirements as well as developing techniques for tracing requirements. Instead of writing and embedding requirements into paragraphs we have become better at writing independent standalone requirements. Instead of tracing requirements by hand we have worked on developing techniques for tracing requirements from hyperlinked documents to spreadsheets. In all, this has only prepared us to take the next step.

4. Advanced Requirements Traceability

The concept of *basic requirements traceability*, linking two system elements, seems inadequate for more complex projects where there are multiple relationships between work products needing to be managed. At Tektronix, we had a need to create connections between original customer needs, product requirements, engineering requirements, and system test cases. A series of two-dimensional links would demand too much overhead to create and maintain. So we investigated a better, what we are deeming advanced, concept for requirements traceability "The ability to follow the life of a requirement, in both forwards and backwards directions"[7], i.e. from its origin, through its specification and development, and through periods of ongoing refinement and iteration in any of these phases.

Advanced traceability starts from the source of each requirement and links through all work products both forward and backwards. Forward traceability identifies the work products and deliverables that satisfy each requirement; providing assurance that each requirement is fulfilled. Backward traceability identifies the source of each requirement (typically, customer need).

When advanced traceability is performed the following can be achieved:

- Inconsistencies between requirements and work products can be identified and corrections can be made.
- Full impacts of requirements changes may become better understood.
- Customer needs can be referenced prior to changing/removing requirements.
- The product can be proved to have met all requirements.
- The status of a project can be better determined. Missing links can indicate work products that have not been delivered.
- Code reuse can be facilitated. When requirements are reused in future products, the associated work products are identified.
- Test cases can similarly be reused. This is especially valuable when working with test automation, if the test cases (and by inference the originating requirements) are granular enough, tests covering similar features in different products are easy to develop.

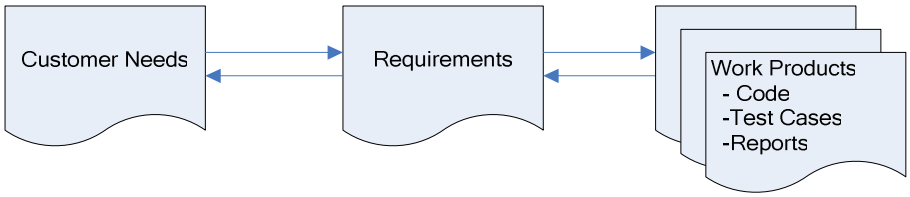


Figure 3 Advanced Requirements Traceability

We have found that by following the life of the requirement in both the forward and backwards directions we are able to realize added benefits of requirements traceability. Through different projects we have been able to:

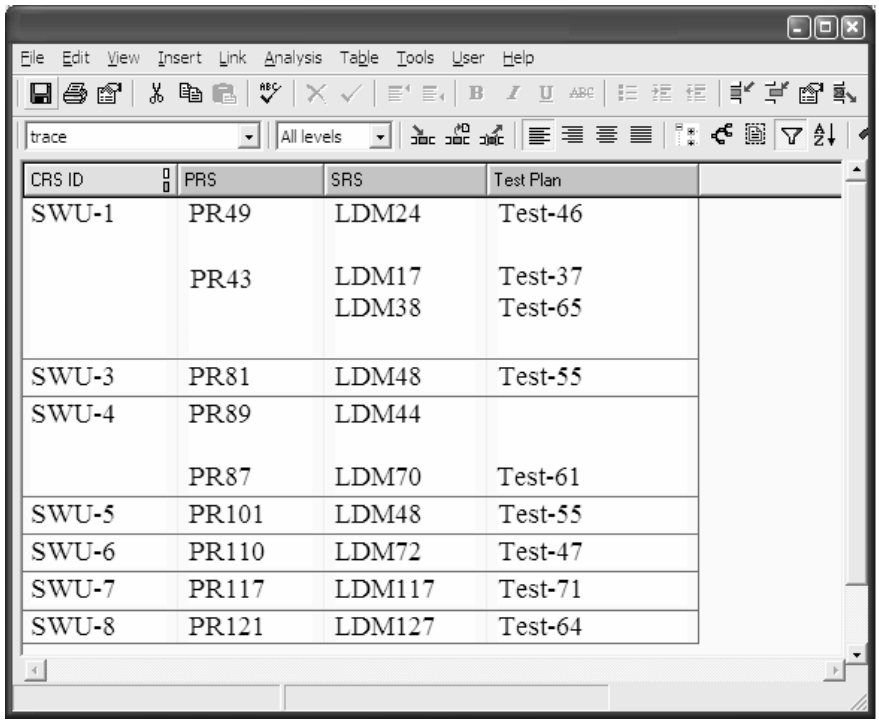
- Make better informed decisions to change or not to change the behavior of an instrument. Through a more complete impact analysis, we can see how a change of a product requirement affects the down stream work products.
- Track real completion of work activities. We found that some projects appeared to be on track because work activities were being completed. By analyzing the amount of traced requirements that had been verified we knew that the requirements were not fully implemented even though scheduled tasks were marked as completed. By using this information we have been able to justify additional resources to complete the projects on time.
- Reuse over 90% of the test cases on a project - only adding test cases to cover new behaviors. Without the traceability this effort would have taken several months to re-develop a new set of test plans and test cases.

5. Requirements Engineering Tools

If advanced traceability is a goal, it becomes too cost prohibitive to do without a tool specifically designed to manage the complex set of relationships. As mentioned earlier, common document and spreadsheet tools are inadequate for basic traceability work; but to achieve advanced traceability, it's downright impossible. The benefits demonstrated above resulted from using a tool to achieve advanced traceability.

Tools help with the essentials of requirements traceability by automatically creating unique identifiers and separating specific requirements from other background information that is sometimes needed in a specification, but are not traceable. No longer do we need to worry about messing up section numbers when adding, moving, or deleting a requirement.

Using requirements engineering tools makes basic traceability easier to do than using spreadsheets. Elements are logically linked and those linkages can be analyzed for unfulfilled requirements or gold-plated software. But unlike manual traceability, it's no extra effort to create advanced traceability as all the linkages are tracked and can be followed in both directions (see Figure 4).



CRS ID	PRS	SRS	Test Plan
SWU-1	PR49	LDM24	Test-46
	PR43	LDM17	Test-37
		LDM38	Test-65
SWU-3	PR81	LDM48	Test-55
SWU-4	PR89	LDM44	
	PR87	LDM70	Test-61
SWU-5	PR101	LDM48	Test-55
SWU-6	PR110	LDM72	Test-47
SWU-7	PR117	LDM117	Test-71
SWU-8	PR121	LDM127	Test-64

Figure 4 Advanced Traceability with Tool

Not only do requirements engineering tools help trace requirements, they can also track, integrate, and share project information when integrated with other project tools (for example, defect tracking, configuration management, project tracking, and test management).

These tools can also help manage requirements and requirements attributes including prioritization, current status, information about the originator (and/or the source of the requirement), the sponsor, implementation engineer and much more. The tools can also track the cost of development, which version the feature was first released, and other notes.

Since most tools are databases these tools can provide the ability to query the data and answer questions. For example:

- In what order should we test the requirements (if time is short)?
- In what order should we implement the functionality in the system?
- Has everyone been notified of a change to the requirement?
- Where did this requirement come from? Is it essential?
- What can be re-used?
- What changes are needed?
- What do we cut first?

Tools can help identify and track project progress, identify risks, know the status of each requirement, trace defects associated with the implementation of a requirement, and know the number and scope change requests applied to a requirement. Metrics can be collected and stakeholders can now identify problem areas and perform root cause analysis. That information can be fed back into the development process to improve upcoming projects.

6. Conclusions

After some experience of being able to perform advanced requirements traceability using a requirements engineering tool we found:

- Teams became more involved in the maintaining and updating requirements and the requirements traceability links (before it was being done by just a few individuals and we did not have complete project traceability). For anything other than a very small project, it is difficult, if not impossible, to keep up with the changes (this can not be stressed enough for those who have not been through it).
- Project requirements were easier to find making it easier to maintain traceability links. Before the requirements were managed by each organization (marketing, hardware engineering, software engineering, mechanical engineering, etc.) and links were provided to the different requirements documents. By using the tool we found that all the project requirements were managed out of a central location. This provided easier access.
- Project requirements were easier to keep up to date. Before, the software requirements were managed as an electronic document under a configuration management system. To change a requirement required a the document to be checked out, edited, checked in, sent out for review, reedited, sent for approval, checked out for approval, approved, labeled, checked in, etc. A good requirements engineering tool managed this process.
- Test coverage was improved. Before, many requirements were missing links and not being fully tested. Just the sheer number of requirements, tracking the links by hand through spreadsheets, we found that we were not testing all the requirements. On one project we ported the requirements and the traceability into the tool and found that we missed greater than 5% of the links – this means that we missed testing greater than 5% of the requirements.

The overall return on investment hasn't been fully realized yet as the tool is still in the rollout stage and teams are in the stage of adjusting their processes and practices. There is some initial extra effort associated with training and familiarization with a new tool, as well as exporting existing work to the tool. We have found that the effort of creating and maintaining traceability links with the tool has been reduced by three-quarters. But there is also some second-order savings to be realized when teams are better able to connect their work to the original customer needs and will be better able to respond quickly to market changes during development. This will produce a better product, exactly aligned to what the customer wants, when they want it.

7. Glossary

Term	Definition
Advanced Traceability	The ability to follow the life of a requirement, in both forwards and backwards directions [7] i.e. from its origin, through its specification and development, and through periods of ongoing refinement and iteration in any of these phases.
Basic Traceability	A logical link between one system element and another.
Measurement	A quantitative assessment of the degree to which a product or process possesses a given attribute.
Requirement	A specification of what should be implemented; a description of how a system should behave, or of a system property or attribute; capability needed by a user to solve a problem or achieve an objective.
Specification	A document that prescribes, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a product, system, or component.
Stakeholder	A person or organization that depends on the requirements. Stakeholders can be customers, but are usually internal to the company.

8. References

[1] Les Grove, Doug Reynolds, and David Suryan, "*Control Your Requirements Before They Control You", PNSQC 2003*

[2] Karl Wiegers, "Software Requirements", second edition, Microsoft Press, 2003

[3] Blackburn, Scudder, et al, "*Improving Speed and Productivity of Software Development: A Global Survey of Software Developers",* IEEE Transactions on Software Engineering, December 1996 (Vol. 22, No. 12).

[4] Alan M. Davis & Dean A. Leffingwell, "*Making Requirements Management Work for You"*, Crosstalk, April 1999.

[5] Watkins, R., and M. Neal, "*Why and How of Requirements Tracing", IEEE Software*, July, 1994, pp 104-106.

[6] *"Requirements traceability for quality management"*, Compuware Corperation, 2004

[7] Gotel, O. and A. Finkelstein*, "An Analysis of the Requirements Traceability Problem",* Proceedings of the First International Conference on Requirements Engineering, Colorado Springs, CO, April 1994, pp 94-101.

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

[9] Conner, C. and Callejo, L., *Requirements Management Practices for Developers*, Rational Software White Paper, 2002

Introducing the foundations of continuous software process improvement

Opal Perry and Laurie Burgher

Abstract

Many people are delighted and inspired when exposed to theoretical or academic models for software process and engineering practices yet “hit the wall” when they try to implement advanced practices before laying a foundation for software process improvement.

In 2004 our organization undertook a process improvement initiative, which represented the first phase of a total software development lifecycle quality improvement initiative. During the course of this initiative, we learned that deploying a solid, common sense process which introduced basic practices could be implemented by inexperienced project managers and developers yet did not impede the efforts of more experienced team members who were already applying advanced practices. In this paper we discuss how we engaged the right stakeholders to identify the vital process improvements to include in our initial process rollout and we expand upon a few key principles and lessons learned.

Authors

Opal Perry is an Implementation Manager with Wells Fargo & Company, focusing on software process development and improvement. During her career she has applied her software engineering knowledge and leadership skills to organizations in the defense, commercial software development, and financial services industries. Opal earned a B.S. in Computer Engineering from Carnegie Mellon University and an M.A. in Computer Resource Management from Webster University.

Laurie Burgher is a Project Manager with Wells Fargo & Company, focusing on quality assurance of high risk projects. During her career she has applied her project management and software engineering knowledge to organizations in the Department of Energy, commercial software development, and financial services industries. Laurie earned a B.S. in Computer Science from California State University, Northridge and an M.B.A from Drake University with an emphasis in Finance. She is a certified Project Management Professional, PMP.

Introduction

Over time, as organizations grow and mature, most discover that an ad-hoc approach to software development projects is insufficient to meet customer needs and expectations for quality - as well as cost and schedule performance. Innumerable studies have proven the value of software process improvement. While the need for quality software process is clear, many organizations and individuals find results more difficult to attain, particularly when they overestimate their organization's capacity for the change required to adopt advanced practices or tools. Many people are delighted and inspired when exposed to theoretical or academic models for software process and engineering practices yet "hit the wall" when they try to implement advanced practices before laying a foundation for software process improvement.

In 2004 our organization undertook a process improvement initiative, which represented the first phase of a total software development lifecycle quality improvement initiative. A 2003 quality assessment of a subset of development teams and business partners had identified opportunities for improvement in six general areas:

- Requirements Assurance
- Design and Development Assurance
- Test Assurance and Defect Management
- Implementation Assurance and Configuration Management
- Release Assurance, Project Change Control, and Resource Management
- Measurements and Process Improvement

It was determined that the first phase of activity should address requirements definition and management for software development projects. During the next nine months we worked with a committed cross-functional group of team members to define requirements process improvements which were subsequently piloted on selected projects and then deployed across one business line of our organization. This successful proof of concept resulted in an expanded process improvement and roll-out which is currently in progress across two other associated lines of business.

This paper discusses the challenges we faced throughout the initiative and the success we achieved by focusing on defining common sense software development process which could be quickly adopted by project teams to improve quality and address the most common problem areas and pitfalls experienced by projects within our organization.

Organizational Background

Our process improvement initiative, dubbed Requirements Assurance (ReqA), originated within the Consumer Credit Group of Wells Fargo and Company. Wells Fargo and Company is headquartered in San Francisco, California with offices throughout the country. This initiative brought together team members based in various geographic

locations, including offices in Arizona, California, Oregon, and Minnesota. While much of the work and collaboration throughout the effort was done using internet office tools and teleconferencing, core team members met together in Phoenix, Arizona approximately once a month for the duration of the initiative. These face to face sessions proved invaluable for accomplishing detailed reviews and reaching consensus on key issues.

Within the Consumer Credit area there were three main organizational entities which collaborated on the process improvements. The CCG Project Management Office (PMO) consisted of project managers and a project control group and represented the lines of business which are served by the technology applications and services. The Consumer Credit Technology (CCT) organization consisted of application and services development groups responsible for creating and maintaining systems using a variety of technologies and tools to serve the needs of the business. This organization contributed analysts, architects, designers, developers, and other technology talent to project efforts. The Home and Consumer Finance Quality Assurance (QA) organization included process improvement, change management, and project status and measurement specialists as well as test leads and test consultants.

These three primary stakeholders, along with other supporting groups, were committed to improving systems quality and organizational effectiveness through the introduction of common, repeatable, project process. Senior representatives from each area championed the initiative and ensured sponsorship and buy-in from all levels. The core team of contributors consisted of twenty two individuals from within the Consumer Credit area; there was fairly equal representation from each organizational entity and functional specialty.

The Pareto Principle

One of the keys to our success was our use of the Pareto Principle, popularly known as the 80/20 rule, as a rule of thumb. Simply stated, the 80/20 rule tells us that 20% of the improvements will yield 80% of the benefit. We used this as a rule of thumb to remind us to focus on the vital 20% for our initial process improvement. In this manner we could deliver much needed improvement on a near term basis, rather than trying to identify and improve 100% at once. Trying to improve all areas at once would have prolonged the improvement effort, and the frustrations of the project team members who were looking for more immediate relief in the form of common, repeatable software process. We did not attempt to define what improvements would comprise exactly 20%, rather we followed the spirit of the 80/20 rule to determine which key improvements would yield the greatest benefits. This approach was applied at several levels of action during the initiative.

The decision to focus on requirements definition and management is the first example of applying the 80/20 rule. It is estimated that 20% of the typical software project lifecycle is spent in the requirements phase (Glass 1992). We chose requirements

as the first focus area for several reasons. First, errors introduced during the requirements phase become more costly to fix as the project progresses through subsequent phases (Boehm and Basili 2001), it has also been shown that the majority of project defects are requirements issues (Boehm 1981.) Project measures and widespread anecdotal evidence confirmed that these issues, so common in the software industry, were also prevalent in our organization. We also recognized that it would be difficult to define downstream processes if the inputs from the requirements activities were neither well defined, nor of sufficient quality. For example, it would have been difficult to define consistent, repeatable, quality design processes if requirements input were of varying quality and format. Likewise it would have been difficult to define process for creating and conducting user acceptance tests if there were not a defined process for producing and reviewing the requirements documents, on which such tests are based.

After the scope of the initiative had been defined we continued to apply the 80/20 rule to determine which process elements were the most critical for the team to focus on. The choice to focus on defining quality checks each project team could perform on documented requirements over details of the requirements approval process was one example. Both areas were important but we determined that poor quality requirements were creating more issues than lack of approvals and so the focus areas were prioritized. Once a focus area was identified we applied the collective knowledge of the initiative team members, and others throughout the organization, to define effective process activities and support mechanisms such as templates, training and mentoring.

Leveraging industry expertise with organizational best practices

We recognize that one of the keys to our success was the fact that we were able to bring together a committed cross functional group of in-house software development and project management experts to identify successful practices from both within our organization and from external industry best practices and to determine how best to incorporate these into a common sense process appropriate for our business and organizational culture. Leveraging well respected team members from inside the company helped dispel the concerns, and occasional cynicism, of project team members who would not have as easily accepted a process developed by external consultants unfamiliar with our culture and specific challenges.

Industry Expertise

This is not to say that we closed our eyes to the use of industry expertise and best practices. Our team identified candidate industry best practices and evaluated them in the context of our specific problems and needs. In the end we used a number of concepts and approaches which were derived from external sources such as the Institute of Electrical and Electronics Engineers (IEEE), the Rational Unified Process (RUP), and the Software Engineering Institute (SEI) and applied them to meet our specific needs and culture. There were other industry best practices which were evaluated but not applied in our initial software process improvement effort; either because they did not adequately address our specific problem areas or because they were advanced practices which would

have been difficult to introduce in our organization without a foundation for process improvement. The important thing was not to take industry best practices and try to insert them in internal process without evaluation and tailoring for appropriateness and fit.

Organizational Best Practices

While root cause evaluation of problems commonly encountered in our software development projects indicated a lack of organizational process, we recognized that there were areas of the organization with local process and practices that had been proven to be successful on a subset of projects. Identifying candidate best practices from areas of the organization and determining how they might be adapted for use by other groups was another important focus area of the initiative. In examining specific problems we would also look throughout the organization to find example projects which had succeeded in avoiding or mitigating the effects of the problems and then examine what practices or process they had used. Once again it was rarely possible to take a process directly from one area and incorporate it into the organizational process without change. However with analysis and tailoring we were able to share best practices across the organization. Use of a traceability matrix to ensure every requirement was accounted for in design is one good example. One area of the organization had a traceability matrix they had used successfully on their projects, we reviewed this matrix and shared it with other teams, gathering feedback and making modifications before incorporating it into our rollout process.

Ensuring every new process element passes the common sense test

While industry best practices and existing local process within the organization contributed to the creation of organizational software process, a significant amount of effort was spent defining and documenting process activities in a manner that could be applied across the target organization. We found that, in the early stages of process definition it is possible that extraneous process steps or excessive detail may be introduced to the process. This was particularly true when process definition was performed by a collaborative team versus one or two individuals. At times, striving to account for all viewpoints resulted in “over engineering” of the software process, which added complexity and, left uncorrected, would have presented additional barriers to adoption when it was time to implement the process across the organization at large. Team reviews were a critical component to detecting these cases and ensuring that the final process was simple and relevant.

It was clear that, for our organization, the benefits of using a larger collaborative cross-functional team outweighed the costs but we also recognized the need to hold the group true to the guiding principles of addressing the most prevalent, most costly issues which were causing the most frustration across the organization. Solving problems which are localized to a small percentage of projects or teams was best left to the experts on those specific teams. Additionally, we believed that solving these localized problems would be easier after a foundational level of process had been introduced across the

organization. During our monthly face to face meetings of the process definition team we took time to question and discuss the level of process which was being defined to ensure that we were focusing on practical process definition which would address the real issues faced by project teams.

As an additional measure to address this issue we appointed a smaller team of three process improvement subject matter experts (SMEs) who focused on ensuring that the process details were clearly documented and applicable across the organization. Activities within the scope of the initiative were divided into five process areas:

- Business Need and Ballpark Sizing
- Requirements Definition
- Requirements Preliminary Estimate
- Design Specification and Sign-Off
- Detail Sizing and Release Assignment

The SMEs worked with business and technical team members in each process area to capture critical process details, identify issues and potential solutions, and create process workflows, activity details, and artifact templates. The initiative lead integrated input from all areas to ensure a consistent, comprehensible end-to-end process definition. This was in iterative process which benefited from cycles of content generation, review and consolidation, and group validation as input into the next cycle of content generation.

It is likely that there will always be some additional opportunity to optimize the process but we were pleased with the results of our approach. During process implementation we held twelve three day training sessions which introduced over 300 project team members to the Requirements Assurance process. Their feedback validated the fact that we had defined a common sense process which addressed their major problem areas without introducing undue complexity or extraneous details. While the effectiveness of the process must be demonstrated by project results, validation from project practitioners in training was a significant indicator that the process could be successfully adopted and institutionalized within the organization.

Avoiding “analysis paralysis”

We recognized that a companion problem to “over engineering” software process was the risk of “analysis paralysis” – the failure to achieve real results because of prolonged discussion of issues for which there were multiple candidate solutions but no universally preferred solution. Analysis paralysis could kill a process improvement initiative – leaving project teams to continue to struggle with day to day problems attributable to the lack of any clear organizational process. Our initiative faced this challenge several times but we were able to overcome it with leadership, trust, teamwork, and a continual focus on the overwhelming organizational need which led to the creation of the process improvement initiative in the first place.

Strong vision and leadership was critical to preventing analysis paralysis. Our initiative champion, the head of the QA organization, was able to keep stakeholders

focused on the ultimate goal of improving software quality through the introduction of common software process. Clearly articulating the 80/20 vision and promoting the concept of continuous process improvement helped the team, and those who will use the process on a day to day basis, understand that it was better to make some improvements that would immediately benefit projects than to stall in trying to define the perfect process in the first pass.

Trust among stakeholders and all process team members was another key to avoiding this pitfall. Once individuals with differing opinions on specific solutions trusted that their interests were truly represented, and that the process would be modified in the future if feedback demonstrated that the chosen solution was not working, they were more willing to agree to an interim solution so the initiative could proceed. In practical terms we found it took a substantial amount of time (approximately four months in our case) to bring together representatives from across the organization, move past previous communications barriers, and focus on the common objectives to the benefit of the entire group.

The fact that the process definition team included representatives who had day to day accountability for projects still struggling under the lack of clear process also helped maintain momentum. The need for process definition and improvement was very clear to these folks and they understood the importance of moving theoretical discussions into the practical realm by developing and deploying process and supporting tools for use on their projects.

To move from theoretical discussion to practical process we found it helpful to test process and scenarios in a limited real world environment and gather feedback to improve the process prior to full deployment across the organization. Incorporating Proofs of Concept (POC) was a key strategy in keeping the initiative moving forward and avoiding analysis paralysis.

Using proofs of concept to help build momentum

After several months of defining and reviewing process activities, templates, and job aids we held a two day review session with sponsors and key stakeholders to evaluate status and determine next steps. This session proved to be a significant event in our initiative for several reasons.

First, we received validation that the process and materials we had created would meet the needs of project teams; some excellent suggestions for changes to fine tune the process were also received. For example, one PMO representative suggested an enhancement to our requirements traceability matrix which allowed project teams to track critical cross-project dependencies.

Secondly, senior management used this forum to reiterate the urgent need for implementing process improvements and we received the go-ahead to begin a proof of concept by introducing the process on several new projects within the organization.

POC Selection

We selected four projects for proof of concept (POC.) Applying the process on multiple projects allowed us to select projects with different parameters (e.g. some projects were on an enterprise release cycle, others had independent delivery dates), different business and technology stakeholders, and different project managers. This approach proved valuable in allowing us to spot trends in the experiences of project teams applying the process and also allowed us to keep the proof of concept momentum when several projects were delayed due to factors unrelated to our initiative.

It was important to select the right projects for the proof of concept. As our process was designed to start at project initiation, we wanted to identify candidate projects who were about to start the initiation process, and train them in the process just prior to applying it on their project. Unfortunately fixed release cycles and the urgency of meeting an end of year rollout schedule prevented us from completely achieving this target. One of our POC project teams had already started the business need definition stage of the process prior to training and this affected their ability to effectively execute the process in later stages as some of the deliverables they had produced before learning the process did not meet the quality criteria required for downstream success. The upside to this experience was that the project team readily recognized the value the complete process would bring to their next project effort.

In addition to project timing there were other factors considered in selecting POC projects. We attempted to select projects that were of high enough business priority that they would receive funding but not of such high priority that team members and stakeholders fearful of applying new process would subvert the process or skip key steps. We also selected projects that were likely to use fairly stable architecture and technology; we felt that it would be difficult to assess the process improvement effects on projects with high risk or significant new technology separately from the other factors and risks inherent to projects of this type.

Project Member Training

Once each POC project was selected we trained project team members in the process, artifact templates and associated job aids. Training the POC teams also served as our pilot of final rollout training materials. Training and support for those involved in the POC was a critical success factor and once again having a fairly simple, common sense process proved to be a huge benefit. We conducted three day training sessions and used the subject matter experts (SMEs) who had helped create the process as training instructors. Using this approach, as opposed to having professional instructors teach the material, was particularly valuable as many questions arose during training which could not have been anticipated prior to the POC. Using the process SMEs ensured that questions could be answered with clarity, in class, and the project team could leave class ready to apply the process effectively to their projects. During class we also logged issues and change suggestions which were raised by team members. This helped fine tune the process before rollout and also increased the buy-in of POC team members who could see

that we were sincere in our desire to work with them in ensuring the process met their needs. Immediately after training we sent each POC team member a list of the issues and action items they identified, along with updates on status, assignment, and ownership. POC teams found this communication very useful in planning their efforts and communicating status and expectations to their management team.

Mentoring by Subject Matter Experts

The same SMEs who conducted training served as support mentors during the POC project lifecycle. We offered support through a number of mechanisms including a weekly support call for all POC teams, a dedicated support mailbox, lessons learned meetings, and proactive calls to each POC project manager to determine what additional help might be of use.

To get the most out of the POC projects it was essential for those involved in the POC to feel that their issues and concerns were heard, prioritized, and acted upon. As those involved in the POC begin to experience the benefits, they become part of the grassroots effort that supported the process implementation. In essence, they became an extension of the process improvement team and were critical spokespeople to the rest of the organization during full process rollout.

Managing organizational challenges and dependencies

One of the challenges we faced was that the pace of work did not slow down to accommodate change. Project teams were faced with the challenge of taking the time to learn the process, and applying it on their new projects, while remaining accountable for deliverables on projects which had started before the process was introduced. This challenge was compounded by resources issues which had previously been created by the problems the process was designed to address.

Resource Availability

Prior to the process improvement effort our projects were plagued by errors and integration issues late in every release cycle, during test, installation, and user acceptance. This resulted in developers, project managers, and other team members spending a large amount of (unplanned, unscheduled) time to resolve these issues. With an aggressive overlapping release cycle this usually meant that project activities for the next release were delayed because the personnel scheduled to perform those tasks were busy coping with problems in the current release. This created a “snowball” effect with each release falling increasingly far behind schedule.

During the process improvement effort we identified that one of the key causes of the problems encountered late in the release cycle was the failure to involve all of the appropriate team members in early project activities such as needs analysis and requirements definition. One of the objectives of the process effort was to identify which roles needed to be involved in early project tasks. Not surprisingly, the people who

needed to be involved in these early activities were the same folks caught up in the late stage firefighting. As the process was implemented on the first wave of projects it was critical that the key resources were made available for new project work, to break the cycle of late stage issues. This required discipline and commitment from all levels of the organization.

This is but one example of resource challenges which surfaced during process implementation. To successfully implement change there must be a real commitment to providing the appropriate resources at the right times for execution of the process.

Attitudes Towards Process Improvement

Another challenge faced when introducing process improvements was skepticism and cynicism on the part of individuals who must apply the process. Many people who have years of experience in corporate life, especially within large organizations, have seen multiple, often short-lived attempts at process improvement. Some of these folks were jaded from their past experiences and saw little use in learning a new process which they believe will be superseded by the next wave of change in six months, a year, or some other timeframe. We found three specific approaches which were helpful in overcoming cynicism, or at least preventing it from spreading to others.

The first approach was making sure that sponsors and management really were committed to the process improvement initiative. Although Executive Management support and engagement was needed for all aspects of the improvement effort, implementation of the process was where the "rubber meets the road". Executive Management support and active involvement can make or break the implementation, and the support we received was critical to our success. The message of support had to be clear and consistent throughout the organization. Lip service was not enough; support for process improvement had to be demonstrated through action as much as through words. Goal and incentive alignment was one management tool for overcoming inertia and motivating individuals to fully adopt the new process. If individuals were rewarded more for cutting costs than for delivering quality software they would subvert quality process if there were an opportunity to cut costs in doing so. It was critical that the process remain aligned to organizational goals, and individual goals remained aligned to the process, thus supporting organizational goals and objectives.

Secondly, it was important to ensure we had the support and buy in of individuals who were "thought leaders" in the organization. Leader in this sense of the word is not meant to imply management, but rather folks who serve as informal (and sometimes formal too) leaders in their area of expertise. When process, tools, and templates met their standards we knew others would be willing to follow suit. When these individuals applied the process on their projects and became vocal advocates for its use, the barriers to organizational adoption were reduced.

Finally, taking the time to listen to, acknowledge, and discuss the concerns of skeptics went a long way to establishing credibility throughout the organization. Once

people had a chance to air their concerns they often found it easier to accept change and move towards implementation of the process improvements. We were willing to have these dialogues one on one, in group meeting, training sessions, and other forums and it paid off. We were sometimes surprised to hear that individuals who had come to class adamantly opposed to change had become some of the most vocal advocates for our process. Of course this would not be possible if the underlying process did not adequately address the needs of the organization and individuals.

Managing Dependencies

Managing dependencies with other efforts was another challenge we faced. It was important to identify, and stay mindful, of other initiatives or organizational factors which were interdependent with the process improvement effort. If these were not managed carefully they could have impeded the process implementation and caused a great deal of frustration among those who needed to learn and apply the process.

We found it relatively easy to manage dependencies with existing static or mature processes, the new process could be designed to integrate with the existing elements and those being trained in the new process would already be familiar with the existing elements. It was much more challenging to manage dependencies when there were many moving parts. Our process improvement initiative took place at the same time a number of other corporate initiatives were being developed. We took the approach of proactively engaging with these other change agents and continuously integrating our evolving process.

Required Skills

The success of every process is also dependent on the skill set and knowledge of those who apply it in their day to day work. While those of us developing process can, and absolutely should, do our best to keep the process simple and to provide good training and support to the end users, there are some processes which, by their nature, require specific skill sets outside of the process itself. An individual or organizational lack of the skill sets required to complete process activities introduces significant project risk.

Establishing and clearly documenting process helped identify required skills sets and sets expectations for those performing the process. This made it easier to identify team members who needed additional training or coaching to bring their skill sets up to par. Aside from helping to manage individual performance, we found that process definition may also highlight needed skills which do not currently reside within the organization. During our process definition effort we identified a couple of project skill sets (e.g. requirements documentation) that were not consistently available to successfully implement our process, these areas are being addressed as a follow on to the process definition effort. To fully meet the objectives of the process improvement, organizations must identify, and staff with, the proper skill sets needed to implement the process.

Every process improvement effort will have some dependencies upon factors outside the control of the process team. It is important to identify such dependencies and manage them to minimize the risk of adverse impact to the process effort or project teams.

Sensitivity to both the organizational and individual capacity for change

It is the nature of process improvement efforts to introduce change within an organization. This change affects each individual as well as the collective whole. While the change is clearly being introduced to bring about improvements in the long term, in the short term change is painful for most people and organizations. Obviously leadership, goal alignment, and individual outlook all play a part in helping manage change constructively. Additionally the process improvement team should regulate the pace of change they are introducing to the organization. Force-feeding too much change at once will only cause frustration for all concerned.

At the first core team meeting for our initiative we discussed the common problems which had been identified, and the analysis which had led to the creation of the process improvement effort. We also discussed candidate solutions for addressing the biggest problems. One proposed measure was to mandate the use of an automated requirements tool on all projects. If properly used the tool could address issues with documenting requirements and establishing requirements traceability. However, some stakeholders had experience with an unsuccessful implementation of the tool and others felt the organization was simply not ready for this level of change. It was decided that we would define processes which addressed the most pressing issues, without mandating use of automated tools for all projects, but was consistent with the use of those tools. In this manner we addressed the needs of all projects, allowing projects that wanted to use the tools to do so, and left the option open for future introduction of tool use to all projects.

We found that there is no simple formula for determining the amount of change an organization can undergo at any one time, and not all change is driven from within an organization, external issues may accelerate the rate of change. When there are multiple project areas that need to be addressed to meet regulatory requirements, such as Sarbanes Oxley, there may be no other option than to introduce more change than would otherwise be advisable. Organizational risk and regulatory and legal requirements must be prioritized and balanced against the organizational and individual capacity for change.

Phased process improvement helps both individuals and the organization as a whole, adapt to, and realize the benefits of process improvement. Phased deployment may also provide opportunities for mid-course correction or enhancement if organizational priorities or other factors change during the rollout. For example, after the initial deployment to a single line of business (as discussed in this paper) our organizational priorities shifted away from line of business specific development to focus on a small number of high-risk high-complexity projects which impacted multiple lines of business. Planned windows of opportunity between deployment phases allowed us to

quickly align our process improvement and training resources in support of this new focus instead of being caught up in the inertia of a large scale all-at-once deployment.

Results

The process improvements have been fully deployed for just over two quarterly release cycles now so it is still too early to completely assess the results of our initiative. Feedback from project sponsors, project managers, and other team members indicates that the process improvements are having an overall positive impact on individual projects as well as the overall management of organizational resources and priorities. There is a clear benefit to having a documented, repeatable process for all project team members to follow, less time is spent trying to decide what should be done or who is responsible for specific activities and more time can be spent accomplishing project objectives and focusing on the needs of the business.

There are still requirements process issues which need to be addressed through management controls, skill set training, and further process improvements. Application of the process on numerous projects has highlighted constraints with the pre-existing release calendar and these are being addressed through ongoing efforts with key stakeholders. Key process elements from this initiative are being leveraged as part of the definition of an Integrated Methodology for a larger part of the Wells Fargo and Company Home and Consumer Finance Group and that in itself is a testament to the valued added by the ReqA initiative. While more time is needed to fully assess the results of this initiative, we feel that the organization is on a positive trajectory towards maturing project process and improving development quality.

Conclusion

As we have discussed, there are numerous components to a successful software process improvement initiative. While needs and priorities vary by organization and industry, organizations just beginning their process definition and improvement efforts will be aided by focusing on critical success factors rather than aiming for perfection in a single improvement cycle.

We learned that deploying a solid, common sense practice which introduced basic practices was implementable by inexperienced project managers and developers yet did not impede the efforts of more experienced team members who were already applying advanced practices. We were able to achieve tangible results by focusing on establishing a foundation for software process that can grow and mature in the coming years.

References

Glass, Robert L. 1992. *Building Quality Software*. Englewood Cliffs, NJ:Prentice-Hall.

Boehm, Barry, and Victor R. Basili. 2001. “Software Defect Reduction Top 10 List.” *IEEE Computer*, Jan.

Boehm, Barry. 1981. Software *Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall PTR.

Bringing Test Forward: Appling Use Case Driven Testing in Agile Development

Jean McAuliffe and Dean Leffingwell

Abstract

Use cases provide a user-focused sequence of events that can serve as a template for acceptance testing activities. In an agile environment, use case-based testing brings testing into the process earlier and also provides early peer review for the logic of the intended new functions. Involving testers in use case development and then harvesting the use case scenarios for acceptance test case development is a powerful technique that has shown practical benefit in a number of agile development environments. This paper provides a brief overview of the use case-based testing method and highlights some practical tips for application and automation of this technique for agile acceptance testing.

Author Bios

Jean McAuliffe (jean@walkingorbit.net) was a Senior QA Manager for RequisitePro at Rational Software, before joining Rally Software Development as Product Manager. She currently is leading the Product Delivery Team for Walking Orbit, Inc. She has 20 years of experience in all aspects of software development (defining, developing, testing, training, supporting) for software products, bio-engineering and aerospace companies. The last two years she has been actively and passionately involved in agile development, and is a certified ScrumMaster.

Dean Leffingwell (dleffing@earthlink.net) is Chief Methodologist for Rally Software and former SVP of Rational Software Corporation where he was responsible for the commercial introduction of the Rational Unified Process. Mr. Leffingwell currently acts as advisor and agile coach to a number of developmental-stage software businesses. He is the lead author of the text *Managing Software Requirements: Second Edition: A Use Case Approach,* Addison Wesley, 2003

From *Test Last* to *Test First* - The Tester's Dilemma

Historically, software testing activities have always come late in the development cycle. Indeed, in the waterfall method of software development, testing was a distinct "phase" at the end of the process designed to assure that the already-written application meets its requirements. Typically, however, the features under test were delivered very late in the cycle, leaving little time for testing activities. Pressure was on testers to do their best–but the job had to be done fast. The release was already behind schedule and testing was going to make it even later.

Moreover, at this late date there was likely to be a significant amount of untested software present and the tester's dilemma was to pick and choose what *was* going to get tested and what was *not*. The team typically discovered a significantly large numbers of defects–counts into the high hundreds were not uncommon–the underlying quality of the application was always suspect and some of the software may not have been tested at all.

To address this development problem, agile methods instead focus on delivering small increments of working, *tested* code as quickly as possible. Functionality to be delivered is sliced into smaller and more manageable size pieces with smaller scope, tighter timelines and more effective testing coverage.

However, the short iteration cycles of agile still provide no slack time for testing. Pressure remains firmly on testers to make sure there the testing dilemma is mitigated in agile. "All" they need to do is make sure that the right things are getting tested, at the right level, with the right effort and all in the right time! Efficiency of the testing process is paramount to success.

This paper describes the application of use cases in agile methods which help testers achieve these necessary efficiencies by bringing test forward. Further, both quality *and* speed can be achieved and optimized by using the developed use cases to directly generate the acceptance test cases.

Agile Software Development

In the last ten years or so, we've seen the rapid adoption of various *agile* software development methods which have demonstrated substantial productivity and quality gains for many software teams. In the following section, we'll provide a short background on the movement to software agility and how it impacts the tester's dilemma.

Background of the Agile Movement

There are a number of differing agile methods in use but they all share many common traits. Specifically, all:

- Promote the rapid delivery of value to the customer by *more rapidly delivering working tested increments*
- Mandate *time-boxing* of iterations and releases
- Provide *timely and regular visibility* of the solution to customers, product owners and stakeholders
- Provide methods to more effectively respond to *changes* that occur

Table 1 lists some popular agile methods currently in use and captures some key characteristics and differentiators for the method.

Table 1 Popular Agile Methods

Source	Key Characteristics
Evo (Evolutionary Project Management) (http://www.gilb.com/)	Started 1960's – 1976 - Oldest iterative and incremental development 1 to 2 week increments, quantifiable measurements of progress and value
DSDM- Dynamic Solutions Delivery Model (http://www.dsdm.org)	Started in 1994 – Based on Rapid Application Development (RAD) framework. Aims to produce and evolve an approach to building systems both quickly and well. Popular in UK and Europe
Adaptive Software Development (http://www.adaptivesd.com/)	An Agile Project Management Framework Five phases: Envision, Speculate, Explore, Adapt, and Close
eXtreme Programming XP (Kent Beck) (http://www.extremeprogramming.org/)	Speaks to developers with specific coding and test-first practices. Attributes: constant informal communication, pair programming, simplicity, feedback, very rapid iterations and releases.
Lean Software Development (Mary and Tom Poppendieck) (http://www.poppendieck.com)	Application of lean manufacturing principles to software development. Eliminate waste; amplify learning, Decide as late as possible, Deliver as fast as possible. Empower the team. Build integrity in. See the whole.
Crystal (Alistair Cockburn) (http://alistair.cockburn.us/)	A family of methodologies that share frequent delivery, close communication, reflective improvement that supports varying sizes of teams and application criticality.
Feature Driven Development (Jeff DeLuca, Coad) (http://www.featuredrivendevelopment.com/)	Domain Object Model-centric techniques. Development by features, component/class ownership, feature teams, inspections, configuration management, regular builds, visibility of progress and results (Practical Guild to Feature-Driven Development).
Scrum (Ken Schwaber) (http://www.controlchaos.com/)	Widely adopted agile software project management framework for rapid iterations and releases. Provides needed visibility to adjust. Attributes: visibility to adjust, one month iterations, daily stand-up meetings, prioritized product backlogs

The Agile Manifesto

A few years ago, a number of agile leaders came together and created a manifesto of guiding principles that are common to all agile methods. (see **http://agilemanifesto.org/principles.html**.) Many of these principles speak directly to the problem of earlier end-user value delivery as well as implied earlier testing. Excerpts from the manifesto include:

- "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Working software is the primary measure of progress.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. "

Benefits of Adopting Agile

There are many direct and indirect benefits that teams have experienced when adopting agile:

- More frequent releases; better response to change. Teams occasionally even decide to release *early* with what they had that was already working and then restructure the remaining work into the next release.
- Immediate, empirical feedback – By having a continuous implementation, build, test and demonstration process, the team always knows where they stand in the iteration or release. New issues are identified and more quickly addressed, as compared to late discovery of issues in the regression test or integration phase where they are far more painful and expensive to address.
- Better teamwork, higher accountability – Agile requires constant communication and sharing responsibility. Everyone pulls together in a synchronized rhythm to meet the value delivery objectives of the iteration. Developers and testers are working on the *same functionality* at the *same time*. The continuous context switching that typically occurs when testing happens out of synch with code development is largely avoided.

Agile Releases and Iterations

Iterative development breaks the software lifecycle into small, sequential iterations. Each iteration is a self-contained project, with the team performing the usual activities of requirements, analysis, design, programming, testing and delivery. The goal of each iteration is to produce a stable, integrated and tested increment of potentially shippable software [5].

Figure 1 below illustrates the basic iteration and release cycle common to all agile methods. Most agile teams release externally at least every two to three months. This may sound too frequent for those used to six to eighteen months cycles but keeping the releases shorter and under more control yields better product quality, customer satisfaction and team morale.

Iterations may be as short as one week, or as long as one month, with two weeks being common. No matter the length, the guiding principle is to load the iteration with the highest priority customer value items to be implemented.

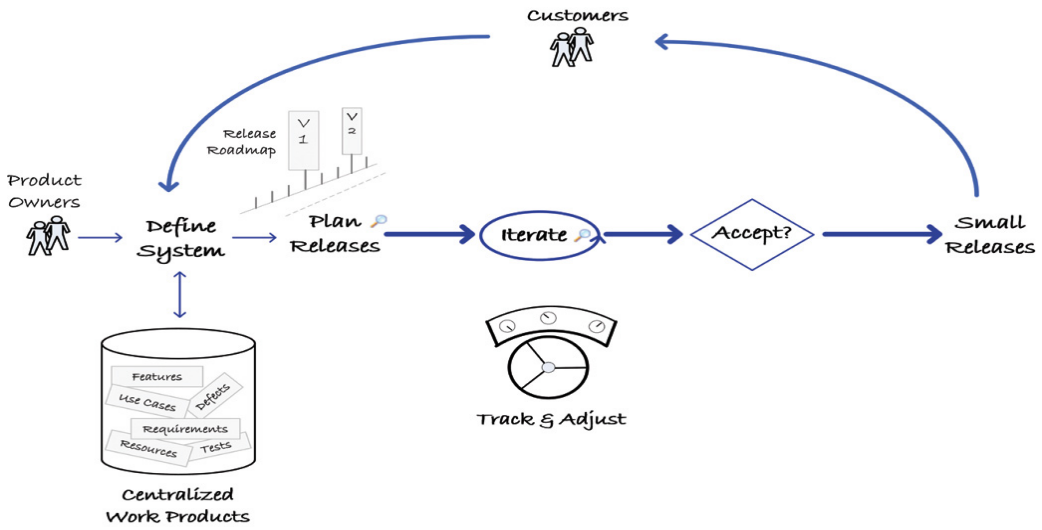


Figure 1 - The Basic Agile Iteration and Release Model.

Acceptance Testing

As each iteration produces *working, tested* code, the estimate for the work in the iteration must include the effort for the whole team, including testing. Testing estimates should include the tasks necessary to create and run acceptance tests.

Agile acceptance tests are typically black-box system tests that verify the specific functionality for the system under development in the iteration. Acceptance tests have been described by Brian Marick (see http://testing.com/agile/) as those tests that are "Business Facing". Others refer to acceptance tests as "Customer Tests". These labels serve to differentiate acceptance testing from *unit* or *developer* testing. Agile testing teams achieve effective acceptance testing by:

- Leveraging developer testing (unit testing) for code level defect discovery, allowing testing teams to focus on higher level acceptance tests
- Sharing the responsibility with the developers for passing acceptance tests and by writing, automating and executing acceptance tests during the iteration
- Setting quality criteria to determine what acceptance tests must pass in order to accept the item of work. There may be "optional" tests that can fail.

How Agility Forces Bringing Testing Forward

Iterations require acceptance testing of all items planned for the iteration. Because of the short iteration cycles, testing is forced earlier in the process. With agile, testing occurs weeks into the release as opposed to many months later.

To be successful using agile, however, the team needs to bring testing *even further* forward. Otherwise the mini-waterfall nature of the iteration might allow for all the code to be delivered on the last day. If all the code for is delivered late in the iteration, the team will fail the "working, tested" principle as there simply won't be sufficient time to test.

To address this problem, many agile methods prescribe a technique called Test-first or Test Driven Development (TDD) [1] that encourages the team to write the tests, (both unit test and acceptance test), *before* writing the code. Focusing on getting the highest priority items coded, working and accepted before going on to the next piece of functionality is a critical step ensuring that stability and quality are built *into* the code from the beginning.

Figure 2 illustrates the mechanics of this method. Before the iteration begins (Iteration N-1 in the diagram), requirements need to be sufficiently elaborated so they can be estimated with a reasonable degree of confidence. During the iteration, the requirements and the tests associated with them are further refined as the team discovers more details. By taking the highest priority items first for development, testing, and acceptance, the team can be sure that the iteration will complete and deliver the highest valued items even if all items cannot be completed. Lower priority items are started only after the highest ones have been accepted. A demonstration of all accepted items is required as part of the acceptance review of the iteration.

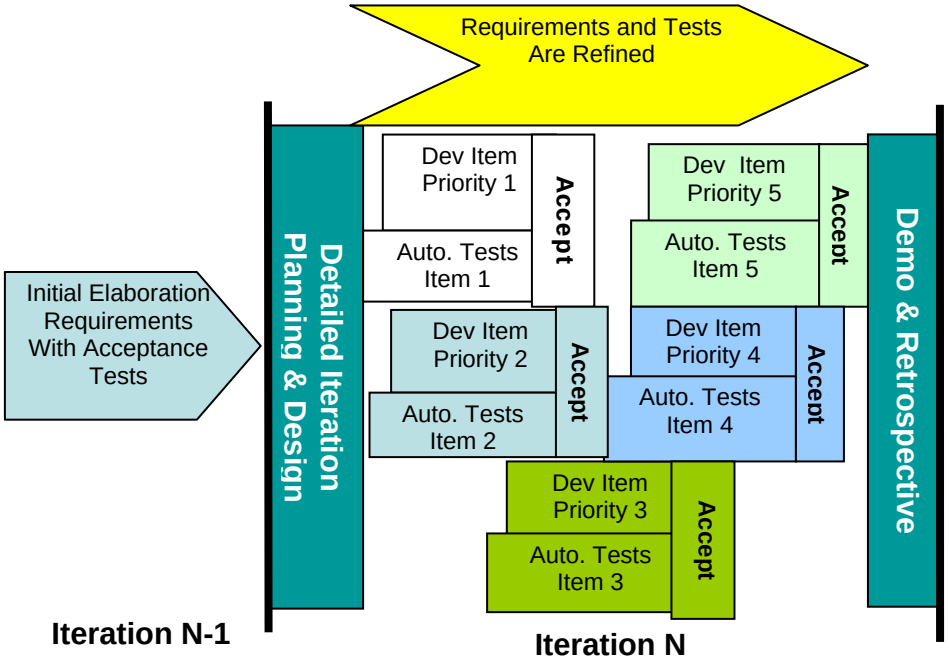


Figure 2 Iteration Acceptance Pattern

Test First

It is easy to *say* "move testing forward" but most teams struggle to implement this concept. Cohn[3] notes: "if you can't seem to find the time for something (testing), then make sure you do it first." This can be accomplished by:

- Sketching out the testing that will be necessary for the functionality during Release Planning, including labor estimates for writing and executing the tests [4.]
- Working ahead to create the acceptance tests for the highest priority items that are planned before the iteration commences. [3]

Writing the testing cases first earlier will provide better definition of the actual work to be performed and assist the team in evaluating and accepting the work that occurs during the iteration.

Agile Methods and Use Cases - the "Sweet Spot"

Use cases and "user stories" are often applied by agile practitioners to describe the intentions of the application. User stories, which are used by most agile methods to describe application requirements equate roughly to a scenario of a use case, which is a semantically richer expression of desired behavior. As Cohn[3] states "user stories are a written description of the functionality for planning and a promise for future discussions to flesh out the details".

Use cases can be the "sweet spot" for accelerating testing when applied on a just-in-time basis. Use Cases provide the context, structure and content to assure that all stakeholders understand what is being developed. They don't replace the conversations about functionality, but serve to document agreements reached for any required system behavior.

The Use Case Approach to Requirements Expression

Use cases are a user-focused sequence of events that can serve as a template for just-in-time functional testing activities. Use cases have a number of advantages over traditional declarative requirements expressions ("the system shall….") Application of use cases accelerates both an earlier understanding of the intended behavior of the application as well as serving as a template for acceptance testing within the course of the iteration.

Elaborating use cases provides early peer review of the logic of intended functions. Harvesting the use case scenarios for test case development expedites the testing while providing a mapping function to ensure acceptance testing for all new functionality. Running acceptance tests against the delivered use cases during the iteration helps assure early testing and optimizes the efficiency of the team as well.

Benefits of Use Cases

Teams that adopt use cases find that they help the development team think through the design of the system from the *perspective of the user.* By having a clear, easy to understand flow of events, the team can engage in a more meaningful discussion involving all users and other stakeholders. Benefits of the use case technique include

- Developers provide technical insights to the intended behavior and better understand technical risks
- Documenters understand how the system will be used and provide valued feedback on complexity and usage issues
- Testers, adept in the art of breaking software, add immediate insights into the alternate scenarios and exception conditions which are often sources for large numbers of defects that would be found later.
- Use case scenarios can then be used to drive the acceptance testing process

Given these benefits, a short summary of the use case technique is now in order.

Use Cases 101

> *A use case identifies a sequence of actions performed by a system that yields an observable result of value or achieves a goal for an actor*[2,6].

An *actor* is someone or something outside the system that interacts with the system. Use cases are an effective way to express the behavior and interaction that an actor (user) has with the system. Use cases reflect the user perspective and how they will optimally use the system as well as how they might misuse it or respond to other exceptions that may occur (alternate flows and exceptions).

A use case has the following standard elements:

- Unique ID
- Name – a description of what will be achieved. Names should be short and unique.
- Brief Description – captures the purpose of the use case in a couple of sentences.
- Actor(s) – The actor could be a real person, or it could be another system that is interacting within your use case. Actors are specific for the product you are developing. Actor–goal and persona–goal lists can help you identify the right ones [2]
- Sequence of Events – This is the body of the use case. Each step contains an actor step followed by a system response.

Additionally, use cases often include pre and post-Conditions that define the required system states before and after the use case execution.

The main success scenario is also referred to as the happy path[1]. Most use cases have multiple extensions that branch the flow when certain events happen. Extensions can either end or return back to where the branch occurred. Extensions can be thought of as mini use cases. They have a condition which sets the reason for when this extension occurs, a sequence of user event steps and system responses, and ends with the final result (or returns back)[2].

Tips and Suggestions for Writing Readable Use Cases

Reading a well-written use case is easy. However, writing a good use case is a learned art and it takes practice and feedback to become facile with both the style and content of use case expression. Some tips for writing effective use cases include:

- Keep the use cases as simple as possible to convey the necessary information.
- Elaborate the main success scenario and a few extensions first. Break use cases up if the scenarios get too complex. The more complicated it is to write, the more confusing it is likely to be to readers.
- Normalize the size of a use case into digestible pieces of functionality. One rule of thumb is to factor use cases into the amount of functionality that can be accomplished by the team in a single iteration.

Example Use Case: "Attach Supporting Information"

Figure 3 shows an example of a use case from Rally Software based on a Cockburn use case template[2]. In this example, a user of a system needs the ability to attach additional supporting information via a file attachment.

This example use case also has an *included* use case, called Cancel Changes. *Including* a use case is beneficial when you find you are writing the same behavior multiple times. Included use cases identify common behavior for reuse from a single point of reference.

This use case also demonstrates a type of extension called an "anytime extension". This extension is not tied to a specific step, as an actor could enter into this extension at anytime during the execution of the use case. These anytime extensions are designated by using a letter format first. In this case, the anytime extension is labeled "Extension b". Each extension is labeled based on the step it extends. As an example, Extension 3a extends from the main success scenario, step 3 and is the first extension to do so. If there were another extension from the same step, it would be labeled "3b".

[1] It's a "happy day" indeed when the user and the system do exactly as anticipated and no errors, exceptions, side conditions or other issues are present.

UC1091: Attach Supporting Information

ID:	UC1091
Name:	Attach Supporting Information
Description:	describes the ability for the actor to attach or remove supporting information
Goal:	to have file attached
Actor:	Systems WP Editor
Priority:	Critical
Risk:	Low
Frequency:	Medium
Owner:	jean
Package:	Collaboration
Pre-conditions:	artifact has been created
Post-conditions:	additional information is attached or removed
Main Success:	**Trigger** actor wishes to attach a file **1. actor** selects artifact for attachment **2. system** presents the details of the artifact **3. actor** selects to add an attachment **4. system** presents the ability to enter in the details of the file and the ability to browse for the file **5. actor** browses to find the file and select to commit the attachment **6. system** attaches the file and displays the information to the actor The use case ends.
Extension b:	**Condition** actor decides to cancel out of the attachment process **1.** UC16: Cancel Changes The use case ends.
Extension 3a:	**Condition** actor instead wishes to remove an attachment **1. actor** select the attachment to remove **2. system** prompts are you sure **3. actor** selects to continue with the removal **4. system** removes the attachment The use case ends.
Extension 3a4a:	**Condition** another user has already removed the attachment from the system **1. system** presents a message that another user has deleted the attachment and nd returns the actor to where they were The use case ends.

Extension 5a:	**Condition** actor instead wishes to add the detail information **1. actor** selects to only add the details and skips adding the file **2. system** attaches information and displays only the details information The use case ends.
Extension 6a:	**Condition** system cannot attach the file- cannot attach the file - file no longer exists in the location specified, server went down, etc. **1. system** Presents a message to the actor that the file could not be attached. The use case ends.
Extension 6b:	**Condition** file being attached is greater than 5 MB **1. system** displays a message that the file cannot be greater than 5MB and suggests that the user zips the file. The file is not attached. The use case ends.
Extension 6c:	**Condition** Atrifact has been deleted by another user **1. system** Displays a message that the artifact has been deleted by another user, file is not attached The use case ends.
Variations:	
Notes:	

Figure 3 Example Use Case

From Use Case to Test Case - A Three Step Process

Time spent writing use cases will reap significant time savings when defining your test cases and will help accelerate the development of the acceptance test case during the course of the iteration. A systematic step-by-step technique for doing so is described by Leffingwell and Widrig[6]. By following this technique, we can help assure that the acceptance testing for the iteration matches the newly included functionality. Each of these steps is described below.

Step 1 Identify all Scenarios in the Use Case

A *scenario* is a complete end to end path through the use case that follows the steps and extension steps of the use case. Each use case will generally create a number of differing scenarios based on the users varied actions in executing the use case. Figure 4 shows scenarios that arise through the alternate flows or extensions.

Many of these extensions are for expected behavior – regular variants as well as odd cases – others capture exception and error conditions that may occur. It is often a good practice to read both the use case and each scenario. Reading scenarios adds value by providing a completely different perspective for the stakeholders, and may uncover flaws in the logic that need to be corrected.

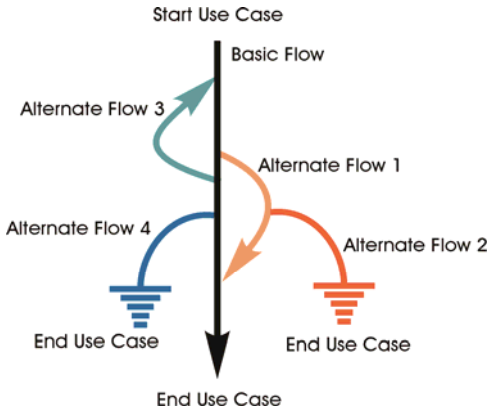


Figure 4 Possible Scenarios of a Use Case

Table 2 shows the scenario matrix for the example use case described above. Elaborating this matrix will no doubt uncover additional undiscovered scenarios. Finding these additional scenarios is a primary reason why testing input is so valuable during the elaboration and analysis activity.

Scenario Number	Originating flow	Alternate Flow
Scenario 1	Main Success	
Scenario 2	Anytime Scenario - Cancel	
Scenario 3	Anytime Scenario - Cancel	Cancel the Cancel
Scenario 4	Main Success	Remove the Attachment
Scenario 5	Remove the Attachment	Multi-user already has removed
Scenario 6	Main Success	Adds detail information only
Scenario 7	Main Success	Failure on attaching
Scenario 8	Main Success	File is too large
Scenario 9	Main Success	Multi-user – another user has deleted the artifact

Table 2 Identify the Scenario Matrix

Step 2 –Identify the Test Cases

As all scenarios are not equally critical for testing, getting consensus from the team to identify the highest priority scenarios is important. Your testing investment should be proportioned to those scenarios that are the highest priorities for the users. Three factors can be used as inputs to determine testing priority:

- Technical Risk – there may be a development risk that developers identify – e.g. new technology was used, or a particularly complex and difficult algorithm was implemented.
- Business Priority – a scenario could be of a high business value and therefore worthy of being more thoroughly tested
- Frequency of Use – a scenario could be one that is most heavily used, or used by many actors

From these inputs, determine which scenarios are of the highest priority for testing and thus which test cases should be elaborated. If time permits, lower priority test cases can then be developed. The example below elaborates test cases only for Scenarios 1, 4, 6-9. Table 3 shows the test cases with their expected results.

You can create multiple test cases per scenario or use a data-driven construct to *write once-test many times* using different input data. Each test case can then be run with multiple data sets that exercise the test cases to their fullest extent. These data sets could include boundary conditions, positive and negative testing, etc.

Test Case Id	Scenario	Description	Expected Result
TC6	1	Main success	File is attached
TC7	4	Remove attachment	File is removed
TC6	6	Add details only	Detail information is added
TC7	7	Can't add attachment	Error message explaining the problems should be displayed
TC8	8	File too large	Message that file exceeds 5 MB is displayed, user is

			allowed to try again with a smaller size
TC9	9	Object that owns the attachments has been removed	Error message that the object has been deleted by another user

Table 3 Test Case Matrix

Step 3 – Add data values for the conditions in the test cases

As the final step, you will need to define the specific data that will fully exercise each test case. Use your normal black box testing techniques by looking at the business rules. Data sets such as boundary conditions and randomly generated data should be used. Table 4 below describes the end result of this process.

Test Case Id	Scenario	Description	Data Driven Input
TC6	1	Main success	• File size < 5 MB • Add multiple file types: txt, image • Add to Artifact Type A • Add to Artifact Type B
TC7	4	Remove attachment	• Remove from Artifact Type A • Remove from Artifact Type B • Attempt to remove when no attachment – should not be able to
TC6	6	Add details only	• Add details < 6KB – success, no truncation • Add details = 6KB – success, no truncation • Add details > 6KB – success, but truncation • Add randomly generated data < 6K – success • Add i18n data
TC7	7	Can't add attachment	• Simulate temporary lost network connection – error message that file was not attached • Simulate database problem - error message that file was not attached • Remove file from the file system - error message that file was not attached
TC8	8	File too large	• Add file = 5MB – file is attached • Add file > 5MB – get message that file is too large
TC9	9	Object that owns the attachments has been removed	• User 1 - Starts to attach a file, User 2 deletes Artifact Type A object. User 1 get message that item has been deleted by another user • User 1 - Starts to add the detail, User 2 deletes Artifact Type A object. User 1 get message that item has been deleted by another user

Table 4 Test Case Conditions Matrix

Automating Test Case Generation

These steps can all be done manually but it may be time consuming and error prone for complex systems with larger numbers of use cases. A tool such as Rally[7] provides the ability to calculate all the scenarios in the use case automatically, selectively generating just the high priority test cases and thereby achieving significant savings in time and effort. Testers are then free to focus on the higher value testing functions such as defining the appropriate data sets, and even more critical to the success of the iteration, automating the test cases that have been defined. Figure 5, provides an example of how Rally[7] captured the nine scenarios of the example use case and provides the tester with the ability to set priorities to determine which scenarios should have test cases generated.

Scenarios for UC1091: Attach Supporting Information

Business Priority: All | Technical Risk: All | Frequency: All | Testing Priority: All | Test Case: All | Go

ID ▲	Scenario Name	Business Priority	Technical Risk	Frequency	Testing Priority	Test Case
1	Main Success Scenario	Critical	Medium	Medium	Critical	yes
2	b	Useful	Low	Low	Useful	no
3	b + 16:3a	Useful	Low	Low	Useful	no
4	3a	Critical	High	Medium	Critical	yes
5	3a + 3a4a	Useful	Low	Low	Useful	no
6	5a	Critical	Medium	Medium	Critical	yes
7	6a	Important	Medium	Low	Important	yes
8	6b	Important	Low	Medium	Important	yes
9	6c	Important	Low	Low	Important	yes

Figure 5 Rally Generated Scenarios

Automated Use Case-Based Acceptance Testing

Teams can choose to execute acceptance test cases manually or write automation code for testing, with automation being the preferred approach. Crispin[4], for example, devotes just one page to manual testing with the admonishment “no manual tests”. For those teams that have struggled to get their test cases automated, this can be a daunting proposition. Teams new to agile methods often establish an initial rhythm of in-iteration acceptance testing by writing the tests and executing them manually within the iteration. While this may indeed be a significant step forward, it can also be a trap. For when it comes time to run your regression suite of tests, all these accumulated manual tests will have been quietly adding up. Executing many hundreds or even thousands of tests manually will become painful and time consuming and *will likely be the limiting factor on the team’s ability to deliver code in small increments*.

Avoiding this accumulated “future testing debt” is extremely important and teams should resist the urge to push off automation into other iterations or to a later time when they think they might be able to get to it. Manual testing debt will only keep increasing and iteration velocity will be inevitably slowed.

In order to achieve in-iteration automation of your acceptance tests, tasks for writing the test automation must be scheduled just like development tasks within the iteration. Experienced agile teams have testing responsibilities that are shared across the entire team, including the developers, and this can be a cultural and or organizational challenge for many teams.

In addition, products such as FitNesse that provide a framework for automating acceptance tests are becoming increasingly popular. Being Wiki-based, it can be used by all team members to support the automation effort. (For more information, see the FitNesse website at http://fitnesse.org/.)

Conclusions

Software development practices must keep up with the rapid changes that are occurring in business today. Faster time to market, smaller team sizes and outsourcing options all place pressure on product teams to make sure they deliver the right product to the market at the right time. Applying agile software development methodologies is one key to success. The agile, iterative and incremental approach facilitates real-time feedback from customers that can be used to more rapidly evolve a solution to meet the needs of the market. Smaller, faster iterations and releases also provide a software management mechanism to more quickly implement what can be achieved while simultaneously providing the visibility to know when issues need to be addressed.

Agile, early, just-in-time testing is the key to an effective agile process. By bringing testing forward in the lifecycle, teams can better assure that the delivered functionality is suited to its intended purpose and that the solution meets the requisite quality and performance criteria. The use case technique facilitates early agile testing by providing a user-focused sequence of events that can serve as a template for in-iteration acceptance testing activities. Harvesting of the use case scenarios for test case development further accelerates acceptance testing. Combining agile practices with use case-based acceptance testing will help your company achieve the *agile edge*.

References

1. Beck, Kent, Test-Driven Development By Example, Addison-Wesley, 2003
2. Cockburn, Allistair, Writing Effective Use Cases, Addison-Wesley, 2001
3. Cohn, Mike, User Stories Applied For Agile Software Development, Addison Wesley, 2003
4. Crispin, Lisa, House, Tip, Testing Extreme Programming, Addison Wesley, 2003
5. Larman, Craig. Agile & Iterative Development – A Manager's Guide. Boston, MA: Addison-Wesley. 2004.
6. Leffingwell, Dean, Widrig, Don, Managing Software Requirements: Second Edition: A Use Case Approach, Addison Wesley, 2003
7. Rally Software Development Corporation: Agile Knowledge, Coaching and Tools www.rallydev.com

Using Metrics to Change Behavior

John Balza
Hewlett-Packard Company
johnjbalza@comcast.net
john.balza@hp.com

Abstract:

In the last several years, the Enterprise UNIX Division (EUD) of Hewlett-Packard has made several transformations in how it develops its software. Each of these transformations has required EUD engineers' and manager's to change their behaviors. Our experience is that because people do what is measured, metrics can be a key element in causing behavioral change. This paper describes how we've taken our standard metric design process and applied it to metrics intended to change people's behavior.

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.

Organizational Change

Organizations are often faced with the need to transform their processes in order to be more effective, more efficient, or better serve their customers. IT professionals are often required to adapt their behavior to fit software engineering process changes. At the Enterprise UNIX Division of Hewlett-Packard, we've found that one of the most effective tools to encourage behavioral change is to create a metric that tracks its progress. People tend to change their behavior if they know they are being observed [1], so a monitored metric accelerates the change in behavior. However, metrics need to be carefully designed to ensure that they encourage the intended behavioral change and minimize risks of undesired behaviors.

Change Management Roles

The most important roles in our behavioral change management process are:

- Champions – individuals who desire the change and attempt to obtain commitment and resources for its implementation.
- Sponsors – managers who communicate the need for change, authorize the change, and assume ownership for change implementation.
    - An authorizing sponsor possesses sufficient organizational power to authorize the resource commitment.
    - A reinforcing sponsor reinforces the change at the local level.
- Agents –people that implement the change.
- Targets –people that are expected to exhibit behavioral changes, emotions, knowledge, etc.

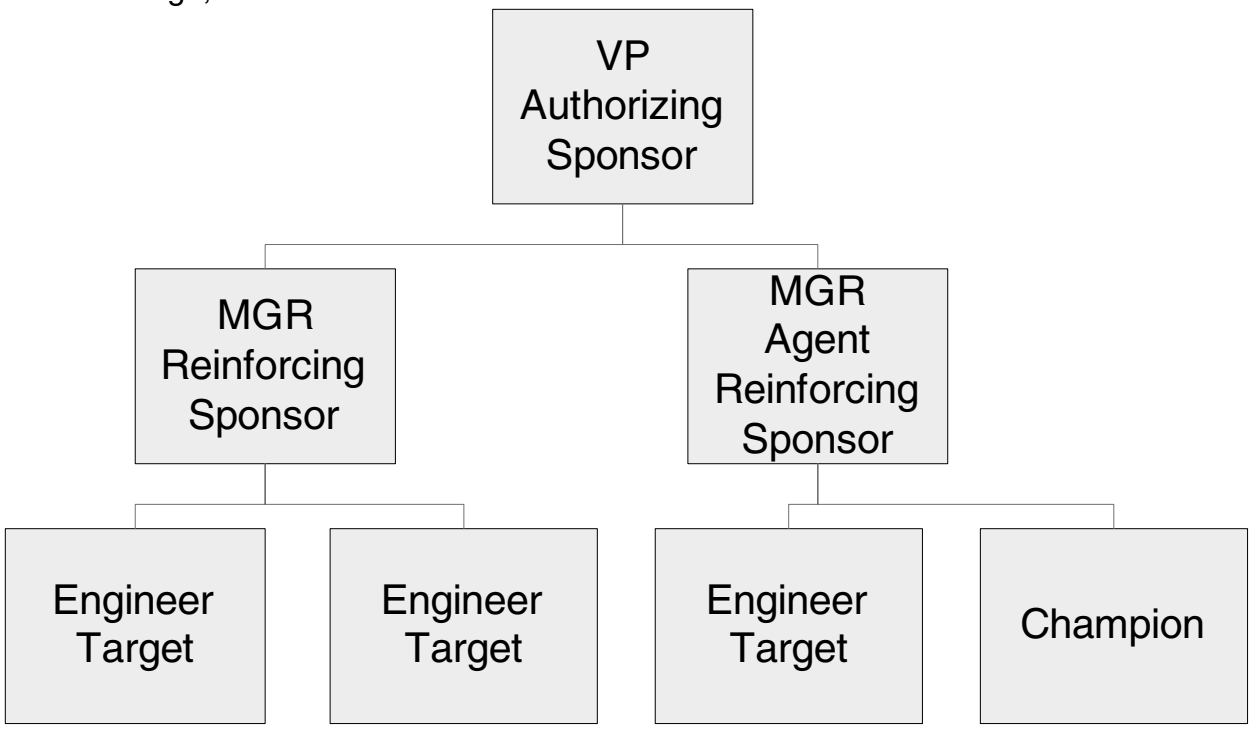


Figure 1 –An example of change management roles in an organization.

In Figure 1, the Vice President (VP) is the authorizing sponsor. He has two managers (MGR) who are reinforcing sponsors. In this case, one of those managers is also responsible for implementing the change, so she is a change agent as well as a reinforcing sponsor. The change champion can be almost anywhere in the organization, but typically reports either to the authorizing sponsor or to a change agent. In this paper, it is assumed that the champion is also responsible for designing the metric. The ideal situation is that the authorizing sponsor is the manager of the entire organization undergoing the transformation. This ensures that the new behaviors are reinforced throughout the entire organization.

The authorizing sponsor has a number of unique responsibilities:

- If a champion does not already exist, the authorizing sponsor identifies a change implementation champion.
- The authorizing sponsor is responsible for authorizing expenses required to cover actual work and the management overhead.
- He/she is responsible for articulating the business case for the change.
- The authorizing sponsor defines the goal for the organizational change and how he/she will know that the change has been successful. This will be a key input into formulating the supporting metrics.
- Finally, the champion and the sponsor agree on a progress review schedule.

Reinforcing sponsors have the following responsibilities:

- Communicate the need for change in terms their organization will understand.
- Promote the change within their organization
- Authorize expenses and resources in their department required for the change
- Monitor the metric within their organization
- Ensure that the reward and recognition system promotes desired behavior and discourages adverse behavior

In large organizations like ours you may need several layers of reinforcing sponsors. One of the most common mistakes made is to neglect to use the mid-level managers as reinforcing sponsors. These managers are the best people to deliver the message of why this change is needed to their people/staff.

Goal – Question – Metric

V.R. Basili and his associates created a concept called Goal-Question-Metric [2] for designing metrics. The basic paradigm is that each organization or project has a set of goals. For each goal there is a set of questions that help you understand whether you are meeting the goal. Then a set of data can be associated with every question to answer it in a quantitative way. We've modified that approach somewhat to address organizational change.

First, the champion works with the sponsor to articulate the organization's goal. As in designing any goal, this goal should follow SMART criteria:

- Specific, state what is to be achieved, clear to anyone in the organization.
- Measurable, can be measured, not qualitative or subjective.
- Attainable, within the power of the organization to attain, but also a stretch.
- Relevant, relevant to the people responsible, related to customer or business needs.
- Time bound, has a time frame for being achieved.

Next, the champion works with the sponsor to define a practical success/fail criterion such that he can determine if and when the goal is achieved.

To increase the probability that the change will be implemented successfully, it's important that people buy into the change and feel that they have some ability to mold how the change is achieved.

After we obtain the sponsor's goal and how the sponsor will measure whether the goal has been achieved, it's time to find a representative group of the targets who will be affected by the change. You are looking for people who have influence in their organization. It's often best to find a mix of people both for the change and against the change. They provide a balanced working group for creating the metrics.

The group refines the goal into several questions that break the goal into its major components. It's important to develop two sets of questions. One set answers whether we've met the goal. What the sponsor said about how he would know whether the goal was achieved can often be used to determine whether we've met the goal. The second set helps us understand if we're making progress toward the goal. The answers to these "progress" questions should help the management team decide whether they need to take corrective action in order to meet the goal. We usually start by identifying (i.e. via brainstorming) and then sorting the initial list of questions in these two categories. We then review the list to refine the questions. A good set of questions should provide different information – each from a different perspective.

The team then begins proposing what metrics could we develop that will answer the questions. Just like we had two sets of questions, we'll have two sets of metrics. The metrics that report whether we met the goal are called 'result' metrics. The metrics that measure progress toward the goal are called 'predictive' metrics. [5] Predictive metrics provide early indications of what the result metrics will measure. Predictive metrics are designed to provide information about the current state necessary for management to take corrective action. For example, a result metric like "increased customer satisfaction", would be associated to a predictive metric like "incident response time". The predictive metric will change concurrent with the behavior change.

Goal-Question-Metric, an Example

The Enterprise UNIX Division at Hewlett-Packard is responsible for developing our version of UNIX™, called HP-UX. This product consists of over 16 Million lines of source code and is developed by a team of several hundred engineers in 7 geographic locations. This product has a major release every 2-3 years and minor releases every 6 months. Several years ago, our division manager wanted to both increase our product quality and our productivity. His personal experience as a development manager had

convinced him that the best way to improve both quality and productivity was to find defects earlier in the product lifecycle. This agrees with much of the literature on the subject. The cost of finding and fixing a defect increases significantly the later in the development cycle it is found [3, 4]. The situation at the time was that most defects were being found through testing by other development teams or system testers. The behavioral change he desired was that engineers would find most of their defects either through inspection or developer testing.

The manager's key goal was 'To deliver our products with increasing quality and productivity'. When asked how he would know the goal had been met, he answered, 'Customers would find fewer defects in our products and we would find more defects in peer review'.

We then convened a small team to develop the questions and metrics to assist in this behavior change. The questions to determine whether we had met the goal were fairly simple:

1) For this product version, did customers find fewer defects than in the previous version?
2) For this product version, did peer reviews find more defects than in previous version?

But neither of these questions would help us determine whether we were making progress toward the goal. In particular, we wouldn't have the customer defects until after a version had been shipped. The questions we developed to measure progress toward the goal were:

1) How many defects remain in the product (asked before shipping)?
2) Where in the product lifecycle are defects being found?
3) What type of verification or validation activities are finding defects?
4) Are the lowest cost defect-finding activities finding most of the defects?

Each of these questions has a different view of the goal – they provide different information for managing toward the high quality goal.

The next step was to brainstorm with the team some metrics that might answer these questions. In our example, we generated the following list:

- Number of defects found in the first year by customers (we chose a 1 year time frame in order to compare with industry benchmarks)
- Number of defects found by peer review
- Backlog of defects outstanding
- Number of defects found in each lifecycle phase
- Number of defects found by activity

We then reviewed and trimmed down the list to the most relevant metrics. The first two metrics were good result metrics and matched what the sponsor said he would use as his measure of success. Our problem was to find a reasonable predictive metric for both quality and productivity. The "backlog of defects outstanding" might be able to predict resultant quality, but couldn't predict productivity very well. The "number of defects found by phase" metric also didn't answer the productivity question very well because in later life-cycle phases defects could either be found by system testing (an

expensive activity) or by peer review (an inexpensive activity). We decided that the "number of defects found by activity" could better answer whether we were finding defects in the most cost-effective manner.

Designing the Metric

After creating a set of potential metrics, it is time to optimize the set and design the metric using a number of tests. These tests can be applied to any metric:

- Adverse behavior – what behaviors might the metric encourage that you don't want to see. This is probably the most important aspect of validating a metric that is designed to change behavior. Think through the other behaviors that people might adopt to meet the metric. The metric needs to be designed so that the metric will not respond positively to unwanted behavior. For example, if you're measuring defect backlog, there is a tendency to hide defects, so they don't get counted in backlog. There may be some adverse side behaviors that have negative consequences. For example, we wanted to encourage 'classification' of customer defects within 30 days. The metric we first used calculated the (number of defects classified before 30 days/all defects that became 30 days old this month). But this metric caused an adverse behavior that once you missed the 30 day window; people ignored classification of the defect, since it would never show in the metric again. For example, if you missed classifying a defect that should have been classified in July, it shows in the July metric, but gets ignored in August. We changed the metric to the (unclassified defects over 30 days old/all defects over 30 days old). Now any unclassified defect that is older than 30 days gets counted in the metric.
- Good performance –Many measures give interesting information, but it's not clear what 'good' would look like. For example, a metric that is just hours of peer review per document isn't a very good metric. Since document size and type varies wildly, it's impossible to define a 'good performance' number. Is spending 4 hours reviewing a document, good or bad?
- Usefulness – put yourself in the future, if you were analyzing these measurements, is the data useful? As part of assessing usefulness, you may need to consider the time period covered by the metric. Will there be enough data points during that period of time or do you need to remove the effects of cyclic activity? For example, you may have to measure a rolling 3 month period, instead of a one month period to get enough data points or to remove the affect of an activity that occurs once per calendar quarter.
- Breadth of coverage – will the metric answer more than one of the questions?
- Ease of collection – is the data easy to collect?
- Accuracy - knowing all metrics have some degree of imperfection, will this one accurately answer the question?
- Normality - will the metric work at several layers of the organization? This often means that the data has to be expressed as a fraction to normalize it in some manner. For example, dividing number of defects by number of development engineers or lines of code change results in a metric that works on small and large projects.

- Credibility with key targets – will the audience for the metric believe that the metric tells them what they want to know?
- Clarity – is the metric easy to communicate and understand?

There is an interesting trade-off between ease of collection and clarity versus accuracy. Metrics can get quite complex in order to be completely accurate. It's often better to choose a simpler metric even if it has some degree of inaccuracy. My rule of thumb is to choose a simpler metric that is easier to collect or explain if it's within 10% of the accurate answer. This may require hand calculating the metric using historical data both ways and determining whether the simpler metric is 'close enough'. I communicate this rule of thumb to others; since it stops a lot of arguments about the inaccuracy of the metric.

Designing the Metric, an Example

Let's continue with the example above by seeing how we designed the metric based on "number of defects found by activity". Following the methodology, we applied the following tests.

- Adverse behavior. Our largest adverse behavior was that engineers would over-report peer review and developer-found defects to get credit for them. But that was fine with us since our processes required them to fix anything they reported. If they fixed more defects, customer quality would be improved.
- Good Performance. What would be good performance? It was impractical to say that all defects would be found by peer review. We decided to model good performance after one of our highest quality products. In the life of that product, 70% of the defects were found through peer review, well over 90% by the combination of peer review and developer test, and less than 1% discovered by customers. Capers Jones [6] had also suggested that a 'best in class' organization would have only 1% of their defects escape to customers in the first year. So we defined good performance as greater than 70% found by peer review, greater than 90% found by the combination of peer review and developer test, and less than 1% found by customers.
- Usefulness. First, we realized that to be useful it had to be computed during the development of the product. Waiting for a year after the product was released would not be useful to managing the process. So we decided we would measure the defects found by all activities including customer found defects over a period of time. Second, we needed to handle the reality that there would be more peer reviews in the early phases of the lifecycle than in the testing phase. We decided to take a rolling 12 month period for the metric. Our typical projects were 6 to 18 months long; 12 months would be long enough to remove most lifecycle effects.
- Breadth of coverage – we had chosen this metric from the list of metrics because it would cover both productivity and quality. If we reduce the percentage of defects found by customers, we should also reduce the raw number of defects found by customers. Finding defects by less expensive activities should increase productivity.
- Ease of collection. This turned out to be a problematic area. Our first problem was that we did not have a good way for collecting data on the defects found by the design engineer during unit test. It was too cumbersome to enter every defect into

the defect tracking system. The engineer would spend more time recording a defect than fixing it. We looked at providing a periodic collection tool, which would ask how many unit test defects the developer had found. We decided that this would be inaccurate; most engineers would log into the tool once a day or once a week and guess how many defects they had found. In the end, we decided not to measure the unit test found defects, the value of the data did not seem worth the cost of data collection.

Our second ease of collection problem required us to rethink the metric. Our defect tracking system didn't require teams to describe the exact type of testing activity that found the defect. (It was an optional field.) It did require they enter what project team found the defect. This caused us to change the metric to 'number of defects found by particular teams', that is, the peer review team, the developer team, partners, and customers. We defined partners to be any other teams that were not working directly on this subsystem of the product. Partners also included the system test team.

- Accuracy. The accuracy of the metric to predict productivity had actually improved with the change from measuring activity to measuring which team found the defect. Our internal data showed that the cost of fixing a defect found by developer testing was 3 times the cost of fixing a defect found by peer review. Similarly, partner-found defects were 3 times the cost of developer-test-found defects. And, of course, customer-found defects were significantly higher (>10 times) than any defect found internally.
- Normality. To normalize the metric so that it could be applied both to small projects and large programs, we decided to measure the percentage of defects found by each team compared to the total defects found. By expressing the metric as a percentage it scaled nicely from projects to the entire division.
- Credibility with the targets. The largest credibility issue came from our conscious decision not to collect unit test found defects. The engineers felt that this was a large source of defects and excluding these defects would give an incomplete picture of where defects were being found. Since our primary focus was to get more teams to use peer reviews, we accepted this credibility issue, but we agreed that we would periodically revisit this decision.
- Clarity. We tested this metric on a number of managers to see if they understood it and to confirm whether they also believed that it would lead to increased productivity and quality. As a result of this test, we clarified our definition of developer-found versus partner-found defects, which at first were poorly defined.

The metric we finally designed was called Defect Finding Effectiveness. Its nickname became '70/20/9/1' after the goals set for peer-review-found, developer-test-found, partner-found and customer-found defects. It measures the percentage of defects over the course of the past year found by peer review, developer-test, partner-test, and customers for a given project or organizational unit.

Change Management

As a final check on a metric, there are a number of other tests that should be applied before implementing it to cause change. The following questions test the likelihood that the metric will cause the intended behavior change:

- Is this metric inconsistent with the goals of the organization?
- Are the encouraged behaviors inconsistent with how individuals and teams are rewarded?
- Are the encouraged behaviors inconsistent with the organizations culture?

Just having a metric in place, won't cause a change unless it is consistent with the culture and the rewards within that culture. When a metric or behavior change is inconsistent with the goals of the organization, the sponsors will need to spend more time and energy communicating that the goals have changed. If the new behaviors are inconsistent with the reward system, the sponsors need to change the reward system and communicate that change. If it's truly a cultural change, the process will take several years and require continual reinforcement by the sponsors. The champion needs to ensure that the sponsors understand the effort and commitment expected from each one of them in order to successfully implement the cultural change.

With any behavioral change, it is important to follow a change management process. The use of a metric to cause behavioral change requires additional considerations.

It is advisable to pilot the new metric to ensure it measures what we expect it to measure. Here's where reality may destroy your theory for the metric. In the pilot, the champion can test whether the metric encourages the desired behavior and see what other adverse behaviors occur. The pilot can also test whether the predictive metric triggers corrective action from management in a timely manner.

After the pilot you may want to keep the metric private for a period of time, that is, the affected organizations only see their own results. Results are not shared with authorizing sponsor until a target date. In my experience, this private period is important to the reinforcing sponsors. They want to show to the authorizing sponsor that they are making good progress on this change. This gives each organization time to use the metric to determine whether the targets are really exhibiting the new behavior. After the target date, we'll often show the data for when the metric was private because this shows how the organization has progressed on adopting the new behavior.

The final step is the review of the metric by the authorizing sponsor. The outcome is comments on good results or corrective actions for when results aren't good. This feedback should propagate down the organization where each of the reinforcing sponsors does the same review with their organization until the targets are reached.

Change Management, an Example

In our case, the target behavior change was consistent with the goals and culture of the organization. We already had a goal to improve product quality, and the organization was well aware that the division manager was convinced that peer reviews would improve our productivity. Thus the behavior change and metric were consistent with the current goals and culture. However, as part of this behavior change, we did need to change our rewards. In this case, the metric became part of every manager's evaluation.

A variation of this metric had been piloted earlier by the product team that had achieved the less than 1% customer escape rate. So we already knew that it caused the correct behaviors.

In our case the management goal was communicated in Quarter 3 of 2002, but the metric would not be made public or included in each manager's performance review until Quarter 1 of 2003. Between Q3 and Q1, the metric for each organization was kept private, only the overall number for the division was reported. This provided managers with two quarters to adopt the new behaviors and improve their performance with the metric.

Today, this metric is reviewed at several levels of the organization every quarter. The division manager reviews the overall division metric, lab directors review their labs' metric, and subsystem owners review their subsystem.

Conclusion

Figure 2 shows what happened at HP. We have seen a dramatic increase in the percentage of defects found by peer review and those found by developers (both peer review and test found). At the same time the percentage of defects found by customers has decreased. Most of the original improvement was just recording peer review data better than in the past, but now each organization is employing defect analysis to determine how to have fewer escapes to partners and customers.

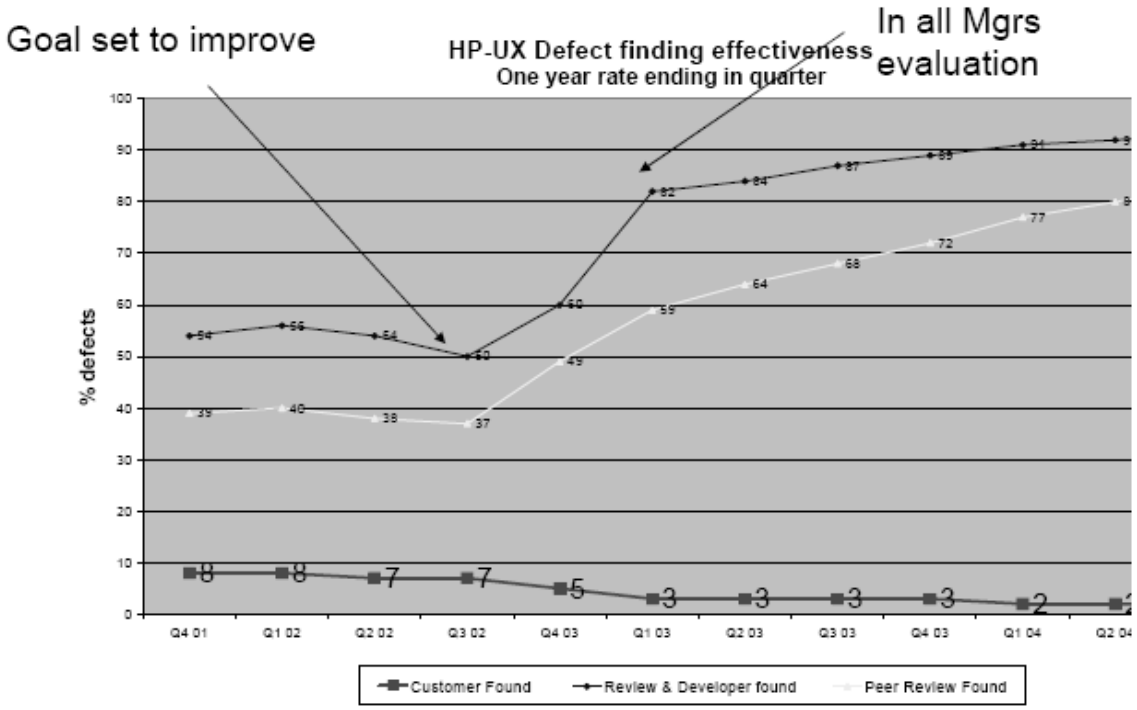


Figure 2 Defect Finding Effectiveness Results

The result metric, 'number of defects found by customers' has also improved dramatically. Customers are finding significantly fewer defects on our latest release compared to our earlier releases. As predicted by finding defects earlier in the lifecycle we have also improved productivity. The product engineering costs to fix customer found defects has been cut in half during this time frame, and our productivity has improved by 30%.

Now that the organization has met our goals of 70% peer review found and a total of 90% developer found, we're talking about what is the next step to improve our quality and productivity. Should we include developer unit test found defects in the metric? This would encourage finding more defects by unit test. We are now investigating our next steps, which may include a change to the metric.

We have successfully used this method on two other behavioral changes. In the first case we wanted to discourage engineers from postponing the fixing of defects in favor of adding new functionality. Existing engineering practices hurt our productivity because these defects could be blocking other teams from making progress. The change in behavior we wanted was that defects should be fixed first then you can add functionality. We put in place an open backlog metric to change this behavior. Our backlog now is very consistent; engineers fix defects as their highest priority.

In the second case, we needed to encourage communicating dates to customers for when a fix would be generally available to all customers. We put in place a metric that encouraged making these commitments within 30 days of receiving it and a second metric to ensure we delivered on the commitment date. At this point, we are consistently

meeting the 30 day goal, but are working on our estimation ability to improve the second metric.

We have found that metrics are a powerful tool to help accomplish a behavioral change in an organization. The key points to remember when implementing a metric to cause behavioral change are:

1) Identify the authorizing sponsor as well as all the reinforcing sponsors
2) Have all the sponsors clearly communicate the business need for the change to the targets (i.e. staff members).
3) Ensure that the metric will encourage the behavior change you desire, and not other adverse behaviors
4) Provide time when the metric is private to each organization, allowing them to show progress before information becomes public.
5) Have the sponsors regularly monitor the metric and putting in place corrective action when things are not progressing.

References

[1] Draper, S. W.: "The Hawthorne Effect: A Note," http://www.psy.gla.ac.uk/~steve/hawth.html(March 12, 2003), September 2003.

[2] Victor Basili, Gianluigi Cladiera, and H. Dieter Rombach; The Goal Question Metric Approach; Encyclopedia of Software Engineering. Wiley, 1994 http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/gqm.pdf

[3] Robert Grady; Practical Software Metrics for Project Management and Process Improvement; Prentice Hall, 1992, p.130

[4] Watts Humphrey; A Discipline for Software Engineering; Addison-Wesley, 1995, p.276-7

[5] Tom DeMarco; Controlling Software Project; Yourdon Press, 1982, p.54

[6] Capers Jones; Software Quality Analysis and Guidelines for Success; International Thompson Computer Press, 1997, p.44

New Tricks with Old Tools

By Kathy Iberle and Bret Pettichord

Abstract

This paper describes how a long-standing technique for analyzing business processes can be adapted to develop effective testing strategies for complex systems. We use Structured Systems Analysis to develop a model, not of a business process, but rather of the key structural elements of a software system under development. This model then serves as a basis for designing testing strategies that can accommodate the staggered delivery of components. This approach can also be used to design test strategies for systems that contain failure-prone modules and systems assembled from off-the-shelf components.

Biography: Kathy Iberle

Kathy Iberle is a senior software test engineer at Hewlett-Packard. Over the past twenty years, she has been involved in software development and testing for products ranging from medical test result management systems to inkjet printer drivers and Internet applications. Currently, Kathy researches and develops appropriate development and test methodologies and processes for different situations. You can contact her at www.kiberle.com.

Biography: Bret Pettichord

Bret Pettichord specializes in test automation and agile testing methods. He is a lead contributor to the Watir and Selenium open-source testing tools and a co-author of *Lessons Learned in Software Testing*. He is Director of Testing Practice at ThoughtWorks, a multinational consultancy providing IT services to Global 1000 companies. Visit his website at www.pettichord.com.

First published in the Pacific Northwest Software Quality Conference Proceedings, 2005

New Tricks with Old Tools

Introduction

Have you ever planned the testing of a fully assembled system, only to find that some components haven't been delivered yet and you can't get at the parts that are finished? This paper presents a method for working around the missing components. We use a diagramming technique borrowed from Structured Systems Analysis to reverse-engineer the system design, and then use the diagram to identify ways to test the parts of the system that are finished.

What Problem Are We Trying to Solve?

The Classic Approach to Testing Complex Systems

The classic approach to testing systems consisting of multiple components built by different teams is to test smaller pieces at lower levels first, and then integrate those pieces together, test again, and so forth until one reaches the system test level. The testing is usually divided into "test levels" or "test phases" such as unit testing, component testing, integration testing, and system testing. The number of levels and the types of testing to be covered in each level depend on both the architecture of the software being tested and the development or integration plan in use. Craig and Jaskiel, in *Systematic Software Testing*, [CRAI02] briefly discuss some of the variations seen in different projects and companies.

An alternative approach to integration and testing is the "big-bang" approach – slam everything together and test the assembled system. Repeat until satisfied (or until the ship date is reached). Essentially there's only a system test level, since the only testable item is the entire system.

A third approach employs a piece-wise integration like the first approach, but integrates features rather than pieces of software. Staged Delivery and the entire family of iterative lifecycles generally follow this approach. Early versions of components are integrated together to create a system with a small feature set, and those features are tested. Later versions with more features are integrated, and the system is tested, and so testing continues throughout the project.

All these approaches share one key assumption – an assumption that is not necessarily valid on many of today's development projects. **These approaches assume that all the development teams are following a jointly developed plan for integrating and testing the system.**

- In the first approach, there is a systematic, hierarchical integration plan. All the participating development teams have agreed to test and fix their parts before integrating them, and they have agreed on an orderly integration plan. The overall test plan is designed around this integration plan, to test the results of each planned integration in a manner appropriate to a piece of that size.
- In the big-bang approach, all the teams have agreed to integrate their pieces at one pre-determined point in the schedule, at which time testing will start. There usually is some agreement on the level of maturity or finished-ness expected at this point.
- The third approach requires a detailed implementation and integration plan that is carefully orchestrated across teams to ensure that all the bits of code required for an individual feature will appear at the designated build for that feature. The test plan is designed around the feature-release plan.

In all three approaches, the development can be viewed as one large project or as a set of coordinated sub-projects.

Reality Strikes

In many situations, one of these three approaches is used. The multiple-team hierarchical approach is common in fields requiring very high-reliability software, such as telecommunications, aerospace, and medical products. The other two approaches are used in many fields, especially on smaller projects.

However, in a number of fields today, it is increasingly common to assemble large software systems out of disparate parts that come from different organizations on loosely coordinated schedules. The various suppliers may be using different lifecycles and have quite different ideas about the test coverage expected of them prior to delivery. A feature may be implemented by the sum of several software modules or components which are not available at the same time or at the same level of maturity. The integration plan is not organized into a simple hierarchy.

Response to Reality

Testers faced with non-hierarchical or unplanned integrations can't use the classic test levels, because this approach assumes that the component suppliers have agreed on the test levels applicable to each supplier and on the coverage expected at each level. Likewise, the iterative approach can easily fail when the integrations aren't coordinated thoroughly enough to make packets of user-visible features reliably "go live" at any particular point in time. The testers generally fall back on acting as if the big-bang approach has been used, even if there are multiple integrations spread over time.

In a big-bang project, the testers generally rely on testing against what the end-user can see - typically user requirements, functional requirements, and non-functional requirements.[1] Testing is usually organized around features. There is little or no testing that is targeted specifically at the software design, and often the testers don't have much visibility into the design.

Unfortunately, individual development teams often assume that the test team is intentionally testing their component, when in fact the test team is testing the feature most closely associated with that component. If the architecture is well modularized and lightly coupled, testing the feature may test the component pretty thoroughly, but more commonly, there are multiple other components involved in implementing the feature. When those components are delivered piecemeal in varying states of completion, we very often find that the incompleteness of a less-visible component is blocking testing of a feature, and thereby preventing testing of the other components which are involved in that feature.

Consider an inkjet printer. The user is notified when the ink cartridges run low on ink, when the paper jams, and numerous other conditions.

This functionality is implemented by several different components:

- Hardware – senses ink level, senses a paper jam
- Firmware – interprets the hardware signals and sends them to the computer
- Operating System – receives signals and hands them to a status monitor
- Status Monitor – reads the signals and pops up dialogs to inform the user of the printer status

From a software perspective, the status messages are a feature of the Status Monitor. However, the feature doesn't work until the hardware *and* the firmware *and* the status monitor have all implemented

---

[1] In this paper, we use Karl Wiegers' terminology for requirements: *User requirements* represent what the user wants the system to do. *Functional requirements* specify what the developers intend the system to do, once implemented. *Non-functional requirements* or *quality attributes* describe other important characteristics such as usability, portability, reliability, and so forth. [WEIG02]

status messaging support. Using a black-box approach, the software that pops up the status message for paper jams can't be tested until the sensing for paper jam is working properly. This means the testing of the status monitoring software basically can't make any progress until the hardware and firmware are nearly complete. The status monitor development team may not find this situation acceptable.

A Solution

A test team faced with this sort of problem needs to figure out how to test individual components on their own rather than relying solely on testing the entire assembled system. Yet it's not possible to tell a program that all the teams involved have to rethink their entire development approach and undertake a completely hierarchical collaborative integration plan. It would be much more practical to do just enough lower-level testing to work around the components that are likely to be delivered late. However, this requires some basic understanding of how the system works.

In short, a high-level design is needed. Often, the high-level design isn't available on paper – either there is very little design in writing, or there are reams of detailed information on each component but no overall high-level diagram. The tester needs to reverse-engineer the high-level design. We have found that a technique from Structured Systems Analysis can be very useful in reconstructing the high-level system design.

What is Structured Systems Analysis?

Structured Systems Analysis is a modeling technique that was originally created to analyze and accurately describe business processes in order to automate them. The "system" in Structured Systems Analysis is any "connected set of procedures (automated procedures, manual procedures, or both)." [DEMA79], generally some sort of business process. Structured Systems Analysis is a process for producing diagrams that are concrete, readily understandable specifications of the process under study at several levels of detail.[2]

Structured Systems Analysis was popular in the 1980s. The description of structured analysis used in this paper is taken from *Structured Analysis and System Specification* by Tom DeMarco [DEMA79].

Why Use Structured Systems Analysis?

Structured Systems Analysis works for reverse-engineering a high-level design because a computer system and a business process are quite similar at this level - they both consist of loosely coupled, dissimilar activities which provide information to each other. When we're reverse-engineering a system design, we need to elicit a description of a complicated system from the people most familiar with that system, which is exactly what Structured Systems Analysis does. An added advantage is that the Structured Systems Analysis notation is designed to be easily understood without special training, so the people most familiar with the system can easily review and correct the description whether or not they know Structured Systems Analysis.

[2] Using today's modeling languages, the process might be drawn as a swim lane diagram, and the user requirements captured with a combination of use cases and business rules. The Structured Systems Analysis approach bears a strong resemblance to a process map.

How to Use Structured Systems Analysis

The Structured Systems Analysis process is pretty straightforward:

- Use structured analysis questioning techniques to elicit information from developers.
- Record the findings in a readily understandable form – the simple graphic notation known as a "data flow diagram".
- Review the system design diagram with developers and correct as needed.
- Use the design diagram to identify ways to work around missing parts of the system – identify what you can test and what is needed to enable that testing on an unfinished system.

We'll work through an example using these steps.

Sue and the Acme Automated Grocery Checkout Stand

Sue is the lead test engineer for the software project Gecko, which is part of the Reptile program. Sue's team has been newly assigned to this program. She sets out to find out what this "checkout stand" is all about.

First, Sue locates the Project Data Sheet. Like most Project Data Sheets, this one bubbles over with enthusiasm and optimistic predictions.

Project Reptile

Product Summary: Reptile 3.0 is the newest, most advanced automated grocery checkout stand. Reptile lets shoppers scan and bag their items themselves, totals up the cost, and allows the user to pay for it via cash or credit card. The average grocery store will save millions of dollars in labor costs per year by making a small investment in this state-of-the-art easy-to-use and very reliable system.

New features: Reptile 3.0 will be able to process produce and other items that are sold by weight.

Teams:
Lizard: scale for weighing produce
Caiman: barcode scanner
Vincent: credit card reader and cash acceptor
Gecko: software

Sue starts out by establishing what a checkout stand is, and is not, using Structured System Analysis techniques.

The Level 0 Data Flow Diagram

The usual starting point for systems analysis is to consider the entire system or product as "inside" and everything else as "outside". Structured Analysis calls this viewpoint the Level 0 Data Flow Diagram (DFD) or Context Diagram.

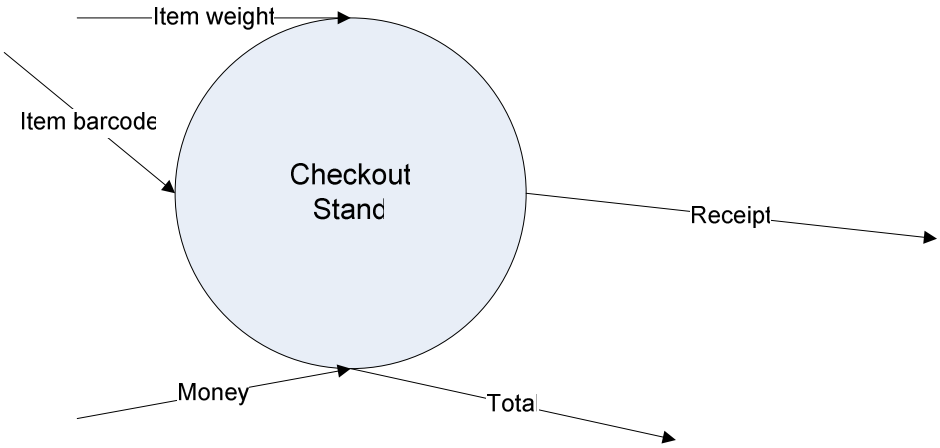
The rules for drawing a Level 0 DFD are very simple.

- Draw one bubble for the process
- Draw all the data that comes into it or out of it. Don't forget the user as a source or destination for data.
- Label all the arrows with the name of the data.

Sue starts by sketching out the Level 0 DFD for using the checkout system, based on the Project Data Sheet.

She identifies input: barcodes and item weights. Oh yes, and money.

She identifies output: a total (otherwise the shopper wouldn't know how much money to give the machine) and, though the data sheet doesn't mention it, there must be a receipt.



The Level 0 DFD establishes the inputs and outputs. This is the same point of view taken by a pure black-box test. The system has inputs, and outputs, and whatever goes on inside is a mystery. So far, Structured Systems Analysis hasn't told Sue anything that she didn't already know.

Sue could stop at this point and create the test plan without understanding anything more about the internals of the checkout stand. A simple black-box test approach calls for applying input and checking the output – this would translate to scanning, bagging, and paying for various collections of items. However, Sue is suspicious that this straight-forward approach is going to run afoul of the multiple development teams. She's already hearing that the team working on the scale is running behind. Sue decides that it would be prudent to know a bit more about how the system works. She sets out to establish the basic system architecture.

Drawing the Level 1 DFD

In structured analysis, the analyst discovers the workings of an existing process by asking questions of the users, drawing a Level 1 DFD, and then asking the users to confirm the accuracy of the diagram. Sue uses the same process to discover the workings of the checkout stand by questioning the developers.

The notation:

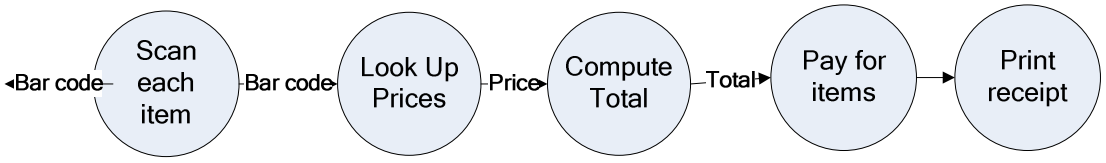
- The bubble: a person or system doing something. Labeled with a **verb**.
- The labeled arrow: data traveling from one bubble to another. Labeled with a **noun**.
- The straight line or cylinder: a file, database, or other permanent data store.

The rules:

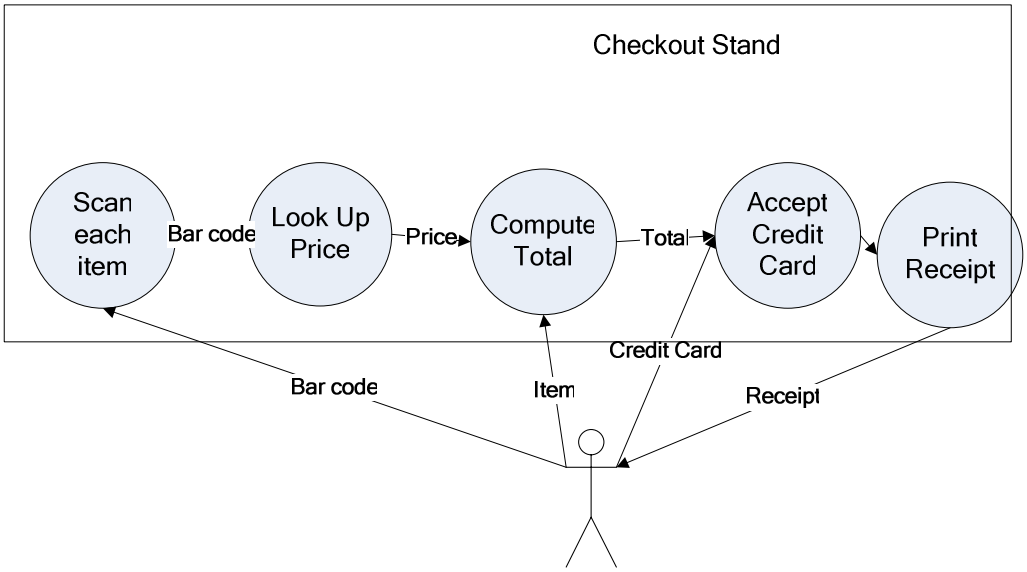
- The data flow diagram always shows flow of data, not of control.
- Decision points are not shown.

Sue interviews Kent, the system architect. Her first sketch is just to establish what the system does.

Sue: "What happens first?"
Kent: "The shopper scans each item."
Sue (draws a bubble for scanning items): "Then what?"
Kent: "The checkout stand looks up the price."
Sue (draws a bubble for looking up the price): "Then what happens?"
Kent: "Well, just keep scanning things until you're done, and then the system computes the total."
Sue (draws a bubble for computing the total): "And then what?"
Kent: "Then the shopper pays for the items and gets a receipt."
Sue busily draws the last few bubbles.
Sue: (showing this picture) is this what happens?



Kent: "Uh, the system doesn't pay for items. The shopper pays".
Sue: "Okay, so what goes into the system when the shopper pays?"
Kent: "You gotta put some money in, or a credit card, or a debit card."
Sue: (erase and redraw) "How about this? Now the checkout stand is doing all the bubbles, and there's a stick person for the shopper".



Sue's first drawing didn't distinguish between the user and the computer system. In her second drawing, the user appears as a stick figure[3] and there's a box is drawn around the system as a whole. Notice that the information going into the box and coming out of the box is the same as the information going into and out of the bubble in the Level 0 diagram. The Level 0 diagram can usually be skipped, but if there are questions about the input and output, it's sometimes worthwhile to draw the Level 0 diagram because the data in and out is more obvious in the Level 0 form than in the Level 1 form.

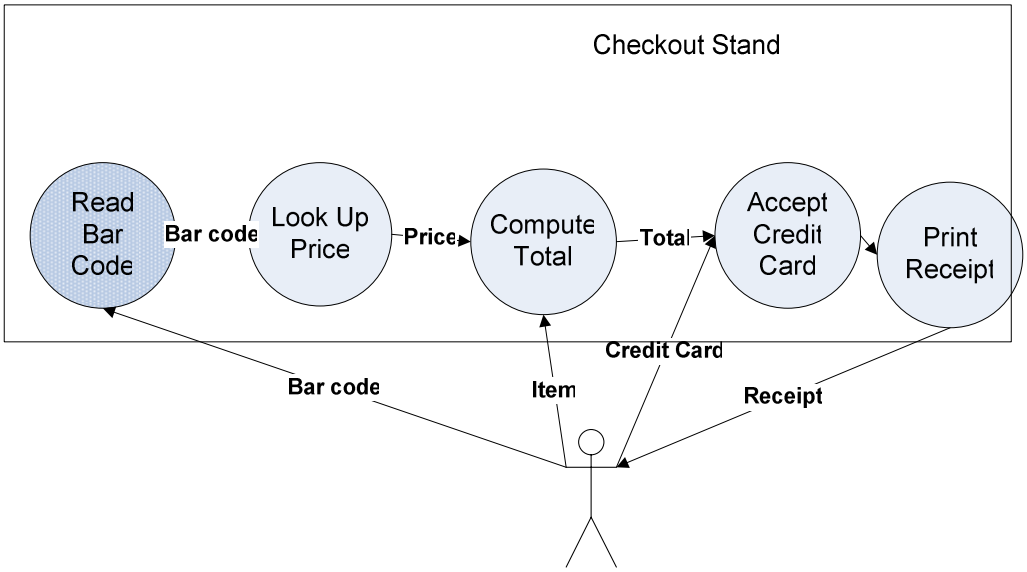
Anything displayed to a user – numbers, error messages, pretty pictures – is output because it goes out of the system to a human being, who is not part of the system. Likewise, anything that a user communicates to the system – mouse clicks, typed inputs – is input. Generally it's more useful in the diagram to depict mouse clicks as the logical input, such as a menu pick, rather than just showing "mouse click".

Note that Sue drew the use case where the shopper pays with a credit card. She didn't try to draw a use case where the shopper pays with cash, because this would involve more output (the shopper's change). When you're trying to understand the basic structure of the system, it's often easier to draw the system needed for the simplest use case first.

Establishing the System Structure

Sue considers her diagram. It's pretty obvious that the software inside the stand isn't divining the bar code via ESP. There must be a physical sensor, and some firmware that runs the sensor and passes numerical data to the software proper. Sue redraws the diagram, showing the hardware/firmware tasks in a different color. She goes back to talk to Kent again to confirm her speculations. She also invites Yumiko, the technical lead for the team that's writing the main processing software.

[3] The stick figure isn't actually part of the original Structured Systems Analysis notation. In the 1980s, the diagrams were drawn by hand, and so the symbols were kept to a minimum.



Kent and Yumiko look over the diagram.

Yumiko: "Why aren't you using UML[4] for this? This isn't a UML diagram."

Sue: "UML doesn't really have a diagram equivalent to this. An activity diagram is similar, but that's got timing and control flow shown which makes it harder to draw and harder to understand. So I use this notation because it's easier."[5]

Yumiko: "Well, OK, I guess we can do it this way."

Kent and Yumiko mull over the diagram for a bit.

Yumiko: "That Look-Up-Price bubble isn't right. Really there's a database, and the data-retrieval dll calls the dbinf dll, then that calls up the database and gets the price."

Kent: "And the Compute-Total calls the database too, to get the sales tax rules."

Sue: "Those dlls are running on the processor inside the Reptile checkout stand, right? And the database is outside the checkout stand somewhere?" (Sue draws the database outside the box)

Yumiko: "Yes. The database isn't part of the Reptile project, really."

Kent: "Where's the produce handling? I don't see where you can enter the SKU. And we've got the scale for weighing the produce – that's not there either."

Sue: "Oh dear, you're right." (busily adding bubbles to the diagram) "There's not any hardware keypad, so the software must be providing the keypad for entering the SKU, is that right?"

Yumiko: "Yes."

Sue: "And is there a way to buy produce like three for a dollar?"

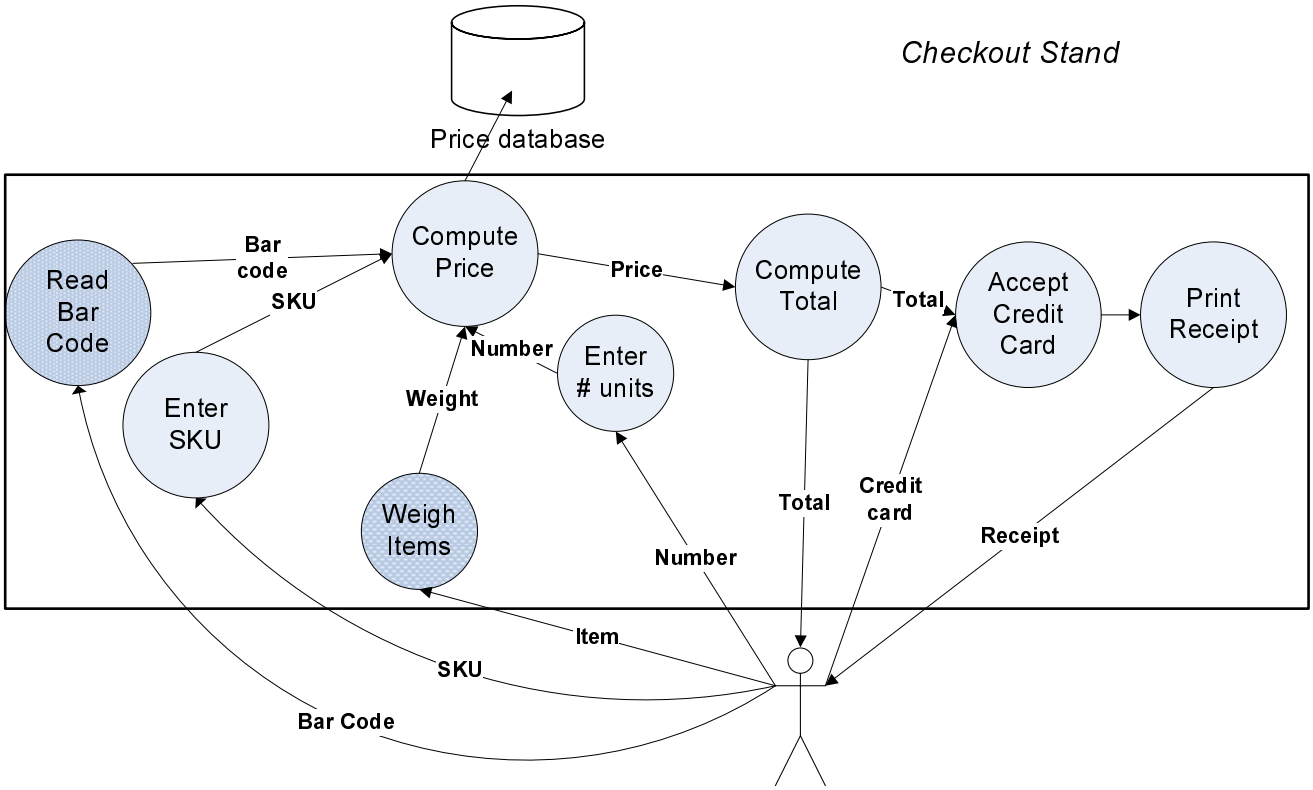
Yumiko: "Yes, there's a screen for entering the number of items." (pulls a piece of paper out of a pile on her desk) "Here's a workflow diagram – it shows the screens and how they flow into each other."

Sue revises her diagram again.

---

[4] Unified Modeling Language

[5] Also, understanding data-flow diagrams doesn't require being familiar with UML.



A few things to note:

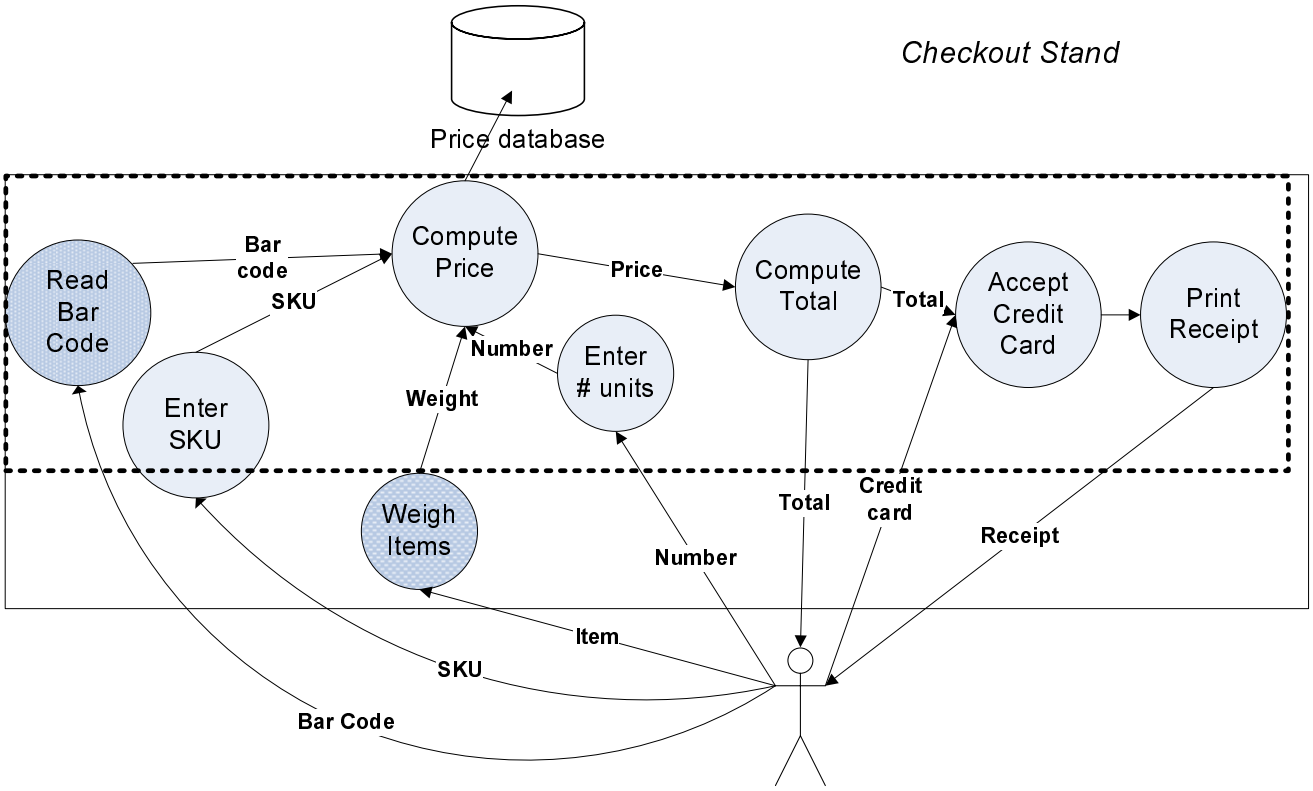
- The database isn't considered to be part of the project, so it's drawn outside the automated checkout stand
- The developers mentioned some dlls by name, but Sue didn't draw them. The data-flow diagram is intentionally kept at a high level of abstraction, often higher than the actual software architecture. The actual software design is often too complicated to help in planning testing at this level.
- The workflow diagram showed that some of the bubbles contained several different screens. In DFD notation, the activity inside a bubble is shown on a separate sheet as a Level 2 DFD. Sue doesn't go to Level 2 at this point because she doesn't see a need for it, although she keeps the workflow diagram to use later in writing test cases.
- The DFD could be simplified by merging some of the bubbles, but Sue didn't do that. For instance, Read-Bar-Code and Enter-SKU could logically be represented as a single bubble Enter-Identifier. Sue hasn't merged those two bubbles because those parts of the system are delivered by different teams, and that might affect the test plan.

The Test Plan

At this point, Sue has enough information to begin laying out a test plan for her team's software. She sets two people in her team to writing test cases based on the functional specifications for the screens. Another person arranges for builds to be received by her team and assembles a test system.

Shortly, one of her staff shows up at Sue's desk. The team delivering the scale for weighing the produce has fallen behind, and as a result the test system can't weigh any produce. The produce handling is a new feature, and therefore one of the features they'd planned to test first. What should they do?

Sue sits down with her diagram. She draws a line isolating the offending scale from the rest of the system. She considers first testing only the items inside the box, then adding in the scale and completing the testing. The early testing would provide information to the software development teams much earlier and thus help the project proceed.



How can they run the system without the hardware and firmware represented by Weigh-Item?

Divide and Conquer: Test Scaffolding

If Sue can find a way to feed weights to the system without the scale, then the routines for pricing produce by weight can be tested without waiting for the bar code reader. Her team needs some *test scaffolding*.

There are several ways to build test scaffolding. Some of them require help from the development team, and others can be carried out by the test team.

USE OLDER SOFTWARE

On projects that are building on pre-existing code, it's often possible to replace an unfinished piece of the system with the older, known good version of that piece and continue testing the rest of the system. There are several drawbacks:

- Whatever bugs are still in the older version can show up during testing and obscure the results of testing the new code
- If the interface definitions have changed, using older software often won't work at all.
- The older software may not be able to supply all possible inputs to the rest of the system – the

new version may have an expanded range.
- People outside the test group often have trouble understanding that it's ok to use an old version of A if you're concentrating on testing B. This is particularly true in groups where all the testing up to this point has been at the system level, so they see no distinction between testing all the code in module B and testing the features in which module B plays a key part...

In this case, there is no older Weigh-Item software, so this isn't going to help.

WRITE DRIVERS AND STUBS

Another way of getting scaffolding is to write it. There are two basic types of test scaffolding, both of which are often referred to as *test harnesses*.

- A *test driver* or *test harness* calls the software under test and passes it the desired data. Typically a test driver will have some kind of simple interface to allow the tester to specify data, either on the command line or by supplying a text file.
- A *stub* is called by the software under test, to permit it to run in the absence of the stubbed-out software. Often stubs do nothing except return success.

Custom scaffolding avoids the disadvantages of using the older software as scaffolding. The main disadvantage is that the scaffolding has to be written. However, often developers have written some scaffolding for their own use during development and are willing to share that with the test team.

On a project with multiple development teams, the test department may find that they need to tell the developers what scaffolding is needed, because the test department is first to know which parts of the system are missing and need to be worked around. The scaffolding can be identified by looking at the DFD. Wherever a data flow crosses the line between the offending component and the rest of the system, there is data flowing either into or out of that component to another part of the system or to the user. These data flows need to be replaced by drivers or stubs. Sometimes both are combined in a single program, and other times there are several bits of code needed.

Sometimes the developers can't quite see how to create a test harness for a tricky bit of code. There is a good introduction to writing test harnesses in Steve McConnell's *Code Complete* [MCCO04]. Michael Feathers presents a variety of design patterns for getting test scaffolding around classes and methods in *Working Effectively with Legacy Code* [FEA05]. Feathers' book is particularly useful in our situation because it assumes that the developer is trying to put scaffolding in place in a few targeted spots, rather than assuming that all the code will have automated unit tests.

HIJACK THE INPUT

Input that is coming from any static data store – a file, registry table entries, a database - can easily be hijacked by changing the file or the registry table.

There is the obvious method of simply substituting one input value for another in the data store. For instance, when a program reads initialization values from an "ini" file, it's easy to change that file.

There's a sneakier method, as well, of hijacking the data midstream to force the system to respond as if a different initial input value had been supplied. For instance, we'd like to test the system with items that are priced $100.00 or more. But all the bar code stickers we have on hand are for items that are much less expensive. We can go into the Price database and change the price of a can of soup to $105.21. Then scan the can of soup, and the system is off and running to exercise the entire data path involving prices greater than $100.00.

The data-flow diagram can be used to determine whether there is any opportunity to hijack data by tracing the data-flow from the initial input to the item-under-test. If the data passes through any static data stores, then additional inputs to the item-under-test can be created by hijacking the data in the data store.

You can get pretty creative if there's a lot of information in static data stores. Sometimes it's possible to change processing rules as well as data to send the system off on the path of your choice. Usually a tester with command-line access to the system and perhaps a few key passwords can make these changes without any development help.

Incidentally, it's pretty common to find systems that include a database server which is not being changed at all and therefore is not considered to be under test. Sometimes the testers are expected to use the "production" database in the testing, and the clients are configured to call that production server. This eliminates the ability to set up interesting test data and to hijack data-flows. A "test" database server which can be freely changed by the test team will speed up testing the rest of the system. (This assumes you have a tester with the skills to change data in a database management system.) Whenever possible, get the developers to set up the client such that the test team can easily redirect the client to a different server.

Remember that any changes you make to static data stores are part of your tests. If you think there'll be a need to re-run the same test or a similar test, then you'll want to save the modified files, or scripts used to make the modifications, or instructions for making the modifications by hand.

Back to Sue and her Team

Sue has to decide which of the above patterns to try. She takes a careful look at the data-flow diagram to understand what part the missing piece plays in the system.

The "Use Older Software" pattern is the easiest to implement, but it's obviously not going to work because there is no older version of Weigh-Item.

The "Hijack the Input" pattern is usually the next easiest to implement, so Sue takes a look at her data-flow diagram. The data-flow diagram doesn't show the data from Weigh-Item going through any static data stores before entering Compute-Price, so apparently there's no way to hijack the input.

Sue thinks about writing drivers and stubs. The people who wrote Compute-Price can't have had the scale and associated firmware while they were writing Compute-Price, since it hasn't been built yet. But they presumably at least tried out the data path for computing the cost of items sold by weight. How did they accomplish this?

Sue goes over to talk to the developers.

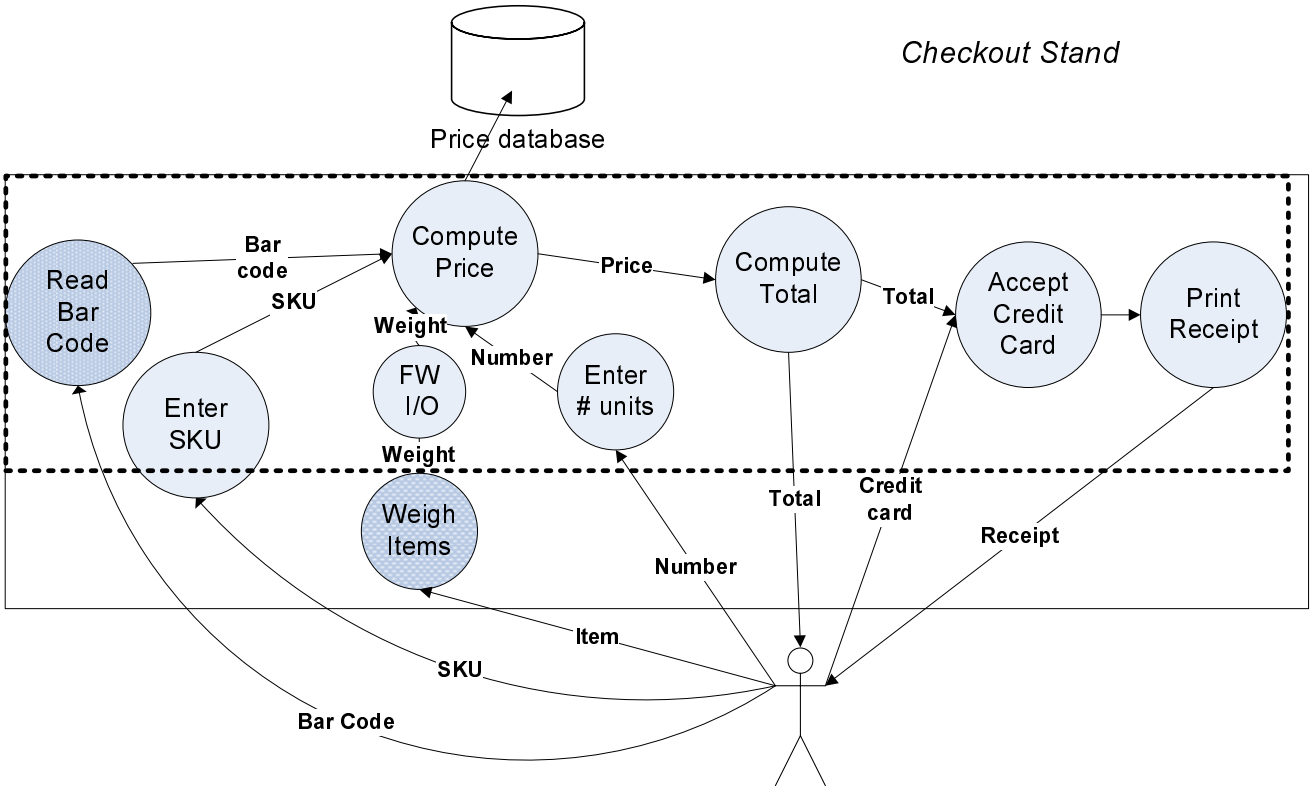
Kent: "Well, Compute-Price isn't talking directly to the firmware, of course. There's a class that interfaces with the firmware."

Yumiko: "I wrote a quick substitute class which pops up a dialog instead of calling the firmware, so I can type in the weight."

Sue: "Can we get that substitute class into the build until the firmware for the scale is ready?"

Yumiko: "I guess so."

Sue has to redraw her diagram again, but now it's clear that there is a way to get input into the system without the scale.



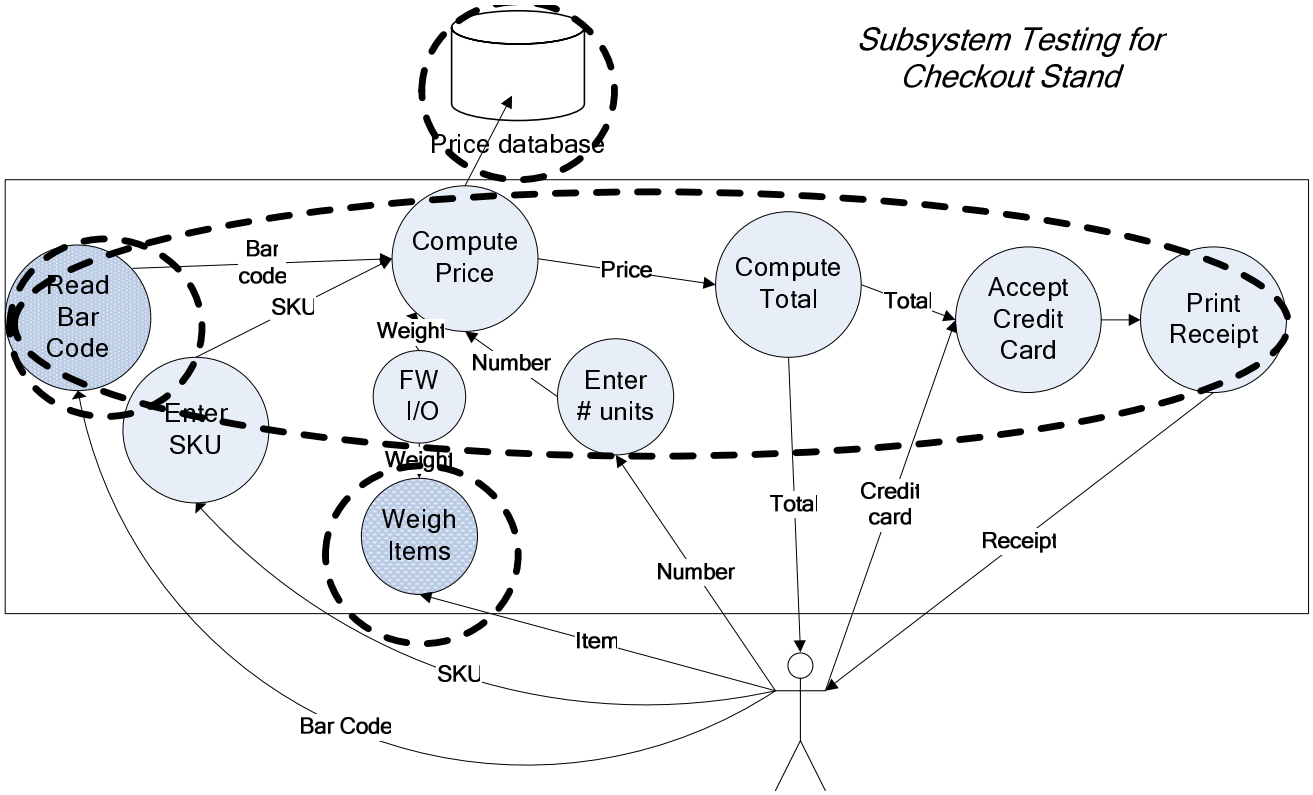
Defining the Test Phases

Sue returns to her test planning. She splits the testing into two parts – testing that will be run on the complete assembled system, and testing that will be run on the software minus the Weigh-Items firmware/hardware. Sue has just defined two test levels – a system test level, and a subsystem test level.

In a system with many parts, it's not unusual to find that there are multiple test efforts run at various levels of integration. Sometimes testing within one team is split into levels or phases, as Sue has just done. Often there is also testing done by some of the supplier teams. In the case of the automated checkout stand, the firmware teams are running their own testing using scaffolding, and the team supplying the Price database has done extensive independent testing.

Sue shows this partitioning in her master test plan by drawing circles around the items tested by each test set. There are four test sets at the subsystem level:

- Price database testing
- Bar code reader testing (firmware)
- Scale testing (firmware)
- Testing of system minus Weigh-Items



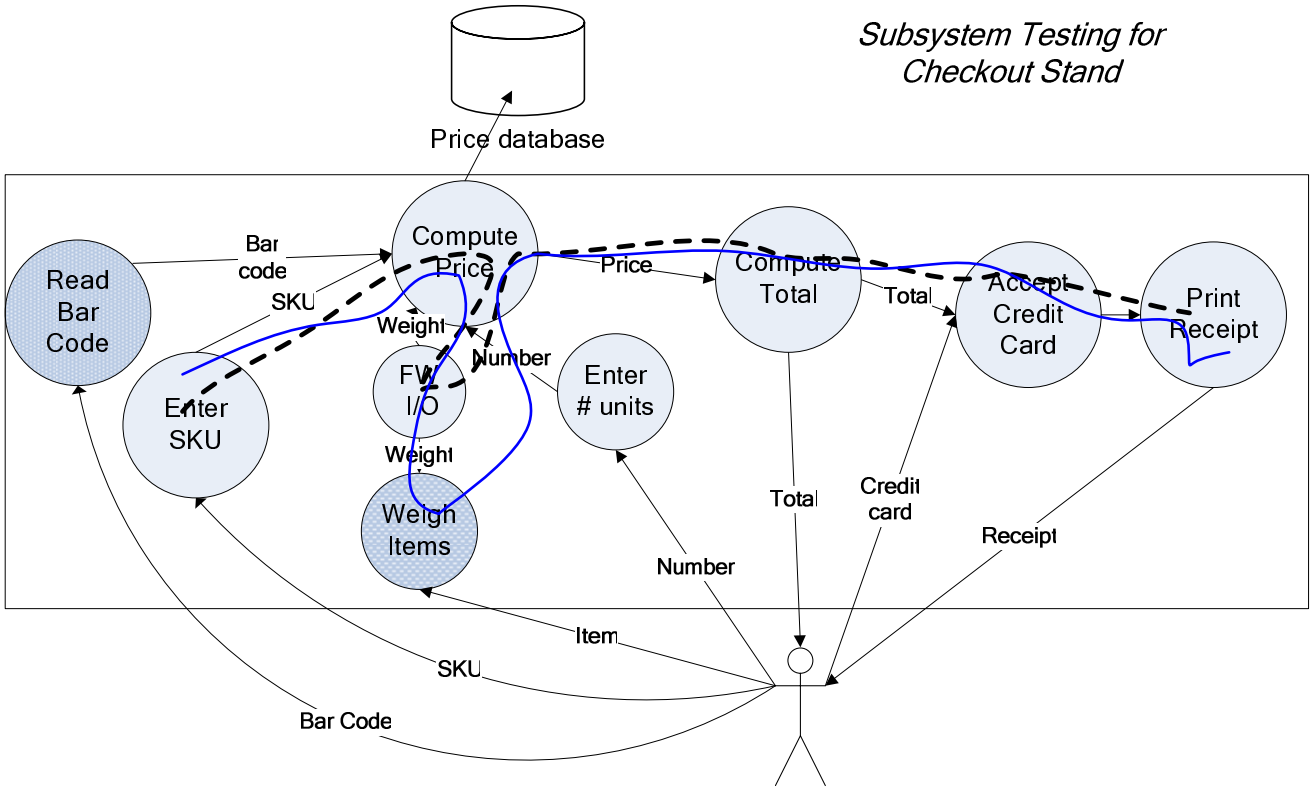
Some people would argue that the first three are a sub-subsystem or component level test, and the fourth is a subsystem level test. This is often a pointless argument, because many systems don't sort out neatly into a hierarchy, and the items-under-test often end up overlapping with each other. In such cases, a drawing of the partitioning usually communicates far more than classifying tests as "subsystem" or "component".

Sometimes the integration is relatively hierarchical, so testing can be split into the traditional system-subsystem-component test levels. Even in this case, if the parts are very different from each other, the test purpose and techniques at a given level may be considerably different for different components. For instance, the database testing and the firmware testing for the checkout stand probably don't resemble each other very much. This means that the content or purpose of "component" testing can't be defined much more precisely than "testing of an item that's bigger than a breadbox and smaller than the whole system". You may end up defining "component testing" as "testing of an item that is the output of a single team". Again the drawing can be much more communicative than a written description of the test levels.

Subsystem vs. System Testing

Sue and her team are able to test most of the functionality of the system with Weigh-Items missing. Once Weigh-Items does arrive, what still has to be tested?

One way to look at this is to consider end-to-end paths or execution threads. You can use the data-flow diagram to keep track of the paths by drawing them on the diagram in different colors or weights and labeling them.



- Black dotted line: test cases run during the subsystem test
- Blue solid line: test cases run during system test

There will be a variety of test cases that follow the black dotted line, using different inputs or making different choices along the way. Rather than simply repeating all these test cases on the fully assembled system, it's smarter to focus on the newly added piece and consider what could be broken. Heuristics for integration testing are usually useful. For instance, can the newly added item send input values to the rest of the system which were not used in the subsystem test?

The key interface here is the interface between the Weigh-Item firmware and the Firmware I/O software (which communicates with the firmware). It would be smart to test a range of weights, check the highest and lowest possible weights, and take a look at the expected precision. Somewhere in the scale or the firmware an event in the physical world is converted into a digital signal, which is then sent to the software. The software may well receive an unexpected input because something that wasn't planned for happened in the physical world. What happens if the harried shopper sets his toddler on the edge of the scale? What if you drop a heavy can on the scale from six inches up?

Other Uses for Data-Flow Diagrams

Now that Sue has solved the problem of the missing scale, and has her test plan in place, she doesn't have any more immediate need for the data-flow diagram. However, there are some other situations in which the data-flow diagram may come in handy.

Data-flow diagrams will sometimes make it obvious that a number of test cases are probably equivalent to each other.

Data-flow diagrams can be used to find defects by walking through the system design. I've more than once found significant design problems during the drawing of the data-flow diagram. I'll ask, "So how does this talk to that?", the answer will be rather vague, and I'll ask again. The developer will suddenly stop being ever so patient with my ignorance, stare abstractedly into space for a few minutes, and then announce that the system isn't going to work. This usually means I have to come back the next day to finish the data-flow diagram, as the developers are distracted by figuring out how to solve the design problem.

Another type of diagram similar to the data-flow diagram is the Deployment Diagram (also know as architectural diagrams, block diagrams, and parts maps). These generally are applicable to web applications but not to other types of systems. The Deployment Diagram is showing physical pieces of the system rather than tasks getting done, but it resembles a data-flow diagram anyways, because the system generally is implemented such that each physical piece does a different job. The operation of the system isn't entirely evident from the Deployment Diagram because the Deployment Diagram typically doesn't say what those jobs are, nor what data is being transferred from piece to piece.

Elisabeth Hendrickson [HEND02] shows how to derive a variety of test cases from a Deployment Diagram, many of them error recovery or stress tests. The thinking process described by Hendrickson should work just as well with a data-flow diagram as with a Deployment Diagram.

Conclusion

Data-flow diagramming from Structured Systems Analysis, despite being an old and hoary technique, is a handy tool to have in one's testing toolbox. This is especially true for systems that are constructed out of disparate parts in a decentralized (or even uncoordinated) project or program, because data-flow diagramming helps the test planner figure out how to test partially finished systems whether or not the integration plan is a neat traditional hierarchal plan.

References

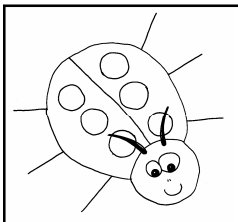
[CRAI02]: Craig, Rick D. and Jaskiel, Stefan P; *Systematic Software Testing*; 2002
[DEMA79]: DeMarco, Tom; *Structured Analysis and System Specification*; 1979
[FEA05]: Feathers, Michael C; *Working Effectively with Legacy Code*; 2005
[FOWL04]: Fowler, Martin; *UML Distilled, 3rd edition: A Brief Guide to the Standard Object Modeling Language*; 2004
[HEND02]: Hendrickson, Elisabeth; "A Picture's Worth a Thousand Words"; Software Testing and Quality Engineering Magazine; Sep/Oct 2002' pp. 26-32
[MCCO04]: McConnell, Steve; *Code Complete, 2nd ed.*; section 22.5 Test Support Tools
[WEIG03]: Wiegers, Karl; *Software Requirements, 2nd edition*; 2003;

Deciding What Not to Test

Robert Sabourin
AmiBug.Com, Inc.

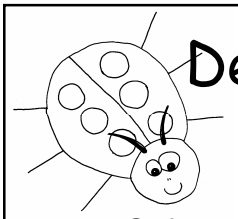
Software project schedules are always tight. There is not enough time to complete planned testing. Don't just stop because the clock ran out. This presentation explores some practical and systematic approaches to organizing and triaging testing ideas. Testing ideas are influenced by risk and importance to your business. Information is coming at your from all angles - how can it be used to prioritize testing and focus on the test with the most value? Triage of testing ideas, assessing credibility and impact estimation can be used to help decide what to do when the going gets tough! Decide what not to test on purpose - not just because the clock ran out.

Robert Sabourin has over 20 years of management experience leading teams of software development professionals to consistently deliver projects on-time, on-quality, and on-budget. He is a well-respected member of the software engineering community who has managed, trained, mentored, and coached hundreds of top professionals in the field. He frequently writes and speaks to conferences around the world on software engineering, SQA, testing, management, and internationalization.



Deciding What Not to Test

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



Deciding What Not to Test

- *Software project schedules are always tight. There is not enough time to complete planned testing. Don't just stop because the clock ran out.*
- *This presentation explores some practical and systematic approaches to organizing and triaging testing ideas. Testing ideas are influenced by risk and importance to your business. Information is coming at your from all angles - how can it be used to prioritize testing and focus on the test with the most value? Triage of testing ideas, assessing credibility and impact estimation can be used to help decide what to do when the going gets tough!*
- *Decide what not to test on purpose - not just because the clock ran out.*

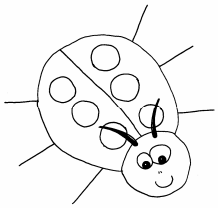
Deciding What Not to Test

- Robert Sabourin , *Software Evangelist*
- President
- AmiBug.Com Inc.
- Montreal, Quebec, Canada
- rsabourin@amibug.com
- www.amibug.com

Yoda

"No! Try not, Do. Or do not.
There is no try."

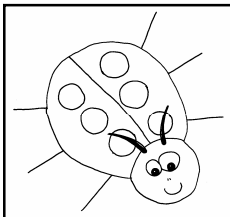
Plan to support change



Testing Ideas

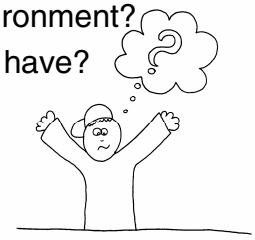
- Collect all testing ideas you can find!
  - List
  - Sort
  - Organize
  - Shuffle

Plan to support change

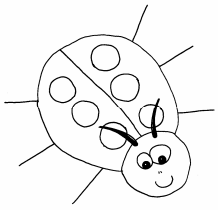


Testing Ideas

- How to find them?
  - Does system do what it is suppose to do?
  - Does the system do things it is not supposed to?
  - How can the system break?
  - How does the system react to it's environment?
  - What characteristics must the system have?
  - Why have similar systems failed?
  - How have previous projects failed?



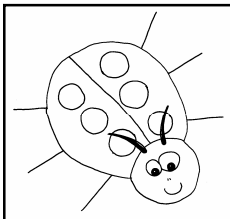
Plan to support change



Testing Ideas

- Collect testing ideas
- From testing ideas build a series of testing objectives
  - Each can be assigned *as work* to a tester
  - Each can include *all, part of, or multiple testing ideas*

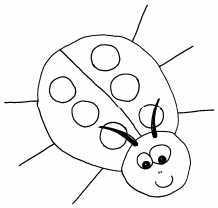
Capture testing ideas



Testing Ideas

- I often use *Index Cards*
  - Unique id
  - One testing idea per card
  - Colour indicates source
  - Shuffled and reviewed
  - Organized and reorganized
  - Sorted, grouped, prioritized and collected

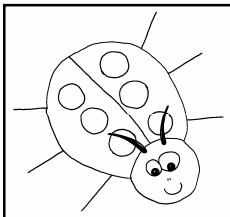
Capture testing ideas



Testing Ideas

- Creative approaches
  - Action verbs
  - Mind Maps
  - Soap Operas
  - Lateral Thinking

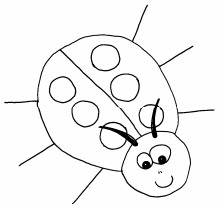
Capture testing ideas



Testing Ideas

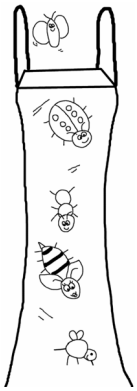
- Investigative approaches
  - We become truffle snorting pigs and try to find useful information in all evidence we discover
  - We can even get good ideas from out of date sources

Capture testing ideas

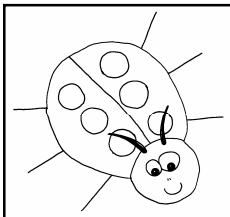


Testing Ideas

- Bug taxonomies
  - Collections of possible bugs
  - Appendix A of *Testing Computer Software, Kaner, Falk, Nguyen*
  - Boris Biezer Taxonomy Otto Vinter manages
  - Shopping cart taxonomy Giri Vijayaraghavan



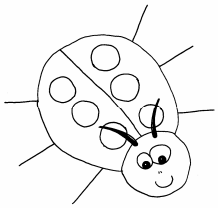
Capture testing ideas



Testing Ideas

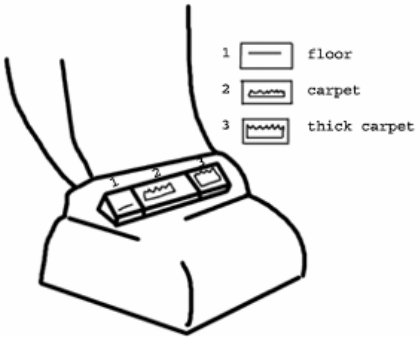
- Requirements
  - Use cases
  - Functional requirements
  - Quality factors
  - Constraints
  - Written requirements
  - Implicit requirements

Capture testing ideas

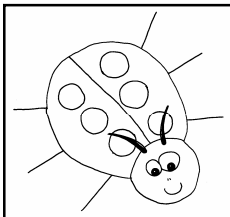


Testing Ideas

- Usage Scenarios
  - Identify classes of users
  - Identify how users will use system
  - Describe scenarios
  - Use Story board or similar approaches
  - Identify variations



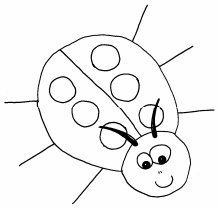
Capture testing ideas



Testing Ideas

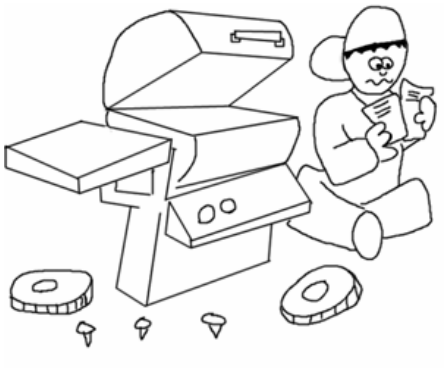
- Functionality Analysis
  - Requirements, Design or Prototypes can give insights into
  - Domain Analysis
  - Equivalence classes
  - Boundary analysis
  - CRUD

Capture testing ideas

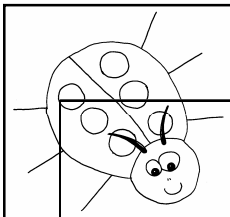


Testing Ideas

- Failure Modes
  - What can break?
  - Reaction to invalid input?
  - How does software behave in constrained environment?
    - Memory
    - Disk Space
    - Network Bandwidth
    - CPU capacity
    - Shared resources
  - Stress, Load, Volume



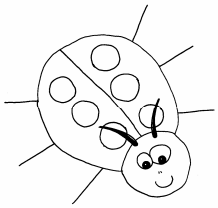
Capture testing ideas



	Adaptability	Accessibility	Auditability	Availability	Continuity	Dependability	Expandability	Functionality	Integrity	Interoperability	Maintainability	Operability	Portability	Reliability	Re-usability	Scalability	Security	Serviceability	Testability	Usability
Application service provider																				
Automatic content generator																				
Customized access																				
Database access																				
Delivery																				
Document access																				
File sharing																				
Informational																				
Interactive																				
Transaction oriented																				
User-provided content																				
Workflow oriented																				
High Focus																				
Medium Focus																				
Low Focus																				

Quality Factors Importance
Different Application Types

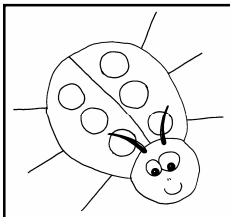
Capture testing ideas



Testing Ideas

Capture testing ideas

- Oracle Collection
  - Strategies to assess correctness
  - Similar systems
  - Old systems
  - Subject matter experts
  - GURUs
  - Standards



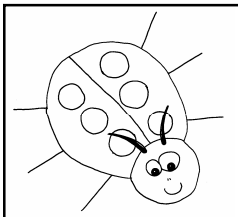
Which test?

Triage testing ideas

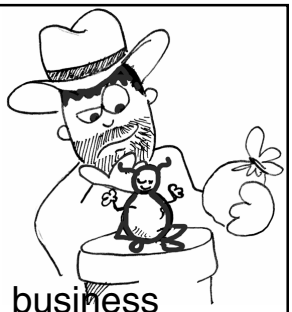
- *Impact estimation*
  - For each test idea guesstimate:
    - benefit of implementation
    - consequence of implementation
    - benefit for not implementing
    - consequence of not implementing
  - How credible is the information?

How to Decide?

Rank	Credibility
0.0	Wild guess, no credibility
0.1	We know it has been done somewhere
0.2	We have one measurement somewhere
0.3	There are several measurements in the estimated range
0.4	The measurements are relevant to our case
0.5	The method of measurement is considered reliable
0.6	We have used the method in-house
0.7	We have reliable measurements in-house
0.8	Reliable in-house measurements correlate to independent external measurements
0.9	We have used the idea on this project and measured it
1.0	Perfect credibility, we have rock solid, contract- guaranteed, long-term, credible experience with this idea on this project and, the results are unlikely to disappear



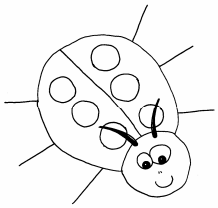
Which test?



Test Idea Rejection – What If?

–If the cost/benefit does not make business sense then consider implementing:

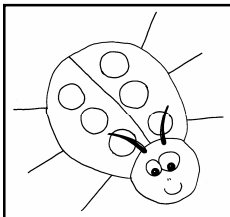
- *part of the test*, could that lead to part of the benefit at a more reasonable cost?
- *more than the stated test*, would that generate more benefit?
- a different test than the stated idea, could that generate more benefit for less cost?



Test Triage

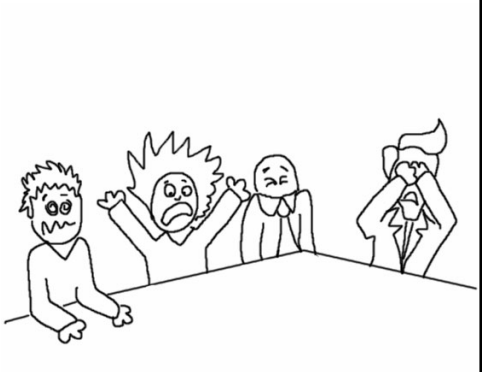
- Test Triage
  - JIT Projects
  - High Frequency
  - Daily Test Triage Session
  - Experience dictates
    - Early AM (Rob Preference)
    - Late PM (several clients)

Triage testing ideas

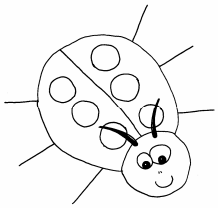


Test Triage

- Test Triage Meeting
  - Review Context
    - Business
    - Technical
  - Information since last triage
    - Test results
    - Bug results
    - New testing ideas

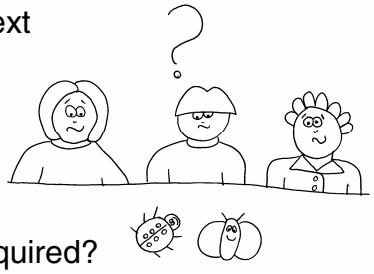


Triage testing ideas

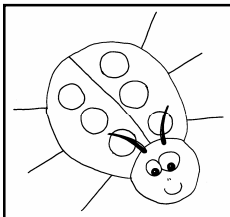


Test Triage

- Allocate Testing Assignments to Testers
  - Make sure testers know context
  - Best thing to test
  - Best person to test it
  - Best people to explore it
  - Best lead
  - Are subject matter experts required?
  - Blend scripted & exploratory testing

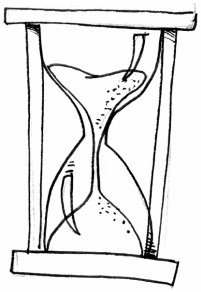


Triage testing ideas

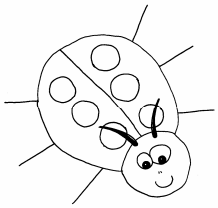


Test Triage

- Requirement Triage
- Change Control
- Test Triage
- Bug Flow
  - Combined
  - Equivalent to CCB
  - Few people
  - Fluid



Triage testing ideas

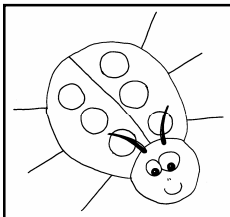


Test Triage

Life of a test idea

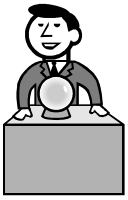
a. Comes into existence
b. Clarified
c. Prioritized
    a. Test Now (before further testing)
    b. Test before shipping
    c. Nice to have
    d. May be of interest in some future releas
    e. Not of interest in current form
    f. Will never be of interest
d. Integrate into a testing objective

Triage testing ideas

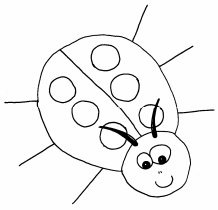


Which test is next?

- Questions
  - *Given state of project, state of business, state of technology, our abilities, our experience and our history, what we know and what we do not know, what should we test next?*
  - *How much effort are we willing to spend continuing to test this project?*
  - Can we ship yet?



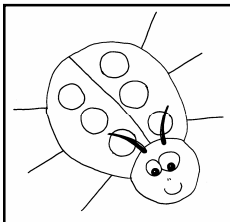
Triage testing ideas



Which test is next?

- Magic crystal ball
  - If it existed how would you use it?
  - What question would you ask?
  - What question would it ask?

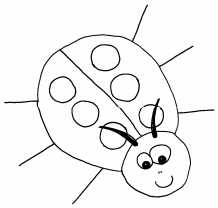
Triage testing ideas



Deciding what not to test?

- Time pressure
  - Should we skip a test?
  - If test failed could system still be of value to some stakeholder?
  - If test was skipped could important bugs have been otherwise found?

Triage testing ideas

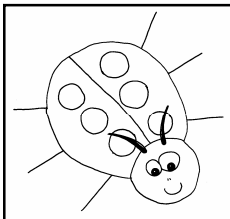


Problem with formulas

```
Exposure(i) = Risk * Consequence
Allocation(i) = (Exposure(i)/Total Exposure) *
                MAX
```

- Units?
- Math?

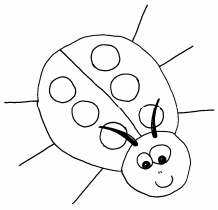
Triage testing ideas



Guidelines and Decisions

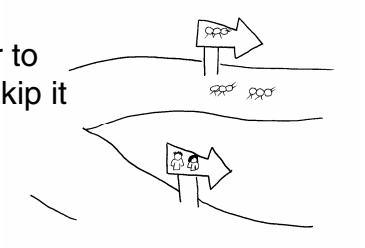
- To each stakeholder
  - risk of failure
  - consequence of failure
  - value of success
  - how much certainty do we have
  - is it a wild guess or an absolute truth?

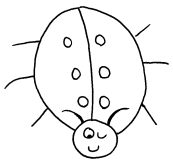
Get Started Right

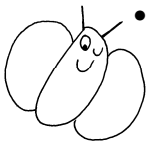


Bottom Line

- *My experience* is that it is better to omit a test on purpose than to skip it because you ran out of time or forgot about it!

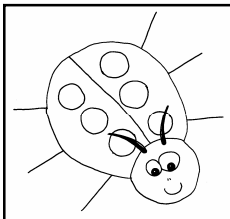






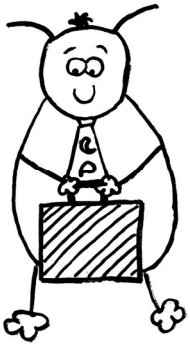
- Systematically collecting, evaluating and triaging testing ideas helps me decide what not to test - *at least for now?*

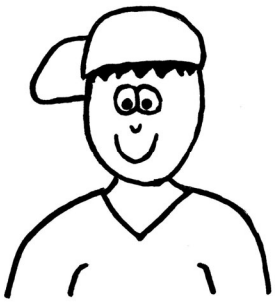
Get Started Right



Thank You

- Questions?





The Value-added Manager: Five Pragmatic Practices

Esther Derby
Esther Derby and Associates

What do great managers do that others don't? Great managers focus their efforts, increase productivity, and develop people. In this session, I'll talk about five pragmatic practices that can improve both results and work satisfaction. We'll look at ways to keep people in the loop, deliver results faster, and increase value to the organization.

Great management isn't easy; it's not rocket science either. Apply these five practices consistently to improve value to the organization (and keep your sanity, too).

Esther Derby is one of the rare breed of consultant who blends the technical issues, and the managerial issues with the people-side issues. She is well known for her work in helping teams grow to new levels of productivity. Scrum implementation, retrospectives, and project assessments are three of Esther's key practices that serve as effective tools to start team transformation. Recognized as one of the world's leaders in retrospective facilitation, she often receives referrals asking her to work with struggling teams. Esther also coaches technical people who are making the transition to management and is a Certified Scrum Master. Esther's articles have appeared in Better Software (formerly STQE), *Software Development, Cutter IT Journal,* and *CrossTalk*. She writes regular columns for stickyminds.com and computerworld.com, and publishes the quarterly newsletter, insights. Esther has an MA in Organizational Leadership and has worked in the software industry for over two decades.

The Value Added Manager

Esther Derby
esther derby associates, inc.
www.estherderby.com
derby@estherderby.com
612-714-8114

 1

The Nature of Management Work

- Managers
  - set direction
  - take a system view of the team or group
  - leverage the work of others
  - develop the capability of individuals and the group

 2

Five Pragmatic Practices

1. Decide what to do and what *not* to do
2. Limit multitasking
3. Keep people in the loop
4. Provide feedback
5. Develop people

 3

Decide What to Do …and What *NOT* to Do

- Base priorities on the mission and goals of your group
- Determine what you can accomplish given priorities and the capacity of the group
- Make a not-to-do list
- Transfer or cancel projects that don't support the goals of the group
- Fully staff projects that support the goals of the group and don't staff those that don't

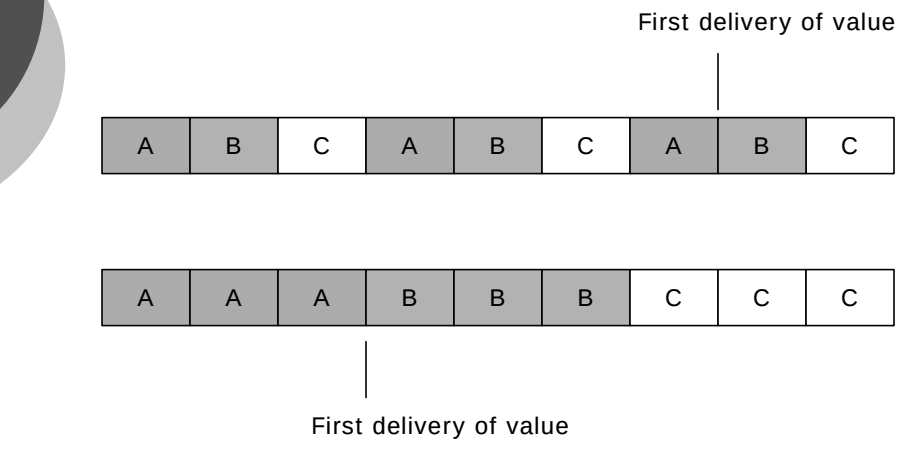
 4

Limit Multitasking

- Understand the affects of multitasking
- Assign people to complete the most important work expeditiously

 5

The Effects of Multitasking on Task Completion



 6

Keep People in the Loop

- One-on-one meetings
  - Greeting
  - Progress and status
  - Obstacles
  - Help
  - Career development
  - Review actions (yours and theirs)
- Team meetings
  - Gossip, rumors, news
  - Issue-of-the-week
  - Review action items

 7

Provide Feedback

- Feedback is *information* not *evaluation*
- Feedback aims to change future behavior
  - Encouraging
  - Change requests
- Feedback messages
  - Create an opening
  - Describe the behavior or result
  - State the impact
  - Make a request
  - If necessary, describe consequences

 8

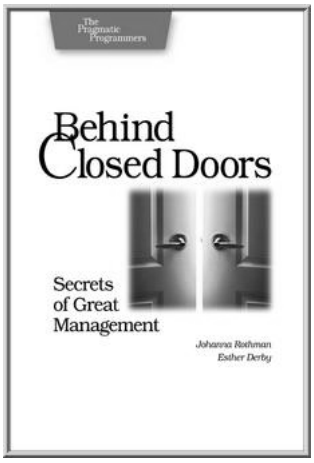
Develop People

- Understand aspirations and career goals
- Delegate where it makes sense
- Create development opportunities within day-to-day work
- Track progress weekly
- Let people know when the work you have doesn't support their goals

Resources and References

- DeMarco, Tom. *Slack: Getting Past Burnout, Busywork, and the Myth of Total Efficiency*. New York: Broadway Books, 2001.
- DeMarco, Tom, and Timothy Lister. *Peopleware: Productive Projects and Teams, Second Edition*. New York: Dorset House Publishing, 1999.
- Hill, Linda A. *Becoming a Manager: How New Managers Master the Challenge of Leadership*. New York: Penguin Group, 1992.
- Katzenbach, Jon R., and Douglas K. Smith. *The Wisdom of Team: Creating the High-Performance Organization*. New York: HarperCollins Publishers, 1999.
- Rothman, Johanna and Esther Derby. *Behind Closed Doors: Secrets of Great Management*. Raleigh: The Pragmatic Bookshelf, 2005.
- Seashore, Charles, Edith Seashore, and Gerald M. Weinberg. *What Did You Say? The Art of Giving and Receiving Feedback*. Columbia, MD: Bingham House Books, 1997.
- Weinberg, Gerald M. *Becoming a Technical Leader: An Organic Problem-Solving Approach*. New York: Dorset House, 1986.
- Weinberg, Gerald M. *Quality Software Management, Volume 1-4: Congruent Action*. New York: Dorset House, 199X.

By Johanna Rothman and Esther Derby:



 11

What to Expect from a Beta Test

By Hal Bryan
Software Test Engineer, Microsoft Corporation
halbryan@microsoft.com

Bio:
Hal Bryan is a dynamic and entertaining speaker, licensed pilot, notary public, former police officer and current software test engineer in Microsoft's Games Studios. He's been testing software for more than 8 years at Microsoft, working on Windows 98, Flight Simulator, Combat Flight Simulator, and other simulation/game titles. For 6 years, he's been known for transforming Beta programs from misguided and distracting wastes of time to valuable partnerships with some of Microsoft's best customers – and for some of the mistakes he's made along the way.

Abstract:
Beta testing is one of the most frequently misunderstood aspects of software development. Many companies put too much trust in their Beta testers, others, not nearly enough. In both cases, a Beta program will be ineffective, at best. When managed effectively, however, a Beta program can provide considerable value, significantly impacting quality at very low cost. The key? There are a few, but the most important one is this: change your expectations.

The number one mistake product teams make with respect to Beta testers is to assume that Beta testers are professional testers, and that their feedback can be judged as such. While averages vary dramatically by company and by product, across most Microsoft Games Studios titles, the percentage of bugs reported by Beta testers that are defined as valid (legitimate, previously unknown, non-duplicate issues) averages between 5% - 10%. Those bugs, in turn, may only make up **a fraction of a percent** of the total bugs filed against a product.

So why bother? If a bunch of amateurs are going flood you with frequently poorly written and largely irrelevant feedback, why take the time, and why spend the money? The answer is deceptively simple – individual bugs, while useful, are the single least valuable contribution of a Beta program. Beta testers are not professional testers, they are customers, and the most important thing to remember is not to judge the value of the test by the quality of the testing.

If not bugs, what? The presentation will answer this question in detail, demonstrating how a well-managed Beta program can improve customer relations, corporate image, provide market research data, test casing ideas and inspiration, and provide widespread configuration and stability feedback via direct communication and / or automated reporting tools. Perhaps most importantly, a productive Beta can directly impact your product's own test team – taking good testers, who know how to think like customers, and helping them become great ones, by showing them how to think like many different types of customers.

Introduction

Microsoft's *Flight Simulator* has been published for roughly 25 years, and enjoys a vibrant and active community of millions of customers from around the world. I've been one of those customers since the very beginning, when I was 12 years old, and have worked on the product's Test team for the last 7 years.

When I started in 1998, the Beta testing programs were insulated and ineffective. We released a build late in our product cycle to a small number of people, most of whom were chosen from a generic pool and did not well represent our customers. Because the builds were released so close to our scheduled completion date, there was effectively no chance that any Beta tester's feedback would be incorporated into the final product. The management culture at that time dictated that all communication between Beta testers and the product development team be filtered through a single person (one of our business development managers - the reason being that he was the primary conduit for dealing with external partners), which was frustrating and time consuming for all involved, and had the net effect of eliminating communication all together.

Since that time, we've worked to make our Beta programs considerably more effective by adopting a dramatically more open and customer-centric approach. This document is intended to provide an overview of how we manage our Beta programs now, based on our experiences over the last several years.

What is Beta Testing?

First of all, a definition is in order. While there are specific technical differences between an Alpha and a Beta, for the purposes of this document, I use the term generically to refer to a program in which an unfinished application is released, under controlled circumstances, to a selected group of customers for the purpose of soliciting feedback.

What are the Benefits?

A well-managed Beta program can have significant impact on both a product's quality **and** its success in the marketplace - as much as we'd like it to be the case, one of those does not always guarantee the other. The first, and often worst, mistake made in running a Beta test happens even before it begins, when a product team decides to run a Beta with inaccurate and unrealistic expectations. Whether they assume that they will get professional-quality testing to supplement or even supplant their own in-house efforts, or they follow something more like our earlier model of assuming that a Beta is a token, rote exercise that is done essentially for appearances and isn't worth any real investment - both of these lines of thinking miss the point. Therefore, it is extremely important to understand the benefits of a well-executed Beta program before diving in.

The benefits of our Beta programs can be broken down into five main areas, in an order of priority that some may find surprising:

1. **Consumer Relations / Corporate Image** - It is no secret that many of our customers labor under the false impression that Microsoft is only concerned with making money, not with making quality products, as if those two things were somehow mutually exclusive. An open, communicative Beta program can demonstrate to a cross-section of our customers, from our strongest supporters to our most vocal (and influential) detractors, that we actually are passionate about building great software, we know a thing or two about what we're doing, and that we are open and receptive to customer feedback. In addition, participating in a Beta test generates advance excitement for a product, and gives customers a sense of ownership and responsibility. These attitudes are carried forward and reflected in public forums (web-based discussion groups, Usenet, blogs, etc)

that can ultimately influence a lot of other potential customers. When they encounter Microsoft bashing, a happy Beta tester remembers his or her friends at Microsoft and frequently comes to our defense.

2. **Customer Research** - One of my personal maxims about testing is this: *A good tester thinks like a customer. A great tester thinks like all kinds of customers.* Interacting with a diverse group of Beta testers is one of the best ways our Test Team has found to learn how our customers think, and the sometimes surprising ways that they use our products.

3. **Leverage for Test Teams to Drive Quality Issues** - Test doesn't win every single battle of quality and customer experience vs. risk, schedule, stubborn Program Managers, etc., but if enough Beta testers complain about a particular issue, many of these battles can be "revisited" with positive results.

4. **Widespread Configuration and Stability Testing** – While finding configuration issues (generally defined as hardware / software interoperability) is traditionally considered to be the primary benefit of running a Beta test, the actual number of valid configuration bugs reported in our Betas is usually rather small. There are two reasons for this: First and foremost, the overwhelming majority of these issues have already been flagged and reported by our internal Test team, and, in the most serious cases, fixed as a requirement for releasing a Beta build. The second reason is the fact that Beta testers will often ignore and / or overlook the most serious configuration and stability issues because they want to restrict their testing to their own areas of expertise, and they will assume that something as serious as a crash will certainly be reported by somebody else. Thankfully, automated reporting tools are dramatically improving the quality and consistency of the reporting of crashes by giving Beta testers a one click mechanism to give us what we need to investigate the problem. Even the lack of reported crashes can be somewhat useful information - even though Beta Testers can opt out of reporting, an overall decline in the trend of reported crashes has proven to be an indicator of greater stability.

5. **Reporting Bugs** - This will be discussed in a bit more detail in an upcoming section, but it is crucial to remember that Beta testers are not professionals. Beta testers will find bugs and in our case, they invariably find some good ones. However, those people that expect that bug reports will be the sole or even primary benefit of running a Beta test will be disappointed, and likely fail to take advantage of the other benefits outlined above. The key to success is in setting and managing expectations among all those involved.

Product Team Responsibilities

Ownership

At Microsoft, while the setup and logistics of releasing a Beta are handled in a painless and turnkey fashion by a Beta coordinator, the product team (which, at Microsoft, includes developers, testers, writers, artists, audio engineers, program managers, designers, localization specialists, and marketers) needs a point person of their own. Some product groups will nominate a Program Manager to oversee a Beta program, but this can generate a conflict of interest.

A Program Manager's primary role on a product team can be simply (**very** simply) stated as "Ship the right product, at the right time." Quality and scope of a product are balanced against available time and resources, with one eye always on the calendar as our shelf dates are frequently critical.

A Tester, however, is a specialist – an expert in reporting information about a product's quality, dedicated to continually demonstrating the ways in which a product can be improved. A test team is, of course, well aware of limited time and resources on a project, but those aren't primary motivators.

People tend to filter Information based on motivation. To put it another way, it is human nature to hear what we want to hear, no matter how objective we try to be. A Tester wants evidence that the product can be improved, and will use all types of customer feedback as a jumping off point to explore and investigate, even if the initial report isn't clear, lacks detail, or refers to an underlying issue that may not appear to be very serious. A Program Manager, on the other hand, is far more likely to interpret the feedback at face value, and will encourage investigation of only the most serious issues.

The second word in "Beta Testing" is "Testing"; our most successful Betas have been those managed by the product's own Test Team, just as they would oversee any other extra-group testing - shared-resource labs, outsourcing, etc.

Recruiting Beta Testers

What kind of people do you need, and where do you find them? The answers to both questions will vary dramatically depending on the type of product you're developing, but the important thing to remember is that, ideally, they should be customers, or at least potential customers.

Our experience has taught us that it is useful to cultivate a pool of Beta testers of varying levels of technical and subject matter expertise. Not everyone who uses Flight Simulator, after all, is a pilot or aviation expert. By the same token, many experienced real world pilots have expertise that is valuable to us, but may not be very technically savvy. Regardless, this diversity has invariably brought new perspectives, from usability difficulties experienced by abject novices to interesting and arcane details (that much of our market thrives on) brought out by our most dedicated enthusiasts.

As to where to find them, as mentioned earlier, first and foremost they should be current or potential customers. Our Beta programs are well established, so we tend to carry a large number of the same testers over from version to version. In addition, we also invite people who have provided interesting feedback on existing products via an external “suggestion” email address or that have made calls to our product support. If we were to build a list from scratch, we would certainly advertise via the web – “fan” sites, online forums, blogs, etc. Our screening process is, admittedly, pretty subjective, but, aside from the diversity of experience and customer status already listed, we look at practical concerns – they, of course, have to have the hardware necessary to use our product – as well as intangibles, such as their enthusiasm for the product, their passion (which may not always be positive) around a particular feature, issue, etc.

When is it Ready?

This, like so many other concerns, will vary greatly by product, but we try to start with a reasonably early build that has no major issues that we would define as “blocking”. Those would include stability problems, privacy / security issues, broken functionality in an area we want specific focus, the presence of any placeholder content that might not be appropriate to ship, and any potential interoperability problems. While we have certain objective criteria, choosing a particular build to release as a Beta is sometimes as much art as science. The earlier a build is released for Beta testing, the better the chances that some of the feedback from the Beta can be incorporated into the product prior to completion. However, releasing a build too early can often result in an overwhelming number of bug reports about known issues, features not yet implemented, and can require sometimes considerable support from the product team fielding questions.

Our Beta testers expect an unfinished product, and we’ve found that a well-timed release can exceed those expectations *("this looks really good, I can't believe it's only an Beta!"*), while still getting it in their hands early enough to potentially make a difference.

Welcome Letter

We send a welcome letter to all Beta testers along with CD's (if applicable), and general housekeeping information like usernames and passwords for accessing web / Usenet-based feedback mechanisms, etc.

We also use this opportunity to introduce the product and the team, offer quick highlights of new features, a rough timeframe, whether refresh builds will be shipped or download-only, as well as any other specific information that might be relevant. We introduce and explain our automated crash detection tool Office Watson, what it does, what information we will and will not be collecting as well our privacy policy, and why it's helpful. In addition, we provide instructions about what information to include when reporting crashing issues, including DXDIAG (Direct X Diagnostic) reports, specific text of error messages, etc. The welcome letter also introduces the terms of the Non-Disclosure Agreement, which our Beta testers accept electronically via their web portal.

Release Notes / README

Each build released includes a README that covers system requirements, setup / installation instructions, and at least a high-level look at known issues and desired areas of testing focus.

There are two major mistakes to be made here:

1. **Too Much Information** – For one product's first Beta release, I overwhelmed our Beta testers with pages and pages of known issues, copied and pasted literally by the thousands from our bug database. I didn't set out to prove to the Beta testers that I was on a team made up of real testers and didn't really need them, but that's exactly what I accomplished. They got the message, and acted accordingly, reporting about 5% of the normal total number of bugs, and exactly 0 of them proved to be valid, useful issues.

2. **Too Little Information –** After that experience, for our next release I opted to include only the most minimal information about known issues, what we wanted them to focus on, etc. This damaged our credibility – a common theme in responses was *"They've got this thing so screwed up, I don't know where to begin!"*

Obviously, this can be a difficult balance to strike, but a good rule of thumb we use is that, for every known issue we point out, for every incomplete area we advise them not to test, we also give them a specific area or feature to focus on. This way, the product team can project its competence without devaluing the Beta testers' contributions, and still solicit the feedback they want.

One final comment I've learned about documents named README: Most people don't. Given the choice of playing with new software, and reading about playing with new software, most people will dive in and start playing. Our solution is to eliminate the choice. We get the README in our Beta testers' hands before they actually have access to the product. There is always a delta between handing off a build (at which time the document has been completed) and the build being duplicated on CD and shipped out, or posted for download. We use that time to our advantage and post and / or email the README out to each tester. If the README is all they have, there's a good chance that they'll read it.

Discussion Forum(s)

At Microsoft, the primary mechanisms of communication between the product team and the Beta testers are restricted-access Beta newsgroups, set up by the Beta Coordinator. At their most basic, there will be two folders specific to a given title: "Announcements" and "General". It is easy to create multiple newsgroups based on individual requests, and it works well to have newsgroups for each major feature area. While the "General" group will almost always see the most activity, breaking them down by feature area helps keep discussions focused, and fosters better communication.

We've found that it is important that each newsgroup have an owner. The owner is not the only team member to post in a specific group, and may not even be the most frequent poster, but they do bear the responsibility of ensuring that all questions get answered, issues addressed, etc. In our case, newsgroup owners are the testers responsible for the given feature area.

All team members (not just Test) should be encouraged to participate in the newsgroups. Our old model was to “sanitize" every post and use a single person as a mouthpiece. This was impossibly inefficient, and did little to help improve our aloof corporate image. The benefits of increased participation and the sense of partnership and community that is fostered now between the product team and the Beta testers far outweigh the potential (and usually pretty easily corrected) risk of a team member misspeaking. There are, of course, certain ground rules – no personal attacks, no profanity, and no proprietary information – but simple politeness and common sense rule the day.

Beta Testers and Their Bugs

As stated earlier, it is important for us to manage the product team's expectations when it comes to the type and quality of testing coverage provided by Beta testers. To do this, it is important to understand a few things about the testers themselves.

First of all, a diverse selection of Beta testers can bring a lot of different levels of knowledge and expertise to a product's development. Obviously, they should at least be *potential* customers and have some level of interest in the product they're testing. But a well-run Beta can include a number of "novices" – for us, that can mean that they very little about computers and software, very little about the subject matter (aviation and flying), or both - and they can provide useful usability feedback via bug reports, online discussions, formal surveys, etc. Don't let their inexperience weaken their credibility - some of the best bugs can come from this type of high-level, new-user perspective.

However, the single most important thing to remember is this:

BETA TESTERS ARE NOT PROFESSIONAL TESTERS

And they should not be judged as such. This is not an uncommon mistake to make, but it is most important that product Test team members, as the primary filters of Beta testers’ information, understand this clearly, or they risk overlooking something valuable**:**

- **Very few Beta testers write good bug reports.** We get bugs with titles like "Application crashes" and the steps to reproduce, if any, may be as terse as "just run it".
- **They report substantial numbers of duplicates**. In our case, this isn't their fault, as we currently provide no way for them to determine if an issue has already been reported if it isn't discussed on the newsgroups or listed in the README. And, no matter how thoroughly we document the known issues, some Beta testers will report those, too. This really isn’t a problem – it takes our team comparatively little time to determine if a reported issue is unique, and it works in our favor to have our Beta testers less encumbered by process. If Beta testers' bugs **weren't** mostly duplicates (of those issues found internally), we would need to seriously reevaluate our own testing.
- **They invariably file bugs on issues that were fixed right after their build was released**. Again - this is obviously through no fault of theirs. (Releasing incremental builds / updates via download can put new content in testers' hands in a matter of hours, which does a lot to mitigate this problem.)
- **They file bugs that aren't reproducible**. This isn't necessarily a bad thing, but it does mean that they are not likely to be nearly as aggressive with their investigations as a professional tester. Even if we can't reproduce the specific problem in house, the investigation can generate excellent test cases.
- **They file all manner of issues, not just bugs.** Examples include bugs with titles like "When do we get another build?", "You should kick this person off the newsgroup", and, my personal favorite, "Setup completed successfully - no problems to report". For many Beta testers, the bug form is their primary means of communicating directly to the product team, so even "no

problems to report" can be very useful feedback for us. We don't specifically utilize a mechanism for positive feedback, outside of the occasional survey and ongoing discussions, but that doesn't mean that sort of information isn't valuable.

- **Self-Styled Experts file bugs about behavior or content that is actually correct** – I suspect that this is more of a problem with simulations than some other types of software because of the objective comparison to the real world. As an especially arcane example, one Beta tester reported that the altimeter (the gauge that indicates an aircraft's altitude) on one aircraft was labeled with text (the word "altitude") that was slightly angled, but should have been straight. What the Beta tester didn't realize was that this word was printed on a portion of the gauge that rotates, albeit not in a very obvious fashion – the behavior of our software matched exactly the behavior of the real aircraft, an aircraft that we had experience flying in the real world, while the Beta tester did not. In this case, we were able to politely re-educate the Beta tester with the added benefit of enhancing our credibility at least slightly in the process.
- **"Mega Bugs"** - These are bugs that are either a laundry list of issues scattered across multiple feature areas, or even an attached prose document with the tester's overall impressions of the product. We try to educate the testers and encourage them to file individual bugs for each issue they report. However, we don't ignore their list and risk missing something important, because we can't always assume that they will take our advice.
- **Some Beta testers think that they are solely responsible for testing our products**. This should never be the case. As we disabuse them of this notion, we try to do it very gently – we make it clear that they are our allies, and vice versa, in the battle for quality.
- **Many tend to be focused on a particular area of interest**. It is common for them to click through and ignore a number of high priority / high severity crashing issues in order to grab a perfect screenshot of a mismapped texture, for example. This underscores the importance of the product team proactively reviewing automated crash reports and soliciting additional information, if needed.
- **Some will be hesitant to file a bug.** If they are active participants in the newsgroup(s), they will frequently post some information about their bugs there, often asking if they are legitimate issues that should be reported. The underlying message is **always**: "if in doubt, enter a bug." We can track bug reports far more easily than ongoing newsgroup or email threads.
- **Some use the Beta as an opportunity to make major feature requests.** Many Beta testers simply don't have a realistic idea of the costs and risks associated with adding even a seemingly minor feature. We treat their requests politely (offering well-practiced answers to questions like "How hard can it be?", not to mention teaching and reminding our employees not to take the criticism personally), we encourage them to focus on testing what's already there, and, most importantly, we track their requests for consideration in a future version.

Given these considerations, it should not come as a surprise that Beta testers ultimately report relatively small numbers of valid bugs. Percentages vary, but an anecdotal average across a number of titles is that somewhere around 5%-6% of all Beta bugs are valid, while the average for our team's Betas is roughly 15%.

Regardless of the percentages, Beta testers can and do find and report some excellent bugs, and it is crucial that each bug be appropriately considered and investigated.

Beta testers may not be professional testers, but they are customers, and that can make their contributions even more important.

Do not confuse the professionalism of the testing with the value of the test - the two are not always directly related.

Microsoft Beta Bug Lifecycle

There is obviously some variation, but these are the typical steps in the life a reported Beta bug:

1. The Beta tester finds a bug. As above, they may also post about their find on the newsgroup(s) - these and subsequent posts can be very informative.

2. The Beta tester enters the bug via a web-based portal used for all Beta activities. The form is very straightforward - testers are required to enter their name, telephone number, and email address, and have the option to receive email when their bugs change. The bug report fields include title, problem type (with an associated severity), a "Reproducible" field (Always, Sometimes, Never), and text boxes for description and repro steps. In addition, the Beta tester can add attachments to their reports. Our product team has complete control of the Bug form and can add / delete any fields which are deemed necessary, at any point during the Beta program. Custom fields can be extremely useful, especially one that requires a version number, if there are cases where we may have Beta testers running different builds.

3. The bug shows up on our radar. The bug is almost instantly copied to a database that is accessible to the product team's internal bug tracking database, Product Studio, and is flagged as Active.

4. Triage. The Beta test lead periodically reviews those Beta bugs assigned to Active (usually twice a day), and assigns them to the appropriate internal Tester for investigation to determine if it is a valid issue or not.

5. If the bug is not valid, then it is resolved and closed in Product Studio by the investigating tester.

6. If the bug may be valid but is not reproducible, or requires additional information, we resolve it back to the Beta tester as "Not Repro", along with a request for assistance. The Beta tester can then reactivate the bug, adding their comments as needed. An alternative is to simply email the Beta tester directly to solicit any additional information required - this can result in much faster responses, and build greater camaraderie, but there is the added risk of being overwhelmed with personal correspondence. Discretion is crucial, as is capturing any additional information about the issue and tracking it accordingly.

7. If the bug is valid a new entry is created in the internal Product Studio bug tracking database by the investigating tester, and any relevant text is copied from the Beta bug and pasted into the new bug. At this point, the investigating tester is acting as a filter, and is expected to provide a proper and professional report based on the initial findings of the Beta tester.

    a. The Beta bug then remains Active, assigned to the investigating tester, pending the outcome of the corresponding internal bug.

    b. Once the internal bug has been resolved, the Beta bug is usually resolved the same way, as long as the investigating tester has verified that the fix will be available in the next Beta build released.

    c. Beta testers will then verify the fixes on their own, and reactivate any bugs that we believed were fixed but are still reproducible. This isn't common, but it happens, and the Beta testers provide a crucial "extra set of eyes" that keeps us honest.

8. Continued investigation and indirect benefits. Regardless of the validity of the bug that was entered, the investigation of one issue can lead to other bugs found indirectly, by updating test cases, etc.

A Few Numbers

When you consider the 15% of Beta bugs reported that are valid, it may only represent an extremely small percentage of the total bugs in our own database. However, consider the following hypothetical scenario:

Suppose an active Beta group enters 2,000 bugs, and 15% of those are valid, which means that 300 of them are migrated into the product team's database. Assuming a reasonable average fix rate of 65%, this means that 195 bugs were found and fixed that may not have been otherwise. Even against an internal Test team's database that likely contains *several times* as many bug reports, nearly 200 "extra" fixes can represent significant polish, at the very least. While it is uncommon for any of those bugs to be major support call generators, “ship stoppers”, or potential patch candidates, it has happened on occasion – and it only takes one.

Bearing in mind that finding bugs is not the primary benefit of a Beta test, it is worthwhile to point out that the number of bugs reported *directly* is usually considerably smaller than the subsequent bugs found *indirectly*:

- Bugs generated from automated crash reports.
- Incidental bugs found during the investigation of Beta bugs.
- Bugs uncovered by new test cases developed while investigating Beta bugs.
- Existing bugs that can be "resurrected" with the added leverage of Beta reports.

Security Concerns

All of our Beta testers are bound by a very specific Non-Disclosure Agreement that prevents them from disclosing anything about the product or the Beta process, including the fact of their own involvement. Most NDA violations are pretty innocent - an inadvertent mention of some innocuous tidbit on a forum somewhere, for example - and can be immediately quelled with a direct email to the violator. However, some violations can be more flagrant - posting screenshots, complaining publicly (outside the Beta newsgroups) about bugs, etc - and require the immediate removal of a Beta tester from the program.

Another potentially major problem for any Beta release is piracy. Occasionally, thorough searches of public newsgroups will turn up unauthorized download locations, and Beta software does show up on eBay from time to time. However, the most common means to find our builds publicly and illegally available has been via peer-to-peer file sharing services. The offenders can be difficult to track, but we work hard to develop good relationships with our Beta testers, and they are usually very proactive about reporting potential NDA violations.

Updates / Downloads

Beta activity winds down considerably as their build becomes more and more out-of-date - it is not uncommon for the testers to declare that they've found all of the bugs in a particular build. While this, of course, isn't true, it is a sign that the returns are diminishing.

To minimize this, we plan on making at least three major builds available during a Beta cycle (many teams try to release one build per month – up to a point, more is better). Depending on the product, smaller updates can be released for download incrementally, but it we have to take the geography of our Beta tests into account. Flight Simulator, for example, sells a substantial percentage of its units outside the US, so our Beta programs are truly global. The adoption of broadband outside the US still lags in some areas and many overseas ISP's charge by the minute for Internet access, so a 3GB+ download just

isn't feasible for everyone. Obviously, however, offering a build for download results in much faster feedback – we can have a download posted and available within one business day, versus at least five days for shipped media.

It becomes exceedingly difficult to track issues when different Beta testers are using different builds, so we do our best to ensure that all Beta testers have access to the same build. In some cases, this means shipping CD's to all Beta sites with each new release, but the benefits and lower overhead on the product team can easily outweigh the expense, depending on budgetary concerns, cost / benefit analysis, etc.

The End Game

As detailed previously, a well-run Beta can build a lot of good will between us and our customers. There are two things that we've found to be crucial at the end of a product cycle to maintain this:

1. **Tell them when we're done.** We announce your project's completion to the Beta testers before announcing it publicly. Beta testers have a strong sense of ownership, pride, and responsibility for a product and it is demoralizing for them if they read about the project's completion on the web before hearing it from us. Those times I've found myself in the unenviable position of working on a product that has been cancelled, this had to be messaged very carefully, but the Beta testers were given at least some basic information. This has proven to be a good opportunity to involve Program Management – it was their purview to begin with.

2. **We always send them a copy of the release version.** We usually don't promise one early on, but we always send one anyway (assuming the project survives and a released versions exist). It's the right thing to do – we don't ask the people that have volunteered their time to help us improve the product to then buy it from us when we're finished. It's an excellent way to say "thank you", and promotes continued customer loyalty and good will.

Post-ship releases can benefit from additional testing as well. In our case, we publish a series of Software Development Kits (SDK's) after our primary release, so we keep the Beta program going. In those cases where we've had to release a patch, then the Beta has been a calming influence, and demonstrates the product team's continuing commitment to customer satisfaction.

Conclusion

By communicating effectively, setting and managing the expectations of all involved and proactively addressing issues as they arise, we've been able to establish a solid rapport with some of our best customers, key members of our community. This rapport has been strengthened with each release, and, once the feedback loop was established, something interesting, but not unexpected began to happen: Beta testers bug reports, even their overall testing skills, have been steadily improving. Aside from the obvious benefits of higher-quality testing, while our core group gets stronger and stronger, we find ourselves in the enviable position of being to recruit more and more novices to our Beta teams, continually bringing fresh perspectives.

Our Beta programs continue to teach us about our customers, and vice versa, and have proven to be an invaluable part of our goal to continually ship higher-quality, and ultimately more successful products.

Get What You Want From the Sponsor! Communicate for Success

Pam Rechel
Brave Heart Consulting
braveheartconsulting@msn.com
503 287-4323

Abstract:

It is widely known that recommendations to save money or boost productivity can die an early death if those who generate the ideas are not able to gain support from the appropriate sponsors or executives. Each person has a preferred style of communication and can take in information more easily when information is presented their preferred style. If you want to influence someone or be certain to have your ideas understood, it is logical that communicating according to the receiver's preferred style will increase effective communication.

The focus of this paper is to provide two methodologies; one is used to assess someone's preferred communication type. The other methodology is to present processes for communicating that are easy and will make it easier for sponsors, regardless of their communication preferences, to hear the message. The assessment techniques include analyzing the communication style by identifying key words and actions they use, e.g. do they ask more "how or what" questions or "why" questions. Also, do they think out loud or take time to reflect before responding. All of these examples point to a preferred communication style. The second method presents important, but simple tips to modify a communication style in order to communicate to all styles. For example, by sending out information in writing ahead of time, even a few hours ahead is useful for those who want to think about something ahead.

All the methods presented will be practical and could be applied the day after the conference.

Author Biography:

Pam Rechel is an executive coach and organization consultant with Brave Heart Consulting in Portland, Oregon. Coaching executives and teams to become more competent is an area of focus. Pam is also a specialist in communication, especially for individuals and teams in high-tech organizations.

She is a certified Myers-Briggs Type Indicator® professional, including MBTI® Step II. She has worked with high tech teams and individuals in the U.S., Singapore, Taiwan, India, Egypt, Europe and Ireland.

Pam received a M.A. in Coaching and Consulting in Organizations from the Leadership Institute of Seattle (LIOS), an M.B.A. from George Washington University in Information Technology, and an M.S. from Syracuse University in Education Administration.

Get What You Want From the Sponsor – Communicate for Success

Introduction

Have you had a good, even great, idea and were not able to get the attention of a sponsor or others that can support your project and/or proposed process improvement? “Recommendations to save money or boost productivity can die an early death if those who generate the ideas do not have the skills to gain support from the appropriate sponsors who can approve their ideas.” [1] Communicating according to the receiver’s preferred style increases effective communication. Techniques for doing that are the core of this paper.

What will be described in this paper:

- Why Modify Communication Style
- Myers-Briggs Type Indicator® Basics
- Assessing the Communication Style of the Sponsor
- Communication Cues
- The Mind of the Sponsor
- How to Modify Communication
- Sample Communications
- Data on type preferences of software engineering managers
- General Rules – what to do if the sponsor is unknown or the group is diverse
- Next Steps – how to apply this information

This paper will describe practical tools to successfully communicate with sponsors and describe how to assess the sponsor’s communication preference and adjust your communication style as needed.

For each personality type, practical suggestions and examples for modifying communication to fully engage the sponsor will be presented. The suggestions are simple enough to be applied the day after the conference and powerful enough to make a difference. Also included are some important considerations that apply to all sponsors regardless of their personality type.

In this paper, “sponsor” describes the person or group that has the power to sanction change. In other words, they decide which projects get funded, where resources of people or money are allocated and what projects are most important in the organization. In most instances, sponsors are managers or executives in an organization. Later in the paper, their thought processes and what is important to them will also be discussed. All of these tools can be used for improving communication with anyone, not just sponsors. For the purpose of this paper, the focus will be on specific requirements for communication with sponsors.

Why Modify Communication Style

Skilled communicators shift their communication to match the language and preferred communication style of the listener. Using the sponsor's preferred methods of communication increases the sponsor's ability to better hear the message in the same way that listening to high quality surround sound is easier and more powerful than listening to monaural sound or sending an attachment in a format the recipient can easily read increases the chances of the message being understood.

If details are presented first and the sponsor prefers to hear the big picture first, chances are the sponsor will be frustrated, get bored and tune out or simply unable to absorb the message. If the sequence in which ideas are presented is not complementary to the sponsor's preferred communication style, it will be more difficult, if not impossible, for the sponsor to take in the idea. If that happens, it's possible to think the rejection of the idea is due to the idea itself instead of an ineffective communication style. The fact is the communication style may have been the barrier. The cues for determining someone's communication style will be presented later in this paper.

Other Reasons to Modify Communication

- Reduce frustration on the part of the communicator and receiver
- Reduce rejection of good ideas
- Improve communication with the sponsor
- Increase confidence and develop new competencies

The Myers-Briggs Type Indicator – The Basis for the Cues

Information is presented here to describe the theoretical foundation for the communication assessment and cues, as well as the communication modifications. The Myers-Briggs Type Indicator® (MBTI®) is the most widely used personality indicator in the world. According to the CPP Inc., which publishes the MBTI® instrument, the MBTI® instrument is taken by over 2 million individuals in the U.S. annually. It is available in sixteen languages and is administered globally. "The MBTI® is probably the most insightful and useful instrument available to very quickly understand how and why people tend to behave and contribute in work situations the way they do. No amount of coaching, well intended resolutions, and pronouncements can really alter what individuals like, dislike, enjoy, detest, want, and do not want…at least in the long haul." [2 p. 35] So, if you know the preferences of the person you are trying to influence, you can modify your style to match their preference.

The MBTI® instrument identifies preferences rather than competencies. It is a framework for analyzing behavior and preferences that allows communication to be tailored to the preferences of the sponsor, thus increasing the opportunity for influence and decreasing frustration-both yours and theirs.

Our preferences are similar to the way in which we "prefer" to use either our right hand or the left hand for writing our name. Some people are more adept in writing with either hand, although all of us can do it. Most of us, however, have a distinct preference for using one hand over the other, just as we have a preference for communication styles.[2] The way that each person communicates correlates to their preferred type 75-80% of the time. That illustrates how strong the preferences are. It also indicates that for 20-25% of the time, communication is flexed to match the needs of the other individual or the situation.

The MBTI® is divided into four pairs of preferences: [3]

Preferred mode of thinking:	To Talk it Out	or	Think it Through
Preferred focus of information:	Specifics	or	The Big Picture
Preferred reasons for action:	Logical Implications	or	Impact on People
Preferred approach to problem solving	Joy of closure	or	Joy of Process

"The pairs of preferences are not traits that vary in quantity, they are dichotomous constructs that describe equally legitimate but opposite ways in which we use our minds."[4] In other words, a preference does not indicate how well a person thinks out loud, for example, but how clear they are that they prefer to think out loud rather than to think it through. Neither of the opposite preferences is better than the other, as we all use all of the preferences and it is important for all the preferences to be represented on a team, in problem solving and in life. The names of the preferences are a descriptive summary of the preference, as well as a cue.

The MBTI® instrument is equally useful outside of the work situation for improving how individuals, children and adults communicate, solve problems together and make decisions. Often the work situation requires that we flex and not use our preferred style. In those instances, it can be awkward, uncomfortable, and more stressful to operate outside of our preferred styles all day, just as it would be to write with our non-preferred hand all day. Anticipating the preferred style of another and flexing our style to adapt is a powerful and competent way to skillfully improve communication, decision-making and problem-solving.

The Type Styles (Table 1) identifies the primary definitions of each pair of preferences and some key identifiers of the preference. The left-hand column identifies the purpose of the preference pair.

Table 1. Type Styles

Purpose	Preference	Opposite Preference
GETTING ENERGY & THINKING PROCESS	**TALK IT OUT** Thinks, processes ideas and solves problems out loud Speaks *before* thinking Gets energy from being with people Extraversion (E)	**THINK IT THROUGH** Thinks internally about ideas and problems Thinks and then may or may not speak Introversion (I)
GATHERING INFORMATION	**SPECIFICS** Wants Facts Details Realities Past or present experiences Wants details in order, does not jump around Practical Decisions: facts first Sensing (S)	**THE BIG PICTURE** Wants Abstract or conceptual Possibilities or meaning The context or "why" Future or what *could* be Random order. Wants the bottom line implications before any details. Decision: gut feel, then checks out the facts. iNtuition (N)
MAKING DECISIONS	**LOGICAL IMPLICATIONS** Detaches from the situation to find a logical approach Principles, especially that follow a non-personal logic (if this, then that) Objective information An outcome that "makes sense" Trusts facts and principles Distrusts emotions only Thinking (T)	**IMPACT ON PEOPLE** Puts self inside the situation Values, especially people oriented values Subjective information A harmonious outcome that "feels right" Trust emotions/personal reactions Distrusts facts only Feeling (F)
ORIENTATION TO LIFE	**JOY OF CLOSURE** Organized Planned Goal oriented Makes decisions quickly and moves on "Let's wrap it up" Avoids stress by planning and scheduling Judging (J)	**JOY OF PROCESSING** Flexible Spontaneous Oriented to gathering information "No decision before it's time" Avoids stress by *not* planning and scheduling Perceiving (P)

Modified from [5] Used with permission

The letters E,I,S,N,T,F,J, and P all refer to the MBTI® names for the preferences: Extraversion, Introversion, Sensing, iNtuition, Thinking, Feeling, Judging and Perceiving. The terms are specific to the MBTI® instrument and do not always match the more common usages of these words. It is not important for purposes of this paper that those terms be utilized. For further information or to take the MBTI® instrument, contact the author, Pam Rechel, at: braveheartconsulting@msn.com or reference www.capt.org, or www.cpp.com to learn more about the Myers-Briggs Type Indicator®.

Assessing the Communication Type of the Sponsor

Think about the person or group you most want or need to influence. First analyze their preferences by thinking about the Type Styles (Table 1) and second by the Communication Cues (Table 2). If you don't know the sponsor well, ask someone you trust who knows the sponsor for another assessment of the sponsor's style. Not all cues listed for a preference may necessarily be present. The key is to look for a preponderance of cues to the preferred, or most used, style. Later in this paper a description will be presented for what to do if you and your trusted advisors do not know the sponsor or all the participants in a meeting.

Table 2. Communication Cues

Preference	Opposite Preference
TALK IT OUT (E) Rapid speech Appears to "think out loud" Interrupts Louder voice volume Reads handouts at meetings[6] Enjoys large meetings and speak often	**THINK IT THROUGH (I)** Pauses in answering or giving information Appears to be thinking things through Quieter voice volume Shorter sentences, not run-on Reads handouts prior to meetings Prefers small meetings and listen more
SPECIFICS (S) Asks for step-by-step information or instruction Asks about the present situation Asks "what" and "how" questions Uses precise descriptions	**THE BIG PICTURE (N)** Asks for the purpose of an action Asks for current and long-range implications Asks "why" questions Talks in general terms and possibilities
JOY OF CLOSURE (J) Impatient with overly long descriptions or procedures The tone is "hurry up…I want to make this decision." Enjoys being done May make decisions prematurely	**JOY OF PROCESSING (P)** Seems to want "space" to make own decisions The tone is "let's explore, what are some more factors to consider?" May even decide at the "last moment" Enjoys processing

[5 p. 26]

Review Table 2 and select what you believe is the sponsor preference for each communication pair. Base your selection on the sponsor's communication cues that you observe, or ask others about the sponsor, a majority of the time.

Data on Type Preferences of Software Engineering Managers

Another way to determine the type of a sponsor who is unknown to you is to use the data on the MBTI® type preference for Software Engineering Managers from the CPP, Inc. 2004 database (Table 3). The data indicates that the greatest percentage, 74.7%, prefer Logical Implications. Therefore, following the actions to modify communication for Logical Implications when you do not know the sponsor is recommended.

Table 3. Percentage of Types for Software Engineering Managers

Type	Count (n)	Percentage
Logical Implications (T)	1052	74.7
Joy of Closure (J)	868	61.6
Specifics (S)	815	57.8
Think It Through (I)	714	50.7
Talk It Out (E)	695	49.3
The Big Picture (N)	594	42.2
Joy of Process (P)	541	38.4
Impact on People (F)	357	25.3

Data provided by CPP, Inc. from the MBTI® 2004 Database. Permission granted.

The Mind of the Sponsor

Sponsors are required to focus on the big picture, the vision of where the project or change will take the organization, managing the resources of time, money and people, the scope of the project and the long term view of the effect the change or project will have on the organization. [1 p. 114] Whether it is their type preference or not, (see preference for The Big Picture in Tables 1 and 2), the sponsor role requires that they manage the big picture as their first priority. Therefore, when communicating with a sponsor that has a preference for details, the big picture must be included first, then the details.

Sponsors do not want to be surprised, be embarrassed or to waste time. Preparing to present a new idea means gaining support before the formal approval presentation. If the sponsor is surprised or embarrassed by your idea or presentation, one Director with over 25 years of Project Management experience said, "Put your head between your knees and kiss your butt goodbye". This includes insuring that your direct manager is not surprised by the communication of the idea.

Important Considerations for All Sponsors

Present the big picture *first* for all sponsors. Ensure that there are no surprises as you present the new ideas. This may require pre-meetings with parties that may be impacted by your idea so no one is surprised or embarrassed. Know who might lose or be damaged if this idea is adopted and who would gain. For all sponsors, get to the point quickly or you'll be tuned out. This is especially important higher in the organization where time is perceived to be a precious commodity.

Modifying the Communication Style to Match the Sponsor's Style

Once the sponsor's style has been assessed (Table 1, 2, 3), follow the suggestions for their style in Table 4 to select how to modify your style of communication. Suggestions will be given later about what to do if you do not know the sponsor or their style.

Table 4. Actions to Modify Style to Match the Sponsor

Sponsor's Preferred Style	Sponsor's Preferred Style
Talk It Out (E) • Send written information ahead of a meeting. It may or not be read, but will confirm that you are organized and professional. • Prepare questions to ask them in the meeting, so they can "think out loud". Some powerful, yet simple, questions include: o What did you hear in the presentation? o What is your reaction to what you heard? o What clarifying questions do you have?	**Think It Through (I)** • Always send information ahead. It allows the sponsor to do their best thinking about the issue. • Send a list of questions you want the sponsor to consider as they are reading the proposal or idea (see suggestions under "Talk It Out") • Be prepared for and plan for some silence during the meeting. Allow time for internal thinking. • Do not interrupt them as they speak
Specifics (I) • Present the Executive Summary of the global concepts (the Big Picture) first • Then describe the impact on the current situation • Sequentially tell all the key facts about the project	**The Big Picture (N)** • Present the Executive Summary of the global concepts (the Big Picture) first • Describe how the idea links to the sponsor's vision and why it is important • Describe the long-term and future • Have the facts ready or attached in an addendum. Present only critical details.
Logical Implications (T) • Present logical reasons for the project • Describe if the project is implemented, then xyz will happen (if…then) • Expect critiques of the idea • Be brief and businesslike	**Impact on People (F)** • Describe who will be impacted – customers, employees, competitors • Tell who was consulted in creating the idea • Speak personally to their vision and pet projects – "I believe this links to your

Sponsor's Preferred Style	Sponsor's Preferred Style
	priority for Customer Service"
Joy of Closure (J) • Say when you want a decision so implementation can be started "I'd like a decision by Tuesday" • Include a checklist or plan for implementation and a timeline with specific steps • An end date for project implementation	**Joy of Process (P)** • Say how long you want to keep the issue open so more ideas can be included "I'd like to keep it open until Tuesday so more ideas could be included if necessary." **The same timeframe can be used for someone with a preference for closure, but described in a more open way)** • Broad guidelines for a project plan • An end date for project implementation

Sample Communications from Engineering Managers and Sponsors

Sample preferred communication for a high-tech sponsor whose style is Think It Through (I), Specifics (S), Logical Implications (T), Joy of Closure (J)

- First – I like to receive an idea in writing. I can read it first, then discussed in a meeting (Think It Through)
- If there is discussion before I have a chance to read it through, it is frustrating. It's too rapid fire. I can't do my best thinking. (Think It Through)
- Not long emails. Write a summary first. Include a plan. (Joy of closure)
- I like to be asked for my preferred communication style. My answer is typically email. (Think It Through)

Sample communication from a sponsor whose preference is Big Picture (N) and Logical Implications (T)

- Tell why the idea is potentially important to the organization to consider this idea (The Big Picture)
- Build a prototype – a 3D model or picture to help visualize. "Here's what you'll get with this project". Use charts and graphs. For example, say "Here's when you'll hit the market" if the idea impacts market share. Use more than just text slides or Excel spreadsheets.
- Ask what concerns they have (Logical Implications)
- What they would do to make the idea more feasible (Logical Implications)

Sample communication for a sponsor whose preference is Impact on People (F)

- Tell why the idea is important to you, the idea advocate. (The Impact on People)
- Ask who else to solicit for input for the idea (The Impact on People)
- Have a personal relationship with the sponsor. For example, eat lunch with them or meet with them beforehand to present your preliminary thinking. (Note:

building a relationship with the sponsor is helpful for all types and especially for those whose preference is The Impact on People.)

Sample communication for a sponsor whose preference is Specifics (S)

- Solve a problem that I care about today (vs. a future situation).
- Don't be vague, fuzzy or force the sponsor to ask a lot of questions because the description was not clear.

Describe how the idea is feasible and that homework was done about how the organization can pull this idea off. Although sometimes tempting, do not say "Here's how the organization can spend $2 million on a neat idea". Describe how the idea is feasible in the near future. Feasibility means common sense information such as:

- Is it the right time
- Can the organization get the resources and people
- Will the culture embrace this or completely reject the idea. Have an answer to these questions.

General Guidelines for Communicating to a Varied Audience

Follow these guidelines when communicating to an audience that you do not know or an audience of varied preferences. These guidelines address all the preferences.

1. **Send information in writing ahead of time**
    a. Including the agenda, the main points of the presentation and what questions you want them to consider. Those with a preference for Talking It Out may ignore it and prefer to discuss it first in the meeting.
    b. Those with a preference to Think It Through will appreciate the opportunity to think about it ahead of time. In any case, you will appear more organized, professional and considerate.

2. **Include a blend of specifics and the big picture in the presentation**.
    a. Always start with the big picture and how the issue relates to the organization goals and priorities. Tie the proposal to the present and past, which is important to those with a preference for Specifics, as well as to the future, which is important to those with a preference for The Big Picture.
    b. Present the important Facts. Repeat the Big Picture.

3. **When presenting specifics, present only the most important facts and indicate that additional details are available.**
    a. For example, present an Executive Summary first that includes the big picture and important details.
    b. Have appendices with additional detail if they want it.

4. **Present the impact of the idea on customers, suppliers, employees and other groups.**

a. Although there are fewer Engineering Managers with the preference for the Impact on People, it is important that information about the impact on others be presented so there is balanced communication.
b. Present impact information after the logical explanations for the proposal.

5. **Take time to personally greet each person in attendance.**
    a. This allows you to connect with the people you have not met before.
    b. This is particularly important to those with a preference for the Impact on People and important for all attendees.

6. **During the presentation, allow time for silence, even 30 seconds, at various intervals.**
    a. This can be challenging for some individuals with a preference for Talking It Out. Count to twenty before speaking.
    b. It gives those with a preference for Thinking It Out time to think and you a chance to breathe.

7. **Include a broad project plan in the presentation.** (Joy of Process)
    a. Ask if a more detailed plan is desired and be ready to present it.
    b. Keep the more detailed plan in the appendices. (Joy of Closure)

Summary and Next Steps

Skilled communicators shift their communication to match the preferences of the listener. A framework for easily assessing the preferences of the sponsor was presented. Methods for changing communication to match the sponsor preferences were outlined. The modifications are simple and can be implemented immediately.

It is not always possible to assess the preferences ahead of time therefore; following the practical guidelines for communicating to varied audiences will improve the success of communication thus reducing frustration for all parties.

Next steps to take include:

1. Identify the sponsors or other individuals with whom you want to improve communication.
2. Assess their communication preferences using Table 1 and 2
3. Identify the actions in Table 4 that you are willing and able to take to modify communication.
4. Implement the suggestions and note the improvement in the responses to your communication.

References

[1] D. R. Conner, *Managing at the speed of change*. New York: Villard, 1992.

[2] O. Isachsen and L. V. Berens, *Working together: A personality-centered approach to management*, Third ed. Irvine, Ca: Institute for Management Development, 1995.

[3] S. A. Brock, "The four- part framework." Gainesville, FL: Center for Applications of Psychological Type, Inc., 1987, revised 1995.

[4] I. Briggs Myers, M. H. McCaulley, N. L. Quenk, and A. L. Hammer, *MBTI manual: A guide to the development and use of the Myers-Briggs Type Indicator*, Third ed. Palo Alto, CA: CPP, Inc, 1998.

[5] J. Allen and J. Gray, *Flex Care: Health care communication skills using personality type*. Gainesville, FL: Center for Applications of Psychological Type, Inc., 2002.

[6] P. Ferdinandi, "Reengineering with the right types," in *Software Development*, vol. 2, 1994, pp. 45-51.

Project Management Tools

for Communicating Scope, Schedule, Dependencies, and Risks

John Balza
Hewlett-Packard Company
johnjbalza@comcast.net
john.balza@hp.com

Abstract:

When dealing with multiple teams in complex projects, communication of each team's scope, schedule, status, dependencies, and risks between teams is essential for a successful integration and execution of the entire project. System specifications, project plans, and detailed work breakdown structures incorporated into PERT charts are absolutely required to describe and manage the project, but aren't the best communication vehicles to management or among multiple teams. This paper provides slide templates developed over the last few years that summarize these deliverables and have proven valuable in communicating among our teams.

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.

Introduction

When dealing with multiple teams in complex projects, communication of each team's scope, schedule, dependencies and risks between teams is essential for a successful integration and execution of the entire project; but even with smaller projects, being able to communicate quickly and easily the essence of the project to the project sponsor and other teams is essential. There are often interdependencies and common resources that need to be managed among projects. Projects create lots of documentation to communicate within the team: requirements documents, specifications, quality plans, project plans, etc. but this information is too complex and difficult to read to be used as communication vehicles between teams and with the project sponsors. Our solution was to create a set of slide templates that doesn't replace this documentation, but communicates the essence of what is in these documents.

These slide sets are supported by a common methodology and set of terminology, which simplifies the communication. Our common methodology includes regular reviews, usually every 2 months, of this material by the program team. The emphasis during these reviews is what has changed in the last 2 months. The terminology includes terms about scope, schedule, dependencies, and issues which clarify the communication.

The Project Box

The most powerful and unique of all the project management tools in our methodology is the Project Box. It provides a concise way to communicate both project scope and schedule in one slide. The terminology we use with this tool is very important because it allows us to capture the degree of flexibility that the project manager may use to balance scope and schedule.

Let's look at the axes of the Project Box (Figure 1). The vertical axis of the box symbolizes feature content or attributes, e.g., the features, quality level, performance, cost, of the project. The horizontal axis symbolizes the schedule aspect. If the project is thought of as a space or "box" within this attribute-schedule plane, then the upper left corner of this box represents the best, most complete attribute set available at an early, realizable date – what we call the "aspirational plan". The lower right corner of the box represents a scaled back feature set available at a more conservative completion date - the "exposed date."

Early in the project, one of the best uses of the project box is to help negotiate your empowerment envelope with your sponsors. The empowerment envelope is the degrees of freedom that the project manager can exercise. It forces early decisions about what features must be in the product versus the ship date of the product. By having these conversations early, you avoid having these conversations in a crisis mode later, when finger pointing is likely to destroy teamwork, right when it is needed most. The discussion revolves around 2 key areas: what is ideal and what is worst case. The ideal discussion focuses on: what does the market require, what is the best the project can do, and when might that be done. The worst case discussion involves topics like: what is the latest date I can deliver and what is the minimum set of functionality that

we need to have a viable product. Thus the box itself becomes the empowerment envelope; the project manager has the freedom to trade-off features and schedule within the box. You're building flexibility into your project plan: what can I sacrifice if I have to. After the Project Box is approved by the sponsor, the team knows that they only have to go back to management if they are outside the box – you can't deliver the Base Plan items or the schedule goes past exposed date.

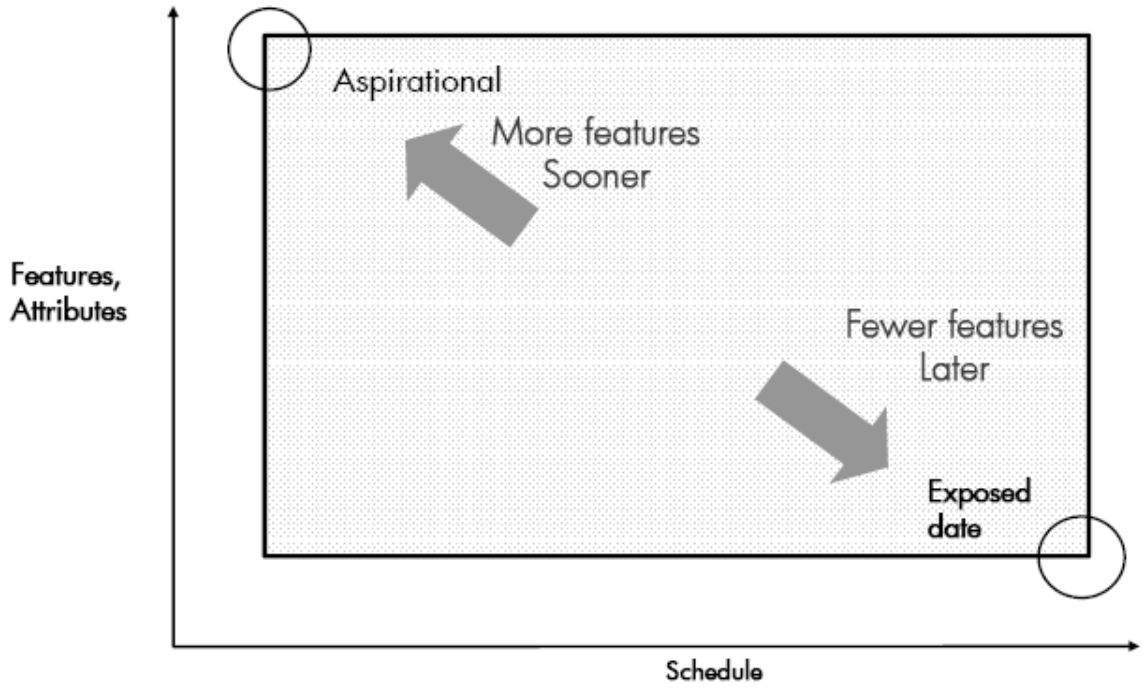


Figure 1, Project Box Dimensions

Before we had this tool, we generally found that management became involved with projects early on as the project was defined, and then later in the project when some crisis developed. Usually the project manager found he couldn't do everything he had promised on schedule, and he needed management help on what to trade off. These were usually unpleasant experiences, caused largely because of unforeseen circumstances. The project manager was put under pressure to explain what had happened and what he was going to do about it.

The other key purpose of the Project Box is to serve as a communication tool between projects to illustrate what is actually in the scope and schedule of the project as well as what are the flexibility options in this scope and schedule.

Let's look at the Project Box Template (Figure 2) to see how you fill out the Project Box. The vertical dimension of the box is scope: the features and attributes of the project.

1) Base Plan Items
2) Working Plan Items
3) Aspirational Plan items
4) Out-Plan items

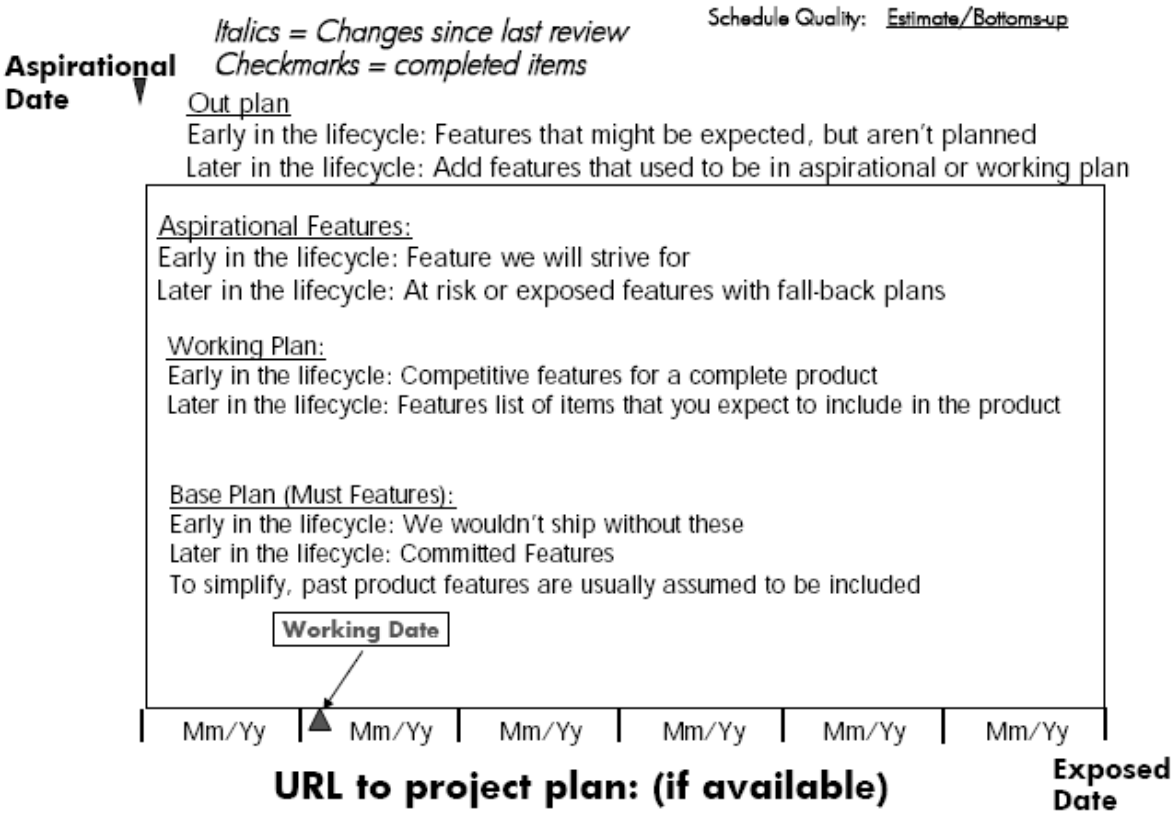


Figure 2: Project Box Template

You see that the vertical dimension has different meanings, depending upon whether it's early or late in the lifecycle. We separate early or late by a checkpoint we call Project Commit. It's the point in the lifecycle where we commit the features, resources, and schedule. The first version of the Project Box is often created at the same time as the requirements are developed. Here we may use the Project Box to talk about market needs. Base plan items are those you wouldn't ship without. Working plan items are the competitive features needed for a complete product. Aspirational features are market differentiators, or features we hope to include.

The most difficult talking point is usually around the base plan items. Typically the project sponsor will want to designate most features as base plan items. Why? Because they are worried that items that are not in the base plan won't be prioritized high enough to be completed. However, with this tool, working plan items are defined as the items needed to be competitive that you're going to work on and have every intention of delivering. The trick to defining what items are base plan items and what items are working plan items is to ask the sponsor to think about the latter part of a schedule, when they've typically had to make a call between shipping on time and having all the features they want in the product. In this context, it's easier for them to determine the base plan items; the ones for which they would delay shipping the product. In other words, they wouldn't ship this product without these features.

Another key feature of this tool is to identify the out-plan items. Those are features or attributes that the sponsor or another team might expect to be covered by this project, but that are not part of your project definition.

Later in the lifecycle, after our project commit checkpoint, the meaning of base, working, and aspirational change slightly.

- Base plan items are those that we've committed to deliver by the exposed date and we will move the schedule out from the aspirational date so that they are included.
- Working plan items are those we plan to include, but would sacrifice, if necessary, to meet the exposed date.
- Aspirational items are those items that are at risk to be included. If things go right, we should be able to include them in the product. Aspirational items need to have formal fall-back plans, that is, if we can't meet the timeframe, we have a plan to remove the feature or attribute.

The second dimension of the box is schedule. We actually state three different dates as the potential schedule:

- Aspirational Date: This is the left-hand side of the Project Box; it is the challenging but realizable date by when the project can deliver the base and working plan items. Of course we would also hope to deliver some of the aspirational features.
- Working Plan Date: This is the most likely date the team estimates program deliverables will be complete – it is represented by the triangle on the bottom of the Project Box. This is the date that partners should align their plans around, with the understanding that there is both downside schedule risk and upside schedule potential.
- Exposed Date: This is the right-hand side of the Project Box; it represents a highly realistic schedule supporting multiple non-best-case-scenarios or paths. It is the more conservative date of the Project Box that accounts for risk and the need for contingency plans. The exposed date is derived by quantifying the exposures outlined in the risk management plan. This is the date we would communicate to customers after our project commit checkpoint.

Obviously the confidence in schedules also changes throughout the lifecycle. That is shown by the Schedule Quality Line on the top of the Project Box:

- Market Need – used during the requirements phase of a project while assessing business needs
- Estimate – top-down management estimate based on previous projects
- Bottoms-up – Based on detailed work break down structures. We require this for our project commit checkpoint.

As the project progresses, we'll mark features that have been completed with a checkmark. In our periodic review meetings, we use italics to mark changes since the last review; this allows the speaker to primarily focus on changes. Of course, a key

change might be that an aspirational or working plan item has now gone out-plan. Another possible modification is a change in the working date. We also find that things that used to be aspirational may be moved to working plan after they've been further understood or potential risks don't occur. So it can go either way.

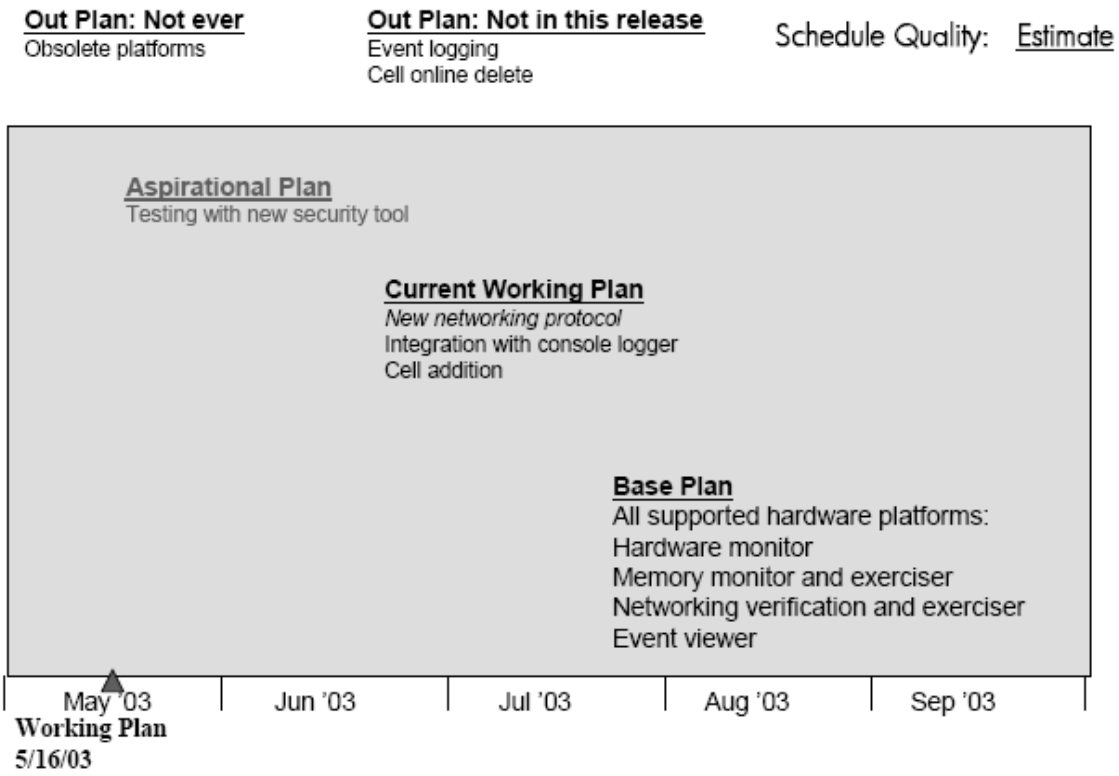


Figure 3: Example Project Box

Figure 3 is a fairly simple Project Box for a diagnostics product. The example will show how the project plan encourages communication between teams. This team has broken out-plan into "not ever" and "not in this release". They "won't ever" support obsolete hardware platforms, but for this release they state that they won't support event logging and cell on-line delete, two new features being done by other teams. These features may be included in a future version of the product. In the program team meeting this will spark the discussion of whether it is acceptable to release these two features without diagnostics support. Looking at the aspirational category, we see testing with a new security tool; perhaps because the Project Boxes overlap (the security tool may not be available in time). This might spark a discussion on whether the security tool is listed as working plan item for the security team, yet the testing of that tool is listed as aspirational item here. The aspirational date is May 1st, 2003; the exposed date is September 30th, 2003. Currently the working date is May 16th 2003, in other words, we already know that we are not likely to make May 1st, so anyone depending upon this functionality shouldn't expect it until May 16th. It is clear that in the worst case we would ship with only the base plan items at the end of September and in the best case we would ship all items on May 16th, though there is risk associated with the testing of the security tool. "New networking protocol" is italicized indicating that this has changed

since the last review. The speaker would indicate whether this was the result of a change request, or that this may have previously been an aspirational item that has moved to working plan.

Dependency Matrix

The dependency matrix usually comes straight out of the project plan, where there is typically more detail. (Some project management courses refer to these as interfaces rather than dependencies.) There are actually two of them: in-bound dependencies (your dependencies) and out-bound dependencies (those who depend upon you). By showing both sides during a review meeting, it helps us find missing or uncommitted dependencies. For example, another team depends on a deliverable from your project, but you don't know that.

dependency	requestor (lab / person)	date required	supplier (lab / person)	date promised
ACPI interface, including name space and ERS	EML: Judy and Helene	External Spec: July 12, 2002 Functional code: Oct. 15, 2002	USEL: Sashi	ERS: 7/12/02 FC code: 10/15/02
Simulator delivered to team	EML: Helene	July 30, 2002	USEL: Lea Ann	Aug 30, 2002
IPMI BT driver Need interrupt version	EML: Judy	Nov. 15, 2002	USEL:Sashi/ Ethan	interrupt version committed for 11/15/2002
Method for getting physical location from HW path	EML: Judy	Nov 25, 2002	USEL: Shashi	TBD by 7/30

Figure 4: Diagnostics dependencies on USEL

Figure 4 shows an example of the dependencies our diagnostic project has on an organizational called USEL. The dependency is described in sufficient detail that it is understandable by both projects. Some teams have begun to give these a common label between the teams. Note that the requestor has both an organization name and more important, a person's name. You know exactly who to go to for information on this dependency from both sides. The requesting organization states their date required and the provider provides a promise date. You'll notice on the first item, they actually have two dates, when the external specification is done and when the code is delivered. This matrix always uses working dates, the dates we expect to need the dependency as well

as the date it is expected to be delivered. If everyone used exposed dates, you would lose the opportunity to take advantage of early delivery of a dependency.

The dashed line is used to separate those items that have promised dates from those that are still being worked. Notice that in this case, it says that the promised date will be determined on July 30th. Red font is use to highlight items where the promised date is later than the requested date. Topics for discussion might include: is the date required still the working date, could this be split into two dependencies delivered at different times, can the quality level be lowered, or can the promise date be moved up? Quality level is often a discussion point, for example, does the code have to be fully tested, or could progress be made if it is just functional? As dependencies are met, they are usually reported once and then removed from the list.

Risk Matrix

The risk matrix also comes directly out of the project plan. The thought exercise required to formulate the risk matrix is usually much more important than the slide itself. In the end, the slide helps us communicate to our partners the high and medium risks we see on our project, which often involve them. Figure 5 shows the risk matrix for our diagnostics project. The risk identification table lists the risk, the potential impact if it occurs, the likelihood of occurrence, the difficulty of timely detection, and the overall risk. We use a fairly simple weighting scheme of high, medium, or low (HML) for each. The unique aspect of this matrix with respect to standard practice is adding the assessment around the difficulty of timely detection. For example, you may not know how solid the quality of a dependency is until late in the project. That may be too late for your needs. Recently, we've added another column, listing the schedule impact if the risk is realized.

The risk planning matrix has two aspects to it – what can be done to prevent the risk, and what is the contingency plan if the risk is realized. The contingency plan has a declared contingency trigger - what event or lack of event by a certain date would trigger the contingency plan. By communicating risks, new ideas for managing the risk often come out. We also find that sometimes dependencies get re-emphasized in the risks, the communication that your delivery may cause risk to them, and what they plan to do about it is important.

This particular project has some interesting preventative actions. Under FW and SW dependency (firmware and software dependency) they have put in place negotiated interface agreements allow both teams to work in parallel after they agree on their interface. In addition, they plan on conducting checkpoints before the due date to ensure that everything is in place to meet the due date. The partner defects risk action plan has the receiving team participate with the sending team in the design and code reviews – they know how they will use the module and therefore are likely to find defects the sending team wouldn't think of.

Risk Identification Table

Risk name	Risk description	Potential impact (HML)	Difficulty of timely Detection (HML)	Likelihood of Occurrence (HML)	Overall risk to project (HML)
FW & SW Dependency	Any software or firmware dependencies not delivered by the due date will impact EML and make it difficult to deliver fully functional tools, monitors, and diagnostics on schedule.	H	M	M	M
Partner defects	Partner deliverables containing many defects (whether they are communicated or not) will impact EML and make it difficult for us to test our tools and meet the schedule.	H	M	M	M
Location from HW path	Full investigation for proposed solution isn't complete. Findings may make implementation not feasible	M	M	H	M

Risk Planning Matrix

Risk name	Preventive Actions	Contingency Trigger	Contingency plan	Owner
FW & SW Dependency	Negotiated interface agreements, check-point before due dates, code to specification , use simulator	Dependency not delivered on time	Withdraw or reduce functionality	Helene
Partner defects	Participate in partner design and code reviews	Once defect found or defect communicated	- Add resources to help with the program to offset any slip caused by such defects, if possible - Disable any functionality that is affected. - Test around defects until fix available.	Judy
Location from HW path	Investigation in progress, in close communication with team	Investigation report on 10/31	Work with USEL to find alternatives. Possibly cut dependent functionality	Jon

Figure 5: Diagnostics Risk Matrix

Project Timeline

The project timeline is designed to communicate more information on the project schedule while keeping it simple. The top section communicates any key dates, the middle section summarizes the critical path, and the bottom section has a pointer to the work breakdown structure for those who need more details. Figure 6 shows the example from our diagnostics project. In the top section, when two dates are shown it represents the last communicated working plan date and actual date achieved. In our example, the italics on the last two items indicate that these dates have changed since the last review. The critical path in this case shows clearly how some dependencies are in the critical path. They have a dependency on the HW path locator, and then have to complete their coding, next they need delivery of hardware prototypes, and finally complete their final testing.

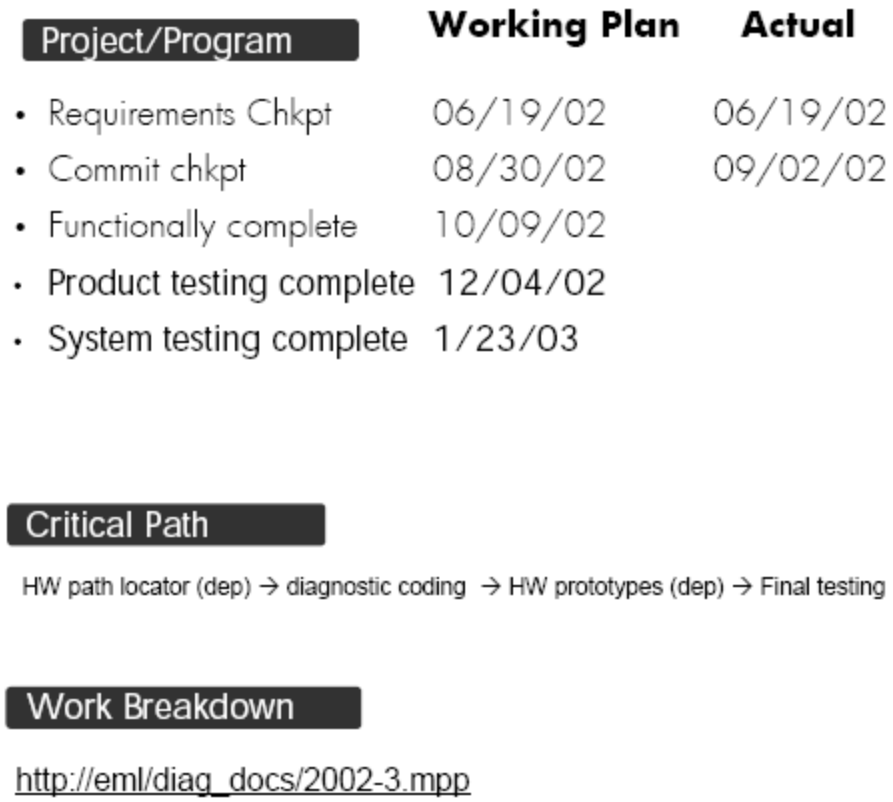


Figure 6: Diagnostics Project Timeline

Issues

The final communication slide is the list of conditions, exceptions and issues for this project. These terms are used in conjunction with key milestones for the project or program. They are defined as:

- **Condition:** Critical issue or deliverable, for which there is no foreseeable plan for resolution at the time of the milestone. Milestones are not considered complete until the condition has been met and signed-off by management.
- **Exception:** Serious issue or deliverable, not resolved at the time of the milestone, but for which there is a well-defined plan for resolution, an owner and a completion date. The milestone is considered to have been completed, but active tracking and reporting of these exceptions is required.
- **Issue:** Significant issue worthy of presentation before upper management with known impact across partner organizations. These have a plan, owner, due date for resolution. These are usually items that are not required to achieve the milestone, but need to be resolved before the project is done.

In our example, staffing is not available for the next phase, which is clearly a condition to successfully complete this phase since a required deliverable is a resource plan through the end of the project. The specification not being done is an exception; it is

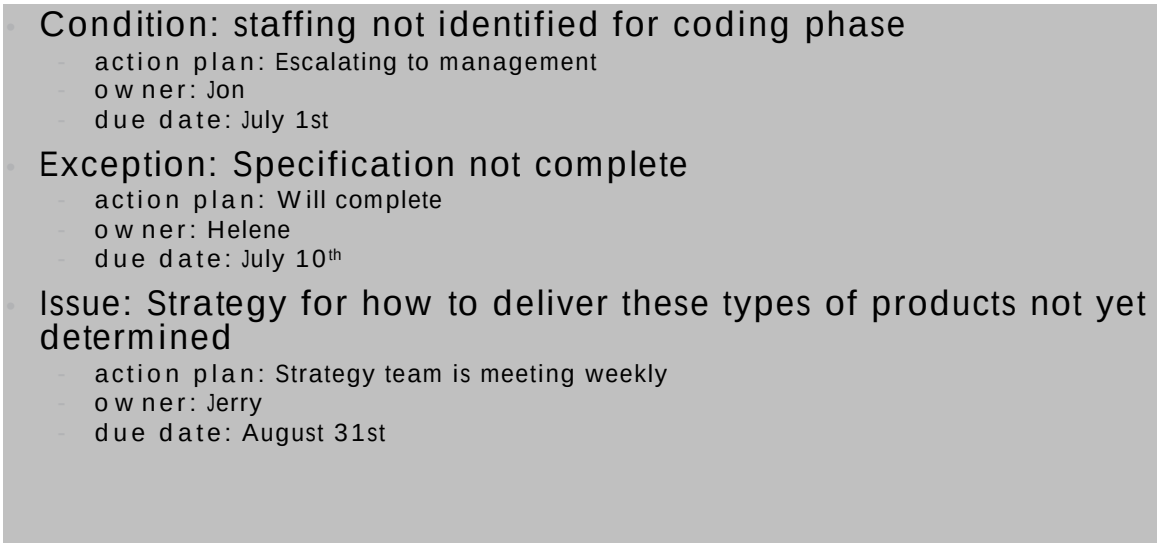


Figure 7: Diagnostics Issue Slide

required, isn't quite done, but has a plan to be done soon. The remaining issue about how to deliver this product is not an immediate concern, but must be solved before we can ship the product.

Conclusion

This use of this simple slide set has greatly improved communication among project teams in large, complex programs by providing critical information about each project in a form that only takes 10-15 minutes to review.

A key to the successful implementation of these slide templates for communication between teams is the periodic review meeting where they are reviewed by representatives from all of the project teams involved in the overall program. These generally occur every 2-3 months. The focus is always on what has changed since the last review, although the entire slide deck is always available.

The reviews have several benefits. They bring out the differing assumptions and dependencies among teams without having to read every project's documentation. It also gives our program management a view of the entire program. The standardized format and terminology simplifies the communication, helping teams focus on what is important to communicate. We always allow a project to add other explanatory slides, as long as they include these key slides.

The Project Box is the most powerful of the concepts presented here. It allows the negotiation of the empowerment envelope with the project sponsor. The conversation around the tough tradeoff decisions early in the project, arms the project manager with a better understanding of the project constraints. It also serves as a simple tool to communicate scope, schedule, and the trade-offs that would be made to other project teams.

Open-Book Testing:
a Method to Teach, Guide, and Evaluate Testers

Jon Bach

OVERVIEW

One technique for teaching and evaluating students in an academic setting is to give them an exam. One method is a "closed-book" exam, where students are tested on what they know and are not allowed to consult resources. The instructor lists questions they want the student to answer, and either the student answers them correctly or not. If a student brings a resource and consults it during the exam, it is often grounds for dismissal and expulsion.

In an "open-book" exam, however, students can bring and consult references. Unlike closed-book exams, open-book exams do not just test knowledge, but also the student's ability to acquire the knowledge on demand and apply it to the question.

As curriculum content was reviewed at Singapore's Nanyang Technological University in 1998, a survey of students showed that in order to develop creative and independent thinkers, "more open-ended tasks that reflect real-life situations – questions involving problem-solving should be incorporated in examinations, especially for higher level education." (Han C, 1998).

Another study said that open-book "removes much of the fear and emotional block encountered by students during examination, while, at the same time, it emphasizes practical problems and reasoning rather than recall of facts." (Theophilides & Dionysiou - 1996)

In my ten years as a tester and test manager, I can attest to the same experience as I experiment with open-book techniques in my role as Managing Test Lead for a software testing lab specializing in rapid and heuristic exploratory test techniques

This paper is about how instructors and test managers can use an Open-Book Testing paradigm on software projects to:

1) Teach: Acquaint testers with the main activity of testing – asking questions of the software

2) Guide: Give testers a context or a mission to frame their questions and get them familiar with the product quickly

3) Evaluate: Assess testers' questioning and critical thinking skills

ORIGIN

Open-Book Testing (OBT) as a paradigm for teaching software testers started in 2002 when I was a tester on Microsoft Flight Simulator 2004.

There were many pilots on that team, and one of the resources offered to the testing staff was a series of Ground School classes taught by one of the test managers who was also a certified flight instructor. Ground School is training that all student pilots in the United States must take in pursuit of a pilot's license granted by the Federal Aviation Administration (FAA). Topics include aerodynamics, weather, navigation, and FAA regulations.

It was not required that Microsoft Flight Simulator testers be actual pilots-in-training, but they were encouraged to attend Ground School because of past testimony from testers who said it enhanced their knowledge of Flight Simulator.

When the 8-week series of weekly classes ended, there was an open-book final exam which allowed many resources to be used to answer the questions, the most valuable being Microsoft Flight Simulator 2004 software itself, which features accurate flight model characteristics of several popular general aviation aircraft, as well as accurate weather depictions, terrain maps, navigational charts, and flight tutorials.

This is an example of one of the open-book exam questions:

A pilot is planning a trip from Singapore to Los Angeles (LAX) and has Samoa as one of the waypoints programmed into the GPS. Just in case he encounters bad weather approaching Samoa, he wants to plan a landing at a nearby airport. What airports will the GPS list as options for emergency airports and what is his best option if the storm is right in his path? (Aircraft is a DC-3 at 20,000 feet with 3 hours worth of fuel.)

In researching the answer, I was exploring features that weren't in my assigned feature area to test. But outside of the exam, I could see how the features I was responsible for could interact with others I had previously not thought about. I found myself more engaged, alert, and remembering more about what I saw when I explored. I was immersed in the product and was learning more effectively in the mission of answering the question than I had been while listening to each lecture or exploring the software on my own. I attribute this to the fact that I had defined frameworks, or missions, to answer all kinds of questions; that is to say, I had new contexts to run all kinds of *tests*.

It left me with a better model of the product, but more importantly, it created memories of my travels, leaving me with more ideas of how the software might be used. It also made me more conscious of opportunities to find failures that a wider range of customers might find because of the varied scenarios and contexts that arose in my exploration in pursuit of the answer.

PROCESS

Struck by my experience with the exam, I set out to define a procedure where testers and test managers could use an open-book framework to guide their testing.

If testing is *questioning*, then Open-Book Testing is a test-idea factory, creating and driving all kinds of question-driven contexts to reveal problems in the software.

Here is a suggested process:

1) Interrogate: The test manager or lead develops a list of questions for the testers to answer.

2) Manipulate: The testers execute actions to answer the question.

3) Observe: Testers take notes on what they find.

4) Plan: Testers determine any follow-up questions (tests) that occur to them, in preparation to debrief their results.

5) Evaluate: Testers and test manager meet to compare answers (test results).

6) Negotiate: After the debrief, testers and test managers talk about the appropriate next steps in mission or coverage

APPLICATIONS TO SOFTWARE PROJECTS

I. Teaching Testers

If an entry-level tester (or an employee transferred into the QA department) is given the question "What are the ways you can launch a setup executable in Windows XP", some of their answers may include:

- Double-click the exe
- Double-click a shortcut to the exe
- Press the Enter key on setup.exe when it has focus
- Start menu / Run / type the path to the exe
- Open a command shell and navigate to the exe

But there's also some slightly less common ways:

- Open Task Manager / New Task / type the command
- Write a batch file that calls the exe
- Control Panel / Add Remove Programs

Since the question is open-ended, it can show how the tester attains and reports new knowledge when the tester has many avenues in which to find the answer, such as:

- Intuition
- Experience
- Expertise
- Online help
- Collaboration
- Written resources outside the software

It is an effective teaching technique because it creates context-driven goals that lead to memories. It also creates opportunity – even room for misunderstanding and misinterpretation of the question could find bugs.

Paired testing is encouraged here as well because "it can result in high productivity and high creativity, serving as an effective training technique" (Kaner & Bach, 2001). Two testers working on the same list of questions can combine the avenues above, collaborating to share expertise.

Confidence can easily be built by choosing open-ended questions that require less exploration, or questions with one correct answer but that have more than one way of getting to that answer.

Here's another example in more detail. It describes the thinking behind an Open-Book question:

> *Using Flight Simulator's GPS feature, what are the 10 airports nearest to Samoa?*
>
> What this scenario is meant to uncover:

1) Is the algorithm for the Nearest (airport) function working?

2) Did the tester navigate to Samoa using:
   Slew
   Initialize position
   Flight Planner – Samoa as init point
   Direct to (Samoa as a destination)
   Samoa as a waypoint
   Airports – typing it in flight sim
   Typing it into GPS

3) Did they use the "Nearest NDB" feature?

4) Did they use the "Nearest VOR" feature?

5) Did they zoom in or out using the GPS Map?

6) Did they "cheat" and not use GPS, but Map View?

7) Did they use the "Choose country" feature and choose "Samoa" as the end point?

II. Guiding Testers

Whether a new tester is brought into an existing project or a seasoned tester is starting a new project, each has to develop a model of the software. OBT provides a mechanism to get them immersed quickly. It gives them a mission, a goal, a charter and a context for exploration.

When a test manager is handed a group of experienced testers, the manager can use OBT to task them, solving the problem that new teams often have – no destination or course to guide them on the first day.

OBT is guided exploration similar to the concept of charters in Session-Based exploratory testing (Bach & Bach, 2000). In SBTM, charters are mission statements to host time-boxed exploratory "sessions". The goal in SBTM is for the test leads and managers to collaborate with testers to write charters that are not so specific that it only takes a few minutes to accomplish the mission.

With OBT, the goal is to get more specific. Lists of questions are designed so that they can be answered in the same time as one session charter (anywhere from 1 – 3 hours). In fact, the questions can almost read like a closed book exam, with one right answer in mind, except for the fact that testers can consult resources.

Here are a few examples of Open-Book questions for exploring Microsoft Flight Simulator:

1) What are the ways to shut off the engine on the DC3?
2) List all the objects that can result in collision crashes.
3) How many approach types are there for Los Angeles International (LAX)?
4) Can the user specify snowstorm weather for Miami International airport?
5) Using the GPS, how many nautical miles is it on a direct route from KSAN to KSFO?

The test manager has to design enough questions and use their judgment in assigning how long it may take for testers to find the answers. The same questions can be given to multiple testers, or different questions, but the aim is to get them tasked and building a model of the product.

III. Evaluating Testers

Just as there is a student/instructor relationship, there is a manager/tester relationship where testers are evaluated on their agility, resourcefulness, and time management.

As the answers and notes are debriefed after a series of Open-Book questions are answered, there is a coaching and evaluation opportunity for the test lead. It's not just a way to see how much of the product has been explored and how, but a way to engage the tester and evaluate their skill, energy, and technique. It also gives a student a chance to assess themselves.

Some criteria to consider:

- Did they get the answer that was expected?
- Were they stuck?
- Did their confusion stop them or did they consult a resource to work around it?
- Did their confusion lead to an interesting misunderstanding of the question, and did that misunderstanding lead to an interesting new test idea?
- Did the tester answer the question in a way that leads to good coverage and effective test design?
- Did it lead to them being able to demonstrate a better understanding of the product?
- Did they have fun?
- Did they remember what they did?
- Did the questions inspire them to come up with their own open-book questions?
- Did Open-Book team debriefings of the answers seem to lead to higher morale?
- Did it promote speed and agility?
- Did they pushback with questions when they are confused, or did they need more context?
- Did they consult a variety of resources?

APPLICATIONS TO ACADEMIC SETTINGS

I don't work in a university setting, but part of my responsibility as Managing Test Lead for Quardev Laboratories is to teach testing classes. As I prepare to present OBT at the 4th Annual Workshop on the Teaching of Software Testing, I know that the audience is comprised of university-level instructors, and I have thought of some practical applications with OBT that may work for them.

These will be discussed more in my presentation, so I will summarize my thoughts here as bullet points in the three categories I have targeted for this paper as useful applications of OBT.

I. Teaching Testers

- Where do questions come from?
- How to pay attention to the questions you have, second by second
- Questioning as a way to learn product modeling
- Paired testing exercises: teach collaboration and test technique
- Class-wide debriefing: teaches testers what test managers expect
- How to "charter" your own Open-Book testing
- Students writing open-book exams for each other to take

II. Guiding Testers

- Types of Open-Book tests – using several types of questions to demonstrate different paths or contexts through a product
- OBT as an exercise in critical thinking when a question is vague or has several answers
- Using personas as a frame for OBT
- Acquainting students with both an intellectual "workspace" where certain answers are expected, and a "playspace" where initiative, creativity, and exploration is encouraged
- OBT as a way to orient students with a piece of software used in class

III. Evaluating Testers

- How do they approach the questions?
- How detailed are their answers?
- What initiative have they taken?
- What energy do they bring to the tasks?
- What kinds of abilities are emerging for them?
- What resourcefulness is demonstrated (i.e. what kinds of literal resources are they consulting)?
- Are their notes and narratives sufficient to convey the answer(s)?
- What kinds of Open-Book tests are they writing for others to answer?

LAB RESEARCH

Over the past 10 years, I have led many testing projects with new testers and veteran testers. But in experimenting with OBT recently, I have noticed some interesting and profound differences in team dynamics than when not using OBT.

In early December 2004, I decided to a 2-day experiment with OBT at Quardev Laboratories in Seattle where I work as a test manager.

On the first day, I assembled 5 testers with various skill sets and experience and told them to explore the Microsoft MSDN and TechNet websites. I gave them little other instructions than that. Their only mission was that in preparing to test the site's function and content, they were to "become familiar" with it in any way they thought useful.

I checked in on each of them at various times of the day, but only very informally to see if they had any questions. There were a few minor clarifying questions as to the scope of the exploration, but on the whole, everyone seemed quiet and did their best to do what they were told.

A one-hour debrief at the end of the day told a different story. I learned that some testers had been bored, others uninspired, and two even overwhelmed with the site's rich content. None of them had any context of what a user was except to see themselves as users, and the different skill sets of the testers yielded different results – some finding minor content and function bugs, others finding no problems and saying everything was "fine" and "straightforward".

The next day, I hosted two Open-Book Testing exercises. In the first hour of the four-hour experiment, they were given a set of 9 questions to answer, and in the second hour, we debriefed them.

The debrief was held in our conference room with a projector and a laptop as we each debriefed our answers to the first set of questions. Right away I saw that they were more engaged then the day before. They seemed more energetic and enthusiastic. The mission, they said, was the key. It helped them focus, they engaged in paired testing on their own, they felt more effective and inspired.

The debriefing went so well that it inspired the need for another level of focus – the addition of fictional personas (user profiles) who may use the site in different ways. As a group, we built 7 different personalities for them to focus on. Each tester took a different persona to think about and a new set of questions.

While they explored answers to the questions, the lab seemed more active with their energy as they conjectured some ways the questions may have different answers. The confidence of the newest tester on the team seemed higher as I checked in with him. He proudly demonstrated his resourcefulness in pursuit of the questions, and I agreed it was a marked difference than the previous day.

During the debriefing in the fourth hour, the success was more profound.

The team was talking much more and each of the testers agreed that today's testing using OBT was much more effective toward acclimating them with the site's contents. They seemed to be getting the idea of how to ask interesting questions despite the site being so complex.

Below are the results:

On the left column is the Open-Book question, on the right, a compilation of their answers. On the following page, I have listed the rough outline of fictional personas used to help give them context for the questions.

OBT question	Answer compilation
1) List the major MS products or apps that can be downloaded from the MSDN site.	Security Tools, Drivers, Service Packs DirectX 9, SQL Server 2005 Express Edition, Tablet PC 2005 SDK, Virtual Server 2005, SQL Server 2005 Beta 2, Server 2003 SP1 beta, Longhorn Alpha, Visual Studio 2005 Beta, Exchange Server, Windows scripts (?), SDK (general)
2) What's the difference between TechNet and MSDN?	One definition: TechNet geared toward IT (business, admins) -- MSDN toward Developers (has beta testing component)
3) For what purpose do you think the MSDN and TechNet sites were built? (1 paragraph)	Self-help -- somewhere to go to get all their answers; marketing for Microsoft, generating buzz, portal for dev, official word from MS (fixes and patches, beta releases, control content), chat function, newsgroup, wiki -- way to mitigate customer support calls and frustration -- helps tech support and build community. TECHNET -- more of a knowledge base and downloads.
4) What are the top 5 downloads?	TechNet: 1. DirectX 9.0 End User Runtime 2. IE 6.0 sp 1 3. Windows media player 10 4. Windows XP SP2 5. .NET Framework 1.1 MSDN: 1. DirectX 9.0 End User Runtime 2. XML parser SP4 3. Windows scripts 5.6 (admin) 4. Direct 9.0 SDK update 5. MS XML 4.0 sp2
5) Of the top 5 hits when you do a search for "Testing" on the MSDN site, which was of the most interest to you?	Tester #1: Community area: how people are related, how people use the site, how information travels, dev to MS and vice versa, Tester #2: bug reporting tool -- comparison and interop with to Product Studio Tester #3: Communication and support -- documentation in general, troubleshooting capability Tester #4: DevCenter -- Longhorn -- probably going to be a big area Tester #5: Longhorn -- XML -- code samples in VB (not in C#, but would be nice), backend database

6) What is Longhorn? (one sentence)	Longhorn is the next operating system (2006) -- 32- and 64-bit (?) -- 2 .NET Frameworks -- new file storage system -- WinFS -- going to incorporate a database -- designed for security -- TCI -- Avalon - presentation aspect to it -- do away with GDI and do away with DirectX, Indigo -- focus on communications and networking, shift from OO to service-oriented, using that to push themselves away from J2EE.
7) What is Whidbey? (one sentence)	Whidbey is Visual Studio Team System 2005 -- enterprise system -- incorporates Dev, Test, and management -- integrated into Longhorn more than previous versions of VS -- is there standalone capability?
8) Where can I find the top 10 MSDN Code Samples? (URL)	http://www.msdn.microsoft.com/code
OBT, part II (with personas)	
9) MSDN: What are the steps to download the SOAP Toolkit 3.0?	Downloads, Click here to see the next 40, #22 http://tinyurl.com/6ac6k
10) How would I find the system requirements for Visual Basic 6.0 Upgrade Samples?	Downloads, Click here to see the next 40, #39, see reqs http://tinyurl.com/4cj47
11) Emma found a bluescreen while ejecting a DVD from her laptop. Using MSDN, how would she find a workaround for this problem?	Knowledge base, search for "bluescreen", third link down http://tinyurl.com/3qe7s
12) What operating system was she running when she found this problem?	Windows ME
13) What type of DVD drive did she likely have?	TORiSAN DRD-U424
14) What 3 other support options does she have?	Contact Microsoft, Customer Service, Newsgroups
15) Mike Kinsman is a real PM on the Subscriptions team at Microsoft. According to his blog, what are the 4 things a user can do to reduce the size of their kit?	If you're receiving CDs, switch to a DVD Subscription, Go through your kit and remove redundant/irrelevant discs, Use MSDN Subscriber Downloads for some Archiving, Order fewer languages to receive on Disc.

16) What are the top 5 new things in subscriber downloads?	Systems Management Server 2003 with Service Pack 1 (French) - (November 5) , Office Publisher 2003 (Chinese-Hong Kong SAR) - (November 5), SharePoint Portal Server 2003 Service Pack 1 (Arabic) - (November 5) , Windows Server 2003 Service Pack 1 Beta - Build 1247 (English) - (November 4) , SharePoint Portal Server 2003 Service Pack 1 (English) - (October 27)
17) How do I find the benefits of being a subscriber?	Click the "Not a subscriber?" link on the subscriptions page -- http://www.msdn.microsoft.com/howtobuy/subscribers/
18) How do I find Chris Sells' blog on Longhorn issues?	DevCenter, Longhorn, Editor's blog
19) How can I read the current issue of the MSDN Flash newsletter?	Dev Center, Longhorn, MSDN Flash link to the right, read current issues
20) Where can I get the latest PowerPoint developer information?	DevCenter, PowerPoint, Technical Articles
21) Where can I find code samples to use to spice up my PowerPoint presentations?	DevCenter, PowerPoint, Code Samples and Tools http://www.msdn.microsoft.com/office/understanding/powerpoint/default.aspx
22) How do I register and get started with .NET Developer training sessions?	http://msdn.microsoft.com/vstudio/using/training/seminarlabs/default.aspx
23) How many international languages does the MSDN site offer?	14

Persona #1 -- Ed

Ed is Brazilian and a technophile. He works for his own software development consulting company, and creates web applications for a few select clients. He has his own website. He wants to know as much as possible about Microsoft technology, goes frequently to slashdot, subscribes newsgroups, used to be a white-hat security consultant. He is self-educated, college dropout (but was president of his high school computer club). Uses DevCenter, Downloads, Library, Code Center, Search Engine. (i.e. search for "buffer overrun") -- uses TechNet as well because he IS the only network admin.

Persona #2 – Xavier

Xavier is a Ukrainian developer. He creates applications for a development company (more specialized). He doesn't care about latest and greatest apps, but has a job to do so needs focused solutions. The meter is always running, so he needs relevant answers quickly. Uses Library, Downloads, Code Center, DevCenter. One problem he needs to solve is implementing a thermometer for his company's fund-raising website and implement rotated text -- see http://blogs.duncanmackenzie.net/duncanma/archive/2004/12/02/913.aspx. He participates in chat rooms and newsgroups for only 30 minutes a day. He's an MSDN subscriber. He creates applications for a development company (more specialized). He doesn't care about latest and greatest apps, but has a job to do so needs focused solutions. The meter is always running, so he needs relevant answers quickly. Uses Library, Downloads, Code Center, DevCenter. One problem he needs to solve is implementing a thermometer for his company's fund-raising website and implement rotated text -- see

Persona #3 – Peter

Peter is a Network Admin -- Mixture of hardware and software expertise, focused on tailoring an existing app to meet specific needs (security or productivity). He needs fixes and patches and a deployment strategy (to combat viruses and worms) -- participates in pilot programs. He has to deal with co-workers who do things they're not supposed to do with their PCs, like opening mail attachments and doing SP updates without permission. He administers new user accounts, email settings, firewall settings, new network topologies, scalability, RAS, compatibility. He's not locked down in terms of network privileges on the server. Uses Exchange, SMS, Server 2003, SharePoint for security, deployment, rollout, management, upgrades, productivity and managing downtime of company staff. He's looking for best practices up on TechNet to stay on top of things like SPs, KB articles. He's on a tight budget to acquire new tools and software -- deals with OS configurations, NetMeetings, setting up accounts, VPN -- subscribes to TechNet Flash Newsletter.

Persona #4 -- Lisa

Lisa is a Program Manager -- Focused on managing application development process, manages by overseeing their work, i.e. -- decision-maker who may need to refer to things like SDLC, whitepapers. Needs to keep up with latest technologies that may be discussed at the triage meeting. High-level overview concept only, can let herself get bewildered by the process or overwhelmed with technical jargon if she's not diligent and careful. Goes to MSDN site a few times per week, linked to TechNet through bookmark -- how often finds only some of what she's looking for on the site.

Persona # 5 -- Oscar

Oscar is a college student interested in training and certification so he uses TechNet and MSDN -- unemployed -- IT (CS) -- limited budget -- looking for free stuff, no previous knowledge with programming. Not a programmer, but looking for understanding and expertise about buzzwords (Longhorn, Whidbey, etc). -- non-professional, reads whitepapers, infrequently accesses the site -- found Via a non-Microsoft search engine (e.g. Google, Yahoo!, Altavista, etc.) -- doing research for a term paper -- code samples for school assignments -- wants to learn about C# -- people tell him it is good to know -- wants to be on the cutting edge of technology to increase chances of being hired -- uses library for research.

Persona #6 -- Rachel

Test Lead / IT rollout "goddess" who specifically works with her company's (Safeco) IT dept to do rollouts for new software, including their own as well as MS patches and betas and service packs -- beta user -- wants to find the newest, coolest app - works in support and needs to know how it interact with Longhorn, is responsible for making sure her company's existing software works with the newest MS stuff so she can deliver a story to her customers about interoperability. Subscriber to MSDN quarterly, Early Adopter -- wants to be part of the MS tech community -- very interested in blogs and user groups, newsletter info, looking for opinions and tech support -- they have bought the subscription, MSDN CDs as well as every access capability there is on the sites.

Persona #7 -- Emma

Sales for firewall application software – product interaction with MS software (firewall interacting with ICF; middleware) – hunter/gatherer of info -- needs to interpret technical stuff for herself and clients (sales pitch) Participates in newsgroups/Usergroups for customer responses/issues, find how her answers their needs – Newsletters/Mags – Technet, rather than MSDN – Less detailed tech info, more general. –- Dev center for quick tips/enhancements that can benefit her customers. Will be selling via presentations…gatherer of info from any and all sources (whitepapers, blogs, etc). Needs to be confident of her sources. Security – Consumer level user of techie info.

REFERENCES

Bach, Jonathan and Bach, James (2000), How to Measure Ad Hoc Testing, *Software Testing and Quality Engineering* magazine

Han, Christine (1998) Singapore: Review of Educational Events in 1997, *Asia Pacific Journal of Education* 18(1), p. 88 – 96.

Kaner, Cem and Bach, James (2001), Exploratory Testing in Pairs, *STARWest 2001 Conference*, slide 4

Loi, Loi Soh and Yuan, Wu (1998), Open Book Examinations, p 3. http://www.ntu.edu.sg/nbs/sabre/working_papers/10-98.pdf

Theophilides, Christos and Dionysiou, Omiros (1996), The major functions of the open-book examination at the university level: A factor analytic study. *Studies in Educational Evaluation* 22(2), 157 – 170.

Tussing, L. (1951), A consideration of the open book examination, *Educational and Psychological Measurement 11*, p 597 – 602.

Lessons Learned in Implementing a Company Metrics Program

Chris Holl

Intuit, Inc.

E-Mail: Chris_Holl@Intuit.com

Bio: Chris Holl is Intuit's Corporate Software Metrics Leader and a member of Intuit's Corporate Software Engineering Process Group. He began his career in Quality Assurance at Boeing Computer Services in the late 1970's. After helping evolve the QA group at BCS, he left to pursue new challenges in 1988. He has worked in a variety of startups and larger companies, such as Sun, Science Applications International Company, and 3Com, gaining different skills from each environment. Chris has been actively involved in the Society for Software Quality, serving in several offices since 1997 and is currently the National Liaison for the San Diego chapter. He has presented on quality and process at conferences and at various companies.

Chris joined Intuit in late 2001, forming the Corporate SEPG and helping evolve process across the company. During his three-plus years with Intuit, many lessons have been learned and significant progress has been made. His current responsibilities requirements engineering, defect management, and metrics.

Abstract: Having useful metrics is a keystone of any quality improvement program. Measurements quantify our impressions. They can help build a case for change, prioritize opportunities, and assess and communicate progress. Metrics tell us if we are changing the right things and if our changes are having the desired impact. However, many efforts to implement a metrics program fail. They are either met with resistance, fade away, or don't deliver the expected results. These consequences are due to a variety of issues in rolling out the program; some technical, most social.

Proper positioning and clear value propositions are necessary for a metrics program to progress. Success is measured at different stages, based on establishing realistic expectations for the evolution of the metrics program. The availability and quality of the data must be evaluated, and consensus reached regarding data collection goals. Initially, success may simply be reviewing good data. Later, analysis for process improvement needs to look at trends, so early "point data" must be represented over time and ultimately tied to process changes.

To gain momentum, metrics must be reviewed, and then shown to provide value. The operating mechanisms for using metrics, and the way in which the data are used, can move the program forward or kill it. At this stage, social acceptance of measurement is critical and how an organization positions metrics and behaves is paramount.

Management support of a variety of aspects of the program, social and technical, determines the ultimate outcomes. After careful nurturing, a metrics program can provide valuable information for project management, process improvement, and other business decisions. This is not the conclusion of the program – only the next beginning.

Chris Holl leads Intuit's software engineering metrics program; learning which approaches work and which fail. In this presentation he will share some of the successes and failures in deploying both local group metrics and corporate metrics and the benefits for all stakeholders.

I. Introduction

It's difficult to imagine an effective process-improvement initiative without some measurement to indicate how things are going. Metrics are critical to establishing a baseline for improvement and to quantify the two key aspects of improvement efforts: Adoption and impact. Metrics are also important for daily project management, providing information to project outcomes and to make tactical decisions with less risk.

When I started with Intuit in late 2001 one of my first assignments was to join a group of executives working to establish a metrics program. I shortly found myself leading the effort, and learning about the Y's and X's of Six Sigma. Since I had many years of experience in Quality Assurance and metrics, I thought this would be a straight-forward project. What I didn't appreciate at the time was true nature of the problem. It was not defining metrics (although this took quite a bit of time). It was not collecting data, or preparing charts. It was learning how to implement change in an organization of roughly 1500 engineers, and underestimating the reluctance of some people and groups to be measured. Implementing a successful metrics program has less to do with measurement than with people.

After an initial false start – and learning a lot about organizations and people – implementing a corporate metrics program became my highest priority. By applying lessons learned for organizational change, key metrics are now in use across Intuit.

II. The Initial Approach

Intuit is largely made up of autonomous business units due to its history of acquisitions (although this continues to change as architecture and technology are shared across the company). Each software product business unit includes a Product Development (PD) organization.

The initial metrics effort began with a group of PD leaders from various business units, such as Quicken, TurboTax, and Payroll, discussing measurement needs in the context of applying Six Sigma to software. We were all learning how to apply Six Sigma to software development (as was the rest of the industry) and working to identify our Y's and X's. The formula Y=*f*(X) says that the output of any process is a function of its inputs. By understanding the function (*f*) and managing the inputs (Xs) appropriately, one can control the output (Y). While the concept seems simple enough, applying it effectively proved challenging. We eventually decided that since our *f* was the process of developing software that our Ys should be the attributes of software most important to Intuit's business:

- Meeting Customer Needs
- Quality
- Schedule
- Cost, Effort
- Security
- Privacy

Intuit Metrics Special Interest Committee

Being charged with implementing metrics for these attributes, I set out by recruiting a group of people already involved in metrics, or interested in the project, to define the measurements. Initially the Metrics Special Interest Committee (SIC) grew quite large (over 50 people) but by natural attrition the weekly meetings soon had between 12 and 20 regular participants. The remaining members were mostly Quality Assurance and process improvement people who were interested in metrics.

Focusing on our six Ys, we applied a variation of Goal-Question-Metric[1] to define several measurements for each Y. For example, one of our schedule goals was to complete projects on time. The appropriate question is "*How reliably are we meeting delivery date expectations?*" We defined **Schedule Predictability** to be the difference between the target completion date and the actual completion date.

[1] Victor Basili's approach to identifying measurements based on goals and questions about the goals.

This immediately led to other questions. What target date? There were several set during a project, with increasing levels of confidence. Which is most appropriate to use? And how would we compare a one-month project that was a week late to a one-year project that was a week late? One would seem to be more predictable than the other.

After several months of weekly discussions, benefiting from the experience of members in the Metrics SIC, about 20 metrics were proposed to the chartering group of PD Leaders for their approval.

Over half of the proposed metrics were approved and thoroughly documented in *Corporate Measurement for Product Development, Implementation Guidelines*. This 35-page publication described reporting frequency, which projects to measure (for example, very small projects should not be measured), roles and responsibilities, and, of course, the metrics. Each measurement was defined in detail, including goals, questions, formulas, and example charts. After many months, the initial version was published and the group declared success. Within weeks the metrics started to be submitted. Not every group was able to produce all of the metrics, but we felt that eventually we would have a complete dashboard from all of the PD groups.

Within four months almost all of the submissions had stopped. Groups were no longer trying to complete their dashboards. What had gone wrong?

III. Lessons Learned

After such detailed definitions were published, crafted by a score of practitioners and approved by Product Development Leaders, why had the metrics program stopped? In retrospect, many obstacles became apparent.

Not Understanding the Use of Metrics

Although the Metrics SIC members were well-intentioned, intelligent contributors, it wasn't clear to the PD groups exactly how the metrics related to daily, tactical work, or how the metrics could help. To some, they were just submitting the "corporate" metrics and finding no value from the effort to gather the data and produce the metrics. Without clear alignment to goals or providing tactical information, the potential value of the metrics was not realized and motivation to produce them waned. While some investment of effort is required, unclear benefits eroded management support quickly, and priority for participating in the metrics program fell below tactical work.

Abuse of Metrics

Since the use of the metrics wasn't clear, there was fear that measurement would be abused to indicate the performance of groups or individuals. And on occasions this was true. There is no faster way to end a metrics program than to use the data to punish. Data may still be reported, but their accuracy will be suspect, and the tarnished spirit of the program will ensure that the metrics and proper use do not evolve.

Difficulty to Produce

The lack of consistent data and tools to compute the metrics made producing the charts labor-intensive. Since common data sources and formats were not in place, there wasn't sufficient leverage to develop shared tools. For example, each group used its own defect management system; some commercial, some developed in-house. The information collected by the tools was different, as were the data schemas and interfaces. Each group either produced metrics manually or developed simple local tools that could not be used by other groups.

Poor Data Availability and Quality

In addition, the data required to produce all of metrics was simply not available. Although almost all groups could count the number of defects they had logged, not all associated the defects with a particular release ID, which made comparing defects from release to release difficult. (Products with annual release cycles could use submit dates to identify the version.) Some sources were not databases: One group used FileMaker Pro as it defect tracking tool. Unfortunately, there were many metrics that could not be easily produced due to lack of existing data. For example, most groups did not count lines of code, and could not compute the defect density metrics defined by the SIC.

While it was certainly possible to overcome the lack of necessary data and tools, doing so would require motivation based on early success with a few metrics. The lack of clear alignment to goals didn't provide sufficient early wins to create the motivation.

Membership Profile Lacking

It became apparent that the SIC membership lacked certain types of participants. For one, it did not include enough *customers*: People who would be directly involved in the collection of the data and receive direct benefits from the metrics. There were two disadvantages to this situation. The first is that we did not get the input from the people closest to the process who could have provided important insights, such as the availability of data in their groups. Second, these practitioners did not get the benefit of our design and intent, which meant some of the things we asked for did not make sense to them and they had no basis for making improvement suggestions.

In his book *Leading Change*, John Kotter cites that one of the reasons change efforts fail is the lack of "a sufficiently powerful guiding coalition." The Metrics SIC needed to include some thought leaders and people of influence in each target organization. Without people who could make or incite change, nothing would happen in the local groups.

Too Many Metrics

The learning curve required to produce high-quality metrics and understand how to use them is significant for a single metric. The SIC had defined over 10 metrics, and the focus was on providing the metrics rather than understanding how to apply them effectively. And this lead to the largest problem in the metrics program: Lack of use.

Lack of Use

There were no operating mechanisms to review the metrics on a regular basis, ask questions, make adjustments, gain benefits, and advance the program. Some PD groups felt like they were just reporting the "corporate" metrics into a black hole. With no use, no feedback, and no evolution based on learning, it's no wonder that the program failed. In fact, given this situation, stopping the production of metrics wasn't the worst result. The worst scenario was that a few dedicated groups continued to spend unrewarded effort to produce metrics that weren't used.

Summary

In summary, the problems were:

- No clear alignment with local tactical and strategic goals
- No short-term wins
- Lack of certain key members in the Metrics SIC
- Fear of metrics abuse
- Lack of tools; high effort to produce
- Inconsistent or ineffective sources of data; no ability to leverage tools and learning
- Complete lack of some data
- Too many metrics
- Lack of knowledge about metrics and how to use them
- Lack of local management support
- Lack of use

And yet, currently metric use at Intuit is on the rise. What has changed to raise this phoenix from the ashes?

IV. Resurrection

Availability of Data

As time passed, the landscape changed so that conditions were favorable for another attempt. One significant change was that Intuit had implemented a common defect management system and so a common source of defect data was available. The fact that I had designed the initial workflow and fields ensured that key metric data would be collected. Or so I thought[2].

Increased Organizational Priority

Another contributing factor was that local groups were realizing the benefits – the necessity – of metrics and had evolved programs of their own. In fact, one group had implemented many of the metrics the Metrics SIC had defined and was reaping benefits, such as the ability to estimate project completion dates and predict product quality. In other cases I used the common defect data to provide metrics to high-visibility programs, giving managers new insight into the schedule and quality of the program and data about which projects were contributing more defects and needed attention. For one program, it was the first time the managers had seen the aggregate defect discovery and closure rates from all of the contributing projects. It should be noted that some executive managers had *always* asked for data and the continued lack of dashboards caused them to raise the priority of a metrics program within their organizations. As a result, corporate metrics became my full-time job.

Alignment with Local Goals for a Pilot

Applying lessons learned from the first effort, several success factors became clear. First, it was critical that the metrics provide clear value to the local groups and that they understand how the metrics support their tactical and strategic goals. The groups must "own" the metrics and realize some short-term benefits to provide momentum to their measurement program. This meant initially focusing on tactical metrics for project management. It was also necessary to have an early win. So rather than reform the Metrics SIC, I started working directly with one PD group called Shared Development and Services (SD&S). This group became the proving ground for metrics development and use.

There were several reasons for selecting this particular group as a pilot: Their leader was a big proponent of metrics; I was located near the group; and I had started an SD&S Software Engineering Process Group that had transitioned to a new Process Manager. I then started an SD&S Metrics Working Group focused on defining metrics that supported the group's goals, such as improved quality, reduction in rework, and less schedule variance. Defining the metrics took a long time (months) and it was occasionally difficult to get the team to unanimously accept the metric definitions. Full consensus was critical in order to get complete support of the effort. When we were done the managers and participating engineers understood the relevance and value of the metrics. I then produced the metrics for them, developing tools (initially based on Excel and SQL) to reduce my own effort. Through this process I learned a lot about the data. It was then I began to understand my more recent mistake.

Improving Data Quality

Although it seemed obvious to me why I put certain metric fields, such as **Fix Time** and **Discovery Phase**, in the defect management system, it turned out that not everyone understood what they meant, how they were to be used, and how to input appropriate values. The result was that a lot of the data were useless. For example, data showed our largest cause of rework was *injected* during the Test phase. To remedy my error I developed a training class called "What Are All These Fields And Why Should I Care?" This class was given to the SD&S Metric Working Group and the new SD&S Defect Management Working Group. This last group helped improve the class and it was then delivered to all of SD&S by the leader of the group. The data quality started to improve immediately.

---

[2] Personal adage: *Why repeat old mistakes when there are so many new ones to make?*

V. Evolution Of The Basics

SD&S held monthly metric reviews called CompStats[3], attended by the senior staff and open to everyone. We started with a few metrics: Availability of hosted services, Fagan[4] Inspection metrics, and defect metrics. In the initial reviews the main discussions were more about the accuracy of the data than the meaning of the metrics, but the SD&S leader made it clear that CompStats would continue. The metrics were being used.

After a few CompStats, I turned over leadership of the Metrics Working Group to a new SD&S QA Manager – a believer and thought leader. His contributions were vital in evolving the meetings so they were seen as assistance to the project managers, rather than incriminating evidence. SD&S also established a *Use of Metrics Policy*, containing the main message that no metric "…will be used as input to an individual contributor's performance[5]." More importantly, this policy was visibly enforced and people's fear of metrics abuse faded.

The CompStat metrics were soon divided into high-level data for review in the meetings, and more detailed data for the management team outside of the meetings. Drafts of the metrics were distributed a week before the review to give people time to interpret the information, ask questions, make corrections, and prepare for the review. This ensured that there were no surprises in the CompStat and people were prepared to answer questions about the metrics. Managers started having their own group reviews.

The initial metrics were for recently-completed projects. This allowed the group to examine various metrics, question and correct the data, and understand how to interpret the information. For about six months we focused on this "point" data, meaning that each completed project provide one point on a chart. Once the data improved and comprehension of the metrics was common, I started to organize the point data into trends. Although in many cases there were insufficient data points to determine statistically-significant trends, this raised the CompStats to a new level: SD&S is now starting to think about how to affect the trends.

VI. The New Metric SIC

About this time, I created the Intuit Software Metrics Council – new names help sometimes too – with deliberately-selected members. Members had to benefit from measurement. They also needed to be in a position of influence to put the metrics into use. This included not only QA managers, but project managers. The group was also smaller that the old Metrics Special Interest Committee. This time it consisted solely of believers who already had some local metrics in place. These people understood the goals of metrics, and their groups recognized they had issues that needed improvement. The strategy was to produce short-term wins with this team that could be cited later to expand council membership.

In our inaugural meeting the group created its goals, mission, and four primary activities, incorporating experience from working with SD&S:

- Define a small set of standard metrics with clear alignment to goals that could be consistently produced.
- Develop/leverage tools to automate data collection and the production of metrics so they would not require significant effort to produce and would be consistent.
- Provide training so everyone would understand the metrics, data, usage, interpretation, etc.
- Facilitate operating mechanisms, such as CompStats, so metrics would be put into productive use.

---

[3] A concatenation of Computer Statistics, after William Bratton's work in the New York Police Department. See *Turnaround*, by Peter Knobler, ISBN 0679452516.

[4] Formal inspections, following the *Fagan Defect-Free Process*, copyright Michael Fagan Associates. See www.mfagan.com.

[5] Group metrics could contribute to a manager's performance.

The members were invited to SD&S CompStats to start sharing metrics and practices. Tools developed in SD&S were designed for general use, and improved by incorporating ideas and code from similar tools developed in other groups. This was possible because all groups were now using the common defect management system. Of course, the QA Manager who led the SD&S Metrics Working Group was a member. Other groups had also developed metrics programs in various states of maturity. Sharing experiences provided significant fuel for the group.

The council then formed its own initial working groups: One to define the common set of metrics and one to develop tools. In both cases, there was now a sizable amount of existing material to draw from. All work products from the council were published on a web site, including distribution of the tools. This enabled groups who were not participating in the council to learn what we were doing and take advantage of the results. Membership has slowly grown as the benefits are adopted by new groups.

The Tax organization is now helping drive the evolution of our reporting tools. As we started to look for more powerful tools, I approached Intuit's Enterprise Information & Intelligence group, which works with Intuit's business data. My assumption that they must have already solved similar data collection and reporting needs and could help us was not only correct, but exceeded my expectations. They had powerful analytics tools at their disposal and were working to standardize on one. To this end, they have been extremely helpful, providing licenses, training, and even automating our initial reports to jump-start our tools effort.

VII. Next Steps

Enabled by the availability of analytics tools, the Metrics Council is starting to incorporate data from sources other than the defect management system. The first integrations will be from our Fagan Inspection database, project schedule database, and source code statistics database. This presents a new challenge. The identification of the projects and products is not consistent across these repositories. For example, in one database a project may be called "Authentication 2.3" while in another it is referred to as "Auth 2.3.0." While most users understand these are the same, this lack of consistency complicates querying for the data. Once again, the data quality will be a hurdle in advancing the metrics program.

VIII. Summary/Recommendations

Implementing a successful metrics program requires the following conditions:

Engage the right people. Start with a small group who has needs or pain and *need* the program to succeed. Ensure that members have significant influence in their organizations, either by position or by character. Ideally the members already have some form of metrics in place.

Select a small number of metrics, clearly aligned with strategic needs of the members. The basics are defects, effort, schedule, and output, but make sure these work for your group.

Remove fear of measurement. Establish a metrics policy including the statement that metrics will not be used as input to an individual contributors performance review. Make sure everyone in the organization knows the policy, and visibly enforce it. (This doesn't mean publicly flogging violators, but when the policy is not followed let people know that infractions were observed or reported and that the misguided individuals have been educated.)

Begin with data that are already available. Defect data is the most likely. If possible, establish a common source for the data. If not, ensure consistency of key fields (names, values, etc.) so the queries to extract data can be same.

If needed, produce the metrics yourself. Eventually people will depend on the metric and you can transition production to the pilot group. They will need to own the metrics end-to-end. This means the quality of the data, the production of the metrics, the regular reviews and interpretation, and the evolution of the metrics.

Develop shared tools to minimize the effort to produce the data. (This may be your effort initially.)

Train people who are inputting the data from which metrics will be derived. Explain how fields are used, the benefit that is derived (ideally for them), and how to input the most appropriate data.

Start with point data. Expect and navigate past debates about data quality. Then move to trends after people understand the data for one point and sufficient points are available. Train people on the interpretation of the metrics. For example, any process includes some amount of variation. Be careful to not react to such normal variations in trends, but to ensure the data are statistically significant before trying to improve the process.

Publicize early wins. At first, simply holding regular reviews of good data can be declared a victory.

Leverage other people's energy. Listen, listen, listen to their needs and ideas. At first it may be beneficial to the evolution of the program to produce metrics they request, even if it's not clear how they will use the result or how it aligns with their goals. It is normal for groups to explore different metrics to find the best ones for their use. Metrics generally raise more questions than they answer.

If you make sure these necessary conditions are met, and set appropriate expectations for the evolution of your metrics program, you will be on your way to implementing a tremendously-powerful tool for your organization. Perhaps more importantly, you will have learned how to change the culture and embed metrics as a part of the operating mechanism for project management and process improvement.

Building more powerful functional and performance tests by blending them together

Michael Kelly

Author Bio

Mike is currently a senior consultant with Fusion Alliance with experience in software development and testing. He consults, writes, and speaks on topics in software testing. Mike is currently serving as the Program Director for the Indianapolis Quality Assurance Association and is a Director at Large for the Association for Software Testing. In his free time, he serves as a co-founder of the Indianapolis Workshops on Software Testing, a series of monthly meetings on topics in software testing. You can contact Mike at Mike@MichaelDKelly.com.

Abstract

Performance test tools sometimes work better for functional testing then traditional functional test tools. They are better designed for high volume automation (as they do not use the GUI) and they can sometimes offer easier access to an application's functionality. These tools often have more robust programming languages then functional test tools, have easier access to the system under test by not using the GUI, and have the ability to run multiple tests in parallel by simulating many users at once.

In addition, functional regression test frameworks often span the critical aspects of the system under test and are capable of providing simple performance information with relatively little modification. With targeted modifications, automated functional tests can provide meaningful, accurate, and clear performance information much faster then most performance-test tools. Given the investment many organizations make in these tools and frameworks, any small change that can yield returns should be explored.

Introduction

Companies invest a lot of time and money into tools for functional test automation and performance testing. I think this tends to lead to the belief that the best way to utilize these tools is to hire “experts” in these tools to get the best return on investment. My observation is that this leads to teams of people who are “very good” at functional test automation or are “very good” at performance testing (or more accurately load testing), but it does not lead to teams that are very good at both.

The problem with this model can be that in specializing, we sometimes lose the ability to choose the most effective tool or technique to solve the problem. Worse, many times we never really learn about other tools or techniques. Many of us suffer from an anchoring bias[1]. That is, we “anchor” our thinking relative to the limited information we have available or to our experience. This bias would direct performance test groups (or even expert performance testers) to think of problems in terms relative to their performance test experience and tools. It would also bias functional testers to choose techniques and tools they specific to their domain (or at least those that are actively marketed as such).

There are certainly many valid scenarios for specialization and for specialist groups. What is important is that we understand what we sacrifice in organizing our teams that way. Many types of testing can and should overlap, and many tools have effective uses outside of the space for which they were designed and marketed. By blending tools and techniques, we can often create tests that are more powerful. A more powerful test is a test that provides *more information* about the application or a test that provides the same information *faster*.

Blending your functional and performance tests is not about replacing traditional performance tests (those often necessary complex scenarios with synchronization points and complex usage scenarios). Instead, it is about a lower cost and faster way to gather performance information. In addition, blending your functional and performance tests is about doing your functional test automation smarter, not about duplicating the testing that you are already doing. It is about finding alternatives to your traditional techniques to allow you to get more testing done faster.

1. Using performance testing tools for functional testing

Performance test tools sometimes work better for functional testing then traditional functional test tools. They are better designed for high volume automation (as they do not use the GUI) and they can sometimes offer easier access to an application’s functionality. These tools often have more robust programming languages than functional test tools, have easier access to the system under test by not using the GUI, and have the ability to run multiple tests in parallel by simulating many users at once. In addition, I have found that performance-modeling techniques can also greatly assist in the development of functional tests.

High-volume test automation

In a recent article on high-volume test automation[2] I related an experience I had using an everyday performance test tool to assist with functional testing. We used a performance test tool to allow us to execute a 300-500 test case parallel test in a fraction of the time it would have taken using our functional test tool. By switching to the specific performance test tool we had available, we gained the following (your mileage may vary depending on your tool):

- We had a more robust programming language, which granted easier access to the test data by allowing us to write advanced conversion and data-parsing methods.
- We were no longer tied to the GUI, cutting execution time to around 45 seconds per transaction instead of 10 minute per transaction using the GUI.
- We could utilize our virtual-user licenses instead of functional licenses. This allowed us to execute tests in batches of 100, while not using all of our testing hardware resources (so we could continue working).

- We would lower our maintenance costs because many application changes (specifically GUI changes) would not result in us having to rewrite the test scripts to re-execute our tests.

We started by recording a simple performance test script and parameterizing the data that was entered into the application. This was not as trivial as we imagined it would be. When reading the HTTP within the script, we had a difficult time figuring out which data was going where. We eventually got it worked out, and documented as much as we could along the way in case we had to run through the process again (which we did). We then wrote a function that converted our test data to a data format that was friendlier to the tool we were using. We initially struggled with this step, but because the tool we were using had an actual development language (not a proprietary language), we were able to recruit developer help in writing some of the more tricky code.

When we got the initial prototype working, not only were we more easily able to enter the transactions, but we even uncovered a load limitation in the web application. Coincidentally, we would not have found this limit until much later in the project, as actual performance testing was not scheduled to occur until the next iteration. All said and done, we were able to automate the highest-risk aspect of our parallel testing (verification of calculations that were performed on the transaction data). The less risky aspects of the parallel testing (such as GUI rules) were then executed manually. We were able to process about 400 transactions (with only a couple of errors in our data translation) in about 20 minutes, as compared to the estimated minimum of 8 hours without the use of the performance testing tool.

Testing applications without a GUI

In addition to working well for high-volume functional test automation, some performance test tools are also better suited for testing specific technologies. Many GUI test tools do not offer the ability to test alternative interfaces (like a web service or an embedded system). Rather than building or buying a new tool for this type of testing, many times your performance test tools can work well in these situations.

On a recent project where we tested a web service, we developed an in-house tool to allow for functional testing using XML test cases. As we developed and used the tool, we encountered many problems relating to multiple users using the tool at the same time, configuration management of the test cases, and poor results correlation (you know all the problems you typically get with homegrown tools). The problems eventually got so bad that we had to abandon the tool halfway through our testing and we had to start to look for alternatives.

One alternative that we found worked well (for anyone on the team who had experience in the tool) was to use our performance test tool to do the testing. Not only did this tool allow us to interface directly with the service without using a homegrown interface, it allowed us to store the test cases and results in a much more manageable way. We also got some reuse by using the same scripts for performance testing since all the test cases were simple XML submissions.

In addition, because of the way the particular tool we were using was licensed, it allowed anyone to install the tool and execute a one-user test without using any licenses earmarked for other teams. The tool was free as long as we did not actually execute the tests under load. Do not let the word “free” be misleading; the organization still incurred the costs of general licensing and the general costs of ownership for the tool. We were simply able to take advantage of the resources we already had onsite.

Modeling the user community

In his article on the user community modeling language (UCML)[3], Scott Barber shows a method to visually depict complex workloads and performance scenarios. When applied to performance testing UCML can serve to represent the workload distributions and operational profiles (among other things). I have used UCML diagrams (and diagrams like UCML before I knew about UCML) to help in planning my performance testing and to help elicit performance requirements.

Models like this allow us to create reasonably realistic performance test scenarios by allowing us to build rich models of how someone might use our application. It shows states, transitions, usage patterns, and dependencies. I have found the power behind a modeling approach like this to be that it is intuitive to

developers, users, managers, and testers alike. That means faster communication, clearer requirements, and better tests. However, models like this allow us to do more then simply develop performance tests. They are also excellent tools for developing model-based automated tests and complex scenario tests.

By having a model of the application you can create a model-based framework to randomly test the states and transitions in your application. Model-based testing is a form of automated testing that generates tests from a description of an application's behavior. In model-based testing[4], you can apply various navigation algorithms to your model so your tests exercise the application in new interesting ways each time you run your tests.

In addition to model-based testing and performance testing, an application model is also helpful in scenario testing[5]. A scenario test tells a persuasive story about a user of the system. The test should be complex and reflect real use of the system. Many times when we develop scenario tests, we have to develop models like this anyway, even if they are not formal and well understood. Using a tool like UCML gives you a tool for performance testing, functional test automation, and manual testing.

2. Using functional testing tools for performance testing

Functional test tools and functional regression test frameworks often span the critical aspects of the system under test and are capable of providing simple performance information with relatively little modification. Some functional test tools provide simple performance data out of the box while other - with targeted modifications like the recording of the system clock - can provide meaningful, accurate, and clear performance information much faster then most performance-test tools. Given the investment many organizations make in these tools and frameworks, any small change that can yield returns should be explored.

Gathering performance data with function tests

In another recent article on gathering performance data[6], I related an experience I had using everyday functional tests to gather basic performance data. The project team was developing a web-enabled application written in Java and the testing team was using IBM® Rational® Robot for automated functional testing and Mercury Interactive® for performance testing. During the first release of the application, we encountered many of the growing pains and uncertainty that most projects go through. Management was very concerned about performance; for various political reasons, application performance gained a lot of visibility early in the project. We needed a way to provide rapid feedback on environmental performance.

Because traditional performance tests typically require detailed setup and a good portion of time to execute, we decided to take a different approach. We were already developing a data-driven framework for regression testing in Rational Robot. We had a significant number of smoke tests, executed several times a day, and traditional functional tests, executed on a regular basis. We decided to include a simple timer mechanism into our automation framework to gather page-load times. We then took that information and wrote it out to a spreadsheet, detailing the page that was loading and the environment in which the script was executing. The next day we ran a macro on the data to process it and format it.

This simple solution solved all of our problems (well, all of *those* problems, anyway):

- We gained a spreadsheet—management's favorite type of document—that we could send to management, with detailed time information for every page and every call to every external web service in the application.
- Because this information was gathered every time we ran a smoke test or any other type of automated test, we always had up-to-date information to help us debug the differences in all of the deployment environments—a task that would have been almost impossible without this rapid feedback.
- This data also served to audit our existing performance test scripts (which turned out to be working just fine). It is worth noting that this type of audit would work for most single or low-volume performance test results, but might not be appropriate for larger tests.

All said and done, gathering this performance data allowed us to fix problems, better utilizing our automation framework, and gave us a gold star with project management. I have rarely seen so little work pay off so richly. Implementation was trivial; four hours for one person, in my case, but it will depend greatly on how your scripts are structured. The benefits are substantial in both practical ways (fix problems) and political ways (wow management). While this is not a complete performance test, it is a first step that ensures that major performance problems are detected early, leaving the performance testing team to focus on load, stability, and concurrency.

Tools with built-in performance trending
Some tools are designed for both functional testing and basic performance testing. SOAPscope (a tool for testing web services) is one example of such a tool. It combines diagnostic tools both with capture based and with custom coding features that allow you to quickly develop and execute XML tests for your service.

One of the more interesting features SOAPscope offers is that it stores the transaction time for each XML test case it executes in a database for you to access later. This tool implements our spreadsheet example above as a basic feature. To generate the same report all I would have to do is query the SOAPscope database. In addition, the tool has a built in trending tool that will show your applications performance over multiple submissions.

SOAPscope is one example of a tool that focuses on both aspects of performance and functionality. In my opinion, tools like this tend to add more value over time in that they provide more information about the application per unit of work. With the testing tools I used before SOAPscope (mostly Mercury and Rational) I would have to do some sort of workaround or custom development to get what this provides. There are other tools that provide these types of features, when you have the choice and they are an appropriate fit, select these types of tools over the ”more traditional” less powerful options.

Using multi-threading to add a load with functional test scripts
If your functional test tool permits multi-threading, quite often you can use your functional test scripts to run low to medium load – low complexity performance tests. Again, if you include a simple timer mechanism to gather page-load times or transaction times (depending on what you are testing) you can capture performance data in a low-tech way. If you then execute the scripts in multiple threads on multiple machines, you can quickly generate load and record all of that information (at a level of detail you specify) in one location for reporting.

In the past, I have used Watir tests (implemented in the Ruby language) for low user performance testing. I have also talked with people who have done similar testing with Rational Functional Tester (which uses Java as the scripting language). Since these languages offer multi-threading, it is relatively easy to get a performance test up and running.

Of course, this method does not support complex scenarios with synchronization points and complex usage scenarios, but it is cheap and quick if you do not *need* a powerful performance test tool. These tests provide relatively good results that can be used to audit other performance tests and can be used in conjunction with your traditional performance tests if you would like to execute performance tests and functional tests side-by-side.

3. More powerful testing
All of the above examples are examples where a tool or technique was used to provide *extra* information about the software under test, not to replace some other method of testing. The basic principle is one of extending your reach into the application every time you sit down to do your testing. Blending performance and function testing is not about attempting to replace two types of testing with one, but about finding ways to learn about the application *faster*. Knowing this simple information allows for more informed decision making and planning and allows you to better use your limited resources.

For even *more powerful* testing, start to mix in a little runtime analysis. Convert your existing automated tests into high-volume automated tests. Use both your functional and performance test tools to help with

exploratory testing. Adapt your tools and practices to allow for combinations of your existing skill sets. If your skill set does not include performance testing or functional test automation, try using simple steps like the ones listed above to get you started.

---

[1] Chapman, Gretchen B., and Eric J. Johnson. *Anchoring, Confirmatory Search, and the Construction of Values*. Organizational Behavior and Human Decision Processes Vol. 79, No. 2, August, pp. 115–153, 1999

[2] Kelly, Michael. *Using performance test tools for high volume automated testing*. InformIT. May 20, 2005.

[3] Barber, Scott. *User Community Modeling Language (UCML™) v1.1 for Performance Test Workloads*. www.PerfTestPlus.com. 2003.

[4] Feldstein, Jeff. *Model-Based Testing: Not For Dummies*. Software Test & Performance. Vol. 2, Issue 1, February, pp. 18-23, 2005.

[5] Kaner, Cem, and James Bach. *Black box software testing: Scenario testing*. www.TestingEducation.org. 2005.

[6] Kelly, Michael. *Gathering Performance Information While Executing Everyday Automated Tests*. InformIT. Feb 25, 2005

Integrating Contextual Design in Software Product Development

Elsie Loh Gerthe
Hewlett-Packard Company
elsie.loh@hp.com

Abstract:

To deliver the right product that will effectively improve the customer-experience, an organization may want to consider adopting a customer-centered development process. A customer-centered process makes an explicit step of understanding who the end users are, their needs, their desires and their approach to the work. Our experience is that the customer-centered process is greatly beneficial to product development because developers meet with customers. This allows them to discover the detailed structure of customers' existing work practices and to see how their product can enable a better way of working. Contextual Design techniques support the organization of such a customer-centered process.

This paper will begin with a discussion about Contextual Design techniques. It then describes how our organization has integrated these techniques in our evolutionary development process for both new and existing products. Finally, it discusses lessons learned in its execution and plans for improvement.

Elsie Loh Gerthe is currently a Customer Connection Program Manager for the Hewlett Packard Enterprise Unix Division. She has more than 20 years of experience with leading edge computer technology companies in roles from software design to project management. In her current role, she championed the effort of integrating Contextual Design into the product development process.

Background

In the past, software developers received data from the marketing department on the key features for a product, but not how those features should operate. When enhancing existing products, software developers resorted to using defect reports and enhancement requests for designing these features. This resulted in the delivery of products and features that did not meet the expectations of the customer.

To deliver the right features that will effectively improve the customer experience, an organization may want to consider adopting a customer-centered development process. Central to a customer-centered process is the understanding of who the end users are, their needs, their desires and their approach to their work. Contextual Design [Beyer] techniques support the organization of such a customer-centered process.

Hewlett Packard came to accept and adopt Contextual Design (CD) while conducting product usability testing with their customers. The development team discovered a mismatch of what the customers expected and what the team delivered. A retrospective of this effort made the development team realize the need for better understanding of the customer. It was concluded that the subsequent product release should be developed with a higher level of customer participation. This resulted in the team investigating CD as a potential approach to achieve better customer participation.

At Hewlett-Packard, management is always looking for ways to improve the team's productivity and the delivery of high product quality with optimal customer value. Since CD promises such a value proposition, the management expressed considerable interest in adopting CD into our software development process for both new and existing products. This allows software engineers to discover first hand the detailed structure of customer's existing work practices and to see how their product can enable a better way of working.

To integrate CD successfully into an established product development process, support from both the development teams and management is crucial. The implementation of a CD approach requires a change in work habits. Developers and management are required to have an understanding of the CD techniques, and the key enablers and inhibitors for success. Changes to the system should be introduced in a controlled fashion in order to achieve a high success rate and to avoid confusion.

Context Design Techniques

There are many different design methodologies available to assist with the development of large software systems and business models. An earlier methodology such as Structured Analysis & Design [Yourdon] was more focused on the data transfers and the data transformations than the work/process it took to get things done, giving an algorithmic point of view. CD is a customer-centered approach focused on users and

tasks that become the driving force behind product development. It unifies the development team's action into a coherent response to the customer. The users' behavior and context of use are studied and the product is designed to support them. Users are consulted throughout all phases of product development and their inputs are taken into account. Such user involvement helps manage the users' expectations and feelings of ownership. All design decisions are taken within the context of the user, their work and their environment. An overview of the CD process that highlights its seven parts is shown in Table 1. This provides us with a template that we can used to tailor our new product development process.

1. Contextual Inquiry (CI) : Gather reliable knowledge about what customers actually do and what they care about
2. Work modeling : Capture the work of individuals and organizations in diagrams to provide different perspectives on how work is done, and create a shared perspective of the collected data
3. Consolidation : Create a single work practice or representation for the entire customer population
4. Work Redesign : Imagine and develop better ways of work
5. User Environment Design (UED) : Structure the system work model to fit into the work
6. Paper Prototype : Verify design early before committing to code
7. Object Modeling : Define the implementation architecture that support the work structure

Table 1. Contextual Design Process

Management and Team Commitment

To provide the team and management a good exposure to the CD techniques and principles, we arranged several sessions where other HP teams who had successfully used CD shared their experiences. These discussions generated great interest in the development team to seriously consider adopting CD into our product development lifecycle.

So, which of our product development lifecycles should we consider incorporating the CD process first? Management was well aware that changes made to any existing process would require ramp up time for the development team, potential impact to our already tight schedule and that benefits of the change may not be realized immediately. As such, the decision was made to initially integrate CD into the development lifecycle of a new product instead of an existing one.

Contextual Design in Practice

This project was to develop a new tool that would simplify the current packaging process and improved the packaging quality on HP-UX. The first thing we did was to create a cross-functional team to work on integrating the CD process into our evolutionary development lifecycle. This team consisted of three developers, one marketing engineer, one learning product engineer and one usability engineer who acted as the team lead. Trained by in-house CD experts, the team was equipped with the necessary techniques and best practices to start adopting the CD process into this new project development lifecycle. Without any prior experience, it was difficult for us to immediately recognize which parts of the process would be most beneficial to the project. However, we recognized that the individual parts of the process can be shortened or eliminated if they weren't applicable. The parts can also be further elaborated with additional techniques if they are important. For the purpose of gaining first hand experience with the CD process, the team agreed to evaluate, assess and scope each part as we saw fit. The steps we took to integrate CD into our development plans were as follows:

Planning of this project
During this phase, we defined the project focus as how the current HP-UX users package non-HP supplied software, scripts and configuration files in the HP-UX native format. We made assumptions about the identified tasks that users do to complete their work. We needed to ensure these assumptions were verified in our interviews.

Next, we needed to determine the type of customers and the target number of customers we would work with. As a rule of thumb suggested by CD, the target number of customers consists of two or three experienced users as well as novice users for each work role, two companies of each significant type of work and no more than four to six companies within one geographical area. This will help maximize differences between companies and geographic areas based on work practice differences such as law and infrastructure. From our marketing customer database, we brainstormed the list of customers that would provide the kind of data needed to validate the project requirements and also its design.

Set-up customer visits
With the assistance of our field support organization, we first communicated with the selected customers our project focus, objective, the nature and benefits of their participation. For some, they readily accepted the opportunity to meet with the product developers to discuss their challenges while others were skeptical about the Return on Investment (ROI) of their participation. As a start, we chose, according to the stated rule of thumb, to set-up the absolute minimum number of eight customers visits (2 companies/type x 4 companies within one geographical area). Once the customers agreed to host such visits, we worked directly with managers to identify the appropriate participants from their team. It was crucial to explain our motivation for doing the inquiry using the CI style interview, to assure them that the interviews would be kept confidential, and to set their expectations appropriately. When possible, we tried to

schedule the interview session at times when the users were doing the work we wanted to observe. If the users work in shifts, the interview team needed to participate in all of the shifts in order to get the complete work practice of the customer's environment.

Conducting customer visits

While it would be desirable to expose as many team members as possible to the contextual interview, we chose to limit the interview team to two where one is the moderator and the other is the note-taker to not overwhelm the customer. It is essential for the moderator to be a trained CD expert in order to be an effective process facilitator. The moderator will be required to prepare for each interview by defining the focus area and identifying a list of possible problem areas to watch for during the observation. He/she also discussed what to look for with the development group. It is easier to conduct the interviews if you have some idea of what you want to build. The note-taker, on the other hand, need not be trained, but having an understanding of the techniques will help with knowing what to pay attention to and what notes to capture during the observation. It was the note-taker's responsibility to write notes furiously and record as many details as possible.

Contextual Inquiry Interview

During the visit, we used the CI technique in our interview session to discover the customer's everyday work practices, needs and desires. The interview is usually a one-team-on-one 2-hour session conducted at the user's workplace and its nature may either be requirement gathering or design validation. It is based on a model of apprenticeship to the customer where the user is expert and the designer or developer is the apprentice. The interviewer is trying to learn how the interviewee does their work and why they do it the way they do. The interview has two parts that can be conducted in either order. In one part, the user is interviewed to understand their role and responsibility in the organization. The other part is a contextual observation where the interviewer watches the user performed his/her normal tasks. Throughout the observation, our team employed the following four principles:

- Context: The developer has to go to where the work is. He/she is to stay grounded in concrete data and doesn't abstract any data while watching the work unfold.
- Partnership: Both the user and developer collaborate in understanding the work being done. The developer must avoid the traditional interviewing relationship which tilts too much power toward him/her. He/she must resist the temptation of taking the expert role back, especially when the user is trying to use a system that the developer developed - rather, let the user lead.
- Interpretation: Both the user and developer are to determine the meaning of the observation made together. This will create a shared understanding of what is going on and help draw out any implications in the user actions and words for the given task performed.
- Focus: The point of view the developer takes while studying the work will help steer the conversation on topics that are useful without taking control entirely back from the user. The focus reveals detail within the area of work it covers, and conceals aspects that it does not cover.

Our interview team followed the users around, paid attention to their attitudes, behaviors, and the workplace and asked for copies of any artifacts and documents used. They learned what activities precede and follow the task they are observing, and got a feel of the environment. When clarification was warranted, the moderator would interrupt the normal work flow and ask questions. At times, the moderator might need to ask provocative questions to look beyond the work that is in front of him/her. The user will be asked to verbalize why they do things in that manner and what do they intend to accomplish.

What are other situations we encountered during the interview? There were times during the interview, the users were just learning about a new task and we would seize the opportunity to see and probe how they learned. The scheduled visit could happen during a user's "slow" week. If this was the case, we would ask the user to recall or simulate recent events. For decision making work which was not observable, we would probe to discover what they were using as criteria; and ask them to verbalize their thought process. If the users were uncomfortable with revealing what they actually do, we assured them the interview is confidential and offered to sign a confidentiality agreement.

Work Modeling

At the completion of each CI interview, it is the rule of thumb to conduct the debrief session within 48 hours of the interview and with no more than 6 people. We would schedule this interpretation session to be of the same length as the interview session but our first interpretation took longer as we were getting used to the process. In this structured meeting, our design team had the opportunity to hear, model and respond to a single customer interview. All team members had the opportunity to learn about each others' perspective on the data. Each team member was assigned the roles of insight capturing, work breakdowns, work notes, design ideas, data holes, and work modeling. At the same time, CD also recommended the following type of work model diagrams that capture the work of individuals and organizations be drawn:

- Work flow model captures communication and coordination between people and roles performed in service of the intent irrespective of time.
- Cultural model captures the overall climate, culture and policy in the customer's environment.
- Sequence model shows the detailed work steps in time performed to accomplish a task.
- Physical model shows the individual physical environment as it supports the work that happens there.
- Artifact model shows how artifacts are used and structured in doing the work.

During our interpretation, we chose to capture affinity notes with sequences and artifact models to provide the context. A sample of the affinity notes taken for one of our interpretations is shown in Figure 1. The affinity notes captured the user and the organizational profiles followed by key ideas emerging from the discussion. Each affinity

note contained only one thought or point with clear reference to who said or what was being done. In addition to these general notes, we also captured design ideas, user quotes, data holes and clarifying questions. The interviewer replayed step-by-step what happened in order while the rest of the team pushed to get the detailed “what” and “why” behind the user’s action. This resulted in an effective interpretation and high quality affinity notes. The note-taker for this session would capture both implied affinity notes and those the team directed him/her to take.

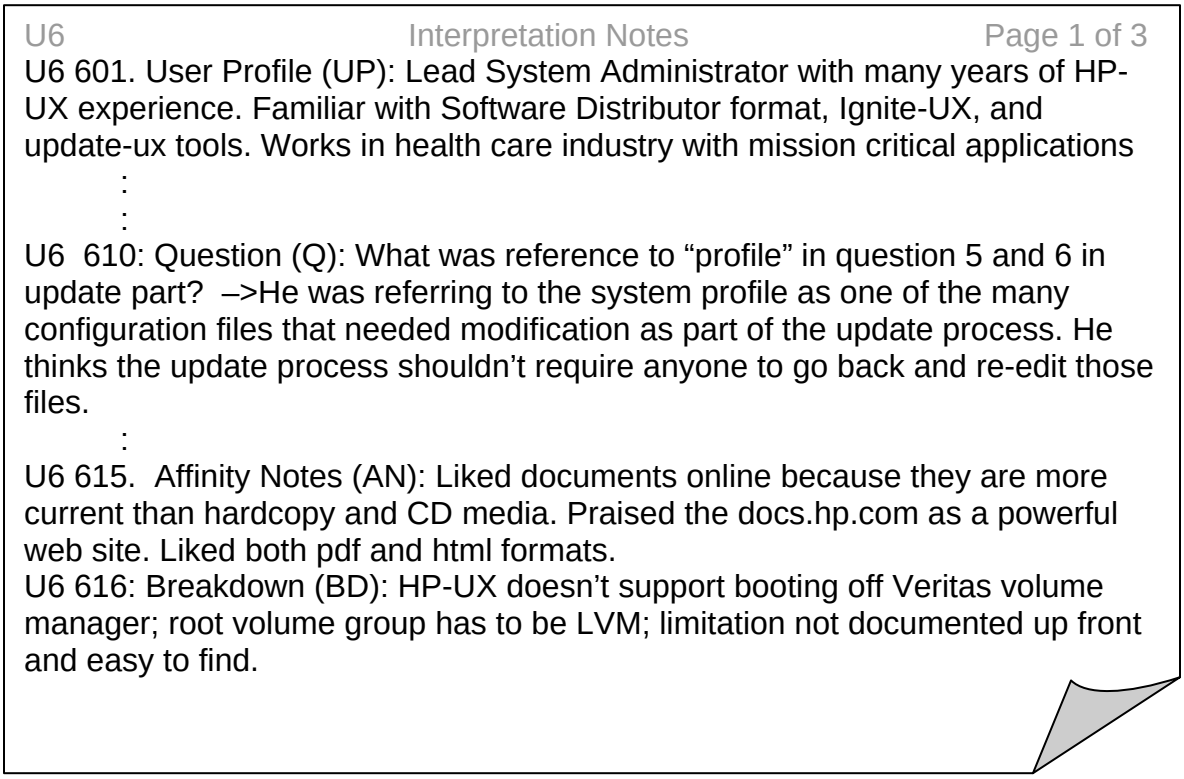


Figure 1. Sample Affinity Note

The work modeler was responsible for creating a sequence model that showed the ordered steps that the user performed to complete a task, and an artifact model that showed how artifacts were used and structured in doing the work. In our sequence model, we opted to capture the work steps of what the user is doing at the level of their action including any thought steps; and user interface steps of how the user clicks and inputs something into the user interface. Figure 2 is a sequence model showing steps captured at a high level of detail.

The collected artifacts were used throughout this session to help us understand how they were used during the user’s activities. As issues, intents, breakdowns, and usage of the artifact were identified, the work modeler would annotate them to our artifact model. At the end of the session, the key insights were also captured. Later, we shared the results generated with the rest of the development team.

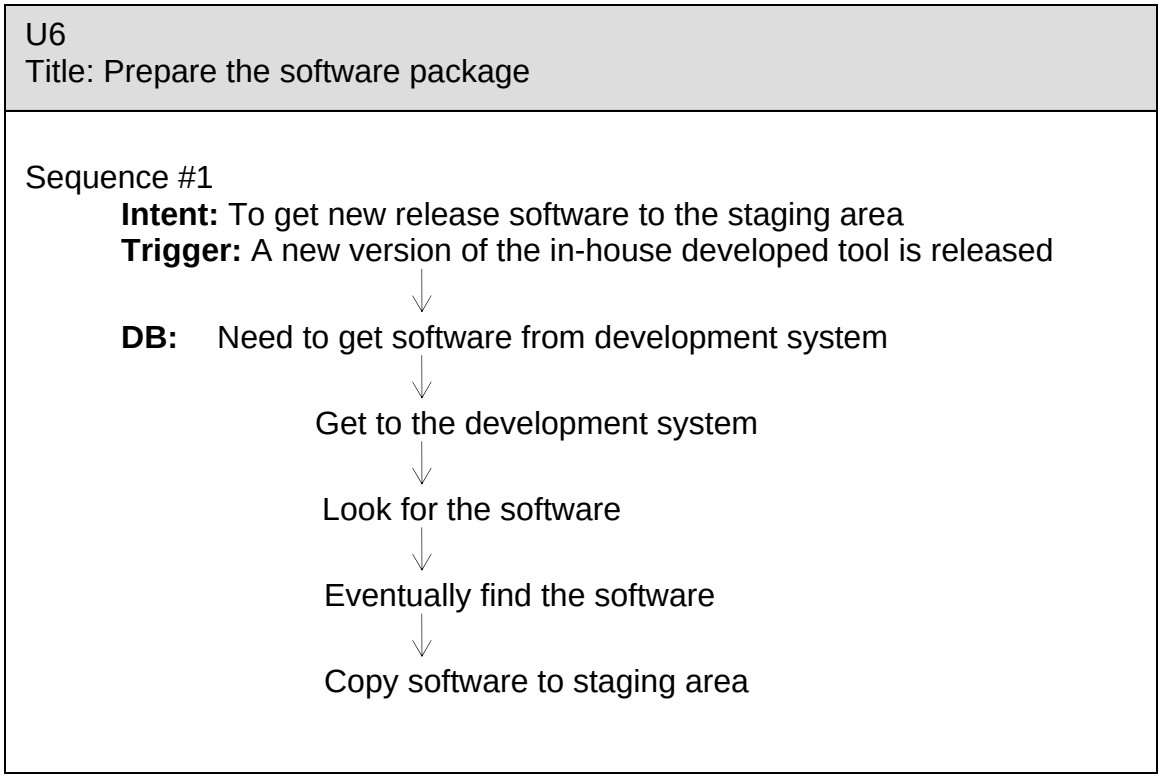


Figure 2. Sequence Model

Consolidation

After the completion of all debrief sessions, we invited the designers of the system such as developers, marketing engineers, usability engineers, and project managers to participate in this group effort. The goal here is to create a single view of the customer and to reveal common underlying patterns and incorporate variations. Each CI session results in a set of models. They need to be consolidated into one view of the work.

In this step, we proceeded with the work model consolidation and affinity diagramming which led to the creation of a single work practice for the entire customer population. The consolidated work models revealed common strategies, intents, structure and scope while retaining and organizing individual differences. The affinity notes from the interpretation sessions of all users were used to build an affinity diagram, as shown in Figure 3, a hierarchical representation of the scope of issues in the work domain. From the affinity notes, we also captured all the requirements into a document to be used for defining the scope of each incremental release.

Next, we worked on visioning and storyboarding the tasks. The vision tells the story of how our customers will perform their work on the new product we designed. It helped our team come quickly together to a shared direction. Details of the vision were being developed into the storyboard which acted like high level use cases or scenarios that showed how people will work with the new product.

1. **I plan my deployments**
2. **I deploy software on my systems**
3. **I verify my filesystem**
4. **HP Give Me Good Information**
   - 4.1 I appreciate the different types of documentation HP supplies
     - A. I like HP's manuals (4.1.1)
     - B. I like the media kit documentation (4.1.2)
       1. Finds release note, upgrade guide very useful. (243 U2)
       2. Media kits and documentations are helpful. Manuals, README are easy to follow.(928 U9)
       3. Recent release documentation seemed better. SD and Omni
     - C. I like HP's online documentation (4.1.3)
     - D. HP does a good job communicating to me (4.1.4)
   - 4.2 I need better documentation content
   - 4.3 I can't find the documentation I need
   - 4.4 I do not need documentation
   - 4.5 I need a better e-delivery solution

Figure 3. Affinity Diagram

User Environment Design

Based on the vision and storyboard tasks, we built a User Environment Design (UED) for our new product that captured the structure, function and flow of the system. The User Environment Design kept our focus on the structure of the system from the users' perspective and leaves out the user interface and implementation details. This is the key value of the UED to ensure we don't lose coherence of the system for the user as we get into the implementation. Using the UED, the project manager and team proceeded to plan and decide what tasks went into each incremental release of the product. Next, the team started the UI design and object modeling tasks in parallel for the next available incremental release.

Paper Prototyping

From the UI design, we developed a rough mock-up of the system. The benefits of the paper mock-ups are that they are easy to produce, tell customers that your product can be changed, and allow you to easily get feedback for several alternatives. They support continuous iteration of the new system and keeping it true to user needs. With this paper prototype, we conducted three rounds of testing at the users' workplace for the purpose of improving the system and driving detailed user interface design. The users were thrilled to see the new system supporting their work process. When they discovered problems with the new system, they were more willing to engage in the redesigning effort to fit their needs. Here the users are made design partners in our new product development. It is important to find and fix errors in the design early before too many resources are invested in the detailed design and code writing. Again, after each

interview session, we interpreted the results within 48 hours. The requirements document, the UED, and the use cases were updated as deemed necessary.

Through several paper prototype sessions, the development team refined the system design and drove the user interface design. Also, throughout the evolutionary development, whenever there was a disagreement among developers over design alternatives, we quickly mocked the alternatives up and tested them with the users. In this way, we shortened the arguments among ourselves and made customer-oriented decisions.

Object Modeling and Incremental Implementation
When the system design was completed, our development team moved into an object-oriented implementation. We walked through the development of use cases based on the UED and storyboards. Next, we translated these use cases into an object design.

With the complexity of the system, the delivery of the product would need to be divided into a series of releases, each addressing a coherent task. At the end of each incremental release, we validated the release deliveries with our users using the CI interview techniques. Based on the release deliveries, our users performed a set of tasks relevant to their environments and provided us their feedback. The feedback was tracked in a database and categorized as ‘issues to be resolved immediately’ or ‘issues to be resolved at a later date’. In this same validation session, we might also validate the design for the next release.

For this new project, we integrated all parts of the Contextual Design process into the early phases of requirements gathering, project scoping and planning, but only the Contextual Inquiry technique was used beyond the implementation phase. There was quite a big learning curve for the development team to come up to speed and be productive. Despite the initial slow ramp-up of the project, we managed to deliver our product on time and met the expectations of the users.

These techniques could also be applied to our existing products in the area of usability improvement, new functionality, or next release process changes. The starting process would be the creation of the reverse UED for both our products and that of our competitors. With the reverse UED of our products, we would quickly identify the structural problems that needed attention while that of the competitors would show where we could gain substantial competitive advantage. Using the reverse UED as the focus for the follow-on paper prototype sessions, the team would engage the users in the re-designing effort.

Lessons Learned

It took us over six months to get the project discussed above off the ground which was slightly longer than what we had scheduled for. Although all team members had good exposure to the CD process, not everyone was initially sold on the CI techniques in

particular. We asked them to be note-takers during the interviews. By sitting with the customer we began to appreciate the user's issues and understand why these mattered. Taking the CD process through a new project once has taught us the following lessons that we can apply to further improve our current processes.

Contextual Design is a collaborative process. Much of the work is done in teams followed with sharing and reviewing by the whole team. To ensure the success of interpretation and sharing sessions for those involved, it is important to set ground rules for communication of the findings. For example, everyone is encouraged to speak out about his/her ideas and these, along with all issues, will be captured as affinity notes, allowing the meeting to move along. Detailed discussion about a specific design idea or issue will be discouraged. Without ground rules such as these, the interpretation and sharing sessions could become unproductive.

When we were generating the list of customers from whom we wanted to gather data, we chose to identify a greater number than the recommended rule of thumb. This accommodated the need to contact another customer when the targeted customer was unavailable or decided not to participate.

We have also encountered situations where customers are really keen in participating but could not accommodate an on-site visit. In such cases, we made an exception and conducted a remote session via conference call and web-based meeting tool. Though the experience may seem less effective initially, we did gather the data we were looking for.

When setting up customer visits we learned to avoid getting too far ahead in lining up the visits and locking down a set of customers that potentially might have the same work practice. As we studied the data from the first couple visits, we had the opportunity to make adjustments to our focus area and the targeted customers for future visits so that we could discover different work practices. Another best practice that we learned during the initial use of CD was to conduct a pre-visit meeting with those we would be interviewing and make sure they understand what will happen during our visit.

For every CI Interview, we also learned we could not assume that the users remembered its principles from previous sessions that might have taken place months earlier. During one of our visits, the interview team realized the user wasn't quite sure about the intent of our detailed probing into the course of a specific action. It is therefore prudent for the moderator to restate the principles at the beginning of the session to avoid confusion or misrepresentation.

During the Consolidation and Work redesign effort, we originally neglected to reserve a room but found out very quickly having a common design room is essential. It should have lots of wall space for hanging vision, storyboards, issues and design ideas where team members can easily review them throughout this effort. Upon completion of this effort, we had posted all the work models, affinity notes, affinity diagrams, issues and storyboards online for ease of access.

Not only would the created vision for the new product be useful to the development team, it had different parts that are important to marketing, the delivery team and the support team. From this vision, marketing could develop its marketing messages and business plans while the development could start investigating the underlying technology. Both the delivery and support teams would have a heads-up on what to expect of them. These groups would start pursuing their separate focus areas in parallel before any detailed work would be required.

The UED is not limited to only creating new products. It could also be used to analyze competitive products because it allows you to compare the products at a structure level instead. The UED may be used to solve usability problems within an existing problem as it will reveal the many structural problems.

Conclusion

The Contextual Design techniques have given our development team the chance to deliver a new product that effectively improves our customers' current experience on HP-UX. Building on this successful experience, our management agreed to integrate a variant of the Contextual Design process into our existing products development effort. Here, the CI interview, EUD and paper prototypes techniques are being utilized extensively while the rest of the CD process had been shortened and incorporated into the interpretation session at the end of the CI interview.

References

Beyer, Hugh and Holtzblatt, Karen. Contextual Design. San Francisco: Morgan Kaufmann Publishers, Inc., 1998.

Holtzblatt, Karen, Wendell, Jessamyn Burns and Wood, Shelley. Rapid Contextual Design: A How-To Guide to Key Techniques for User-Centered Design. San Francisco: Morgan Kaufmann Publishers, Inc., 2005.

May, Elaine L. and Zimmer, Barbara A. "The evolutionary development model for software – includes related article on HP's Software Initiative – Company Operations," Hewlett-Packard Journal, August, 1996

Yourdon, E. Modern Structured Analysis. Englewood Cliffs, NJ: Prentice Hall, 1989.

Title:

An Interactive Workshop – Quality Systems, the Virtual Wall, and Integrity in Outsourcing

Author:

Catherine Casab

Manager, Quality Systems

The EmTek Group

3563 Investment Blvd., Suite 3

Hayward, California 94545

casab@emtekgroup.com

Biography:

Catherine holds a B.S. in Computer Science and a M.S. in Business Management from the State University of New York Institute of Technology at Utica, NY. She has over 15 years experience in regulated industry development under DOD, NRC, and FDA regulations that include positions in software development, requirements and test development and execution, systems engineering, and project and quality management. She is currently employed as the Manager of Quality Systems for The EmTek Group. The EmTek Group is a company specializing in design, development, verification, and test of Embedded Computer Systems, primarily for use in Safety Critical applications.

Abstract:

Companies that outsource engineering work need assurance that their information and development activities are secure and held private, often in an environment servicing multiple customers who may be competitors. The outsource service provider needs to demonstrate best-in-class integrity and process control so that their current and potential clients are assured that their development secrets and trends will not be compromised. Additionally, these systems must be compliant with the requirements imposed by both regulated (FDA, ISO, DOD, etc.) and non-regulated customers.

A comprehensive quality system for outsource companies will enforce the virtual wall for the protection of their clients information.

Introduction

This interactive workshop is aimed at three audiences:

- Clients who engage outsource services will learn and discuss how to identify processes and office environment clues (is the office environment quiet, are projects discussed in open areas, etc) supporting the processes to ensure their confidence is properly placed.
- Service providers who will learn and discuss how to protect their clients proprietary information via the implementation of a virtual wall, and how to identify employees with the integrity to support client protection.
- Persons who are looking for a position within outsourcing companies will learn what employers are looking for and why. People looking to redirect or enhance their employment position will understand that integrity and information protection is vital to companies today.

As part of this workshop, we will complete three (3) discussion exercises.

- The Quality Systems exercise aims to identify policies and procedures for information protection.
- The “Virtual Wall” Chart, is aimed at signs of integrity / client protection that guide outsource service providers on policy decision and guide clients on what to look for.
- The discussion chart on “Signs of Integrity” aims at identifying clues to integrity and intellectual property respect from the standpoint of service providers, clients and potential employees.

Proprietary Information and Outsourcing

Let’s start with some definitions to facilitate communication.

Proprietary Information: “The level of confidentiality given to a document or information by the owner. The owner of information considered proprietary will use the designation to limit dissemination of the information. The term is sometimes interchanged with “trade secret”. Examples include: financial, marketing, research & development, and manufacturing information.”[1]

Intellectual Property: “Any intangible asset that consists of human knowledge and ideas. Some examples are patents, copyrights, trademarks and software. Most such assets cannot be recognized on a balance sheet when internally generated, since it is very difficult to objectively value intellectual property assets (slightly different rules apply in the case of software). They can… be included in a balance sheet if acquired, which allows a more accurate valuation for the asset.”[2]

Business has always relied on the ability to protect information and intellectual property to remain competitive. Government recognizes this need through the enforcement of various laws, for example, copyright, patent, and ethics laws (white collar crime). In more recent years, outsourcing various components of the development process has become more popular. Companies that outsource need assurance that their information and development activities are secure and held private, often in an environment

servicing multiple customers who may be competitors. Those providing the outsource service need to convey the integrity and process control necessary to instill confidence in their current and potential clients.

The service provider, as an independent company, also has intellectual / proprietary information to protect and the systems that protect the clients will also protect the service provider.

Copyright, patent, and other laws related to business ethics provide some protection in the form of a deterrent, but if you are at the point of enforcing those laws, the damage has already been done. The key here is prevention of both intentional and inadvertent transfer of information. Prevention requires integrity, process, and the necessary paperwork needed for protection under the law. Although paperwork does not quality or process, particularly in the case of copyrights and patents, its mere existence is a deterrent as legal remedies require a documentation trail.

Quality Systems

Quality is more than a policy or a team. While most companies have a “Quality” group or division, the protection of customers / clients, employees, and business is the responsibility of everyone. Processes must exist to instill quality as a goal of all departments; a part of everyone’s job. Companies have multiple excellent sources to use for the development of policies, such as the IEEE standards, ASQ training and certifications (for example, ASQ CQM for Certified Quality Manager), and standards developed for ISO, FDA, NRC, DOD, and other regulated bodies. But quality policies need to be backed up with procedures understood and followed by the employees of a company. Clients look for quality policies, backed up by clear procedures, that identify an understanding and ability to provide well designed, safe and maintainable products that comply with regulations and/or project requirements. Today, we are concentrating on information and intellectual property protection.

Protecting your information - Service Provider Quality System (Use discussion to add other questions as well as to discuss how to prove your process to auditors.):

Question to Answer	**Answer Sources (people / documents)**
Is there a clear policy for protection of Intellectual Property and Proprietary Information?	Quality Manual, Mandatory non-disclosure documents, Company Policy manual
Are non-disclosure agreements required of all employees?	Quality Manual, Company Policy manual, company brochures
Are there written policies for protection of documentation and electronic media?	Quality Manual, Document / Media Control Procedure(s)
Are there identified people / departments for protection of information?	Quality Manual, Document / Media Control Procedure(s)
How is information for individual clients protected within the service providers work environment? (virtual wall)	Quality Manual, Company Policy manual, company brochures (to be discussed further)

Question to Answer	Answer Sources (people / documents)

The Purpose of the "Virtual Wall" (line in the sand)

Steve Mezak, CEO of Accelerance, Inc., identifies being "Promiscuous with your Intellectual Property" as one of "The Seven Deadly Dangers Of Outsource Software Development" (Reference #3 at the end of this paper provides source information for this article if you have a desire to explore all "Seven Deadly Dangers"). Quality is part of every step of the software development process, however integrity must be maintained, between development engineers and test engineers, as well as between groups within the same outsource provider working on projects for different, possibly competing customers. One needed element is the "Virtual Wall".[3] The virtual wall is a critical element for preventing this "deadly danger".

In the context of outsourced engineering projects, there are two conceptual walls. For the integrity of the service provider and the protections of clients, policies need to be in place to prevent casual project conversations in areas that could be overheard, for example hallway or cubicle "meetings". Project information must be protected from any person not directly involved in the project at hand.

A second virtual wall is between design / development tasks and verification / test tasks. I am not advocating division of the troops. There must be a "line in the sand" that the two groups do not cross. There are test activities in both groups. Typically, development is responsible for Unit Test and some level of performance test before "freezing" the code. The Test Engineers take the code received from development and perform functional, customer acceptance, and regression test activities. For example, quality and test engineers, depending on their background, may identify an error in software code, an electrical schematic, or mechanical drawing, but they are not responsible for making the changes to correct the error identified. By the same token, Software / Electrical / Mechanical Engineers should not be responsible for final test of their own designs for release.

In very limited cases, if called for in the contract, an exception may be made. For example, our company performed a code difference on modified code to determine the scope of the changes. This was followed by review of modifications we were delivering for use in formal V&V testing. In general, our in-house process calls for a compare of the "original" and "updated" code from configuration controlled directories, completion of the difference listings and review, "freezing" the software and a second quick check of

the frozen code against the reviewed “updated” code. The contract called for our team to work directly under the clients quality system and direction. We were given the 2 directories of code on a CD, performed our listings and review, documented the results and returned a report with a CD we burned with the updated code. There is no way for us to verify that what we reviewed is what is being used for V&V exercises. The client and project details cannot be released / discussed.

At this point, lets take a look at the “Virtual Wall:” chart and identify policies to look for (or develop / request), along with client benefits.

Virtual Wall (Line in the Sand) Chart

(Initial example partially documented to start the discussion)

OUTSOURCE POLICY	CLIENT BENEFIT	SIGNS TO THE CLIENT
All project related discussions must be held in an office or conference room with the door closed.	Protection of proprietary information discussed during execution of the project. Client ….	Policy is documented in the Quality Plan or other appropriate documentation. Quiet work environment No hallway business discussions …

OUTSOURCE POLICY	CLIENT BENEFIT	SIGNS TO THE CLIENT

The Name of the Game - Integrity

Company policy and the “Virtual Wall” alone cannot ensure protection of proprietary information and intellectual property. If the individuals working on or with the information do not have integrity, written / stated policies will be worthless.

Integrity is defined in the Scott, Foresman Advanced Dictionary as “honesty or sincerity; uprightness”.[4]

The client needs the integrity to provide the agreed tools, information, and payment called for by contract. The service provider needs the integrity to deliver the expected results and to keep the client informed throughout the process. Both need employees with integrity to carry this out consistently.

Integrity is also one of those human traits that can be difficult to assess. There is not an established “integrity test” to rely on. Let’s use the chart “Signs of Integrity” to see what we should all be looking for when interviewing potential clients, service providers (future employers?), and potential employees.

Signs of Integrity

It is all well and good to say that we need employees with integrity, service suppliers with integrity, and clients with integrity. I would like to end the presentation by sharing signs we look for to identify integrity in these areas. For suppliers, it will help to hear from potential clients what they look for. For clients, what is the supplier looking for, and if the supplier is not looking for signs of integrity, what does that tell you. For anyone who may be in the position to hire another or provide feedback into the hiring process, what should we all be looking for. Let's brainstorm for everyone's benefit.

How do clients identify suppliers with integrity	How do suppliers identify clients with integrity	How do we all identify employees with integrity
		Background check.

References:

1. University of Texas Libraries, www.lib.utexas.edu/engin/guides/proprietary.html
2. Investorwords.com, www.investorwords.com/2526/intellectual_property.html
3. The Seven Deadly Dangers Of Outsourcing Software Development, And How Companies Like Yours Can Avoid Them, by Steve Mezak, CEO, Accelerance, Inc., Copyright © Accelerance, Inc., 2005 , www.Accelerance.com
4. Scott, Foresman Advanced Dictionary, by E.L. Thorndike / Clarence L. Barnhart, Copyright © Scott, Foresman and Company, 1997

Improving Software Quality with Static Analysis Tools

John Lambert
Test Lead, Microsoft
JLamb@microsoft.com

Abstract

Static analysis is any analysis of a program that is performed without executing the program. It can be anything from reviewing source code, to compiling with warnings enabled, to running a spell checker on the documentation. Although static analysis finds defects, most people have had negative experiences with static analysis tools: using them can be a time-consuming, expensive, arbitrary process that generates noise (not defects). But, with careful investigation and thoughtful planning, static analysis tools can be a very cost-effective technique to improve software quality by finding important defects early in the project cycle – and preventing them from re-appearing. This paper will give an overview of static analysis tools, describe the benefits and drawbacks of using them, show how to define a static analysis process, and provide tips based on my experience.

Author

John Lambert is a test lead at Microsoft and works on the next-generation web services runtime ("Indigo"). He manages the team responsible for testing XML messaging security. John has bachelor of science and master of science degrees in computer science from Case Western Reserve University; his thesis was on using stack traces to automatically identify failed executions in a distributed system. John spent a summer as a program manager intern at Microsoft working on server appliances and a summer as a research intern at Cigital investigating malicious software detection techniques.

Introduction

Static analysis is defined by the National Bureau of Standards (NBS) as "analysis of a program that is performed without executing the program" [22].[1] Another definition from the Institute of Electrical and Electronics Engineers (IEEE) is "the process of evaluating a system or component based on its form, structure, content, documentation" [2]. Although these definitions include many techniques, this paper focuses on using tools to improve software quality.

1 Tools

1.1 General architecture

The general architecture of a static analysis tool follows. Not all tools support all the concepts: the simplest text searching tools don't assign importance to string matches.

1.1.1 Inputs

For software products, the input to a static analysis tool is source code, binaries, or both.[2] A few tools use both the source code and binaries: one common case is to analyze the main product executable for issues and then use the debugging information in the symbols to display relevant source code. The inputs can also be scoped using various techinques: classes, files, new code, existing code, etc.

1.1.2 Hints

Hints are guidance the static analysis tool can use to improve the analysis: "a parameter `len`X is the count of elements in array X," "function names starting with `Display` should display information to the screen," "this method might throw exception E," etc. Hints can be acquired from annotations in the product source code, user-editable files separate from the product, binary metadata, tool-specific rules, or inferences (based on usage or heuristics) about the product behavior.

1.1.3 Rules

A static analysis rule is a statement about behavior: something that happens, something that the product should do (but isn't doing), something the product shouldn't do (but is doing), etc. A rule can include a description of the problem, whom to contact about the rule, when the rule will apply, when the rule won't apply, etc. (Example rules are presented in 1.3.)

[1] Most testing would qualify as dynamic analysis according the NBS: "analysis that is performed by executing the program code" [22].

[2] While not the focus of this paper, static analysis tools can also examine logs, websites, databases, documents, system properties, alternate representations (UML, state machines), and other artifacts.

1.1.4 Violations

The output of a static analysis tool is a list of violations. Each violation contains the rule that was violated, the importance of fixing the violation, the certainty that it is a real issue, the location, how to fix the issue, and the consequences of fixing the violation.

Rule Which rule caused the violation?

Importance How important is it to fix this violation? Some violations are more important than others: a potential buffer overflow versus a spelling mistake in an error message.

Certainty How likely is it that the violation is correct? If the rule has high confidence that there is a real problem, then certainty would be high. But, if there are many incorrect firings of the rule, then the certainty is low.[3] Certainty helps with prioritization: first investigate important issues with high certainty, then important issues with low certainty. Still, most tools don't explicitly use certainty: they assign a lower importance to lower certainty (or "flaky") rules, and a higher importance to higher certainty rules.

Location Where is the problem located? This is often a physical source code location, a function name, or a binary offset. The location might also include the specific conditions (paths, assignments, external events) required for the violation to occur, or the hints that were used by the rule.

Resolution How can you fix the problem? Depending on the rule, this can be very general guidance or an extremely specific list of changes.

Consequences If you fix the violation, what will be affected? Renaming a method could cause changes to other source code, renaming a parameter could cause changes to an error message string, and throwing a different exception could cause updates to the documentation. Some changes are more breaking than others: anything at compile-time is caught quickly, but changes in behavior might require dynamic testing to verify.

1.1.5 Exclusions

Some violations aren't real issues that need to be fixed – like false positives or violations that exist to present information about the product. Each exclusion refers to a violation and includes justification and tracking information: the reason why the violation was excluded, the person who made the determination, when the violation was excluded, etc. Like hints, exclusions can be stored as annotations in the source code or in user-editable files. (See 4.5.2 for more information on exclusions.)

[3] "Certainty" reminds us that the static analysis tool isn't always right. It's promising that tools and their authors can be aware of the limitations.

1.1.6 Filters

A filter is a view over the inputs, violations, and exclusions: "all violations in X.dll that would cause breaking changes," "excluded violations with a high certainty," etc. Project exit criteria are often expressed as one or more filters.

1.2 Categories

This section describes some general categories of static analysis tools, from the least complex to the most complex, and gives a few examples for each.

1.2.1 Simple source code matching

Text search programs like `findstr` and `grep` can help check conformance to simple rules ("never call `strcpy`") and help identify potential problems ("review every call to `strncpy`"). When run against source code, these tools can be easily – and quietly – misled by macro expansions, formatting differences, pointers, etc. You can also run these tools against macro expanded source code, intermediate language, or assembly code. Learning about regular expressions [7] will help improve the quality of your analysis, but there are limits on the effectiveness of these tools [28].

1.2.2 Formatters

Some tools focus on source code formatting issues: misplaced braces, incorrect (misleading) indentation, comment headers, etc. These tools may automatically fix the problems, too. Most integrated development environments now include some built-in support for reformatting source code. GNU indent [8] is a popular tool for C source code. (These types of tools may not find too many customer-focused, runtime-visible bugs.)

1.2.3 Compilers

Compiler warnings (and other compiler settings) are highly effective static analysis techniques. One issue is that the source code can disable the warnings. Although it is easy to integrate the compiler into your build process, it is often not possible to add custom rules to the compiler. It may be difficult to vary the settings for different inputs: new versus existing versus changed code, specific classes in specific files, etc.

1.2.4 Informational tools

Informational tools help you understand the product; they usually don't check conformance to rules. These tools may provide visualizations of relationships, information about the system's structure, measurements of system complexity, or other statistics. McCabeQA [1], .NET Reflector [25], and Klocwork inSight [13] are examples of this type of tool.

1.2.5 Syntax analyzers

Syntax analyzers are primarily language-specific tools that perform syntax analysis beyond simple pattern matching: they might track issues across multiple files, support macro expansions, or perform some simple control and data flow analysis. The "lint" family of tools are examples of these tools [6, 11, 26].

1.2.6 Simple binary reflection

Java and .NET provide simple API's that allow you to write a program to reflect over the object model and metadata in binaries. These API's are less powerful than general static analysis tools, but easier to use. Even though they do not provide the static analysis tool infrastructure (exclusions, hints, reporting), they can still be very useful for simple rules ("don't refer to unreleased products or code names in the public API") or to estimate the initial scope of a problem (and determine if it is worth "fully automating" with a more complex technique).

1.2.7 Static verifiers

A static verifier checks program conformance to various properties without executing the code. These tools generally use symbolic execution to examine all possible execution paths and program states. For C/C++, Microsoft Static Driver Verifier [19] will check conformance to various driver development guidelines, as will PREfast [18].[4] For Java code, ESC/Java2 [5] and the Java Modeling Language [10] are verification tools. Microsoft Fugue [4] is a static checker for .NET systems.

1.2.8 Analysis toolkits

Analysis toolkits usually include all the features of the above tools and provide user extensibility. Microsoft FxCop [16] will automatically check .NET assemblies for conformance to a large part of the .NET Design Guidelines [15] and supports custom rules.[5] Parasoft's C++Test and Jtest [23, 24] each provide many rules and include support for source-level issues.

1.3 Rules

Each static analysis tool has a different set of built-in rules. Some tools also allow you to add custom rules via an application programming interface or a tool-specific custom language.

[4] These tools take advantage of code annotations present in the system headers [17].

[5] FxCop focuses on the design guidelines shared by all .NET languages; it does not check source code for formatting issues.

1.3.1 General

Here are some general categories of defects that static analysis can find.

Formatting Is the source code formatted consistently and correctly?

Design Is the API and object model well-designed and consistent with the platform expectations and guidelines?

Usage Does the product correctly use libraries and language features?

Safety Does the product have potential race conditions or locking issues?

Resource Does the product correctly create, manage, and dispose resources? Resources include memory, database connections, files, objects, etc.

Security What potential design and implementation vulnerabilities are in the product? Detecting potential buffer overflows by looking at dangerous API's is one example.

Intent Does the product implementation match the developer's intent?[6]

Review Are there items which need to be reviewed by others?

Informational Are there properties or statistics that are informative or useful?

1.3.2 Microsoft FxCop

Microsoft FxCop [16] checks for conformance to over 200 items in the .NET Design Guidelines [15]. Some of the categories include rules that are "review" issues requiring human intervention. For example, FxCop adds a violation for each SQL query in the product and the user excludes the violation after reviewing the query for security vulnerabilities.

Design design and "shape" of the object model

Globalization running on international systems

Interoperability COM and P/Invoke (Win32) integration

Mobility power consumption, suspend states

Naming naming methods, parameters, classes

Performance constructs and implementation choices affecting performance

Portability considerations for other platforms

Security problems that might affect the security of the program

Usage proper usage of various API's and platform features

[6] "`if (p = 0)`" is a canonical example: was the intent to check for 0 or to assign 0?

1.3.3 C++ Coding Standards book

One popular book for C++ guidance is "C++ Coding Standards" [27]. It includes 101 suggestions for developers; some of the items can be checked via static analysis tools, and others which require human review. The main sections of the book follow.[7]

Organizational and Policy Issues

Design Style

Coding Style

Functions and Operators

Class Design and Inheritance

Construction, Destruction, and Copying

Namespaces and Modules

Templates and Genericity

Error Handing and Exceptions

STL: Containers

STL: Algorithms

Type Safety

2 Benefits

This section describes the benefits of static analysis.

2.1 Finds defects

The primary benefit of static analysis is that some of the violations are defects. In addition, as you investigate issues, you can find other, non-static analysis defects in the product.

2.2 Automated and deterministic

Static analysis helps you automate something that you might have done manually. This frees up your time to focus on the things that the tools can't do.[8] Some of the issues found via static analysis could be found by humans – but the tools are deterministic and don't get "burned out."

[7] The full table of contents is at `http://www.gotw.ca/publications/c++cs.htm`.

[8] It is difficult to review a document for logical fallacies if you first need to find and fix all the spelling mistakes.

2.3 Early detection

One of the biggest benefits is that static analysis promotes early detection of issues by finding the problems before executing (or even compiling) the program. This especially benefits the "hard-to-reach" parts of the program which are usually covered later in the cycle by dynamic testing: error handling routines, rarely-used features, platform- and configuration-specific code, etc.

2.4 Early removal

Static analysis helps developers fix defects earlier (soon after introduction) and in less time (because they don't need to spend time remembering what they were trying to do). This "tight feedback loop" also promotes prevention: after making – and fixing – the same mistake several times over, developers will internalize the "right" way to do it.[9]

2.5 Promotes understanding

2.5.1 Problems

As you perform static analysis and analyze the results, you will need to learn more about each rule: why it's a problem, when it isn't a problem, similar issues to look for, etc.

2.5.2 Product

Static analysis forces you to look closer at the product. As you "dive in" to investigate violations, you increase your understanding of what's there: how the product works, other components in the system (and how they work).

2.5.3 Relative risks

Static analysis defect density can be used to discriminate between components of high and low quality [21]. It can also be used to predict pre-release defect density at statistically significant levels.

2.5.4 Planning

Information from the tools can help your planning efforts. An area with a relatively high number of violations could mean that you'll need more time for testing, bug fixing, and regression. It could be more difficult to test a component because it uses many libraries. If you need to rename many of the functions in a library, it might make sense to not write a lot of test code yet.

[9] This is like a writing with a spell checker always enabled: your spelling will improve, but you may never be able remember how to spell certain words. Still, you can end up relying on the tools too much – see 3.6 for more information on dependence.

2.6 Move from detection to prevention

From a quality assurance perspective, static analysis helps you progress from probabilistic problem detection, to deterministic problem detection, to proactive problem prevention:

- find a bug in a class (adhoc)
- find some bugs in a feature (simple or manual static analysis)
- find many bugs across the product (automated static analysis)
- prevent an entire class of bugs from ever appearing again (productization of static analysis)

3 Drawbacks

Static analysis also has drawbacks.

3.1 False positives (noise)

The most common problem people have with static analysis are violations that aren't actual problems. In [29], researchers examined several static analysis tool for C source code and found that for the two tools best at detecting buffer overflows that "average false alarm rates were high and roughly 50%." Also, on correct programs, the two tools produced "one false alarm for every 12 to 46 lines of source code and neither tool can accurately distinguish between unsafe source code where buffer overflows can occur and safe patched code." For some tools, and some (simpler) rules, this percentage will be much lower – but it may never reach zero.

3.2 False negatives (misses)

Static analysis can also miss issues due to technical or people reasons. (I think it is still a "miss" if the tool caught the issue, but a person incorrectly decided "Won't Fix.")

3.2.1 Technical

Limitations or bugs in the tool can cause you to miss product defects. One situation where the tool might have issues is when the product is "newer" than the static analysis tool: newer language features, new design patterns, new third-party libraries, etc. Another class of issues can be exposed by "mixed-mode" products: multiple programming languages, multiple language/library features,[10] multiple coding and naming styles, multiple versions of the same library, etc.

[10] Using both `malloc` and `new`, for example.

3.2.2 People

Operator error can also cause misses: not running the tool at all, running the tool against the wrong targets, running the tool with the wrong rules, running the tool and ignoring the results, running the tool and fixing the wrong issues, introducing exclusions/fixes that unintentionally (and quietly) affect future static analysis runs.

3.3 Tools

3.3.1 Tool acquisition

Not every static analysis tool will fit into your organization, process, and budget. One tool might be great fit for one developer on the same workstation, but may not work in a multi-user parallel development scenario. If you want to use a tool that costs money, then someone needs to pay for it. This can be very difficult: a survey for the embedded tools market[11] showed that 42% of the companies the respondents ($N = 322$) worked at would pay at most $100 for "'optional tools' like source code analyzers." (Almost one out of five respondents (17%) answered "We'd never use such a thing.")

3.3.2 Tool maintenance

Another part of the cost of static analysis is from the hardware, software (operating systems, databases, web servers), and system administration needed for running the tool and storing the results. If you want to maintain a complete history of a complex tool's analysis for a large system, this cost can be non-trivial.

3.4 Time

There are a few things to learn about: why certain things are problems, how the tool works, limitations of the tool, how to use the tool, and how to use the tool wisely.

3.4.1 Problems

To decide whether or not to fix a violation, you will need to learn more about the problem: why it's bad to call X before Y, the cases when it is actually okay to do so, what happens if you do, etc.

3.4.2 How the tool works

As discussed in 1.2, different tools use different techniques and each tool handles the details a bit differently.[12] It takes time to understand how the tool deals with programming constructs like macros, include files, function pointers, among others – especially in the context of your project.

[11] `http://embedded.com/pollArchive/?surveyno=12900001`

[12] A comparison of how different tools handle static call graph extraction is presented in [20].

3.4.3 Limitations of the tool

Each tool has limitations: code it can't parse, rules it doesn't include, customization it doesn't allow, rules it can't ever support, etc. [14] evaluates C++ static analysis tools and found that, of the tools available in 1997, "none offers truly comprehensive coverage of the language."

3.4.4 Using the tool

As software written by programmers, for programmers, about programming, static analysis tools can be complex and hard-to-use. Integration with commercial IDE's like Microsoft Visual Studio and Eclipse make things better, but there is still a learning curve.[13]

3.4.5 Optimizing for the tool

Part of good tool use is learning how to best optimize for the tool: if it flags your product's name as a spelling mistake, do you exclude all 500 violations – or do you add your product's name to the custom dictionary? These tips and tricks will take time to discover and apply.

3.4.6 Scale

Some static analysis tools have difficulty scaling to larger projects: a tool that might work fine for a single developer might not be efficient (or cost-effective) when used by several hundred developers across multiple continents. Also, some tools may not scale to large code bases [3].

3.5 Finds issues

Another "drawback" of static analysis is that it finds issues which are then fixed. I think that the problems, and costs, are the same whether the problems are found with dynamic testing or static analysis. Because static analysis finds many issues in a very short period of time, the costs are more noticible.

3.5.1 Analysis

It takes time to review the violations – and noise (3.1) increases the time. However, making targeted changes to the product's most frequently used code and tweaking a few settings in the tool can significantly shorten the time. For example, if the same tracing macros are used everywhere in your product, and they're being flagged every time they are used, you might want to annotate (or exclude) those macros globally. Similarly, you could add the product's name to the tool's dictionary so it is no longer flagged as a spelling mistake.

[13] One promising approach used by [18] is to run the tool (`prefast`) instead of the compiler (`cl`), and even perform this substitution transparently.

3.5.2 Standard costs

There are costs associated for every bug fix: identifying the issue, making the fix, verifying the fix, rebuilding the product, re-running tests, adding regression coverage, marking the fix as complete, etc. This also holds for static analysis bugs, although you may be able optimize your process for it.

3.5.3 Cascading changes

Although a static analysis tool can usually indicate fixes that would be a breaking change, the tool might not be able to predict "cascading" changes related to the breaking change. For example, if you add a parameter to M, you will need to change the immediate caller C of M (otherwise, the code won't compile), but what about the methods that call C?

3.5.4 Tough fixes

Since static analysis can discover complex, hard-to-find, deeply technical problems, it follows that you might need complex, hard-to-find, deeply technical solutions. These solutions can be correct according to static analysis, and wrong according to dynamic testing.

3.6 Dependence

Because static analysis finds problems, teams can become overly dependent on it: instead of doing it right the first time, they'll write a lot of code and then go back and fix all the problems detected by static analysis.[14] It's important to remember that never introducing a defect is much better than introducing a defect, detecting it (quickly) with static analysis and *then* removing it.

4 Defining a static analysis process

Starting a static analysis process is similar to other process improvement efforts: it depends on the organization and your role in it.

4.1 Determine the motivation

The first step is to determine the motivation for a static analysis process. Why do you want to do static analysis? Why now? Why do you think it'll work? What does success look like?

4.2 Define scope

The second step is to define the scope of your static analysis effort.

[14] What happens next time when static analysis isn't in place?

4.2.1 Targets

What will be the target of static analysis? Will you perform static analysis on new code, existing code, changing code, or everything? Will you run it on test deliverables, too?

4.2.2 Frequency

Will you run static analysis tools for every checkin, every build, every week, every milestone?

4.2.3 Exit criteria

How will you know if you are done? Which rules do you need to run? Which violations do you absolutely have to fix? Which problems can you fix later? Will new code have different exit criteria than existing code?

4.2.4 Timeline

What is the timeline for starting static analysis? When do you measure against exit criteria? What are the deadlines for meeting our exit criteria? What is the relationship between static analysis and starting your testing?

4.3 Acquire tools

Which tools are you using? Who pays for them? Do you need additional machines? Who will integrate the tool into our own build/release process? What happens if a new version of the tool is released during the project?

4.4 Identify roles

QA should oversee the entire static analysis process, using other disciplines as needed. It's useful to identify an "overall owner" for the static analysis effort – someone responsible for the success of the effort and who can be the "first contact" for questions about the tools and process. The overall owner will need to be somewhat technical, but they may not necessarily be a QA person.

Who will run the tool? It may be useful to have developers do the "first pass" on issues individually before moving to a centralized team. You may also choose to integrate static analysis into your build process.

Who handles tracking and reporting? This is discussed more in 4.5.

Who decides to fix or exclude? This may be left to individual developers, a centralized team, feature teams, or based off strict guidance. (This is similar to the process used for deciding to fix bugs.) In the case of breaking changes (4.6), additional participants may be involved.

Who verifies the fix or exclude? For certain test teams – and certain violations – it may not be effective to have the testers verify (and regress) every fix.

4.5 Tracking and reporting

4.5.1 Violations

Do violations go in the bug database? If so, who files them?

4.5.2 Exclusions

How do you track exclusions? One way is to use an exclusion list parallel to the source code. This keeps the source "clean" – which may be a requirement if the people running the tools aren't allowed to change the source. But, an exclusion list can be difficult to share and update in a distributed development environment. A different option is to annotate the source code: source-based mechanisms can prevent things from ever coming back, but they are difficult to find and review, they can quietly introduce unintended side effects, and they make it difficult to do a "full run" of the static analysis tool.

4.5.3 Progress against exit criteria

Who reports your progress against the exit criteria? What format do they use? Who receives the report?

4.6 Handle breaking changes

Breaking changes include anything from static structure (function renames) to dynamic behavior (throwing a different exception) to user-visible changes (shortcut keys, screen layout). Below are some additional considerations when fixing a violation causes a breaking change.

4.6.1 Approval

Who decides if you should accept a breaking change? What data do you need to make the decision? Are certain breaking changes okay to make without getting approval? (If the class hasn't been documented yet, you might rename parameters and methods.)

4.6.2 Communication

How do you tell the participants affected about the breaking change? Is it one-way or do they have a chance to "veto" your request? Should you collect our changes into one big breaking change, or should you have many small breaking changes?

4.6.3 Follow-through

Once the change happens, is there anything else you need to do? Do you need to update your test code or test documentation? Do other team members need to update the user manual, product specification, etc.?

5 Hints

This section gives some tips based on my experience.

5.1 Timing

5.1.1 Don't start too early

If you're prototyping, does it really matter if the parameter name is `anc` instead of `anchor`? Maybe, but the point is to not "thrash" fixing a bunch of violations in code that might disappear anyway or in code that you know will change.

5.1.2 Don't start too late

It can be discouraging to see thousands of "must fix" violations appear overnight: how did you miss them for so long? How will you be able to fix them all in time? Starting early will keep this number down and reduce late surprises.

5.1.3 Don't [not] fix issues for the wrong reasons

The possibility of a breaking change can discourage fixing the problem. Although it can be painful to take a breaking change, it is usually better to pay the cost now instead of later. (Once you ship, you may not be able to make the change for a long time – if at all.)

Another problem is when fixing static analysis issues (which may or may not have a direct customer impact) distracts you from fixing other bugs (which have a known customer impact): are you fixing the violation because it is the right thing to do – or because some document says you have to?

5.1.4 Don't block testing on static analysis completion

It makes sense to wait for complete some static analysis before starting to test, but you should not wait until static analysis is completely finished before starting to test. Start both static analysis and testing early – let them overlap. But, don't duplicate effort: it is generally unproductive to have testers look for (and file) bugs that will be found soon via static analysis.

5.2 Tracking

5.2.1 Don't track too closely

For some of these issues, do you really need a tester to write an automated regression test? Should testers even verify the fix? Does an entire team of people need to approve each fix?

5.2.2 Don't track too loosely

Decisions (rules to use, exclusions, fixes) and results (number of violations, progress against exit criteria) should be recorded and visible.

5.3 Results

5.3.1 Don't [not] trust it

There will always be false positives, but you shouldn't ignore or suspect the tool is wrong in all cases. Similarly, there will always be false negatives – your tools won't catch everything. Sometimes people "accidentally" disable or ignore things.

5.3.2 Don't treat violations too differently from bugs

Using bug counts to measure testers can be problematic [12]. Many of the same rules for bugs hold for static analysis violations: don't punish people for high violation counts, don't reward people who have low violation counts, etc.

5.4 Testing

5.4.1 Use static analysis for test code

It's instructive to experience the static analysis process for test code: you will learn a lot about the tools and usefulness of static analysis.

5.4.2 Use static analysis for testing the product

Some product requirements – functional and non-functional – can be expressed as static analysis rules: never call this method, always call this method before throwing an exception, etc.

5.4.3 Use static analysis for tracking/measuring

Static analysis can be useful for measuring the "surface area" of a product: what methods are exposed, what libraries does the product use, etc. This information can help with estimation, traceability, and evaluation of security properties [9].

5.5 Implications

5.5.1 Don't over-extend the conclusion

You don't want to get a false sense of security: a low number of violations doesn't mean that you are done with testing or that the product is a quality product. Rather, getting to a few violations is more like "the end of the beginning" – the product is slightly better and the code is more stable and it's probably okay to increase the amount of testing you're doing.

5.5.2 Don't ignore the implications to risk

If component C has many violations, and component D has relatively few violations, you may want to start testing/investigating component C before component D: C probably has more bugs (including non-static analysis bugs). (The use of static analysis as an indicator of pre-release defect density is covered in [21].) Alternatively, you may want to look at D first because it's less likely to change.

5.6 Tools

5.6.1 Improve the tool

If you find bugs in the tools, report them to the authors. There might be legal or company policies around sharing code with third-parties, so you might need to create a smaller, non-product repro.

5.6.2 Write rules for those who use your product

You might consider writing a set of add-on rules to guide users toward proper usage of your product. This helps users experience the benefits of static analysis on their own projects. Be sure to test the product behavior when the user doesn't know about (or chooses to ignore) the rules. Also, if correct use of your product is only expressible in natural language and cannot be expressed via a programming language, then your product might be too complex.

5.6.3 Play nicely with static analysis

Make a "hello world"-level program with your product and run various static analysis tools with their "out-of-the-box" settings. (You don't need to write any custom rules: just use the built-in rules.) Can you reduce the violations (whether real or not) that are present in user code because of your product? (This suggestion also applies to static analysis tool developers: does your tool give useful results for default code generated by popular development environments?)

References

[1] McCabe & Associates. McCabeQA. http://www.mccabe.com/iq_qa.htm.

[2] IEEE Standards Collection. Software engineering, 1994.

[3] Manuvir Das. Esp: Program verification of millions of lines of code. Microsoft Research Faculty Summit, July 2002. http://research.microsoft.com/~manuvir/talks.html.

[4] Robert DeLine and Manuel Fahndrich. The fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft, January 2004. ftp://ftp.research.microsoft.com/pub/tr/TR-2004-07.pdf.

[5] ESC/Java2. http://secure.ucd.ie/products/opensource/ESCJava2/.

[6] David Evans and David Larochelle. *Splint Manual*. Secure Programming Group, University of Virginia Department of Computer Science, June 2003. http://lclint.cs.virginia.edu/manual/.

[7] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly, 1997. ISBN 1565922573.

[8] Gnu indent. http://directory.fsf.org/GNU/indent.html.

[9] Michael Howard, Jon Pincus, and Jeannette Wing. Measuring relative attack surface. Technical Report CMU-TR-03-169, Carnegie Mellon University, August 2003. http://www.cs.cmu.edu/~wing/publications/Howard-Wing03.pdf.

[10] Java Modeling Language (JML), 2005. www.jmlspecs.org/.

[11] S. C. Johnson. Lint, a C program checker. Technical Report Computer Science Number 65, Bell Telephone Laboratories, 1977.

[12] Cem Kaner. Don't use bug counts to measure testers. *Software Testing and Quality Engineering*, page 80, May/June 1999. http://www.kaner.com/pdfs/bugcount.pdf.

[13] Klocwork. insight. http://www.klocwork.com/products/insight.asp.

[14] Scott Meyers and Martin Klaus. A first look at C++ program analyzers, February 1997. Pre-publication version of article from Dr. Dobb's Journal http://www.aristeia.com/ddjpaper1_frames.html.

[15] Microsoft. Design guidelines for class library developers. http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconNETFrameworkDesignGuidelines.asp.

[16] Microsoft. FxCop 1.312. http://www.gotdotnet.com/team/fxcop/.

[17] Microsoft. Header annotations, July 2005. http://msdn.microsoft.com/library/en-us/winprog/winprog/header_annotations.asp.

[18] Microsoft. PREfast, 2005. http://www.microsoft.com/whdc/devtools/tools/PREfast.mspx.

[19] Microsoft. Static driver verifier, 2005. http://www.microsoft.com/whdc/devtools/tools/sdv.mspx.

[20] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, 1998. http://doi.acm.org/10.1145/279310.279314.

[21] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 580–586, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-963-2. also available at http://research.microsoft.com/research/pubs/view.aspx?type=Publication&id=1358.

[22] National Bureau of Standards (NBS). Special publication 500-75: Validation, verification, and testing of computer software, 1981.

[23] Parasoft. C++test. http://www.parasoft.com/jsp/products/home.jsp?product=CppTest.

[24] Parasoft. Jtest. http://www.parasoft.com/jsp/products/home.jsp?product=Jtest.

[25] Lutz Roeder. .NET Reflector. http://www.aisto.com/roeder/dotnet/.

[26] Gimpel Software. PC-Lint. http://www.gimpel.com/.

[27] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: Rules, Guidelines and Best Practices*. C++ In-Depth Series. Addison-Wesley, 2004. ISBN 0321113586.

[28] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. Its4: A static vulnerability scanner for c and c++ code. Technical report, Cigital, 2000. http://www.cigital.com/papers/download/its4.pdf.

[29] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-855-5.

High Level Petri Net approach to Model Testing

Marco Piumatti

This paper describes a technique, based on High Level Petri Nets (HLPN), to generate a formal description of a distributed system and automatically generate test coverage. It is based on the experience of the Windows Collaboration Technologies team adopting this technique to test a peer-to-peer collaboration platform for Microsoft Windows.

1 Biography

Marco Piumatti is a Software Development Lead with Microsoft. For the last 3 years he has been working in the Windows Collaboration Technologies Team. The team is delivering a peer-to-peer based collaboration platform as part of the Windows operating system.

2 Abstract

As software development is evolving toward distributed architectures, testing technologies are facing new challenges in providing the required level of quality assurance.

The analysis of a distributed computer system requires the ability to describe the logical interactions between multiple components participating in a common task; informal models or plain text descriptions are no longer a suitable language for describing such complex systems. Based on our experience, specifications generated in plain text are necessarily verbose, subject to interpretation and not suitable for formal analysis. The use of High Level Petri Nets (HLPNs) provided us with a compact graphical and formal notation to describe distributed systems; the well defined formalism allows for an easy translation of the model to programming language code that can be built and executed to perform analysis and generate test cases. In conjunction with our distributed test case execution engine we could accomplish our goal of an end-to-end automated solution from the model generation to the test case execution.

3 Introduction

The theory of Petri Nets [PN] originated from the doctoral thesis of C.A. Petri in 1962. Since then the formal language has been developed and used in many theoretical and applicative areas. The PN formalism has been extended to integrate new concepts such as time (Timed Petri nets [TPN]) and to provide a more abstract, high level description language (High Level Petri Nets [HLPN]). All these variations over the original formalism are equivalent classes and the same analysis techniques can be applied.

We started exploring the HLPN formalism while trying to model our peer-to-peer platform for Microsoft Windows. The distributed nature of our features exposed a number of difficulties while trying to use our existing tools based on Finite State Machines [FSM]. The FSMs that resulted were not efficient at modeling parallel execution, synchronization and variable number of peers.

We considered HLPN as an alternative way of describing our components and test scenarios. Based on a set of high level requirements (ability to describe multi-machines distributed scenarios, reviewability of the models, integration with our existing test infrastructure and flexibility to adapt to design changes), we designed a tool that provides the functionalities to graphically draw the HLPN model of a system and automatically generate test coverage.

The tool is based on Microsoft Visio for drawing capabilities and on Spec Explorer from Microsoft Research for model analysis and test case generation.

To understand the effectiveness of this new methodology and how it would affect our testing process we collected and compared data for two components, one tested using the technique described in this paper,

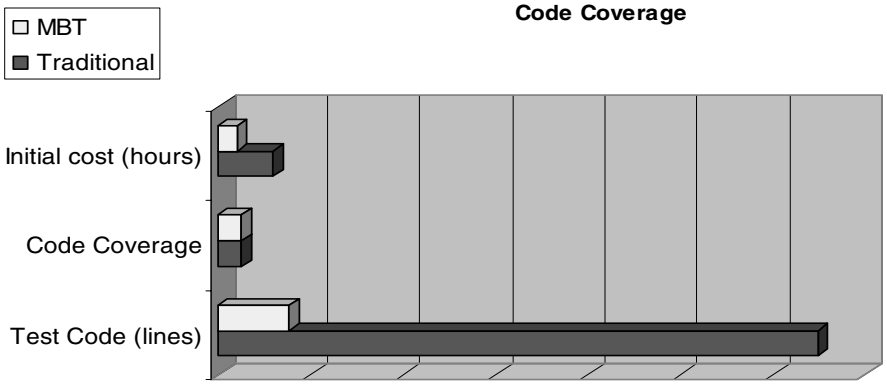
the other using our traditional approach consisting in manually developing test scenarios in C# and using our test harness to generate and control multi-machine scenarios. We based our evaluation on the following set of metrics:

- Code coverage;
- Number of bugs found;
- Test/dev bug ratio;
- Lines of test code manually written by the test developer;
- Feedback collected during model reviews.

Code coverage

We collected this metric by generating test coverage for the same component using both the HLPN based technique and by manually writing test cases. The study was performed by only generating coverage for valid scenarios and does not includes API parameter validation and invalid test sequences. The chart below shows the results in term of block coverage, number of lines of test code and initial cost. (Cost of development of the test suite).

fig.1



Both the test suites provides similar amount of code coverage (around 50%), the biggest difference is in the number of lines of test code that is reduced by a factor of about 10 using the HLPN approach. This also affects the time of development that dropped from 3 weeks to 1. From this data we also expect a reduction in maintenance cost due to the reduced amount of code.

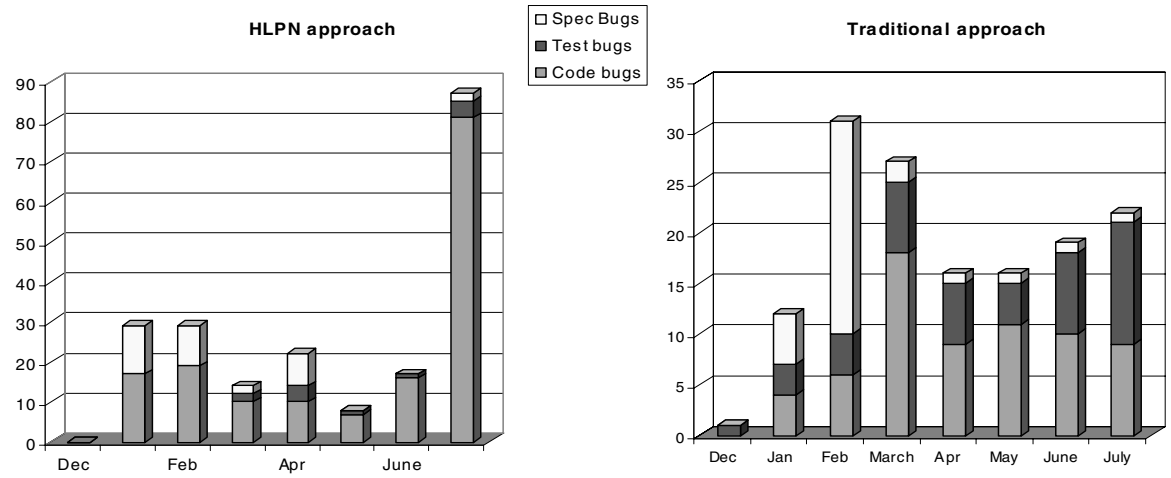
Bug trend

The chart in *fig.2a and fig.2b* show the bug trend, in an 8 months timeframe, on two components developed and tested by our team using the HLPN model-based testing approach and our traditional automation approach. The months of February and July correspond to development code complete for two milestones of the product.

In *fig.2a* the component tested using the HLPN approach shows a bigger concentration of bugs around the two milestones. This is because the models could be developed in parallel with each feature based on the same set of specs the dev team was using. After each feature was completed, the models were executed to exercise the feature functionalities. This approach created some problem to our process because of the unanticipated peak in incoming bugs immediately after each feature code complete.

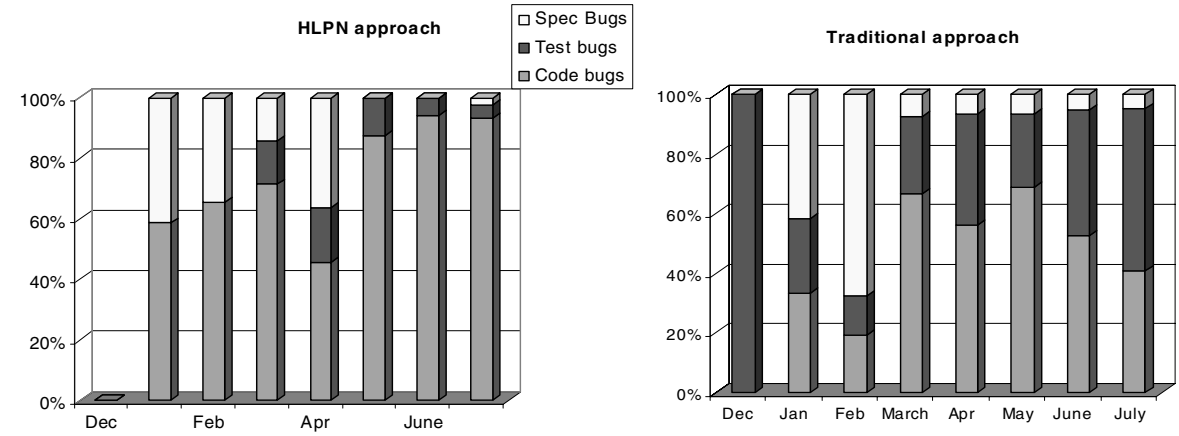
The traditional approach shows a more consistent trend in the number of test and dev bugs during the entire time frame. The test case development was mainly performed after each component was code complete and continued throughout the entire period.

Fig.2a



The ratio between spec, dev and test bugs (*fig2.b)* shows a consistent advantage of the HPLN technique over our traditional approach. The ability to automatically generate test code not only contributes to reduce the number of test bugs, but also makes our test suites more flexible to last minutes design changes. Incorporating a design change often only requires modifying the model and regenerating the test suite.

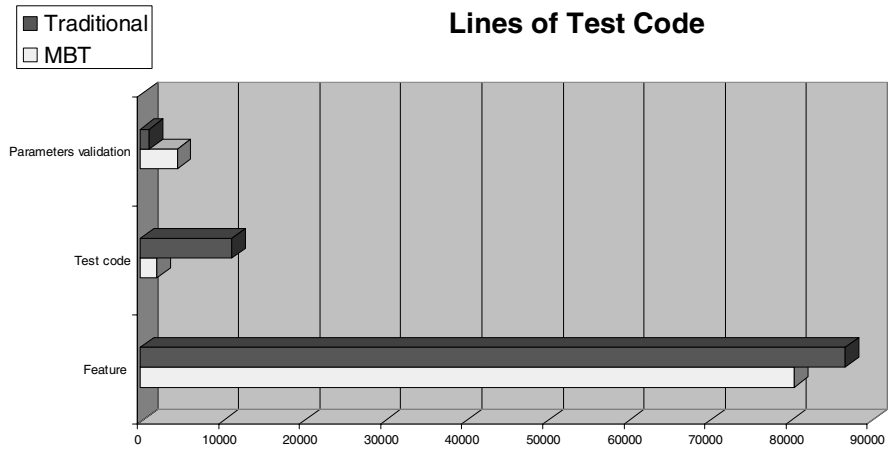
Fig.2b



Number of lines of test code

Comparing the number of lines of test code manually written confirms the data collected in our initial investigation (*fig.1*) with a reduction of about 1/10 in the HLPN approach. *Fig.3* shows dev and test code comparison for the two components used in this analysis. In both cases API parameter validation testing was implemented without using models, the difference in the size of test code for this area is justified by the different number of API exposed by the components under test.

Fig.3



4 High Level Petri Nets

Petri Nets are a graphical and mathematical notation to describe a system. The areas of application for Petri Nets include: communication protocols, distributed software systems, distributed database systems, concurrent and parallel programming, multiprocessor systems.

Petri Nets provide a formal mechanism to describe concurrent, distributed, asynchronous and non-deterministic systems. An inherent problem with Petri Nets is the explosion of the number of states in the graphical representation. HLPN help mitigate this problem by introducing higher level concepts such as structured data to represent tokens and expressions, token identity for resource identification, probabilistic execution and time.

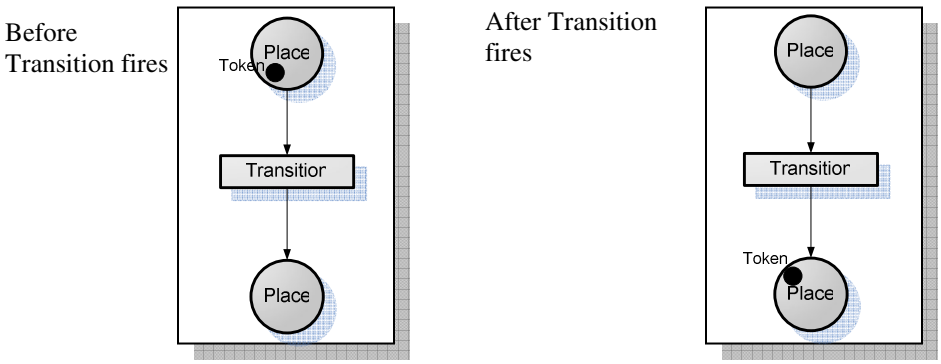
Definition:
A HLPN is defined by:

- A directed bipartite graph composed of two disjoint sets: {Places}U{Transitions} and a set of arcs connecting Places with Transitions such that no two Places and no two Transitions are directly connected;
- A set of {Token} where a token represents and models a resource of the system;
- An Initial Marking that determines the initial state of the system by giving the initial distribution of the Tokens in the Places.

A transition T is characterized by two expressions that determine its behavior:

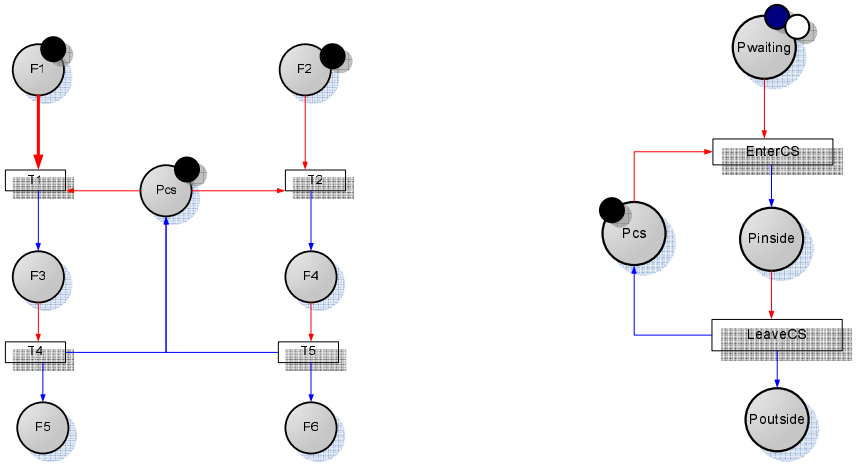
- A firing rule: this is a function of the input arcs of the transition; it is an expression that evaluates to true/false and determines when T can fire (execute);
- An Action: this is an expression of the input and output arcs of the transition and determines the behavior of T when it is fired. The gross result of firing a transition is to move tokens around the HLPN graph thus modifying the status of the system..

Example (Firing a transition):



Petri Nets are a rather low level modeling paradigm, it is often more convenient to use the more high level formalism described by HLPN. Fig.4 shows the comparison between a PN modeling two processes competing for access of a critical section and the same model described with the HLPN formalism. In the case of the PN description, all the transitions involving the two processes need to be separated in order to maintain the identity of each process; in the case of HLPN, the token itself will keep the process identity information (by the color in the case), so the common transitions *T1*,*T2* and *T4*,*T5* can be collapsed into *EnterCS* and *LeaveCS*. Also the HPLN model is able to describe the behavior of any number of process without any modification to the model.

fig.4



5 Modeling a system

The use of HLPN to model a real system consists of defining a mapping between the properties, events, actions and initial configuration of the system and the elements of the HLPN (Places, Transitions, and Tokens).

- Places identify properties of the real system (working, waiting, listening, failed, etc…). The state of the system at any given time is described by the set of Places that are marked. For example a process that is actively listening on a connection will expose both the working and listening properties;
- Transitions describe events or actions that modify the system state;
- The marking of the HLPN at any given time describes the state of the system in term of exposed properties and enabled actions/events.

The following examples illustrate typical situations arising in system modeling

5.1 Concurrency

The HLPN in *fig.5* and *fig.6* show three different concurrent scenarios; in *fig.5* transitions *t1* and *t2* are enabled simultaneously by the marking of places *P1* and *P3*. The firing of one transition does not modify the state of the other and the two can fire concurrently.

fig.5

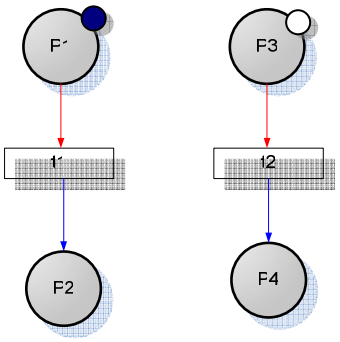


Fig.6 shows two situations where the transitions *t1* and *t2* shares the input place and are both enabled by its marking. In *fig.6a P1* contains enough tokens and the two transitions will fire in parallel; in *fig.6b* the firing of one transition will disable the other one, only one transition will be allowed to fire. In both cases the behavior of the HLPN is non-deterministic.

fig.6a

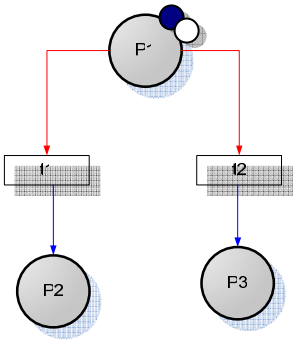
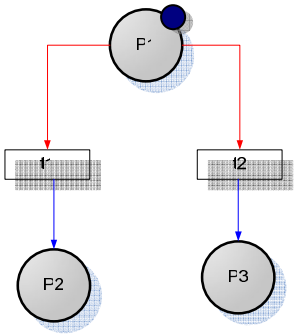


fig.6b

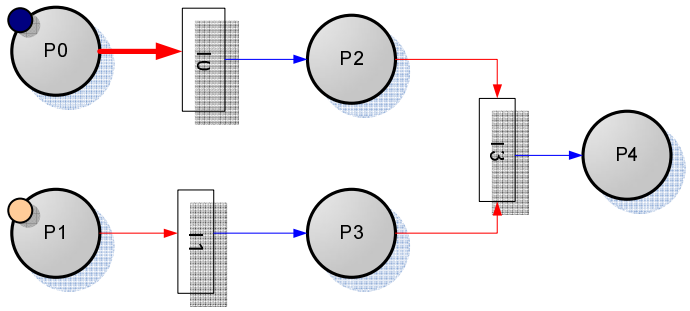


5.2 Synchronization

Synchronization typically involves two or more processes executing concurrently and needing to synchronize at some point of their execution path.

In *fig.7* the activities modeled by transitions *t1* and *t2* execute concurrently, however they need to be completed (synchronize) before activity *t3* can execute. This situation is modeled by defining the firing rule for the transition *t3* to depend on the marking of both *P2* and *P4*, thus requiring both activities represented by *t1* and *t2* to be completed.

fig.7



5.3 Mutual exclusion

Mutual exclusion is used when multiple processes in parallel execution need exclusive access to a shared resource. *Fig.8a* to *fig.8c* show one possible evolution of an HLPN modeling a critical section with three processes competing for accessing a shared resource.
The *EnterCS* transition forces only one process at a time to access the critical section by requiring the place *Pcs* to be marked in order to fire and by removing the token from *Pcs* upon firing. The transition *LeaveCS* will reset the token in the *Pcs* place, enabling *EnterCS* to fire for the next process.
The identity of each single process is maintained throughout the model evolution by using colored tokens.

fig.8a

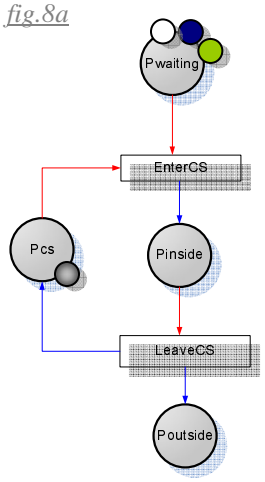


fig.8b

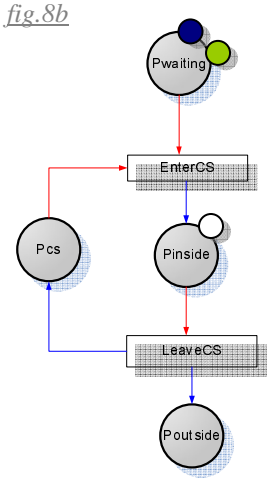
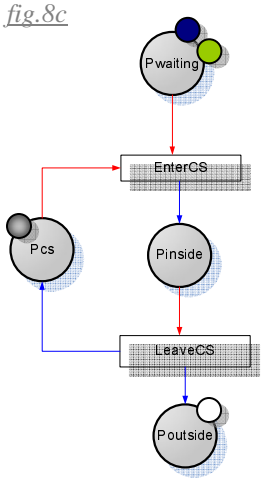


fig.8c



6 Formal analysis

The effectiveness of a modeling tool depends on two factors: its modeling power and its decision power. The modeling power refers to the ability to correctly represent the real system; the decision power refers to the ability to analyze the model and determine properties of the modeled system. In the previous sections we covered some examples of the modeling power of the HLPN for common programming paradigms like concurrency and synchronization.

Two of the most common techniques to perform a formal analysis of an HLPN model are reachability graph and reachability matrix. Both techniques involve describing all the possible evolution paths of the model and deriving system properties from them.

In our implementation we choose the approach of generating the reachability graph of the model. This allows both visual inspection and formal verification of its correctness.

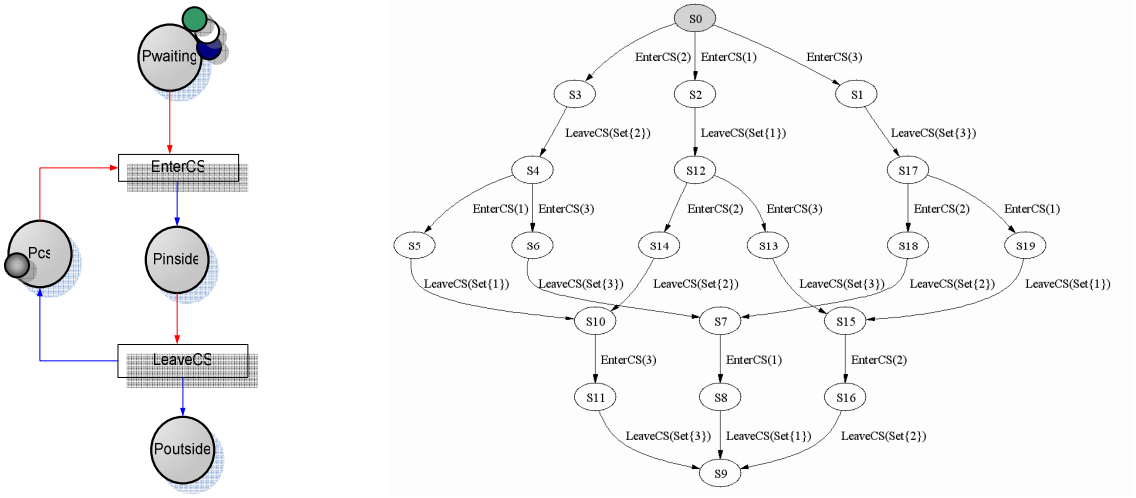
6.1 The reachability graph

The reachability graph *Gr (M1)* is a directed graph composed of the set of all the markings *Mi* reachable from *M1* by firing the enabled transitions. The arcs connecting two nodes *Mi* and *Mi +1* represent the ordered set of transitions that produced *Mi +1* from *Mi*.
By defining the following three conditions that define an end node in the graph, it is always possible to generate the reachability graph in a finite set of steps:

1) Terminal (dead) nodes: nodes where no transitions are enabled;
2) Duplicate nodes: nodes that have already been generated in the graph;
3) Infinitely reproducible nodes: nodes that are infinitely reached with an increasing numbers of tokens.

Fig.9 shows the produced reachability graph for the mutual exclusion model with three processes (tokens) competing for a shared resource.

Fig.9



7 From the models to the Test Cases

Our team is responsible for producing a collaboration platform based on peer-to-peer technology. The platform enables discovery and interaction between peers by running distributed algorithms between all available peers eliminating the need for a server infrastructure.

Such architecture poses a unique challenge for testing: it requires test cases to focus on scenarios where multiple machines are involved and coordinated to perform a common task.

We identified a set of requirements for our testing strategy that can be summarized as:

- Test harness should be able to execute, control and enable debugging on multi-machine test cases;
- Test cases should focus on distributed scenarios;
- Test cases should be invariant on the number of participating peers;
- Test scenarios should be easy to maintain and review.

We focused on model-based testing as a way to formally represent our test scenarios and automatically generate test coverage that could be easily integrated and run in our existing test environment.

HLPNs gave us the modeling power to represent complex distributed scenarios and a formal representation that facilitates implementing the models in a programming language.

The missing piece was an effective way to produce our models as HLPN graphs; we developed a tool to build graphical representations of HLPNs and to automatically convert them into a programming language for analysis and test case generation.

The Petrinetor tool provides the following features:

- assisted graphical development of an HLPN;
- automatic implementation of the model in the C# programming language;
- reachability graph and test case generation;
- model simulation to visually verify the evolution of the system and to gather execution metrics.

Fig.10 the Petrinetor tool

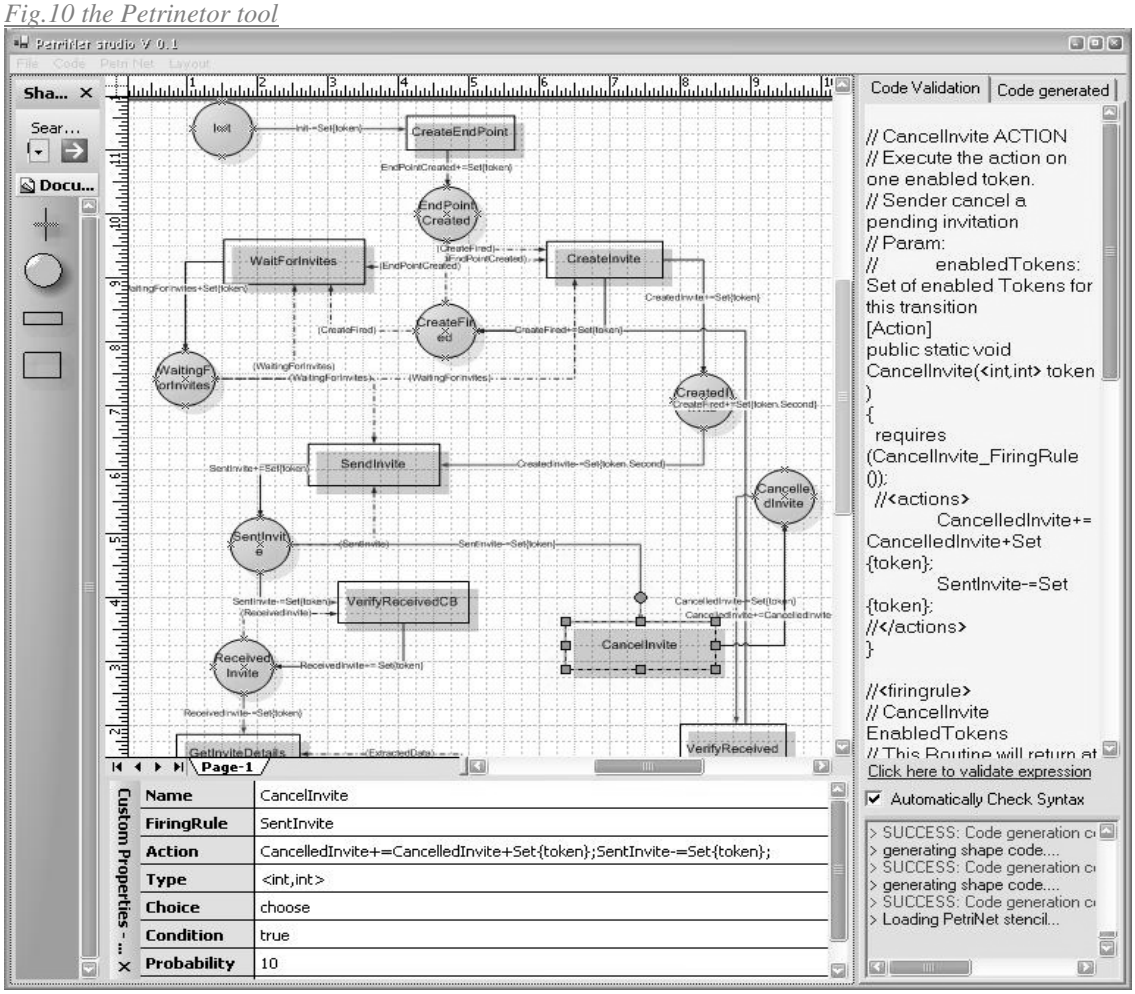
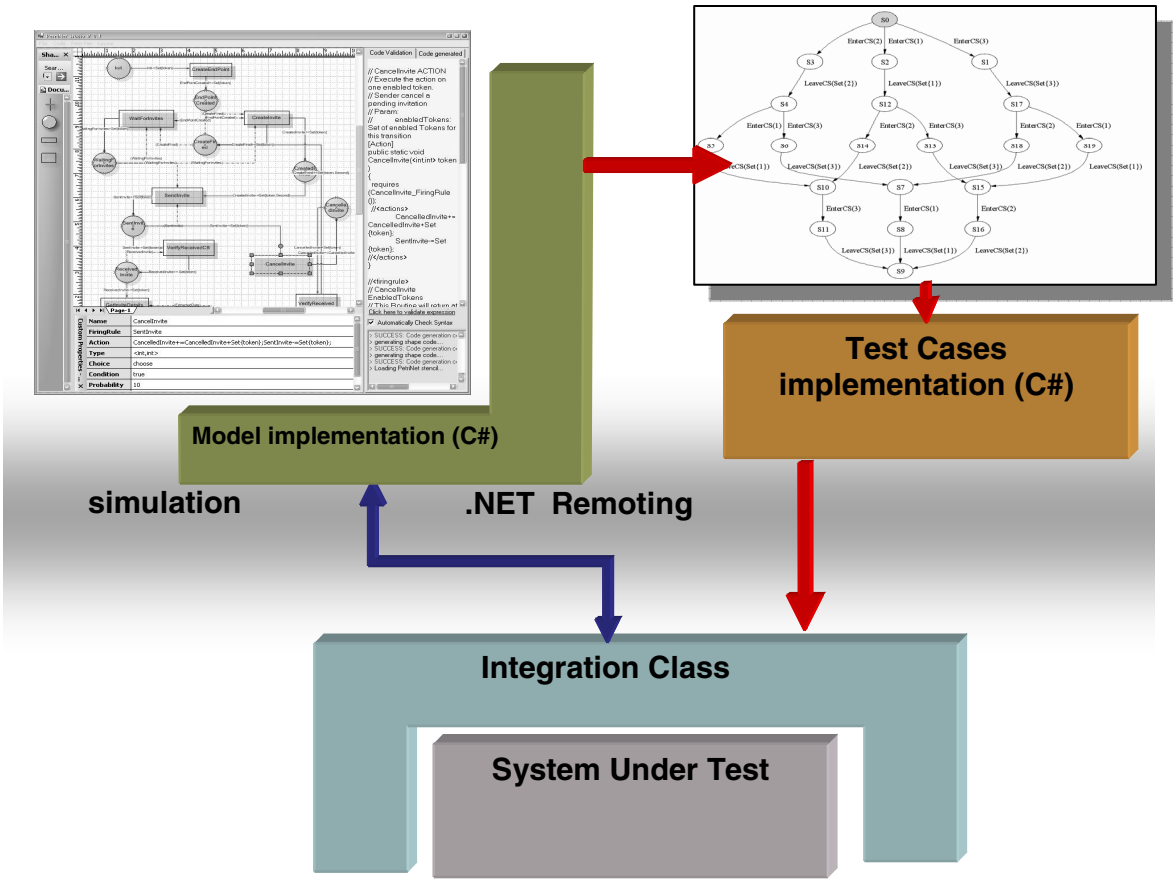


Fig.10 shows a screenshot of the tool with a portion of a model; the left pane of the tool shows the available drawing stencil, the central pane is the model, here circles represent Places, rectangles represent Transitions, the oriented arcs are automatically drawn by the tool to represent the flow of evolution of the system and can be decorated with the associated action. The bottom pane is the property page for the selected object and the right pane contains the implementation code for the selected object and status information such as tool status and error messages.

Our approach to model-based testing using the Petrinetor tool consisted of the following steps (*fig.11*):

1) Graphical model development;
2) Implementation of the model;
3) Component integration;
4) Model analysis and test cases generation;
5) Simulation runs.

Fig.11



7.1 Graphical model development

Model design is performed using a set of custom shapes representing the HLPN elements: (Places, Transitions), and a set of shapes acting as placeholders for data definitions such as structures and enumerations.

Places are containers for tokens; their properties describe the place name, the type of the tokens and the initial marking. Transitions describe the behavior of the model and are described by a name, a firing rule, a set of actions and a probability. Transitions also define how the action will handle multiple tokens that could fire at the same time. Depending on the target scenario, a transition can fire all available tokens in parallel, one by one or just pick one randomly from the available set.

7.2 Model Implementation

The translation of the graphical model in a programming language code is automatically performed by the tool.

During this phase Places are mapped to collections represented as Sequences or Sets. Transitions are mapped in sets of three functions:

- *transition*_Action( );
- *transition*_EnabledTokens( )
- *transition*_FiringRule( ).

The *transition*_FiringRule( ) is a boolean function returning whether the transition is enabled. The execution engine of the tool invokes this function to build a list of enabled transitions at each step.
The *transition*_EnabledTokens( ) is a function returning the set of tokens currently enabled to fire the transition. This set is passed in input to the *transition*_Action( ) function whose responsibility is to fire the input tokens moving them from the input to the output places.

7.3 Integration

This step consists in writing a layer of code to integrate the model implementation with the component under test. The main function of the integration component is to map functions, parameters and return codes of the component under test into the corresponding model's actions. Logging and debugging utilities are also part of the integration layer. This is the only test code that needs to be manually written by the test developer.

7.4 Reachability graph and test case implementation

After completing the model, a set of test cases, in the form of C# code, can be automatically generated.

Test case generation is completely automated and consists of the following steps:

- The model implementation is built into an executable file;
- The executable is loaded into the tool using .NET reflection;
- Model actions are associated with the corresponding methods exposed by the integration layer;
- The tool builds the reachability graph of the model by performing a complete traversal of the HLPN graph;
- Test cases are derived from the reachability graph in the form of XML or C# code.

7.5 Simulation

A simulation consists of an on-the-fly run of the model. The HLPN is converted in the corresponding executable module that is loaded by the tool and executed to simulate the evolution of the modeled system.

The Petrinetor will automatically update the marking of the HLPN drawing for a visual verification of the evolution of the system.

Simulations can be used to inspect the behavior of the model under different initial conditions (markings) and can be used to generate simple metrics such as the number of steps required to execute a path between two states in the model.

A simulation can also be bound to the integration class to perform an on-the-fly execution of the system under test.

8 Test Process

In our experience testing a distributed peer-to-peer platform, we applied the model-based testing methodology at various steps during the production process. *Table1* summarize the testing phases during the production cycle. Model-based testing was applied mainly to functional verification (Functional test and stress) while other test areas like parameter validation testing, performance, real world testing and security were implemented adopting different specific approaches for each area.

table1

Development phase	Test phase
Features definition and design	Feature models design based on specs, Model reviews, Simulation runs.
Implementation phase	Test scenario models design, Models review, Integration module design and implementation, Test cases into the Test Cases Management tool, API testing development, Code reviews,
Feature code complete	Models integration with feature code, Test pass execution in the lab, Failure investigation / Bug fixes, Stress models definition, Stress runs, Code coverage, Ad hoc testing: (real world testing, performances, scalability, security) Static code analyzers.
Bug Fixes	Regressions test passes, Private test passes, Test bug fixes, New models / test cases definition based on code coverage and bugs.

On individual basis, each test developer was involved in the following activities:

- Participate in feature definition reviews;
- Produce and review a test plan presenting test areas, test strategy, risks and needed resources;
- Design a model for the feature based on preliminary specs, point out specs bugs (at this point mostly inconsistencies and missing parts);
- Once the specs are stabilized enough, have a review of the model with the feature team (dev, test and pm). This helps in having all the feature team on the same page on the expected behavior of the feature;
- Design and develop an integration module between the dev code and the model;
- Generate test sequences; this phase consists of using the tool to produce a set of test sequences for the feature. This involves multiple steps where the scenarios are refined through the model to obtain test sequences for each desired scenario, while trying to eliminate less meaningful ones;
- Test cases definition; the generated test sequences are analyzed to identify different levels of priorities (Build Verification Tests, Functional pri0, Functional pri1 etc.). Test cases are entered in the Test Case Manager tool accordingly to their priority. During this phase each test developer has to go through the test sequences to identify different level of priority based on the test coverage;
- Work with test execution team to integrate the new test cases into the team's test pass;
- Analyze test pass results, log and follow up on bugs.

9 Conclusions

During our experience with model-based testing we focused on modeling distributed networking systems to analyze their behavior and automatically generate test coverage.

We identified in the HLPN formalism a valid representation tool for these kinds of systems and we built a tool around it to help us generate models and obtain test coverage for our product.

We used the modeling technique to validate functional requirements and stress of our components while adopting different techniques for other test areas like security, performance, integration and real world testing.

By adopting a mode-based testing approach we could improve our process in multiple areas:

- Quality assurance: by validating the design of our components we were able to discover design defects very early in the process;
- Collaboration and communication: by formalizing on paper our testing strategy using models, we could easily and effectively communicate our approach facilitating feedback and collaboration;
- Test cases management: by formally documenting the behavior of the system we could effectively manage design change by easily visualize on the model what areas of the system would be impacted by the changes;
- Reduction of our test code base: less code to maintain and less test bugs to fix;
- Repeatability of tests, across multiple configurations and multiple version releases was made easier: the integration module provides an abstraction level between the model and the real implementation, this allows for executing the same models on different platform configurations or feature releases by simply using the corresponding version of the integration module. For example we were able to run the same models on 32 bit, 64 bit machines and mixed scenarios by simply compiling the integration model for each platform;
- Finally the model-based approach, by providing a low cost regression test suite, provided us with more time to focus on corner cases and specific scenarios.

The main issue we faced while adopting model-based testing in our process was the big number of test sequences generated by the tool even for simple configuration. We identified the main causes of this explosion as:

- API parameters modeling; introducing API parameters into the model produced an explosion in the number of test sequences generated by the tool. Each functional sequence was replicated for each possible combination of all involved parameters;
- Number of peers participating into the model. In general the number of test sequences generated was growing exponentially with the number of peers involved in the model. This was consistently limiting our ability to test with high number of peers;
- Invalid scenario modeling. Creating models to generate invalid scenarios was relatively easy; by simply altering the firing rule of selected transition we could bring the system in an invalid status. The issue was to be able to verify the right behavior of the system in those conditions.

We adopted a set of strategies that allowed us to generate good test coverage while keeping the number of test sequences into a manageable number:

- We reduced the variations of parameters in the model; the integration module abstracts most of the parameters that are not directly involved in the functional evolution of the system. For example the name of a peer is internally generated by the integration class instead of being exposed to the model.;

- Full parameter variation and verification was covered in a separate test suite not using model-based testing;
- We focused on designing test scenario models; instead of just designing a single model describing the component under test, we generated a set of models focusing on specific scenarios. For example we had a model covering all the possible initialization paths and one focusing on discovery and connecting scenarios between peers. The second model did not have to explore again all the initialization scenarios but could just implement one straight path to initialize all participating peers saving test sequences for the real scope of the test;
- We realized on-the-fly models where the HLPN is directly driving the evolution of the system under test. In this case there is no test sequence generation, the HLPN exploration algorithm is directly connected with the integration module and drives the execution of the test while exploring the HLPN. These are more generic models than can involve a bigger number of machines and can be run for long periods of time;
- We designed specific models for the invalid scenarios we wanted to cover. By having models focusing on specific scenarios we could embed the expected behavior into the model for verification. We had to accurately select and review the invalid scenarios that we wanted to cover. This is not different from what happens in a more traditional testing approach where the test developer defines the set of invalid scenarios to implement.

10 References

1. Final Draft International Standard ISO/IEC 15909
   "High-level Petri Nets - Concepts, Definitions and Graphical Notation"
   Version 4.7.1
   http://www.informatik.hu-berlin.de/top/PNX/pnstd-4.7.1.pdf

2. Wolfgang Reisig
   "Elements of Distributed Algorithms"
   Springer, ISBN 3-540-62752-9

3. Andrea Bobbio
   "System Modeling with Petri Nets"
   Istituto Elettrotecnico Nazionale Galileo Ferraris
   www.mfn.unipmn.it/~bobbio/ BIBLIO/PAPERS/ANNO90/kluwer**petri**net.pdf

4. Spec Explorer tool from MSR
   http://www.research.microsoft.com/projects/specexplorer/

5. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte
   "The Spec# Programming System: An Overview"
   Microsoft Research, Redmond, WA, USA
   http://research.microsoft.com/specsharp/papers/krml136.pdf

6. Margus Veanes Colin Campbell Wolfgang Grieskamp Wolfram Schulte Nikolai Tillmann
   "Online Testing with Model Programs"
   Microsoft Research, Redmond, WA, USA
   http://research.microsoft.com/~schulte/Papers/OnlineTestingWithModelPrograms(FSE05).pdf

7. Lev Nachmanson Margus Veanes Wolfram Schulte Nikolai Tillmann Wolfgang Grieskamp
   "Optimal Strategies for Testing Nondeterministic Systems"
   Microsoft Research, Redmond, WA, USA
   http://research.microsoft.com/~schulte/Papers/OptimalStrategiesForTestingNondeterminsticSystems(ISSTA200.pdf

Open Source Is Coming: Here's How to Deal With It

Craig Thomas, Manager Engineering and Performance Group
craiger@osdl.org

Abstract

Free and Open Source Software (F/OSS) is becoming more prevalent in the work lives of developers and QA engineers each day. It has infiltrated into the IT departments, and is being used to aid in product development and testing. Some organizations are even integrating F/OSS into their own integrated solutions.

The benefits of F/OSS have been reported in many trade publications. However, they don't cover the topic most critical to an engineer's job: How can I effectively take advantage of F/OSS software to make my job more efficient? Tools and methods can be used to reduce some of the problems faced by developers and QA personnel when using F/OSS products in their development and deployment cycles.

Application developers understand the problems associated with integrating commercial-off-the-shelf (COTS) packages or depending upon them. But for F/OSS packages, how can developers keep up with the constant flux of new releases? How can they work closer with the F/OSS communities that supply key package dependencies to their products and assure that they stay on top of changes?

This paper will outline some activities application developers and test engineers can do to help them become more effective working with F/OSS and their communities. Open source software is not something to be dealt with, it is something to embrace and utilize.

Craig Thomas is the Manager of the Test and Performance Group at OSDL, an independent vendor neutral organization that works with open source communities to improve the Linux* operating system. With over 20 years testing in Unix* and Linux environments, he has been involved with functional test, integration test, system test, and performance test for enterprise systems. Craig has a BS in computer science from Washington State University and has served as the co-chair of the ASQ Certified Software Quality Engineer examination review committee.

Assessing free and open source software

Many Independent Software Vendors (ISVs) are assessing the potential market of Free and Open Source Software (F/OSS), either as a platform where their applications run or as dependent modules to support the

operation of their programs. They must weigh the potential costs of integrating or utilizing F/OSS programs with the potential increase in revenue they could get from new market opportunities. They ask themselves "Will porting my application to Linux open new markets so I can make more money? What are the costs to port?"

Developers of those applications are looking for ways to bring their products to market faster and with better quality. They want tools that help them develop, debug, and manage their code. Budgets for tools are usually very small. As a result, developers sometimes resort to F/OSS tools to help them develop their products. They ask themselves "How can I manage my source code? What tools can I use to help me develop my code?"

Quality and test engineers need tools to help them manage the quality and reliability aspects of software products. Tools to manage defects, automated test environments, and test suites are typical items on their wish list. Many times commercial test tools are not flexible enough to support the testing efforts of developing applications. Quality and test engineers ask themselves "What can I use to record and track defects better? Are there automated test environments that will test my application? What kinds of stress tests are available?"

Engineering departments are turning to F/OSS to help them in their jobs. Whether they use a tool to help them perform static analysis of a module or simply compile and debug their latest code changes, their activities are increasingly being supported by F/OSS products. The simple reason is money; lack of sufficient funds for a department creates very resourceful engineers. Developers and testers alike are finding new F/OSS tools to increase their productivity every day.

Engineers can embrace and integrate F/OSS into their daily activities. Many good tools exist that can help developers and testers become more productive at work. But there is more to F/OSS than just taking programs for free. An entire ecosystem exists that allows all software programmers to benefit. It is important for anyone who uses F/OSS to understand and be part of that ecosystem as well.

OSDL was founded as a vendor neutral, non-profit organization in 2000 with the purpose of accelerating the adoption of Linux into the business enterprise. To support this adoption, OSDL works with a variety of member companies and open source communities to help create stable and robust F/OSS products for the enterprise. We have enabled initiatives for carrier grade systems, data centers, and enterprise clients (business desktops) to improve the capabilities of F/OSS software in those areas. Because of our involvement with open source developers as well as commercial companies, OSDL is in a unique position to understand the needs and desires of both organizations.

Do you have open source software in your environment?

In simplified terms, Open Source Software consists of programs that are typically distributed with its source code. It may or may not be free of cost. Typically, licenses for F/OSS allow recipients to modify and redistribute the code without further need to pay royalties or other compensation to third parties. The Open Source Initiative (OSI) currently has 58 approved open source licenses; each has small variations on this general theme[1].

Several F/OSS products could already be in use at your company or even in your development environment. They can range from a content management system and internet tools used to support your company's infrastructure to compilers, debuggers, bug trackers, and script languages used by the development and test teams. The table below lists some common F/OSS programs that may be used in your work environment.

Apache HTTP server	Web server
Mozilla, Firefox, Opera	Web browsers

Perl, Ruby, Python	Programming languages
Bind	Internet DNS naming
Linux	Operating System
CVS, RCS	Source control managers
OpenOffice	Office productivity suite

Increasing your development and test productivity with F/OSS

Software development teams use a variety of tools to develop their applications. Traditionally, they are either purchased from software vendors or are developed in-house and used from project to project. With demands on product functionality increasing and project schedules decreasing, a test engineer is usually taxed to find time to analyze test results and report on the product quality and reliability. New tools and methods are constantly being examined in order to make the tester's time more productive.

Additionally, commercial tools sometimes don't fit the current processes employed at an organization. Sometimes the development tools dictate the development processes. Early CASE tools in the 1980's exhibited that problem. To make changes to a commercial tool, one would typically submit an enhancement request to the company and wait for a new release or patch that may or may not contain the enhancement. In any event, facilitating a customization change could take some time.

F/OSS tools have a unique feature; the source code is available for customization. Programming teams can allow the processes to dictate the types of tools they need to make their activities more efficient. F/OSS tools can be customized for the specific needs of a developer or a tester. Many tools exist to aid in the development of products or help to allow programming and test teams to collaborate on project tasks.

Communication among programming teams:

Communicating with programming teams is quite easy if they are all located in adjacent work spaces. However, communicating with other departments within an organization or to manage remote programming teams is a harder task. Open source tools for communicating the status of a project can be done with a few simple tools, that have proven to be fundamental for many open source projects themselves:

Internet chat programs such as Xchat IRC[2] and Gaim[3] are used for interactive, real-time discussions relating to project issues. This is extremely valuable when you need to talk to another engineer on the project and he/she is not next to your work space. Less real-time intensive means of communication can be managed by discussion mail lists. All project members can subscribe to a project mail list and communicate all project issues within the mail list. Mailman[4] is an essential tool for project team communication.

To keep an editable web page of project status and other valuable information that can be read by other departments within the organization, Wiki [5] is a valuable web application. A Wiki allows readers to add and edit content for a web portal of project information. This allows constant updates to project information, whether it be status of progress, changes in the product design, or other items of information delivered to project members. If a Wiki is not for you, a simple web page managed by an Apache HTTP server is valuable to provide all project team members readily available information about the project.

Programming Language tools:

Programming tools are easily found in the F/OSS community. C and C++ compilers such as gcc[6] and build tools such as Make[7] and Ant[8] have been around for quite some time. Interactive languages such as Perl[9], Python[10], Ruby[11], and TCL/Tk[12] are used to develop test scripts and file manipulation programs.

Integrated Development Environments:
It is possible that developers need an Integrated Development Environment (IDE) to make their products successful. Qt[13] and Eclipse[14] are two good candidates. Of course, if the project team wishes to manage their source code with a F/OSS source code management tool, they can use one of several: CVS[15], Subversion[16], Arch[17], or Monotone[18].

Test Frameworks:
Testers can find many test frameworks available for their specific needs. For testing Java applications, Junit[19] is well known to many developers. STAF[20] is another framework used to aid in testing applications. OSDL is working on a project known at the Binary Regression Test platform[21]. Its purpose is to run applications on a integrated set of products to determine if changes to any of those integrated products affect the application that depends upon them.

Defect Tracking and Other Test Tools:
Even defect tracking tools, such as Bugzilla[22] can be acquired and customized to fit the specific needs of the programming team. For specific test needs, a plethora of tools exist. Several sites exist that contain reviews of open source tools and a test engineer should visit them if there is a need to acquire a specific tool. One particular site provides product reviews of test tools from the point of view of a tester's needs. The site is maintained by Danny Faught and is located at www.testingfaqs.org. Other sites are noted at the end of this paper.

By finding an open source tool that supports the development and test efforts, engineers find that the efficiency of the project team improves.

Integrating open source tools into developed products

Product Dependencies
Many development projects are focused on a single application that must run with other software products. Usually, that application has dependencies on other binaries on the system such as C libraries or other programming system API's. Some development efforts require an integration of a set of packages to complete a suite of tools for the end user. An example would be a product suite that allows an engineer in the EDA (Electronic Design Automation) industry to create a system model using VHDL (Very high speed integrated circuit Hardware Description Language) and then analyze the model through various tools and simulators. Another example could be a system management suite of applications made for an administrator to configure and monitor several computer systems from a single console. These types of package suites require several individual software components, each highly dependent upon each other.

It is possible to replace some of these individual components with F/OSS packages, thus potentially saving development costs. However, there are caveats to consider. Many open source projects lack a reliable road map for new features that your development team may like to have available for its product. It is impossible to manage an open source community as one would a 3rd party partner. Many corporate strategies are based upon a contract; working with the open source community involves trust, influence, and respect. One cannot legally hold an open source developer responsible for failure to deliver a specific feature. An open source developer cannot get fired for failing to complete a feature or bug fix by a specific date. Therefore, it is difficult to synchronize your release cycle with those of an open source project whose dependencies on new features you desire. That is one of the findings OSDL had noted after conducting surveys with ISVs about their porting issues to the Linux operating system.

However, it is possible to work in harmony with open source projects and achieve a successful integration between F/OSS and commercial applications. OSDL has several member companies who work within the open source community, namely within the Linux kernel community. They sell applications that are ported to Linux and hence, they are dependent upon its runtime libraries. They provide patches to the community so

that their applications will perform better under Linux; they are involved in reviewing new features in the kernel to see if their applications can use those features to their advantage and increase the value-add of their products. They provide test resources to assure that distribution vendors, who take many of their packages from the F/OSS communities, have high quality code so that their applications will run more effectively.

Licensing

An additional problem with the integration of F/OSS tools into a commercial product suite is the consideration of license issues. Although this is usually a concern for the legal department, developers and QA engineers must understand the impacts caused by F/OSS licenses and the consequences of license violations; they must be able to clearly explain the technical aspects of how products are integrated in a way lawyers can understand so that they may make the correct decisions regarding policy governance. The most common licenses that an open source developer or user will encounter are the GNU General Public License (GPL), the the GNU Lesser General Public License (LGPL) and the BSD License. The virtues and complexities of these open source licenses are beyond the scope of this paper and it is best to leave those discussions to lawyers.

In general terms, the GPL code has a provision in its license that states if a developer uses GPL code, then all code integrated with it must be released as source to anyone who wishes to view it. LGPL code is usually found in libraries where an application development team wants to link with the library but doesn't want to provide source code of the application with every copy they ship. Linking with LGPL licensed libraries allows the development team to distribute object binaries that are linked. A BSD license allows a development team to make modifications to the original source and then distribute it without the need to release the source code. All that is essentially required is that the original copyright notices stay with the modified source. Of course, one must be aware of other provisions each license may contain.

In the case of the EDA system model product suite that utilizes a F/OSS VHDL compiler containing a GPL license, the integration of that F/OSS package may require your company to provide source code for all of the products you developed for that system modeling product suite. Surely, that would not be beneficial to the safe-keeping of your company's proprietary value-add. On the other hand, if the VHDL compiler was under a LGPL or a BSD license, the release of the product suite's source is not necessary although other restrictions may apply. It is important to know that different F/OSS products contain different licenses and the best way to stay clear of license violations is to be aware of potential license conflicts and consult your legal department when you believe a conflict could exist.

In early 2005 Harald Welte, an open source developer and copyright holder of some networking code in the Linux kernel, went to CeBIT, a technology trade show in Germany 2004 and proceeded to distribute letters to 13 companies claiming they have failed to comply with the GNU General Public License . He submitted his code under GPL and discovered several corporations integrating his code into theirs and violating the GPL agreement[23]. Most of the companies didn't fully understand the terms of the GPL and thus unknowingly violated the license. Harald has created a web site to raise public awareness about past and present infringing users of GPL licensed software and has set up a Frequently Asked Questions site to explain the issues involved with using or integrating GPL code[24].

Embrace F/OSS within your organization and make it work for you

You may come to work one day and discover that the project team down the hall is incorporating F/OSS tools to help them with their development. They may even be talking about integrating some F/OSS packages into their product suite. Judging from the feedback of OSDL member companies, F/OSS usage within corporations is growing quickly. How should you react to the growing usage of F/OSS? Don't fight the adoption of F/OSS within your organization – embrace it! Learn how to select good utilities that will help you become more efficient. Learn how the open source process works and how you can play a role contributing to F/OSS products you use.

Open source software can be used to increase the productivity of a project through the use of tools that aid development. Test suites can be acquired to exercise the application, saving hours of new test development.

Open source software can also hinder the development process if the wrong tools are chosen or expectations of support from the maintainers exceed reality. But how does a developer or test engineer determine a good F/OSS tool from a bad one?

Selecting Tools that Work:

Many F/OSS programs are available on the internet. Some are very robust and well maintained by a large community of developers; others are essentially dead projects. Good F/OSS tools all share some common characteristics. Engineers should ask the following questions when considering the adoption of a tool for their use:

1. Check the number of releases posted for download. How often has the package been released: once a year, every month? When was the last release issued? The frequency of releases could indicate how active the the project is. If the project releases too frequently, it may not fit well with your development cycle and you may need to either adapt to their release cycles, take their releases less frequently, or pick another project.
2. Check the activity of defect fixes and feature request implementations. Are defects fixed in a timely manner? How responsive is the project community toward implementing feature requests? If defects are very old, there could be a legitimate reason, but it could also be an indication that the community may not be responsive to issues you may face and you would have to maintain patches yourself.
3. Check the mail archives. How many unique senders are on the discussion list? How many mail messages are sent per month? This is a good indication on how active the project community is and how many people are using or helping to maintain the product code. Lots of activity indicates that the community may be more responsive to answering questions you may have.
4. Check for a home page. How much information is available? A project home page should contain a source download site, documentation, and the list of communication mechanisms of that project community. You should be able to find the contact for the maintainer here as well. If the home page is not well maintained, then the project may not be well supported.

Once a good tool is selected, it may be necessary to modify the code to fit the needs of the project. In many cases, commercial and open source tools rarely meet the exact needs for a development project. However, with F/OSS tools, the source code is available to customize the functionality in order to make it perform precisely as needed. But to work with code in the open source community, engineers must be aware of concurrent development activities for that package. It is possible that people from all over the world are contributing changes to that package and chances are that your customization efforts may have already been performed by someone else. It is wise to get involved with the community that supports the F/OSS package to make sure that any work performed by your development team has not already been done before before devoting a significant amount of effort to make a change.

Once the tool has been modified to fit the needs of the development team, it is important to share the experiences working with the F/OSS package with the community and with others within your organization. It is considered good practice within the community and it allows others within that community to learn from your experiences and make further improvements to the package. In the course of modifying and testing the F/OSS package, a defect may be discovered and fixed. It is important to tell the community that supports the package of the fix so that the fix is incorporated in future releases and the developer will not need to maintain and support a separate version of the package.

Developers that want to work with F/OSS projects should understand the communication process, the proper methods to address the community in order to get developed code accepted by them, and an understanding that pushing fixes into the project mainline source tree allows those changes to be maintained by the entire community. Communicating on group mail lists and Internet Relay Chat (IRC) are ways to stay current on the feature implementations and reliability of various releases.

Not all F/OSS products follow the same development process. Each project is supported by a community of developers and users that establish their own culture. However, nearly all employ aspects of agile programming practices. If your development team is already using agile techniques, then it is far easier for management to embrace an open source product for use within the organization.

Good citizens of a community

To make F/OSS products successful for future use, people who use the packages should make an effort to help support that community. Engineers and testers need to factor additional time in their project development cycles to dedicate to the development efforts of F/OSS projects they use. They need to improve the packages so that it benefits them in the future.

The first thing anyone should do is to learn how the community operates, who the main developers are, and how they communicate. Be smart; before joining a community, get a feel for the priorities of the project. Read the archives in the mail list. If you have a question about how to use a particular tool, do your research before asking the question on the project mail list. Chances are that the question had already been asked and answered somewhere in the mail archives. Also the archives can tell you who the most active contributors to the project are by the fact that they are the ones that typically dominate the content of the discussions or are the ones who respond to questions about the F/OSS package. Join the mail lists and follow the discussions. Many projects also have IRC channels and this tends to be the form of choice for core developers that are collaborating together to work on a specific feature of their product. Joining the IRC and following its discussion is a good way to discover the most pressing issues of the project. Learn how to ask questions and to contribute to the project; as stated earlier, do your research before talking to the community developers.

Offer test results if you are evaluating a product or trying to make it stable for use in your organization. If you find bugs, submit them to the mail lists. Many project developers are grateful that defects are found in their packages and they often fix them almost immediately after they are notified. They are even more grateful if you provide fixes to the discovered defect in the form of a patch. Offer code patches in small manageable pieces. Larger patches (for example a 500 line change) is usually rejected by the maintainers due to its large size. The patch must be small enough to be reviewed by others involved in the project and to give them comfort that it will not introduce further defects in the package when it is incorporated.

Offer code, not specifications. New ideas are best presented as code not specifications or requirements. F/OSS is often driven by ideals and pragmatism – bottom up development, whereas commercial products are typically driven by marketing requirements. Therefore the drive to influence open source developers to develop new features by providing specifications is typically ignored (best case) or violently flamed (worst case).

When contributing code to a project, there is a possibility that not everyone in the community will accept your contributions. It is unfortunate that some in the open source community have little tolerance for new people unfamiliar with their projects. For example, the Linux kernel community is known to be intolerant of code patches that don't follow their coding style. It is best to have a thick skin and be persistent; modify your code based on the community's feedback and respond to feedback promptly. That is the quickest way to create an acceptable patch to be incorporated. Realize that in the open source community contributions earn respect.

Above all, provide feedback to the developers of the tools you use. OSDL has developed a set of open source database workloads that can be used to stress a system. Over the course of a year, there have been over 2000 downloads from the sourceforge project site. But within that same year, less than 10 people have taken the time to submit a patch that improves the operation of the workloads or even bothered to provide any feedback at all. Everyone wants to use tools for free, but the tools can be so much better for all if those who use the tools spend some time to make them better.

The success of F/OSS comes from the economy of scale provided by the multitudes of development and test engineers giving small portions of their time to make the packages they use better. It should be an obligation to give back to the community that provided you with the F/OSS tool you used to help make your job more efficient. Developers and testers alike can provide feature enhancements or patches to fix defects. Quality Assurance engineers and testers could share test results of F/OSS products with the rest of the community so that defects can be fixed sooner. The effort spent will be rewarded. Your development team gains status and

recognition within that community and it can help provide greater influence over the community development priorities if your team needs a particular feature to be added to the F/OSS package.

Corporate citizenship within open source communities is becoming more common. Several of OSDL's corporate members hire engineers to work exclusively on open source products. This gives the open source community additional resources to modify and improve their packages and it gives the corporation added recognition within the open source industry as those that help open source projects.

Management considerations

Managers need to understand and support a development team's efforts to incorporate F/OSS packages if it helps deliver products with better quality and in a shorter amount of time. Managing a project that involves the use of F/OSS packages may affect project milestones. It may be difficult for a manager to depend upon the release of a F/OSS package that they may have little control over. That makes them uncomfortable.

So, how do you convince managers that using F/OSS is a good idea? Here are some arguments to support the use of free and open source software:

1. *Open Source can be modified to fit your needs.* Open source can be customized in house, saving time to submit defects and feature requests to commercial software houses and waiting for a new release or update.
2. *Defect corrections and features can be maintained by others.* Patches can be made and submitted to the project maintainer; once your patches are in the main line source tree, they are forever maintained by the community, reducing your costs of maintenance for those fixes.
3. *Being active in development can help predictability of features.* Developer that participate in a project will know what current features are being addressed by the community and they can monitor their implementation. Furthermore, active developers can help their own causes by contributing code to the project and help deliver features their company may need.
4. *F/OSS provides a wide selection of tools to fit your process.* It may be easier to browse through a wide variety of open source projects and select tools to fit your process, rather than utilizing tools you get from commercial sources that may cause you to change your process slightly. Furthermore, tools that fit your current process reduce training costs as opposed to learning new tools that could alter your process.

Management must understand that involvement in the open source community will require some time from their development engineers. It is estimated that to be involved in driver development for the Linux kernel, it would take a little over one dedicated full-time software engineer to keep up with mailing lists and kernel changes, to stay current, and to become part of the development community. For example, the Linux kernel changes are enormous. Within the first six months of development after the 2.6 Linux kernel was released, 1.5 million lines of code changed for a 6.2 million line program![25] It is important to stay abreast of the F/OSS project email discussions where developers talk about implementing features so you know what to expect and determine if the changes impact your product or process. Projects in your company may not require that much dedication, but managers need to be aware that time must be accounted for in project planning. The commitment needs to be a continuous, on-going affair, not an infrequent cameo appearance. Once the decision is made to be part of an open source project, an engineer must work hard to maintain their level of credibility within that community.

If the manager understands the F/OSS development process, they can use their skills to manage the deliverables of their product with little or no additional considerations. Open source methods spawn a unique form of open communication and depends upon a flow of information to succeed. These techniques are not so readily found in the corporate world, and development teams could be better if they practiced this communication trait more often.

Conclusion

While F/OSS may be free of charge to download and use, there are other aspects that should be considered. Engineers have an obligation to be good community citizens and support the application they are using. If a team of developers are tying to integrate F/OSS code into their released product, license issues must be addressed. Developers and testers need to stay informed of the latest changes to the tools they use. Since this is a community of peers, any contribution one can provide to the F/OSS product would benefit the entire community of users and developers; time should be allocated from your job to help support the tools that are used.

Corporations are discovering an advantage for working together to solve industry wide problems that benefit all corporations. Initiatives such as the Service Availability Forum, the X.org Foundation, and OSDL's Carrier Grade Initiative are examples of corporate “co-opetition” that allows them to share engineering resources from several companies to fix a problem. This is a lesson they learned by working in F/OSS communities.

Tomorrow's developers and QA engineers will be working on global collaborative projects spanning beyond corporate boundaries to solve industry specific issues. Those that have experience working in the open source community will be the most qualified to take on these types of projects.

Sites for free/open source tools

One of the most obvious ways to find a potential tool is to google for it. Another method is to go to the project home page, such as Apache, samba, etc. Most organizations that create F/OSS products will end in .org (such as samba.org, or postgresql.org). Other web sites organize F/OSS packages and provide a large repository of information or host the actual projects. Below are a few of the sites one can use to find tools that may help in their day to day activities:

1. http://opensourcetesting.org/
   This site aims to boost the profile of open source testing tools within the testing industry. It provides a list of good tools that are organized into testing functions such as functional test tools, performance test tools, test management tools, etc.
2. http://www.freshmeat.net
   This site maintains the web's largest index of software packages for various operating systems. The site maintains a large database of open source packages organized into specific categories. This is a good site to begin your search for any open source application, but be warned, this is a huge database of applications; it may take you some time to find the right application. The site is actually a portal to many project communities. It offers documentation, code, chat channels, etc.
3. http://www.sourceforge.net
   Touted as the world's largest open source software development web site, it provides free hosting to tens of thousands of projects. This is one of the most common repositories of F/OSS products. Each entry is actually an open source project page where one can participate, or just download the latest code.
4. http://savannah.gnu.org and http://savannah.nognu.org
   These companion sites host free software projects. The former URL contains projects accepted by the GNU organization, while the latter host projects that are defined as free software, but are not part of the overall GNU project. Lists of software packages can be found by navigating the “Full List” link under the Hosted Projects section.
5. http://www.tigris.org
   This site is a portal to an open source community focused on hosting tools for collaborative software development. This site provides a much narrower field and removes some of the noise attributed to dead projects and those unrelated to software development tools.
6. http://www.koders.com
   This is a free search engine that provides developers with a way to find source code examples and discover new open source projects which can be leveraged in their applications. It searches a large database of open source code, so if code fragments are copied, one must be careful that copyrights and licenses are not violated.

References:

[1] The Open Source Initiative collection of approved open source licenses are found at: http://www.opensource.org/licenses/

[2] Xchat IRC can be found at http://www.xchat.org/

[3] Gaim can be found at http://sourceforge.net/projects/gaim/

[4] Mailman can be found at http://www.gnu.org/software/mailman/

[5] Information on Wiki's can be found at http://wiki.org/wiki.cgi?WhatIsWiki/ A good wiki package is Twiki. It can be found at http://www.twiki.org/ A very interesting example of a wiki in practical use is wikipedia: http://en.wikipedia.org/wiki/Main_Page/

[6] The GNU Compiler Collection (gcc) can be found at http://gcc.gnu.org/

[7] Information about GNU Make can be found at http://www.gnu.org/software/make/ The actual software package is found at http://savannah.gnu.org/projects/make/

[8] The Ant project can be found at http://ant.apache.org/

[9] A good starting point for Perl information can be found at http://www.perl.org/

[10] A fairly comprehensive web site for python can be found at http://www.python.org/

[11] A comprehensive web site for Ruby can be found at http://www.ruby-lang.org/

[12] The Tcl Developer Xchange site can be found at http://www.tcl.tk/

[13] An open source version of Qt governed by the terms of the GPL license is available for download at http://www.trolltech.com/download/opensource.html/

[14] Eclipse can be found at http://www.eclipse.org/

[15] Information for CVS can be found at the GNU site http://www.gnu.org/software/cvs/ a community web site for CVS can be found at https://www.cvshome.org/

[16] The subversion project home page can be found at http://subversion.tigris.org/

[17] The Arch project home page can be found at the GNU site http://gnu.org/software/gnu-arch/

[18] The project home page for monotone can be found at http://www.venge.net/monotone/

[19] The Junit project home page can be found at http://junit.org/

[20] The Software Testing Automation Framework (STAF) home page can be found at http://staf.sourceforge.net/index.php

[21] The Binary Regression Test (BRT) project page is found at http://developer.osdl.org/dev/brt/

[22] The project home page for Bugzilla can be found at http://www.bugzilla.org/

[23] One source to this story is carried by zdnet Australia at http://www.zdnet.com.au/news/software/0,2000061733,39184947,00.htm

[24] Licensing issues can referenced at http://gpl-violations.org/index.html For help on the issues about GPL code, there is a vendor faq that answers some questions at http://gpl-violations.org/faq/vendor-faq.html

[25] Dunlap, Randy. “Linux Kernel Development: Getting Started”, Tutorial presented at IEEE Northcon, May 19, 2005. http://www.madrone.org/mentor/linux-mentoring.pdf

Supporting 'Agile' Development with a Continuous Integration System

Christopher P. Baker
Chief Technology Officer
CrossCurrent Inc.

Chris Baker is Chief Technology Officer at CrossCurrent Inc. Chris holds an M.S. in Management in Science and Technology from Oregon Health and Science University and a B.A. in Computer Science and Cultural Anthropology from Hamilton College. He has been active in the IEEE as Chair of the Oregon Chapter of the Engineering Management Society and past chair of the Developer Special Interest Group of the Software Association of Oregon.

Abstract

Continuous Integration systems are a critical component of Agile software development processes. CI systems employ principles from the process improvement and total quality movements such as 'make problems visible' and 'find and fix problems as early in the process as possible' to increase code quality and to avoid the integration 'Train Wreck' common to many projects.

This paper discusses the benefits of continuous integration systems and how such a system supports Agile development processes. Then we take a detailed look at the anatomy of a real world system in use at CrossCurrent Inc. and provide pointers for the reader for further exploration.

Introduction

Central to an Agile software development process is a continuous integration, build, and test environment. In this paper I discuss the benefits of such a system, how it supports important Agile development processes, and then take a detailed look at the anatomy of a the system in use at CrossCurrent Inc. which is composed of mostly free, off-the-shelf, open-source components.

The primary goal of a continuous integration, build, and test environment is to always have working software because as stated in the Agile Manifesto: "Working software is the primary measure of progress." More commonly known as a Continuous Integration (CI) system, it supports and enables many of the key principles of Agile development and is core to the daily software development workflow.

Benefits of a Continuous Integration System

A CI system has the following benefits:

- Code (almost) always builds, runs, and passes all unit tests.
- Frequent integrations avoid the integration 'Train Wreck'
- Build problems are caught and made visible
- Code check-ins that break other code and made visible

Every check-in to the software configuration management (SCM) system should result in software that builds, runs, and passes all tests. A successful build means that new code has been added to the code base, been successfully integrated, and shown to work correctly with the existing code. This provides many benefits which I shall discuss below.

The CI environment at CrossCurrent consists of an integrated but independently developed set of (mostly) open source tools providing:

- Software configuration management (SCM)
- Scripted automatic builds of the company's entire code base and all related projects
- Automated unit and acceptance test execution
- Web dashboard for initiating builds and viewing status of the previous activities
- E-mail and system tray plug-ins for communicating status.
- Code test coverage metrics
- Code complexity measurement
- Rules based coding convention checker
- Redundant code finder

The Software Development Workflow at CrossCurrent

At CrossCurrent the development workflow is as follows: each developer, or developer team when we are writing code in pairs, is responsible for checking code in often (at least daily). As new code or changes are checked in, the build system retrieves the latest code

from the SCM system and executes all build scripts. All projects relevant to the modified code are built.

After all the code is built, the system runs all unit tests and acceptance tests. If the code builds and all tests pass, the build is deemed to be successful. If the code does not build (compile and link) or if one or more tests do not pass, then the build 'fails'. If the build fails, it is the responsibility of the person or team that caused the failure to fix it as soon as possible.

All development team members are notified any time the build changes state (from success to failure or from failure to success) via email and a system tray plug-in. A web based 'dashboard' is available that gives real time feedback of the status of the build in addition to reports on the tests and test coverage and other metrics.

Developers fixing the build go to the project status and report web site to get detailed information about the cause of the failure. They work on the build until it is working or, if it seems like it will take a long time, back out the check-in that caused the build to fail, fix the problems and then check-in again.

Developer Workflow

1. Developer retrieves latest code from SCM system.
2. Developer merges changes and reconciles conflicts on local machine..
3. Developer checks in reconciled code to the SCM system.
4. Build system retrieves changes from SCM system.
5. Build scripts run to build all projects with changes.
6. Scheduled builds run for release versions twice daily.
7. All unit tests and automated acceptance tests are run on newly built projects.
8. If code builds and all tests pass, build is good, if it fails, developer must fix immediately.
9. All developers are notified of build status changes via email, system tray plug-in and website status and report dashboard.

The Benefits of Continuous Integration

Make Problems Visible

One of the key principles of process improvement is to make problems visible. Visible problems are much easier to identify and are therefore more likely to be fixed. As seen above, a CI system gives immediate feedback of problems to the development team when new code has broken the build or when code changes cause unit tests to fail. This immediate feedback allows us to fix a 'broken' build quickly. This means that we almost always have a code base that builds and runs.

Further, it means that anyone can get the latest code from the source control system and know that it will compile. We also have the assurance that the code will run with reasonable reliability as proven by passing all the unit tests. Always having a successful build means that we spend almost no time with broken builds nor do we waste time trying to fix the side effects of someone else's changes.

Avoid the Integration Train Wreck

Another principle of using a CI system is that code is integrated continuously. By integrating code continuously, the integration train wreck common in many projects simply does not happen. Checking in changes to the system frequently means that any given check-in is only a short 'distance' away from the code on the head of the source control tree. When this is true, conflicts are rare and usually easily managed.

Continuous integration works by doing many small integrations rather than fewer large ones. The longer the time between check-ins, the more code changes there are to reconcile and the longer it takes to integrate the changes. Merge conflicts are more common because with more code and more time passing, there is a greater chance that different developers have modified the same code in incompatible ways requiring reconciliation. Our experience shows that we spend almost NO time with three-way or other complicated merges.

Integrating continuously finds and fixes integration issues much sooner than the 'big bang' method of integration and prevents problems from snowballing into more costly issues. For example, it was common at previous employers for development to proceed thusly:

> A developer would work for days or weeks writing code to implement a new feature. When the code was 'written' he would then 'compile' the code and work for several more days fixing the compile errors. Once the code compiled, he would check in to the source control system and then work for more days to get the code to actually work.
>
> Meanwhile, other developers have been working for weeks on their own branches not wanting to do a checkout from the head because the code almost certainly wouldn't run if it would even compile. In some cases, the source control system would be rendered almost useless because a number of developers would have vast numbers of files 'checked out' with exclusive write permission.
>
> Then, integration hell would happen, halting forward progress for days as everyone worked to broker multi-way merges.

The CI system enables many developers to work on the same code base, integrate their changes, and know that their changes have not destabilized the system. Each developer can get the latest code changes and be confident that the code will build and pass all tests before he or she adds their code. Small merges are much easier to handle than large ones.

Another benefit of successful builds build prior to the check-in of new code is that when a new check-in causes a problem, the problem space is usually small. Also, by requiring the code base to pass all tests after a check-in, changes that negatively affect dependent code are discovered immediately. One can be fearless when making a change to a library, for example, because if a changes cause a problem, the tests will most likely catch it right away.

When one knows that the code base passed all the tests before a check-in, but does not after, the scope of the problem is limited to only the most recent check-in. And because the number of lines of code in the check-in is also small because of frequent integration, it is almost always easy and quick to get the tests passing again.

Agile Processes and Principles Supported by a CI System

The continuous integration system at CrossCurrent directly supports a number of the key process areas and principles of our development process and agile development in general. Below I mention each principle or process area and then discuss how the CI system supports each.

Find and Fix Defects as Early as Possible

Edward Deming and the process improvement pioneers of Japan[1] discovered a key quality principle at the heart of a Continuous Integration system: Problems become exponentially more expensive to fix the farther away in process space or time they are from the point at which the problem was caused. Essentially: Find and fix defects as early in the process stream as possible.

The goal is to not allow defects to progress to downstream processes where they will be more expensive to fix[2]. For example, a defect quickly found by a developer's unit test while developing the code lives only a few moments. A behavior change in a library that affects client code caught when running the full unit test suite never gets entered into the issue tracking system, never causes another developer to need to debug a problem because their code is not working for some reason not apparent to them and never gets delivered to the customer thereby never causing support issues.

When tests fail upon check-in, the developer can address the problem while the mental model of the code is still in his or her mind. Further, other developers are much less likely to spend time debugging problems introduced by other check-ins to the code base. When one knows that the code worked a short time ago but doesn't now, the amount of code that needs to be considered for the fix is often quite small. The code and issues are still fresh in one's mind and the problems are usually quickly dispatched.

A CI system makes problems visible early allowing defects to be discovered and fixed with all the downstream benefits that accrue.

Deliver Functionality Incrementally

If the code works after each feature is added, in principle, the product can be delivered after each increment. Even if the product is not delivered after each increment, having

[1] The Toyota Way

[2] It is best to prevent defects in the first place. Agile development has other methods for this such as working directly with a customer to prevent requirements defects and pair programming to prevent design and coding defects upfront. While valuable, these methods are beyond the scope of this paper.

complete and stable code, at each iteration, enables flexibility in the order of features added. Also, developing software in this fashion tends to favor loose coupling between components which increases testability and decreases schedule dependencies.

The Continuous Integration system at CrossCurrent supports this goal at the micro level by helping to make sure that we always have working, high quality software. Checking in often and requiring that the code build and pass all tests, favors decoupled designs that have good testability. Each check-in represents solid, measurable progress.

Embrace Change

At CrossCurrent, we are able to be nimble in responding to customer and marketing needs because we are never very far away from a solid version. Further, the continuous integration system predisposes the development team to write code that favors flexibility in the order in which features are delivered. Our designs are, by necessity, much more decoupled than before we adopted this system.

We no longer fear Marketing's inevitable change requests.

Test Driven Development

Critical to a CI system is that the code base is covered by a comprehensive set of unit test. These unit tests, when run after each build, are necessary to know that a change has not broken existing code. While it is possible to write tests after writing the code, our experience indicates that writing tests while developing the code produces superior results. Writing unit tests concurrently with the code has made the biggest difference to our downstream code quality.

Writing the tests along with the code makes sure that the code is of high quality to begin with and passing the tests with every check-in makes sure that bugs do not reappear and that existing code continues to work as expected.

A high quality code base makes adding new code much easier. Our experience is that we are more productive because we rarely waste time with bugs in other code. Working with quality code greatly limits the problem space when new code is not working.

All the Tests Running All of the Time

Our experience shows that it really is important to run the unit tests after each check-in to the code base: We never had "All the Tests Running All of the Time" until we added them to the CI system. We just never ran them all after making changes. We would run the tests for the code we were working on, but the remainder we'd 'forget'.

What tends to happen is that a test here and there will break and then when one runs the whole suite, one cannot tell what check-in broke the test. Getting the tests to run again is now much more difficult because problem space extends back to the last time all the tests

ran. Running a test suite where they don't all run now becomes less helpful or is a distraction from the current development task.

It really is important to have all of the tests running all of the time.

Continuous Refactoring

Refactoring as new code is written or old code is modified is a central precept of Agile development methodologies. Refactoring is not rewriting the code to do something new, it means improving the design of the code without changing its functionality. Refactoring is often a precursor to adding functionality.

In practice, we find there is not always a bright line between refactoring and writing code for a new feature. What we do strive for however is making changes or adding functionality to the existing code base in micro steps with tests written first to prove each change or addition works as expected and has not broken code elsewhere.

Test-driven-development and refactoring go hand-in-hand and knowing that all tests pass is generally a prerequisite to refactoring. Because our CI system makes sure 'all tests pass all the time', it gives us the confidence to refactor as necessary.

Enabling refactoring has long term positive consequences for CrossCurrent as a company. Proof will take a bit longer but I believe that refactoring is a method and capability that will prevent or attenuate the need to rewrite whole products from the ground up.

Code bases, like bureaucracies, accrete functions over time and become increasingly calcified. It becomes more and more difficult to make substantive changes because of coupling of components, muddled architecture and design, and code that no one knows anything about.

Refactoring allows the product design to evolve with the product's requirements over time and allows software engineers to continuously make the code base better by making frequent incremental changes supported by unit tests to a very large base of code that would otherwise be very difficult or impossible to manage.

Refactoring, incrementally, enabled by a CI system, is our best chance at avoiding the need to throw it out and start over again at some point in the future.

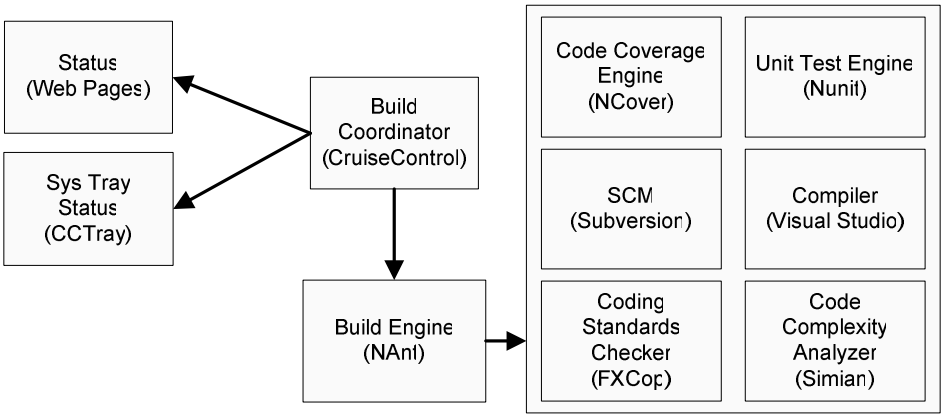
Anatomy of a Continuous Integration System

The Continuous Integration system at CrossCurrent delivers many benefits to the engineering team and the company as a whole by making us more nimble in meeting the needs of our customers and in enabling us to deliver higher quality code more quickly.

CrossCurrent develops software primarily for Microsoft operating systems. While much of our code is C# running on the .Net framework, we have various other projects that use C++, Java, XSLT, Flash, assembly language, ASP, Perl and MS SQL server. Knitting all these technologies together is a challenge.

Our CI system essentially is composed of a main build engine that monitors the SCM system for newly checked in code. When changes are detected, or on a schedule, the build engine checks out the new code and builds it. Then in turn, the resulting binaries are loaded and unit tested and then other services are run to provide code coverage metrics and to check the code for compliance to coding standards.

The diagram below shows how these components interact:



Dedicated Build Machine

At CrossCurrent, we have dedicated a reasonably powerful machine to do little else than build and test the code at every check-in. The build machine has become a central part of every developer's workflow and therefore is powerful enough to do a full build quickly. One does not want to wait long to get the results of a check-in.

Software Configuration Management (Subversion)

The software configuration management system of choice at CrossCurrent is Subversion[3]. Inspired by CVS[4], Subversion is the workhorse of our CI system. Subversion is the repository for all source code and other files needed to build and run unit tests with the exception of operating system and database software. The goal is to have everything needed to do a build and run the tests available in the repository.

The SCM system is a common repository for all code. As mentioned above, developers check-in often for best results.

[3] Subversion and various companion tools can be found at http://subversion.tigris.org/.
[4] Concurrent Versions System (CVS) can be found at http://www.gnu.org/software/cvs/.

Any number of SCM systems would work well. We find that Subversion, however, has a couple of unique features that make it especially well suited to a CI system.

1) Subversion is transactional: Multiple files are checked-in as a group and either all succeed or fail. Also, the check-in fails if any of the files collide with changes checked-in to the head allowing changes to be reconciled locally before check-in to the repository. This prevents a partial check-in that leaves the repository in an inconsistent state where the code will not compile.

2) Check-in's are multi-file: Each check-in (one or more files) to the repository is given a 'commit' number where the entire check-in (even if it consists of many files) can be backed out easily. This makes it easy to keep the files on the head build-able at all times and comes in handy on the rare occasion when a build breaks and it will take some time to fix.

3) Subversion follows the 'Copy-Modify-Merge' paradigm: Unlike some systems that lock files to prevent others from making changes, Subversion focuses on merging changes at check-in time. Subversion is a better fit for parallel development than a 'check-out locked' system.

Important features not unique to Subversion are:

1) Secure internet access to the repository. There are many ways to make Subversion accessible to remote developers. We have configured Subversion so that it is available to developers in or out of the office.

2) Plug-ins for integration with our IDE[5] (VisualStudio). Ankh[6] allows us to interact with the Subversion within our main development environment.

3) TortoiseSVN[7] is another indispensable tool for Subversion. This tool has many intuitive features that make managing code easy. It's especially good at handling merge conflicts and navigating the repository to discover who made changes to what when.

Automated Build System

We use NAnt[8], inspired by Apache Ant[9] for Java. It is essentially a very flexible build tool. NAnt is extendable by adding .Net objects that implement a standard interface thus making it possible to add almost any function to a build.

[5] An Integrated Development Environment (IDE) combines numerous development tools into a single product.
[6] Ankhsvn, a plug-in for integrating Subverion with VisualStudio can be found at http://ankhsvn.tigris.org/.
[7] TortoiseSVN, a terrific GUI for Subversion is available at http://tortoisesvn.tigris.org/.
[8] NAnt, a build tool for Windows focusing on .Net builds can be found at http://nant.sourceforge.net/
[9] Ant for Java is available at http://ant.apache.org/

NAnt is the glue that combines that various tools. Our NAnt build scripts check out code for each project, compile and link the code, build installers, assemble database creation scripts, and a number of other tasks required to fully build each project. Then the scripts run the unit and acceptance tests and reports the results.

NAnt build scripts need to be hand built for anything but fairly simple tasks. There are tools available however that convert VisualStudio project files into scripts which make keeping the script files up-to-date much easier. We use the 'Slingshot[10]' command.

NAnt has hooks to many SCM systems, code statistics, a command to copy files to a remote server, allows calls to code in libraries one provides, or can call anything one wishes via the command shell.

Unit Testing Framework

Unit tests are run using NUnit[11] which was initially a port of JUnit[12] for Java. NUnit is an outstanding tool and was the impetus for CrossCurrent adopting Agile methods. We loved it so much; we decide to investigate the rest.

NUnit is run with a user interface or as a command line tool and therefore is very useful while developing code interactively and as an automated process as we do on our build machine.

At CrossCurrent, we generally write NUnit test class below the class that it is intended to test. We surround the test class with a compiler directive to compile out the tests for release builds. Creating a test is as easy as adding attributes to a class that indicates that the class is a 'Test Fixture'[13].

NUnit uses reflection to dynamically load each dll and executable. It then finds the tests it is to run, calls each test and reports the results to an xml file which are then used by other tools such as NAnt and CruiseControl (discussed below).

A refinement to unit testing we have recently added is coverage metrics with NCover[14]. These metrics are giving us a good sense of how much of the code is covered by our unit testing. When the unit tests don't cover a significant percentage of the non-trivial pathways in the code, then one can be sure there are undiscovered bugs waiting to be found and that the developer needs to write more tests.

Related to test coverage is code complexity. Complex code is difficult to modify and maintain because it is difficult to understand what it does. Often this kind of code has a

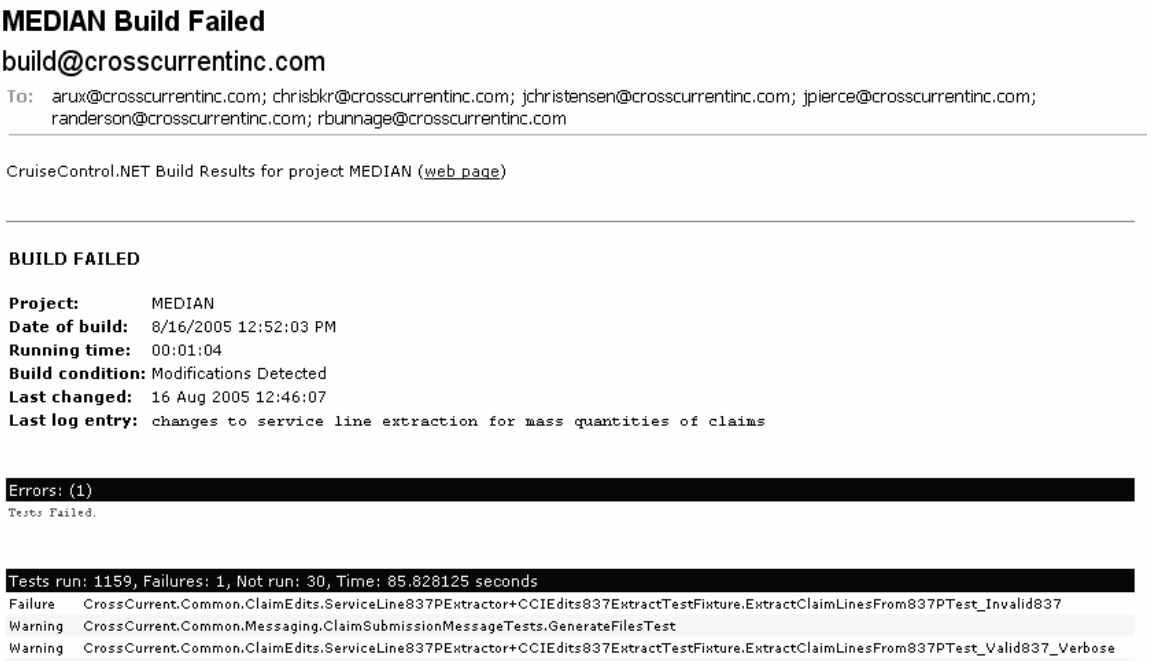
[10] Part of the NAnt contributors section of the project, it is available at http://nantcontrib.sourceforge.net/
[11] NUnit can be found at http://www.nunit.org/.
[12] JUnit can be found at http://www.junit.org/
[13] A good explanation of Attributes in C#/.Net can be found in MSDN magazine: http://msdn.microsoft.com/msdnmag/issues/05/05/BasicInstincts/default.aspx.
[14] NCover, as code coverage measurement toll can be found at http://ncover.sourceforge.net.

lot of execution paths that are difficult to test--many of them are error conditions and corner cases that acceptance level testing does not attempt to explore. We have not yet added this report to our system but intend to do so.

Build Manager

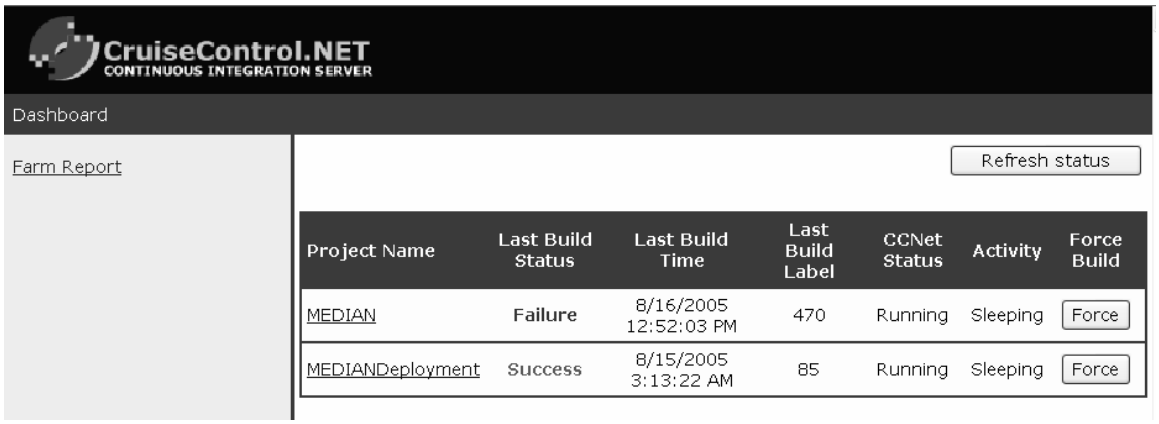
CruiseControl[15] is the build manager and user interface to the results of builds on the build machine. CruiseControl starts regularly scheduled builds and starts new builds when code is checked in to the Subversion repository using NAnt. It also sends email and communicates status to developers via a system tray plug-in when the build status changes.

By navigating to the projects page, one can examine the results of a build including compiler errors and warnings. In addition, there are reports for all the unit tests including their run times. A user can also manually start a specific build with a single click of a button.

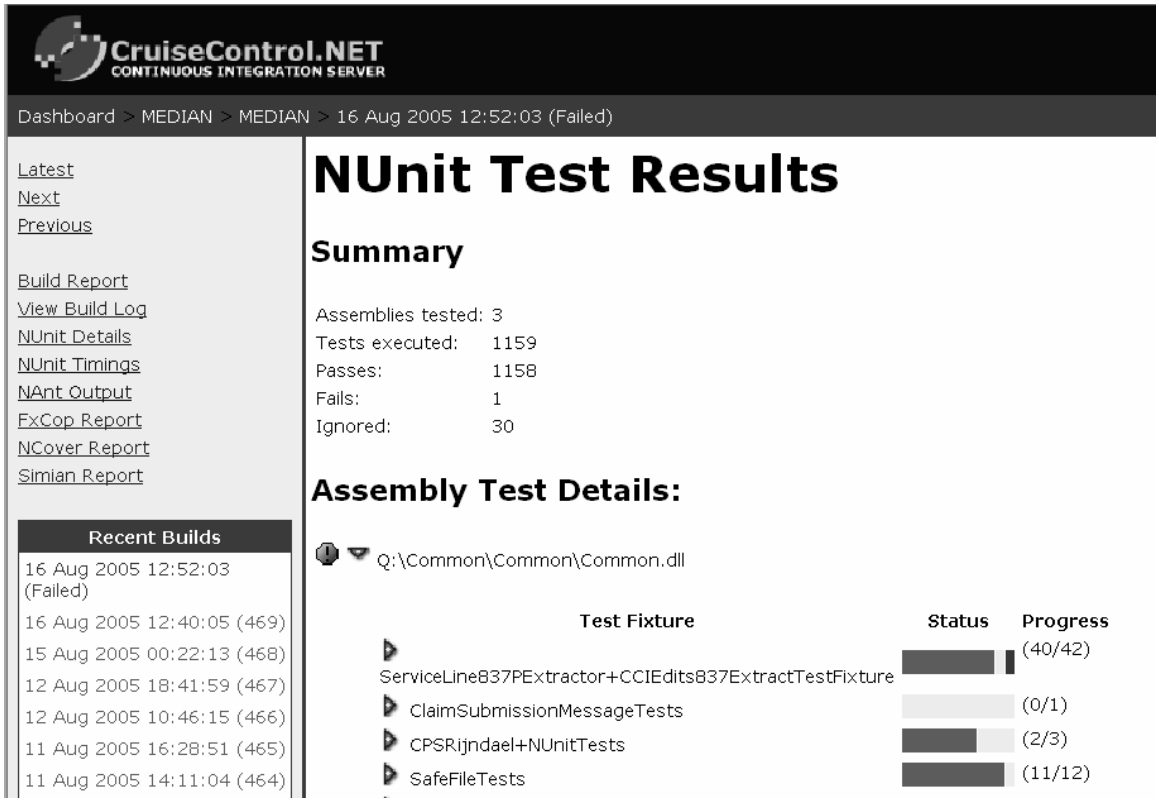


Developers receive an email informing them of a build failure.

[15] CruiseControl, as CI build manager can be found at http://cruisecontrol.sourceforge.net/main/index.html



One can visit the website to see the status of various builds and to manually start a build.



Each project has detailed reports available. In the example above, a developer can see the results of a check-in that caused the build to fail. It is easy to find the failing test and fix the problem.

Conclusion

A continuous integration system is an important tool in support of Agile software development methods. A CI system prevents the integration 'train wreck' common to many development projects while increasing and maintaining code quality and catching errors early in the development process.

To make CI work, a development team needs to practice other Agile methods such as frequent small integrations, and test driven development with unit tests covering all non-trivial code. In addition, a CI system enables other Agile practices such as refactoring by making impacts to other code quickly visible so that they can be fixed.

CrossCurrent has benefited greatly by using CI as a critical component in our software development process by helping us produce better products more quickly. CI also enables less tangible results such as allowing the team and thus the company to respond rapidly to market opportunities.

Using an Open Source Test Case Database: A Case Study

Doug Whitney
Software QA Engineer
McAfee, Inc
503.466.4520
dwhitney@mcafee.com

Have you ever spent all day looking for the perfect test case database? Ever found one that fit your enterprise perfectly? What about the cost? The team at McAfee took a different approach. We found open source software and modified it to suit our multi-national and multiple project needs.

Doug Whitney is a Senior QA Engineer at McAfee with 12 years of testing experience. He chaired a panel discussion at Quality Week 2000 titled “Protecting Intellectual Property in an Open Source World”.

HOW IT ALL STARTED AT MCAFEE

This process started when the Director of QA at McAfee asked me, within my first month of employment, if I knew of any good test case database programs. McAfee had created a "roll your own" program that was not being adopted due to a number of deficiencies – most notable was the lack of reporting and the overwhelming structure needed for test case input. The QA engineers had continued to use their custom spreadsheets that allowed for special reporting to track progress. They were used to "striping", which is running the same test on various computers, operating systems or localized languages that was not supported on the roll you own version. I told him that I knew a good one, QASYNC, although it could have its limitations for usage at McAfee. I suggested that I search online to see if there would be any other test case databases that could better meet the needs of our QA group at McAfee.

StickyMinds

The test tools area on the website StickyMinds.com had always provided me with ideas for other type of testing tools. I had read reviews and looked for information on configuration management, requirements and defect tracking. I searched for test case database and got several hits. Most were commercial and I was looking for something inexpensive and for one that we could modify if necessary. I looked at most of the StickyMinds choices and then decided to go to SourceForge, because it is an open source site and you can easily modify open source software.

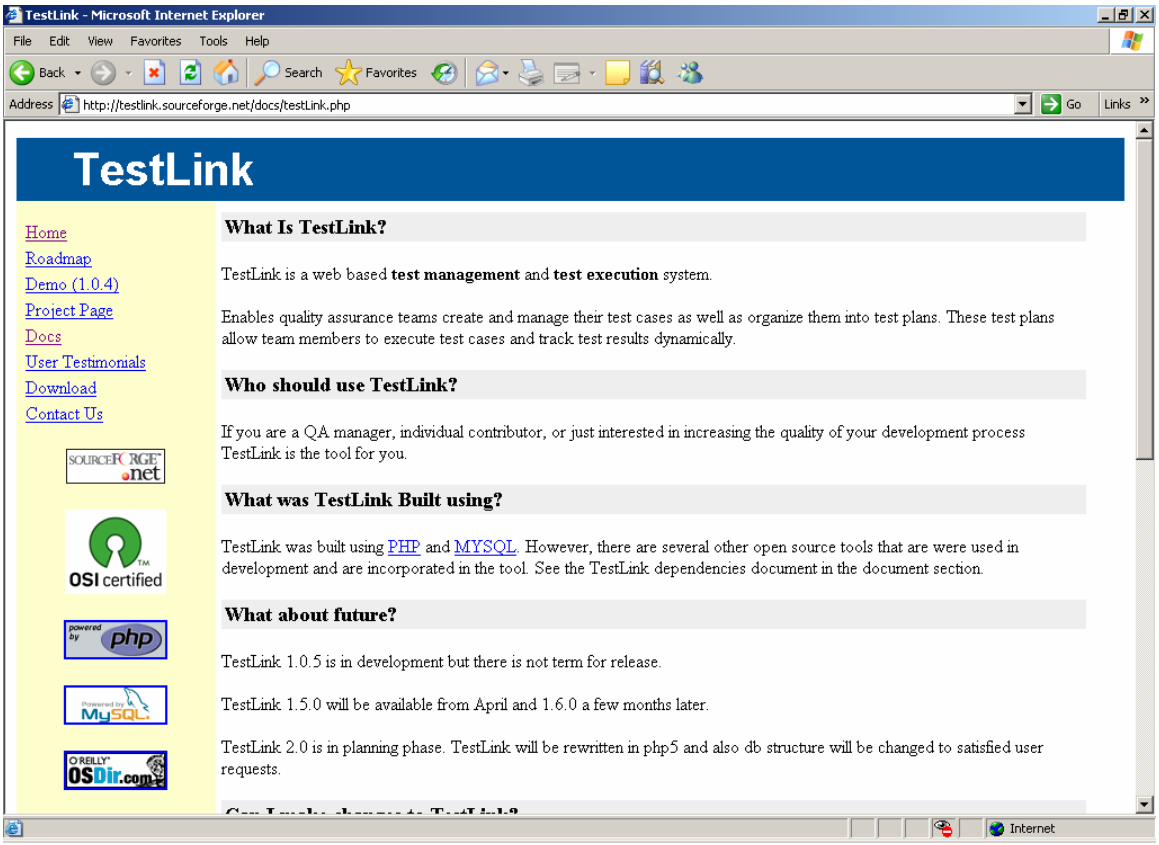
SourceForge

On the SourceForge web site I searched for "test case". There were several hits and the second on the list was Testlink. After viewing the TestLink data, it appeared to have what we were looking for. The SourceForge web site provides summary overviews of projects that include the ability to drill down for additional information. You can find TestLink on SourceForge here: http://sourceforge.net/projects/testlink .

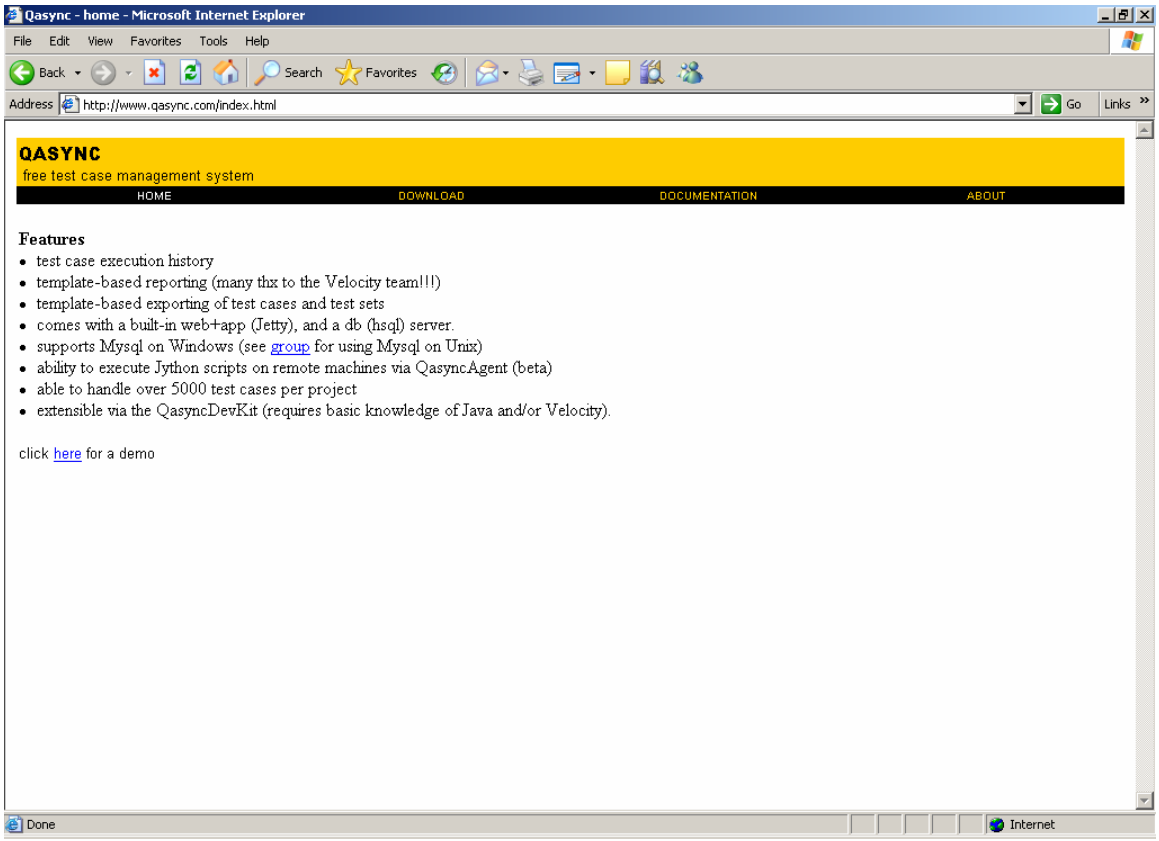
Contenders

The following pictures of the TestLink and QASYNC home pages give you a look and feel of both these test case database projects. In the case of TestLink, the Home Page section provided the most useful data. There are links to the roadmap, documentation, testimonials, and also a demo of the product. We found the documentation very useful, especially the information on the installation process. All in all, for an open source project it is refreshing to see things like bug numbers and suggestion totals. In contrast, the information available on the QASYNC website is quite sparse.

Testlink Home Page:



QASYNC Home Page:



EVALUATION PHASE

Evaluation criteria

From a functional perspective, what we were looking for had to replace macro filled spreadsheets and the failed attempt at creating a "roll your own" test case database internally. Although the spreadsheets allowed for a repeatable process, they did not allow for sharing of test cases and were not consistently backed up. The internally created test case database had issues with reporting that did not allow management to track progress of who was completing the testing. We also wanted something we could modify.

From a scalability viewpoint, several geographically distributed teams would have to use the new system at McAfee worldwide. From an initial small number of engineers, usage could eventually grow to include about 500 QA engineers in 5 countries who would be using the new test case database.

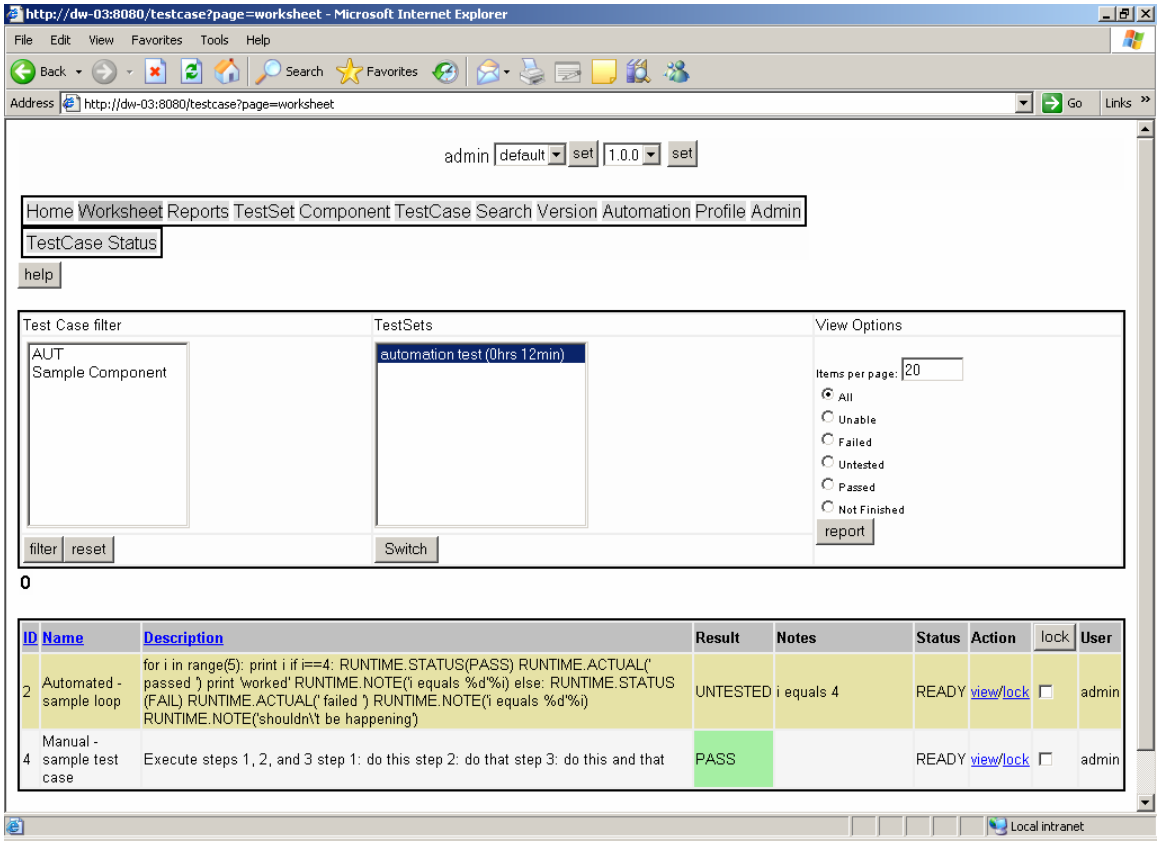
Picking one - QASYNC or TestLink

At my previous job I had used QASYNC. So when the director of QA asked if I knew of any test case databases, I had ready knowledge. My previous job was a single project with a team of seven engineers and two QA. (Yes, I know – it was a little out of balance) The software worked well for us as it allowed for a repeatable test process. It was pretty straightforward to set up and test case input was easy. QASYNC had a Yahoo group associated with it that I had been a member of for a couple of years. The binaries from QASYNC were free, but the code itself was not available for modification. I had sent private messages to the developer, but received no reply. QASYNC has a web site that was maintained by the main developer. It is: http://www.qasync.com.

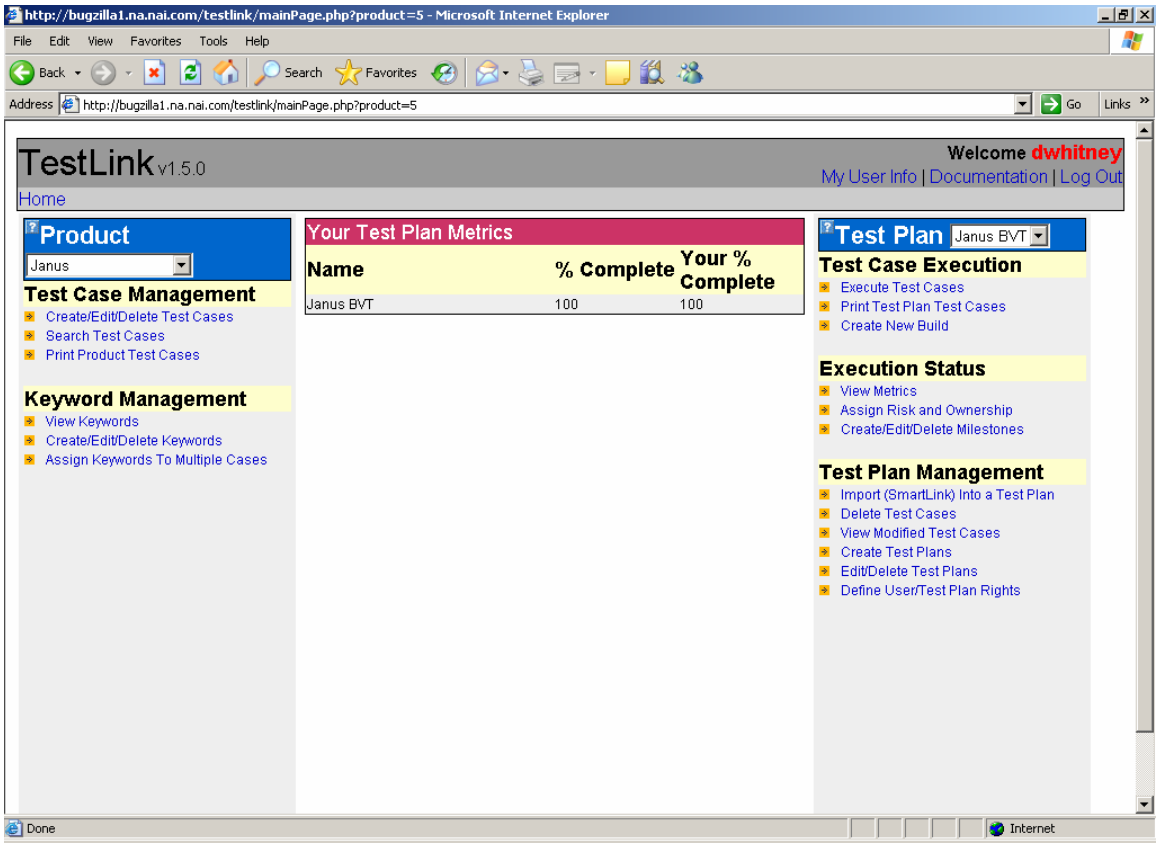
We set up QASYNC in a test environment for a couple of the managers and the Director of QA to evaluate. We also set up TestLink. Both systems allowed for the creation and reuse of test cases. They both had reporting functions and could track testing progress. Since both had database back-ends, they could easily be backed up. We saw early on that TestLink was more adaptable to our needs than QASYNC. There was a better relationship between test case and test plan. It had a more extensive report function (down to the point of who was the last to run this specific test and on what build was it ran). It also allowed us to assign test cases and evaluate not only the entire projects progress, but the individual testers as well. TestLink also provided an ability to designate importance to test cases and to assign keywords for sorting. We came to the conclusion that TestLink was our selection. It also gave us an opportunity to modify the source code, since it was freely accessible, in the event we needed or wanted to add functionality.

The next two pictures give you a look and feel of the client interfaces of, respectively, QASYNC and TestLink. Note that the client interface of both systems – a web browser – shows the platform independent nature of both these systems.

QASYNC example:



TestLink example:



TESTLINK TEST SITE

Creating a test site

Since McAfee was already using Bugzilla, which is an open source bug tracking system, and the installation requirements were similar, we deployed TestLink to the test area we had for Bugzilla. This area, which has Perl and a MySQL backend, is used to validate changes for Bugzilla that needed testing prior to rolling out updated versions. In the case of TestLink, we deployed it to the same test environment and started the evaluation. We used this test site for about a month to create test cases, test plans (suites), and to show the metrics reports to the managers and QA Director prior to moving some projects to a live site.

A live site for early adopters

My team was selected to participate in the trial on the live site. This site was based in Plano, Texas where we have one of our main data centers. We wanted to make certain that our world wide team of developers and QA would have access 24/7. The data center has a backup battery supply, a larger Internet connection to our network and daily data backups. During this initial time on the main site and prior to the first modifications, over 11,000 test cases were added.

Meeting of main users and stake holders

A team of stake holders was created. It included the QA Director, IT Manager, QA Managers and QA Leads for teams that could be early adopters. This team decided that we would move forward with a live site for a couple of teams while the rest would continue to evaluate and propose changes. We met to discuss what was working well for us in either the production environment or in the test environment. We also requested changes to be made that would make TestLink more in line with our existing work flow. The biggest issues to start with was striping and reporting.

Test Site - Also used to validate modifications made

Once changes were made to the code, we needed a place to test the changes. The test site was continued to be used for this purpose. The teams that were not on the live site continued to test and evaluate the changes that were being implemented. We continued to meet as changes were made to make certain that nothing critical broke and that the changes were implemented as we expected. After all, we are QA.

Bugzilla project created

Since the project was becoming more visible and we wanted to track changes, we created a Bugzilla project. In the project we proposed changes or Feature Modification Requests (FMR’s) that would make TestLink more useful. We also created defects for bugs in the changes and updates that were being made. Our infrastructure team dedicated one engineer (Brendan Leber) to making the changes and feature requests for TestLink. This engineer also works with our Bugzilla deployment.

TESTLINK ROLLOUT

Announcement

We have a process at McAfee that when we make a change to an internal tool program, we send out a notification via email to all concerned. In the case of the first update to TestLink, Mr. Leber sent the notification indicating that we would be migrating TestLink on a Friday evening. We set a deadline of 5:00pm PDT to have all items completed. The process completed successfully and on Monday we were back up and operational.

Migration of existing data

Mr. Leber created scripts that exported the data from the underlying MySQL database, updated the tables, and imported the data back into the new structure. He then ran a series of tests in order to validate that all data was restored correctly. This was necessary since one of the enhancements modified the database schema.

Current Status

Currently, several teams at McAfee are using TestLink. The number of users is running about 50 QA engineers in our Oregon, Bangalore, India, and Cork, Ireland, facilities. As existing projects reach their respective milestones, more engineers are expected to start using it for their next projects or milestones. Eventually over 500 McAfee QA engineers in 5 countries could be using our version of the TestLink product.

LESSONS LEARNED

Potential TestLink gotcha's

The only area that caused any issues during the migration was our use of the test case numbers. We use an iterative development process and a collaborative work environment to track tests for each release. In our iteration user stories, we had used the test case numbers for tracking the completion of the acceptance tests. The test case number was the database index number. When we exported from the current version and imported into the updated version the data all of the indexes changed. Mr. Leber understood this would take place and provided us with a decoder ring. This was a file that listed the previous test case numbers and the new test case numbers. Another area that could cause issues during a migration would be test case formatting. TestLink allows test cases to be formatted with bullets, numbers, bolds and indents. When we migrated, we lost the formatting. I will note that the changes we made to the source were substantial and most updates would not require the data migration that we needed.

Updating Open Source Software

The changes that we have made to TestLink were not in line with the future direction of the core TestLink team. Mr. Leber has created our own branch of the TestLink code. We have been "doing our own thing" to make certain that the needs of McAfee have been met. We will not publicly "release" our version; it is for internal use only. We will however, release our changes back to the core TestLink team, which fulfils our obligation under the GNU public license (or GPL) used by TestLink. If they choose to incorporate any of our changes, they certainly can. That is the beauty of using Open Source software.

CONCLUSION

We decided that using an open source test case database would allow us the flexibility to make changes to the source if the need arises. This flexibility makes the use of TestLink at McAfee an easy choice when compared to a proprietary software package, either commercial or free. Our world wide team of QA engineers can now standardize on a product that we can modify to best suit our needs.

FreeBSD: The Biggest Toolbox In The World

Chris McMahon

christopher.mcmahon@gmail.com

Biography:
Chris McMahon has been for nearly a decade a tester of library software, telecom, stock trading, and networked video systems. He has published a number of articles aimed at technical testers, but his true career is at being an enthusiastic beginner. Chris works for ThoughtWorks, Inc. as a tester, and can be reached at christopher.mcmahon@gmail.com.

Abstract:
From late 2003 until early 2005, I was a tester in an all-Windows environment. Although unlikely on the face of it, FreeBSD, an open-source Unix-like operating system, became a valuable test tool platform in that context. FreeBSD contains useful and powerful applications for any tester in any environment.

Unlike Linux, FreeBSD is a single monolithic project rather than a collection of disparate parts assembled into a distribution. And the most attractive part of FreeBSD for a software tester is the FreeBSD ports collection—a very large, managed set of software applications with a single simple and uniform installation procedure.

Following an introduction and some background on FreeBSD, this paper describes several software test tools from the FreeBSD ports collection that the author used in an all-Windows environment.

Introduction

Software testing environments are radically more complex than software development environments. Interconnected systems to test, network entities, databases, and filesystems present challenges to testers that developers can for the most part mock out and essentially ignore. Software testers need more tools, and more complex tools, than do software developers.

On the other hand, software development tools are much more highly evolved than software testing tools. There is no Eclipse or IntelliJ or even Visual Studio aimed at testing. Testers struggle and scratch to find tools appropriate to their test environments and appropriate to their Systems Under Test (SUTs).

Every test environment is different. Finding appropriate test tools is challenging. And when a tester has identified a set of useful tools, introducing them to the test environment and integrating them with the test environment is a challenge.

The set of tools available with the FreeBSD Operating System (OS) is amazing. The FreeBSD ports collection contains more than twelve thousand separate applications, all of which have a standard installation procedure and conform to a set of guidelines that make them reliable without the need to manage dependencies, appropriate versions, and all of the other problems that affect even the most well-managed Linux distribution or the various versions of Microsoft Windows. The monolithic nature of FreeBSD and the FreeBSD ports collection removes much of the trouble of integrating tools with the test environment, regardless of the OS under which the SUT runs. FreeBSD is a highly evolved server environment, and contains so many reliable applications, that every tester should consider adding a FreeBSD machine (or several) to their test environment.

The 12,000 tools available in the FreeBSD ports collection are designed to support a highly evolved server environment—not necessarily a test environment. Of course, some of those tools are not useful for testing—you probably won't need seven hundred eleven games, or four hundred ninety-two utilities for the GNOME desktop environment. But you might need some of the forty-nine tools for accessibility for disabled users, or some of the one hundred nineteen data archiving tools, or possibly even one of the fourteen tools for computing in Ukrainian. Some more obvious choices are the six hundred three system utilities, the one thousand forty-one network tools, or the fifty benchmarking tools. Using the FreeBSD ports collection for testing requires a certain amount of creativity, improvisation, and perseverance. Whether your test environment is Windows, UNIX, Linux, Mac OS, FreeBSD itself, or some combination of any of them, FreeBSD and the FreeBSD ports collection is a great place to look first.

This paper describes some of the (sometimes highly creative) ways that I have used some of the tools available on FreeBSD for software and system testing, particularly in an environment of networked Windows machines. I hope to encourage others to add their own FreeBSD applications to their own toolboxes.

FreeBSD: Background

In the early days of sophisticated operating systems, there were two main implementations of the UNIX standards: the AT&T version, developed mostly in and around The Massachusetts Institute of Technology (MIT); and the Berkeley Systems Distribution (BSD), developed mostly in and around the University of California at Berkeley. BSD UNIX became open source in the early nineties as a result of a lawsuit. The now-free BSD system spawned several offshoots: the most well-known is FreeBSD, which yahoo.com (and about.com, etc.) runs on; also important are OpenBSD, the most secure OS in history; NetBSD, which runs on almost any hardware; Darwin, the guts of Apple's Mac OS X operating system; and DragonflyBSD, which is experimenting with kernel architecture. Except for Apple's Mac OS X, FreeBSD is far and away the most popular, versatile, and widely-used of the BSD systems.

Besides the OS itself, the FreeBSD license is an important contribution to the universe of software. Where the General Public License (or GNU public license, or GPL) requires the user to publish any changes made to the licensed software, the BSD license only requires that, if the user changes the software, that the BSD copyright notice be displayed prominently. Because of the BSD license, providers of proprietary software feel free to use BSD code in their products. The core of Mac OS X is BSD. Yahoo.com runs on a BSD system that has probably been modified to suit Yahoo's purposes. There is even BSD code integrated with the Microsoft Windows IP network software.

It is worth noting that the BSD systems are remarkably stable and robust. FreeBSD routinely accounts for around half of the most reliable systems in the world as measured monthly by the reliability standards at netcraft.com. It is hard to imagine that a tester would ever need something as large and solid as what runs yahoo.com—but FreeBSD scales down as well as up, and FreeBSD running on a single old Pentium II machine is just as reliable and complete as the software system that runs yahoo.com.

FreeBSD Documentation And Resources

FreeBSD's system documentation and the documentation for the tools in the ports collection are consistently excellent. The resource of first resort is the FreeBSD website itself, freebsd.org/docs.html. My own experience is that regardless of what other documentation I might consult, I usually end up on the FreeBSD web site for the final answer.

An excellent overview and first reference to FreeBSD that is frequently updated is *The Complete FreeBSD* by longtime FreeBSD committer and promoter Greg Lehey. My own opinion is that this book sacrifices organization for completeness, but it is a wonderful first reference (given that the FreeBSD documentation pages are available).

Finally, the O'Reilly publishing company has a large number of articles on its website devoted to FreeBSD. Many of these are written by Dru Lavigne, and many of the articles are aimed at beginners and at people with experience in other OSs that might be new to FreeBSD. Skimming these articles gives the new FreeBSD user a very good sense of the power and utility of FreeBSD.

How To Use The Ports System

This is how you install an application “foo” from the FreeBSD ports collection:

```
cd /usr/ports/foo
make install
```

and the system does the rest. It reports build status and test status, and installs all the relevant documentation as well. You can see why this is attractive to a tester, who typically is pressed for time!

FreeBSD For Testing

The test environment should be more stable than the SUT. Once the tester decides to use the tools available on FreeBSD, FreeBSD’s long record of reliability makes it an easy choice for a test tools platform.

My own introduction to FreeBSD came like this: I was hired by a major vendor of large-scale network security video services to be their “network testing” person in an all-Windows environment. My first assignment was to replace the obsolete, buggy, disk imaging system. I chose to do that with an open-source disk imaging system called “Frisbee” which was implemented originally on FreeBSD. I built the system—a feature-for-feature replacement for an expensive proprietary system—but, for a number of good reasons, we never actually used it in our production system.

In the meantime, I had discovered the FreeBSD ports collection and started to use some of those tools for testing; and I had discovered the power of disk imaging with Frisbee, especially for smoke testing and installation testing; and FreeBSD became a permanent part of my test lab. The test lab I built, and the FreeBSD systems I created still exist, and still provide value to the testers there.

The rest of this paper describes my experience with various FreeBSD applications that I used as test tools in an all-Windows environment with multiple network devices.

FreeBSD For Collaboration: Twiki

Although I started using FreeBSD because of the Frisbee disk imaging system, this paper will first discuss Twiki, my favorite wiki, and how and why I chose to use it. My experience with Twiki illustrates many of the skills necessary for managing other FreeBSD applications.

A wiki is a simple set of web pages to allow many users to share information and collaborate on all sorts of documents. Twiki is a wiki fine-tuned for document management. It has built-in version control and security implemented at the user/password level. I used it for requirements management and traceability.

Twiki is implemented in Perl, so I felt comfortable using it. (Although reasonable people dislike Perl for creditable reasons, Perl is deeply integrated with FreeBSD, and is extremely powerful, particularly on FreeBSD.)

The company I worked for managed their requirements in a proprietary tool with a truly byzantine hierarchical structure. Although the tool was fine for generating requirements, it was horrible for testing. Also unfortunately, the tool supported very few export mechanisms. From the proprietary tool, I had to

- Export to Access
- Export from Access to Excel
- Export from Excel to a delimited file
- Import the delimited file to the wiki by means of a Perl script

And in Twiki I was able to implement version control, traceability, and test coverage, all features missing from the proprietary tool. This gave me a remarkable ability to trace requirements coverage with my tests. Having imported a complex hierarchy of sets of requirements, I could use Twiki's simple markup features to show each requirement as tested, passed, or failed. I could attach test documentation to wiki pages, and since I had already scripted the process, it was easy to update the requirements as they changed. And Twiki itself handled version control for such updates.

As with all of the examples in this paper, installing Twiki on FreeBSD is fairly simple. It takes just a few minutes on a FreeBSD system. However, if you want to use Twiki on a Microsoft Windows platform, I strongly suggest you read the Twiki documentation extremely carefully. I know someone who installed Twiki on Windows, and it took him several days. Twiki on Windows requires not only knowledge of Windows, but also deep knowledge of Cygwin and Perl.

Furthermore, at one point in the project I had to migrate my wiki from a machine running FreeBSD 4.8 to one running FreeBSD 5.3. The migration consisted merely of installing Twiki on FreeBSD 5.3; using FreeBSD's "tar" feature on the FreeBSD 4.8 machine to gather all of the Twiki data files specific to my testing; FTPing the gathered files to the new FreeBSD 5.3 machine; and untarring the file. The complete set of Twiki documents migrated with no issues or problems at all. That's the power of a unified system like FreeBSD.

FreeBSD For Disk Imaging: Frisbee

A disk imaging system is a mechanism for saving and restoring all of the data on a physical disk. The most popular commercial system for doing this is probably the product Ghost from Symantec.

Frisbee is an open-source disk imaging tool from the University of Utah that conforms to the same guidelines as the applications in the FreeBSD ports collection.

The Frisbee enterprise disk imaging system mentioned above had a lot features I never implemented in the test lab. Using Frisbee and an open-source tool called PXELINUX, I was able to

- Boot the Windows client machines from the network
- Make an image of the client
- Update the BIOS on the client
- Lay down an existing image on the client machine
- Make a set of "restore CDs" for the client

In the test lab, I only needed to boot from the network, make an image, or lay down an image on the client machine. Both Frisbee and proprietary imaging systems allow the user to image individual drives on the client, but I never had a need to do this.

Installation testing was a large part of my duties at the company where I used FreeBSD. To do this testing, I would typically use Frisbee to make an image of a machine containing only a Windows OS, install the SUT, and run a smoke test. The smoke test typically left the test machine in a very bad state. But instead of having to painstakingly clean up the mess left by the failed installation, I simply re-imaged the machine in question with the bare-OS image and started over. A typical re-image containing only the Windows OS and a few test tools took less than three minutes. Using Frisbee, we could run smoke tests on about six builds per day; before Frisbee, we could run smoke tests on about three builds per week.

Of course, Ghost or other proprietary tools also image machines quickly under these circumstances—once you buy the tool, license the software, install it on an appropriate server, and configure it properly. I prefer Frisbee to Ghost because: Frisbee is marginally faster; Frisbee is very easy to install on FreeBSD; and Frisbee is very efficient. Adding a couple of small Perl scripts to the normal Frisbee distribution gave me an imaging environment tailored for the test lab. (The most complex of these scripts is available freely from Better Software magazine. Better Software published my interview with Frisbee's lead developer in March 2005: http://stickyminds.com/bettersoftware/magazine.asp?fn=cisnd&tsnid=401.)

I also used Frisbee to preserve the state of a machine after I had uncovered particularly complex defects. That is, if it took a large effort (many steps and/or a long duration of time) to demonstrate a defect, I could make an image of the machine at the point at which the defect was visible, and restore the image at will to demonstrate the defect to developers or managers.

FreeBSD Security Testing: Nessus

Whenever you have more than one entity on a network, and whenever you expose a server to the wider internet, security of the machine itself is always a concern. Nessus is an open-source remote vulnerability scanner for security/penetration testing that consistently is "rated among the top products of its type throughout the security industry", according to the Nessus website.

Nessus (running on a host machine, in this case a FreeBSD machine) probes a remote machine over the network for security vulnerabilities. It does a port scan, finds what ports are open, then investigates the software that has those ports open for a huge number of security risks, for all major OSs. It generates detailed reports in a number of formats that anyone can understand. The number of security probes available in the default installation of Nessus is very large, but sophisticated security/penetration testers take advantage of NASL, the Nessus Attack Scripting Language, a domain specific language that allows testers to craft their own attacks using Nessus' available features.

Of interest is that, while Nessus is a free download for UNIX-like systems (and is available in the ports collection of FreeBSD), it is available on Windows only as a commercial product from a company called Tenable. The Tenable product is "NeWT", "Nessus on Windows Technology".

FreeBSD Network Tools

FreeBSD is most widely used as a robust server platform. It follows, then, that tools related to network analysis and performance will be highly evolved on FreeBSD. What follows in this section is a brief description of network diagnostic tools that I found invaluable in testing in a networked environment.

FreeBSD Network Tools #1: ntop

From the name, one would assume that ntop emulates the functions of the UNIX "top" command, but for the network rather than for the local machine. Perhaps the first version did; currently, ntop is capable of providing detailed information about a huge number of hosts and their status and activities on the network.

For testing, two features I found very powerful: at a high level, ntop shows the amount of network traffic on the entire network segment minute-by-minute, hour-by-hour, and day-by-day in graphical format. Also, ntop shows information about recent connections between individual hosts on the network.

It is easy to see traffic trends on the network as they are occurring; also, if something anomalous appears, ntop records detailed information about network connections between hosts, including the ports over which the connection happened. This was critically important when analyzing software issues.

In particular, I would monitor the traffic graph minute-by-minute and hour-by-hour. If ntop showed a period of time for which traffic was particularly high, I would find out which host was generating the traffic. I would examine the software running on that host, over that port. Often it was a new build with a bug.

FreeBSD Network Tools #2: ettercap

Of all the tools mentioned in this paper, ettercap is the most morally troubling. It is a complex tool. Before I discuss how I used ettercap, I will define some terms:

IP address: usually displayed as four numbers separated by periods, for example 128.162.100.100, this is the address over which a host sends and receives network traffic.

MAC address: the unique identifier assigned to every piece of network interface hardware at the time of manufacture.

ARP: Address Resolution Protocol: the means by which a particular network assigns an IP address to a MAC address.

ARP poisoning: the practice of corrupting the local ARP tables in order to direct network traffic to a destination not intended by the sender.

Man In The Middle (MITM) attack: implementing ARP poisoning in order to eavesdrop on the traffic between two hosts in the network. MITM attacks implement ARP poisoning such that both network hosts send and receive messages normally, and are unaware that an attacker is in fact reading and forwarding the messages both ways.

Ettercap is a tool for ARP poisoning. It can decipher passwords on the fly, corrupt IP traffic, and do any number of other morally questionable things.

I used ettercap as a performance tool. In my test labs, all of my FreeBSD machines ran on discarded hardware, Pentium II processors. I found that when I used ettercap to sniff traffic between two hosts, the lack of processing power caused ettercap on the slow MITM machine to start dropping packets, making it look to the client machine in the SUT as if there was interference or other trouble on the network. And by varying the load on the FreeBSD machine, I could in fact control the number of packets being dropped: running ettercap and the UNIX "yes" utility caused 100% packet loss.

This is my most creative use of a FreeBSD too for testing. In a more straightforward application, any time a tester needs to eavesdrop on traffic between two hosts on a network, ettercap is an excellent choice because of its power and ease of use.

FreeBSD Network Tools #3: Perl

Perl gets a special mention here because Perl's network utilities are outstandingly good compared to other languages. Perl Net::* modules and IO::Socket::* modules and other features for network management are robust and powerful. But they often fail to compile on Windows. It is the ease of use of Perl's network utilities on FreeBSD that gets Perl the mention in this section.

I used Perl's network utilities to impersonate network clients and servers for test purposes. On one occasion, I was required to test software that was a client to an interface on the New York Stock Exchange. Unfortunately, the NYSE test server was down about nine days out of ten. I wrote a little network server in Perl to emulate simple functions of the NYSE server in order to test the client software.

On another occasion, I had to test some functions on a web server. It was not difficult to write a simple Perl HTTP client to validate that the server was functioning properly.

I also used Perl to validate the output from a server sending to a multicast address. I wrote a simple Perl multicast client on FreeBSD to monitor the traffic on a multicast address. Lincoln Stein's excellent IO::Socket::Multicast module made it easy. (Note: I never got IO::Socket::Multicast to compile on Windows. I tried it on Windows 2000 and on Windows XP.)

FreeSBIE

I hope that this discussion of testing with some of the tools available on FreeBSD has piqued your interest. If you would like to explore a FreeBSD system without committing yourself (or your hardware) to a full installation, there is an excellent version of FreeBSD available in the form of a bootable CD, very similar to the Knoppix Linux distribution. The system is called FreeSBIE (Free System Burned In Economy) and is available from freesbie.org. While not all of the FreeBSD ports are available on FreeSBIE, it is still an excellent way to get a feel for the system. Some of the tools I discuss in this paper are available in the FreeSBIE distribution, but not all of them: nessus, for instance, is available in version 1.0 of FreeSBIE, but not in version 1.1.

Conclusion.

I used tools from the FreeBSD ports collection in four areas: in the network, where the operating system has very little impact on how software behaves; for remote security testing and performance testing where I can manipulate remote machines over the network regardless of the operating system; for disk imaging of Windows, Linux, and FreeBSD machines; and on the webserver, where Twiki was my collaboration tool of choice.

Because the installation procedure for all of these tools is standard, I spent relatively little time installing and configuring my tools. Because all the tools were hosted on a single platform, I had all of my configuration and diagnostic information located in just a few places. I kept all of my potentially dangerous security tools on a single machine, which made my presence on the network tolerable to the company's network management staff. And the compatibility between FreeBSD versions made it fairly simple to upgrade and to manage multiple FreeBSD machines. And of course, I could rely on the correctness of my test results, because the system itself is so reliable.

I have tried using Linux in a similar way, but my experience is that package management quickly becomes tedious if not overwhelming. The FreeBSD ports collection handled that for me. And many of these tools are simply not available on Microsoft Windows. And when they (or their equivalents) are available, their cost, both financial and in terms of overhead, was simply too high.

But FreeBSD's simple installation procedures and robust ports collection makes it easy to experiment with the huge number of tools available. I often found myself browsing the ports collection looking for interesting applications to install, just to see how they worked. I found ettercap by browsing the ports collection, a tool that became very useful very quickly. It became clear that the more tools I used on FreeBSD, the more economical became the management of those tools.

The next time you need to reach into your toolbox for some sophisticated, reliable, and powerful testing tools, I hope you find them in FreeBSD.

Combining Traditional and Open Source Testing Processes for NFS Version 4

Bryce W. Harrington
bryce@osdl.org
Test and Performance Department
Open Source Development Labs

Abstract

As use of open source technology expands into the enterprise, a challenge emerges for the end user: the testing of these applications is often not as rigorous and comprehensive as is typically the case for proprietary applications. Yet despite this, many open source applications have gained a reputation of quality and robustness, using an evolutionary "release early, release often" approach.

The open source process has important implications for achieving quality, yet it is very different from traditional formal testing. This raises an interesting question: could traditional testing and open source testing be conducted in a way that taps the strengths of both and avoids the weaknesses of either, and thus better address the challenge faced by end users?

This paper presents a case study of fusing traditional testing in the Linux NFS version 4 open source community. The development and testing efforts of a number of different companies and universities are coordinated using a typical open source processes, enabling the sharing of results and the distribution of test development, testing, and analysis work. A test matrix is used to break out the tasks, identify owners, and track progress. Existing open source tests such as Connectathon, PyNFS, and Iozone are identified, enhanced for our purposes, and standardized across the community.

NFS is an interesting case study, as it is a critical technology for many businesses and has very high expectations from end users in terms of functionality, robustness, interoperability, performance, and security. Yet the Linux kernel and its NFS stack is not owned or controlled by any single company; thus no single entity has responsibility and accountability for conducting formalized testing. While for some applications, an evolutionary approach to quality may work well; for NFS the level of tolerance for issues will be much narrower.

This provides an excellent opportunity for testing to play a central role in enabling the adoption of the technology by a) identifying issues early in development so they are fixed quickly, b) providing data to potential adopters about the risk/reward they can expect to gain, and c) producing information and tools to help end users identify, report, and overcome problems they run into.

Based on our experiences, several technical, sociological, and organizational ideas for improving the overall state of testing in open source are outlined. As open source becomes increasingly more critical in the enterprise, small steps to improve testing processes today could have large benefits for the future.

About the Author

Bryce is a Senior Performance Engineer at the Open Source Development Labs (OSDL) in Beaverton, Oregon. He graduated with a BSc in Aerospace Engineering from the University of Southern California and an MSc in Aeronautical Engineering from the California Institute of Technology in 1995. Currently Bryce is leading OSDL's NFSv4 testing project.

INTRODUCTION

There is no single best way to test a piece of software, and testers have learned, combining multiple methodologies can result in much better coverage than using a single methodology alone. In this paper we treat the open source development process as a new testing methodology and explore how it can be combined effectively with more traditional methods like regression and performance testing.

The idea that the Free/Open Source Software (FOSS) community could produce higher quality products through use of formalized testing should be no surprise; the challenge is in gaining the community's adoption and appreciation of it.

The Open Source Development Labs (OSDL) became interested in NFSv4 due to interest in it by enterprises using Linux in their data centers. We found that a number of organizations were already involved in testing NFSv4, and that the desired role for OSDL was to facilitate, organize, and improve the quality assurance processes. OSDL exists to find ways to help enterprises and the open source community work together to address common needs. Previous efforts have included stabilization of the 2.5 Linux kernel, publication of capabilities needed by the carrier-grade industry, and facilitation of technical discussions between vendors, customers, and the community.

BACKGROUND

Network File System (NFS)

NFS [1] is a UNIX protocol for large scale client/server file sharing. It was originally developed by Sun Microsystems, and has become a de facto standard on all UNIX platforms via RFCs 1094, 1813, and 3530. It is analogous to the Server Message Block (SMB) and Common Internet File System (CIFS) protocols on Microsoft Windows, however it maps better to the UNIX Virtual File System (VFS) semantics, has an easy to understand protocol, and is relatively simple to implement on UNIX-based systems.

Version 1 was first introduced in 1984 but is not typically used on Linux. Versions 2 and 3 of the protocol were 'stateless', meaning the server did not keep track of which clients were using which files. Locking and caching were implemented externally to the protocol, and no specific provisions were included for security. Version 3 marked a shift to using TCP as a transport; the previous versions had used UDP exclusively, which had proved difficult to use on a wide area network (WAN).

The need for state tracking and security caused many users to shift to the Andrew File System (AFS), developed at Carnegie Mellon University (and named after Andrew Carnegie and Andrew Mellon), as a more capable alternative. AFS provided better security, performance and scalability than NFS, and has thus been adopted by a number of organizations for large scale file sharing.[2]

Influenced strongly by AFS, a new version of the NFS protocol was introduced with the publication of RFC 3010 in 2000, and RFC 3530 in 2003.

This development work has been undertaken by a number of companies, with most of the developers working as part of the Center for Information Technology Integration (CITI) at the University of Michigan [3]. However, development alone will not be sufficient to drive adoption of NFSv4. Many of the key usage models for NFSv4 involve large numbers of machines; thus even a short duration downtime can lead to extensive costs in lost productivity. Because of this, testing will be critical, both from the standpoint of eliminating defects and to give potential adopters a confidence about its capabilities.

Need for Testing

OSDL became aware of the need for testing of NFSv4 through formal and informal discussions of testing priorities with its member companies in the latter half of 2004. It was remarkable that while NFSv4 testing was not the number one priority for very many companies, it appeared to be somewhere on everyone's top ten list. OSDL was looking for testing projects that would benefit as many companies as possible, that were in need of formal testing similar to what OSDL had done previously, and that were not already getting sufficient attention from testers. NFSv4 fit the bill.

Specifically, OSDL's goals were:

1. Ensure that the testing of NFSv4 is effective at improving the quality of NFSv4
2. Open visibility of the testing so everyone can see the current status
3. Share the effort of conducting the testing to many companies

4. Help new testers join the testing effort
5. Participate in the testing efforts by gathering, writing, and running of tests

Testing Objectives for NFSv4

Testing is an extremely broad topic, covering standards conformance, validation of new features, performance measurements, code audits, and so on. The sheer number of ways for testing the software is overwhelming. Thus for organizational sanity OSDL, several member companies, and the NFSv4 community started by identifying the high level goals for the testing efforts to achieve.

The first goal is to assure the Linux NFS version 4 implementation be as complete and correct as any proprietary NFSv4 implementation. This helps the developers gage their progress, identify regressions, and notice where their implementation work needs further attention.

The second goal is to showcase the new NFSv4 features and benefits. We expect to be able to show significant improvements in performance and security compared to NFSv3, and want to make measurements of these changes to help prove the benefits of NFSv4 adoption. Doing this will also help identify best practices and provide users with example implementations, and help developers in maximizing the magnitude of these benefits.

The third goal is to give assurance to end-users that NFSv4 will not impose new limitations compared to what they are currently using. We want new adopters' initial experiences with NFSv4 to be satisfactory, and to meet their expectations of its quality.

Business Interest in NFSv4 Testing for Linux

The next step was to identify the reasons that other companies were interested in NFSv4 testing. The purpose of this analysis was to validate the goals and to help find rationales for convincing additional companies to participate. [4]

- *Technology Consolidation* – Shift customers from AFS and other file systems to NFS to provide cost savings from simplification of migration paths and reduction in supported technologies.
- *Promote Technology Adoption* – By ensuring Linux has good NFS support, it enhances or enables other products and services the company provides.
- *Strategic Investment* – The company relies directly on Linux and wishes to ensure its NFSv4 support is on par with commercial implementations for its long term needs.
- *Internal Adoption* – The company intends to implement NFSv4 in their environment to replace NFSv3 and wish to verify it meets their immediate needs.

The companies contributed in different ways: Direct funding, equipment, tests, testers, use cases, defect reports, and documentation.

THE OPEN SOURCE TESTING METHODOLOGY

The notion of open source as a testing methodology was perhaps most famously expressed in Eric Raymond's essay "The Cathedral and the Bazaar" with the famed adage attributed to Linus Torvalds, "Given enough eyeballs, all bugs are shallow". [5] While not every project follows the approach Raymond outlines in his paper, it has served as a useful model for characterizing how open source projects can achieve high levels of quality without relying on traditional testing methods.

The Process in a Nutshell

A more technical description of the methodology might characterize it as an evolutionary development approach that heavily leverages its user-base to form an integrated feedback cycle. It blurs the distinctions between users, testers, and developers, and strives to involve everyone as co-developers. It is a highly organic process, informal and rarely

organized, yet tends to be able to converge on a stable product with noteworthy speed. It is good at identifying robustness, interoperability, security and portability problems, and weaker at performance and conformance testing.

The open source model is often contrasted with the more linear and compartmentalized "waterfall model," such as in Figure 1. The waterfall model is not a completely accurate model of the way software is developed today due to the proliferation of alternative development methodologies, yet due to its production-line mentality it fits with expectations of how the process "should" work, and thus from a conceptual standpoint it serves useful for distinguishing how open source is different. [6]

The testing process for open source projects typically begins with a public release of the code, often coupled with an announcement of some form to a mailing list, blog, and/or software registry. The initial release of the code is often very incomplete, but it serves a particular need. Users with an interest in that software download it and try it out, and report back on errors they encounter or ideas for features they'd like to see added. Developers work on whichever of these reports seem most interesting or necessary to them. [7]

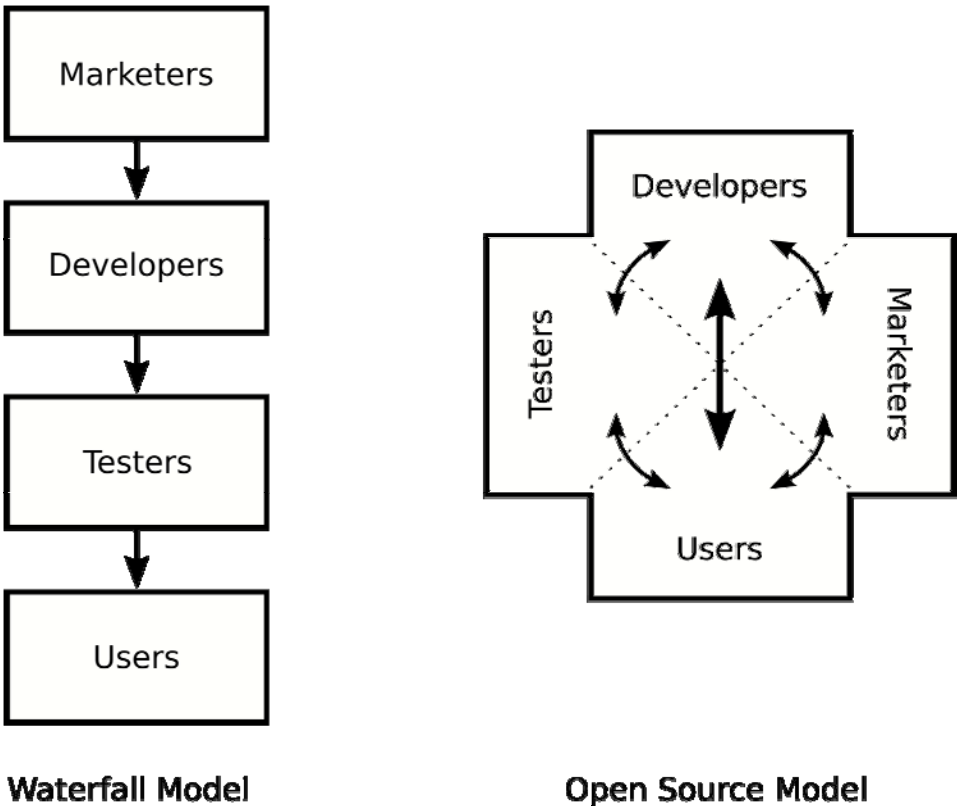


Figure 1: Comparison of traditional "waterfall" model with open source

Users also get involved by writing patches for the code, fixing defects or adding features that they care about. These changes are submitted to the core developers for incorporation. The patches go through some sort of review process, which varies from project to project (some apply all patches without question, others have stringent quality requirements).

Marketing plays a very different role in open source than in traditional development processes. Traditionally, marketing plays both an internal role for defining requirements and creating specifications, and an external role for promotion of the product to support sales. In a typical open source project, marketing tends to play a reduced role, confined to external promotion of the project's work. [8]

Evidence of the Successes of Open Source Processes

That this simple process can result in high quality code seems counter-intuitive. Organization in these projects often seems haphazard. Many projects do not have delivery roadmaps, task assignments, or test plans. Bugs seem to get fixed unpredictably, seemingly important features languish forever unimplemented, and designs are often left undocumented.

Despite this, the evidence is strong that open source processes can produce some of the highest quality applications in the industry. David Wheeler's paper "Why Open Source Software / Free Software" [9] summarizes published information about the reliability, performance, scalability, and security of open source applications as compared with other alternatives.

Studies of the robustness of several major open source projects have demonstrated significant advantages of the open source program over closed source alternatives.

Studies have shown the GNU/Linux OS has fewer flaws, fewer critical system failures, and fewer crashes ([10], [11], [12], [13]). Coverty [14] conducted a four-year research effort comparing the Linux kernel against the industry average. They reported 985 defects in the 5.7 million lines of code, and compared this with data from Carnegie Mellon University that would predict 5,000 defects in a similar sized proprietary program.

A Swiss evaluation of website uptime by Syscontrol AG [15] showed that the average down-time of the Apache web server as 3.23 hours per month, compared with 11.03 for Microsoft IIS, 3.35 for Netscape, and 9.21 for other servers. A Netcraft uptime study [16] in 2001 reported that of the 50 sites with the highest uptime ratings, 92% were running Apache, and half were run on open source operating systems.

A code analysis [17] published in the "Communications of the ACM" reviewed 6 million lines of code for a number of programs over time. They reached the conclusion that open source "code quality appears to be at least equal and sometimes better than the quality of [closed source software] code implementing the same functionality."

Beyond robustness, open source applications such as Linux, Samba, MySQL have also repeatedly been found to be equal or higher performance than their proprietary counterparts, according to studies by PC Magazine [18] [19], eWeek [20] and more.

Open source also scores well on security. Informal studies show that Linux vendors have faster response rates to security issues than others [21], that GNU/Linux systems are relatively immune from outsider attacks [22], that Apache has much fewer security vulnerabilities than Microsoft's IIS [23], and that viruses are much more common on Windows than Linux [24]. It is sometimes claimed that the higher virus numbers for commercial products is due to their larger installation base, however according to Netcraft's August market share survey across all domains, Apache commands a 60.5% marketshare, compared with 20.4% for Microsoft IIS [25].

Clearly, there is mounting evidence that open source development methods are able to produce code that scores well on key quality metrics. For this reason, these methods are worth exploring so that the characteristics or mechanisms that make them work so well may be more generally applicable.

Open Source Processes Alone Do Not Guarantee Quality

Despite the growing evidence showing that programs developed using open source methodologies can be very high quality, there are of course, no guarantees.

Indeed, it is relatively easy to find unusable, buggy, and ill-performing open source programs. In some cases this can be explained as due to the programs being early in their life-cycle. In other cases, lack of proper adherence to open source processes may be a cause. In still other cases the cause can be attributed to lack of sufficiently large user-bases. Yet even for programs that have active development communities, issues that would not be found in commercial software can be easily found.

There are several reasons why the open source methodology does not ensure better code more often. To name a few:

1. *Non-user facing defects.* Defects that are not known will not get attention. Since developers often rely on users to report defects as they find them, defects that don't affect users directly may remain hidden until new features

reveals them. Unfortunately, time passes between the cause of the defect and its discovery, and the knowledgeable individuals are no longer available. If these latent defects are revealed earlier in development, they would stand a better chance of getting fixed swiftly and provide savings later on.

2. *Tolerable defects.* Testers and users take different mind-sets to a program. Testers are actively looking for issues and motivated to report even minor quirks. Users, on the other hand, desire to see the program work correctly, thus may overlook minor discrepancies that none-the-less are legitimate issues. The accumulation of these minor issues can give a program an appearance of being poorly tested, when in reality its core capabilities may be best of breed.

3. *Local optima.* In open source, development is often organized around the individual program level. Great attention will be placed at making a particular library, application, or server high quality, easy to use, and powerful, yet each program may be implemented in incompatible ways. Different interface standards may be employed. They may use different configuration approaches. They may have notably different look and feel designs. Viewed at a higher level, while each program may be great in and of itself, the inconsistencies from one program to the next can result in serious issues when using the system as a whole. Bugs seem to prosper in the cracks between integrated components.

4. *Slow convergence.* Programs like Apache, Linux, Samba, Sendmail, Bind, and so forth have been the result of years of attention. In some cases, proprietary alternatives will more swiftly meet customer needs in the short term, and users may expect open source projects to add and perfect features more quickly than is seen in practice. While the particular open source application may be of high quality eventually, early in its life-cycle it may not be able to fully meet all of the user's needs.

5. *Defect propagation.* A unique issue with having source code easily available for examination and/or re-use is that a defect in a piece of code that is highly copied or reused will get propagated into a number of other open source programs.

While they can present serious issues in the short term, in the long term projects find ways to work around or avoid the issues, especially when ample resources are made available and open source methods are mixed with more traditional methods.

Fostering the Open Source Methodology

The practices in successful open source projects often seem very natural and low-key, and can lead to the misconception that the working style arises spontaneously and effortlessly. Simply making code available under open source terms is not sufficient to guarantee the project's success within the open source model. In fact, achieving viability with open source requires equal parts technical quality, sociology, timing, and luck.

The magnitude of the challenge is evident from a cursory review of the SourceForge open source project registry. As of June 2005 there are over 100,000 registered projects, operated by over a million open source developers. Only a portion of the total number of projects register through SourceForge, so the actual number of open source projects is well above 100,000. Yet the number of packages available for a typical Linux distribution is much lower. Red Hat Enterprise Linux, a corporate-oriented distribution, includes approximately 1500 packages [26]. Debian Linux, a community-oriented distribution, includes about 15,500 packages [27]. Freshmeat, a registry of released open source software, lists about 38,000 projects that have produced at least one release [28]. It is not uncommon for one project to produce multiple packages, so the package number gives a very conservative estimate for the number of viable open source projects. These numbers would suggest a rough rule of thumb that of 100 open source projects started, no more than about 10 will succeed at producing a package that is included in a Linux distro, and only around 1 will be of sufficient interest for inclusion in an enterprise distribution. Clearly, successful open source projects are the exception, not the rule.

Even among those projects that have been accepted into a Linux distro, not all would be considered as high of quality as projects like Apache or the Linux kernel. The interesting question to ask is what processes, practices, and characteristics are enabling the low-quality projects to grow into high-quality ones. In particular, what testing activities are they employing to help foster their successes?

A position paper from Carnegie Mellon University [29] outlined a study examining the publicly visible portions of successful open source projects from November 2001 to March 2002. It noted several several major characteristics that the projects shared:

1. The life-cycle of the projects studied were all in the software maintenance phase. The projects had benefited from having a good initial design and an architectural foresight to enable modular, incremental growth to be achieved without heavy risk of introducing major quality setbacks.

2. All projects used nightly builds. This ensures that the changes made to the codebase do not break the build system, as well as providing a downloadable binary for users to test. There are also tools such as Tinderbox that assist in identifying the precise change(s) that caused the breakage.

3. Publicly visible bug tracking is performed by nearly all of the projects examined. A bug tracking tool enables users to post issues and enhancement requests, and opens a dialog with the developers and other users with similar interests. The tracker also provides a vector for users to become directly involved in development, by performing traces and tests on their own system, or even creating and sharing fixes on their own. The trackers also fulfilled a project management role by allowing prioritizing defects or flagging issues for closure prior to a particular planned release.

As will be explored below, for NFSv4 the first characteristic has been followed for quite some time, the second has been recently adopted, and the third is in the process of being adopted.

The Carnegie Mellon researchers also modeled open source projects, examining the input/output flow in a "Free-Body Diagram" style. The leaders of the projects control the engineering practices inside the boundary of their project, and establish practices that simultaneously maximize the outgoing flow of information to the community while strictly limiting and controlling the flow of incoming information. This allows the project to select critical aspects of software best practice felt to be of relevance to the project, without overly burdening the external developer. They add, "Any quality-related intervention must respect this overall approach." [29]

Another aspect they noted of quality management in open source projects was the aggressive use of software tools to manage it. The leaders strive to shift policy enforcement from people to tools. This not only frees up the leaders to focus on more important development activities, but also ensures the rules are applied consistency and that they will remain in place even if the project participants change or if activity levels wax and wane. It also provides an effective way to mediate between participants who may be widely separated both in geography and time.

Based on their analysis, the authors recommend that the following criteria are required of any quality-related technology or process improvements introduced to a project:

1. Incremental model for quality investment and payoff

2. Incremental adoption of tools and methods

3. Allows for untrusted input from anywhere, but produces a trusted version of the code for universal use

4. Tool interaction style is easily adopted by practicing open source developers.

An effective way to institute better Quality Assurance practices in an open source project is to go slow, changing the processes incrementally little by little and avoid processes or tools that lay too far outside what the community is already using.

IMPROVING TESTING METHODOLOGIES FOR NFSV4

With NFSv4, the desire is to leverage the full strengths of the open source methodology, and to couple it with some of the more formalized methods of traditional testing. This objective was attacked from a number of angles.

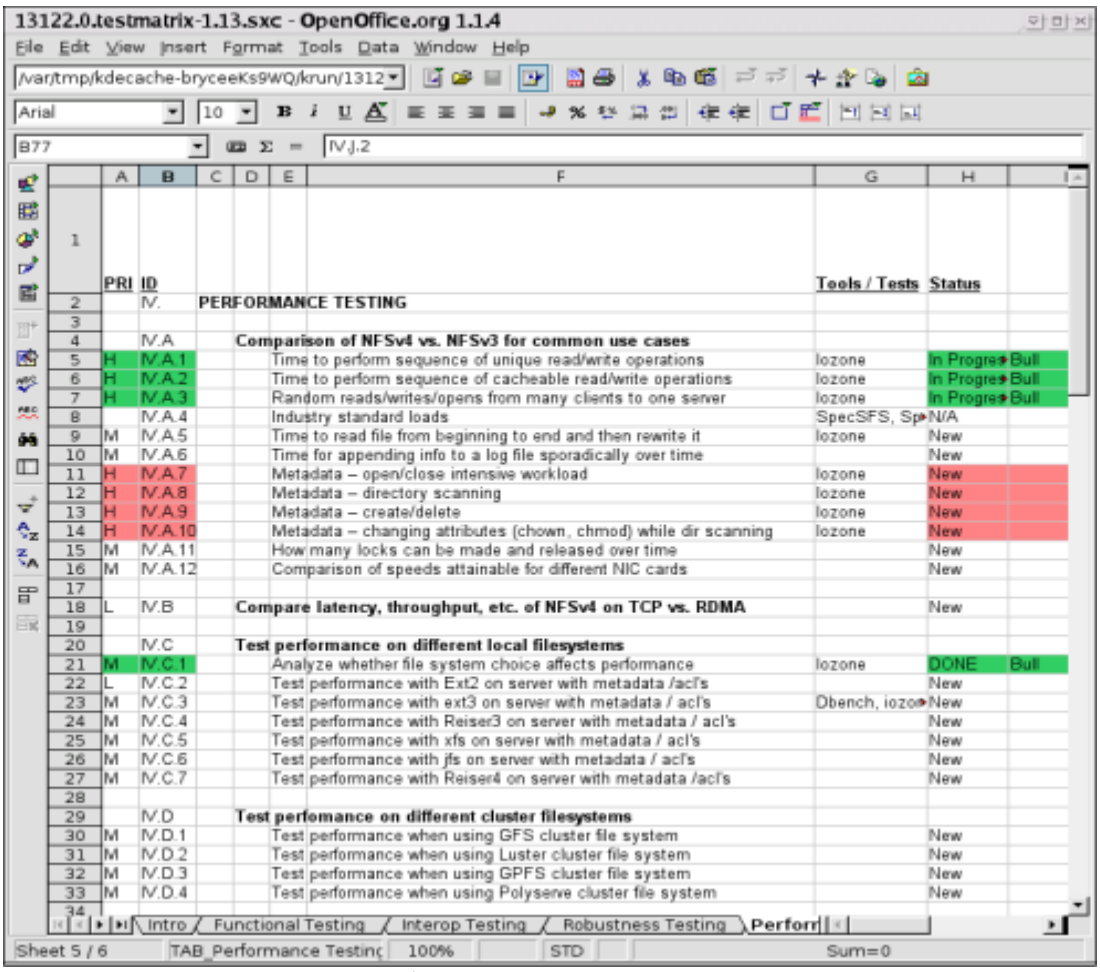
Test Matrix

When OSDL first took interest in assisting with NFSv4, our first question was what form our assistance should take. To answer this, we first analyzed the existing testing efforts and the quality questions being posed by the users and developers.

In formal testing, it is traditional to lay out a test plan detailing all of the items to test. However, as described in the previous section, open source projects rarely follow such formal processes. Thus we wondered if it was possible to find a comfortable middle ground, that would give companies the confidence that the testing was well planned, yet not impose procedural hurdles that would fail to work in practice.

We also reviewed existing presentations, documents, and test plans that addressed testing needs. Within a month, we found ourselves with a comprehensive yet overwhelming list of items to test.

The community needed a way to collect and organize these disparate ideas and plans to communicate testing needs coherently. Early on, Mary Edie Meredith of OSDL, Data Center Linux (DCL) Initiative Manager, suggested the notion of a test matrix, to correlate test items with test programs and reference testing resources and staff. The form this document took was a listing of testing items in a spreadsheet that resembled a work breakdown structure (WBS). Like the typical WBS, the NFSv4 test matrix organized testing tasks into a numbered hierarchy, as seen in Figure 2.



	PRI	ID		Tools / Tests	Status	
2		IV.	**PERFORMANCE TESTING**			
3						
4		IV.A	**Comparison of NFSv4 vs. NFSv3 for common use cases**			
5	H	IV.A.1	Time to perform sequence of unique read/write operations	Iozone	In Progre	Bull
6	H	IV.A.2	Time to perform sequence of cacheable read/write operations	Iozone	In Progre	Bull
7	H	IV.A.3	Random reads/writes/opens from many clients to one server	Iozone	In Progre	Bull
8		IV.A.4	Industry standard loads	SpecSFS, Sp	N/A	
9	M	IV.A.5	Time to read file from beginning to end and then rewrite it	Iozone	New	
10	M	IV.A.6	Time for appending info to a log file sporadically over time		New	
11	H	IV.A.7	Metadata – open/close intensive workload	Iozone	New	
12	H	IV.A.8	Metadata – directory scanning	Iozone	New	
13	H	IV.A.9	Metadata – create/delete	Iozone	New	
14	H	IV.A.10	Metadata – changing attributes (chown, chmod) while dir scanning	Iozone	New	
15	M	IV.A.11	How many locks can be made and released over time		New	
16	M	IV.A.12	Comparison of speeds attainable for different NIC cards		New	
17						
18	L	IV.B	**Compare latency, throughput, etc. of NFSv4 on TCP vs. RDMA**		New	
19						
20		IV.C	**Test performance on different local filesystems**			
21	M	IV.C.1	Analyze whether file system choice affects performance	Iozone	DONE	Bull
22	L	IV.C.2	Test performance with Ext2 on server with metadata /acl's		New	
23	M	IV.C.3	Test performance with ext3 on server with metadata / acl's	Dbench, iozo	New	
24	M	IV.C.4	Test performance with Reiser3 on server with metadata / acl's		New	
25	M	IV.C.5	Test performance with xfs on server with metadata / acl's		New	
26	M	IV.C.6	Test performance with jfs on server with metadata / acl's		New	
27	M	IV.C.7	Test performance with Reiser4 on server with metadata /acl's		New	
28						
29		IV.D	**Test perfomance on different cluster filesystems**			
30	M	IV.D.1	Test performance when using GFS cluster file system		New	
31	M	IV.D.2	Test performance when using Luster cluster file system		New	
32	M	IV.D.3	Test performance when using GPFS cluster file system		New	
33	M	IV.D.4	Test performance when using Polyserve cluster file system		New	
34						

Figure 2: NFSv4 test matrix

The testing task ideas were divided into five areas, with objectives defined as follows:

Functional Testing - Ability to do what it's supposed to do. Standards compliance, regression, compatibility, static code analysis, etc.

Interoperability Testing - Ability to work with other versions of NFS, other operating systems and other software/filesystems/etc. Generally associated with NFS.

Robustness Testing - Remains stable and recovers even in extreme situations. Stability, error recovery, race conditions, etc.

Performance Testing - Able to perform well under real and theoretical workloads. Load, stress, destruction, scalability, etc.

Security Testing - Resistant to being compromised and difficult to attack.

Within each category we identified particular types or aspects of testing. For instance, in robustness testing we identified the following aspects: basic stability assessments, resource limit testing, stress load testing, scalability (robustness), recovery from problems while under light/normal/heavy loads, race conditions, and automounter robustness.

In each of these sections, the testers and developers then itemized the specific tasks that were felt to adequately cover the need. For example, with resource limit testing we identified the various resources that can be limited, such as memory, disk space, pid, inode, and swap space, and created tasks to perform a test of each situation.

Prioritization was then performed to highlight the specific testing tasks that were felt to be most important, versus ones that were less necessary or that could be postponed for a time. To establish the priorities, considerations were made for whether the given test item helped us towards achieving one of the community's objectives, as listed above.

It was important for the viability of the effort that the priorities represent a consensus between users, developers, and testers. Initially, we attempted to gain this through email discussions, but found that interest was low. We shifted to hosting weekly conferencecalls and working through the matrix. Over the course of three months we worked our way through the test matrix, prioritizing and fleshing each section out. This work was completed in May 2005 and is available online at http://developer.osdl.org/dev/nfsv4/testmatrix/.

In a number of cases the team found that community members were already working on testing tasks. Tracking this existing activity in the test matrix helped other testers avoid duplicating efforts. This correlation also helped to identify gaps where specific forms of testing were needed, but where the existing tests lacked the necessary coverage. This information has proven especially interesting for the test authors, giving clear direction about what to add to tests, and why.

With the help of this prioritized matrix, we have been able to group the items into sequential milestones towards achieving enterprise-readiness for NFSv4. These activities include:

- Detailing steps needed for performing the testing tasks
- Writing tests to support the testing tasks
- Setting up test cases using the tests
- Performing testing tasks
- Analyzing and reporting on results

Using a spreadsheet format was convenient for printing and for making changes to many items at once, but it was found to not be an ideal match to the community's needs. The main issue was that it was cumbersome for more than one person to update at the same time; this was particularly problematic as testers needed to update their status. It was also difficult to hyperlink items to their tests, results, or commentary.

For these reasons, the test matrix was converted from a spreadsheet to a wiki. The community members were already comfortable using wikis from other projects for collaboration. This has improved the ability to share access

to the data, hyperlink to additional materials, and give the NFSv4 community ownership over it. There are some minor remaining issues but it is expected that these can be resolved in time.

Synthetic Testing

Groupe Bull, an IT solutions company headquartered in France and emphasizing a focus on open environments, has been contracted to perform dedicated testing of NFSv4 on Linux. Their efforts have included focused testing on a particular characteristic of the system, and has worked with the development community to fix a number of issues that their testing uncovered.

For example, they set up a test to place as many (zero-byte) files into one directory as possible [30]. They tracked the time needed to create 100 more files in the directory. As they conducted these tests, they would find and report issues that caused NFSv4 to take excessive amounts of time to create the files. The developers took interest in these issues and quickly resolved them, enabling the testing to continue to new levels. At last count, they had achieved 1.6 million files in one directory. This level is well beyond what a typical user would expect to work, and thus would be unlikely to be reported as a problem until perhaps late in development. By conducting this artificial, synthetic testing early in the development process it identifies and eliminates defects well before any users would encounter them, thereby ensuring a better experience by future users.

The test matrix has been of particular use for Bull. It helps them to scope out and prioritize their own work and it ensures their efforts address the primary needs of the users and developers.

Bull follows a traditional testing methodology, writing test plans, conducting the tests, and creating reports from their results. However, these efforts have been adapted to fit within the open source model used by the NFSv4 project. They report their findings on the mailing lists, and work directly with developers to close the issues.

These efforts are also proving to be the most effective way at accomplishing the tasks set forth in the NFSv4 test matrix. It is hoped that additional companies can become involved in ways similar to Bull, in order to make good progress through the testing tasks.

Automated Build Testing of NFSv4 Patches

The NFSv4 development community uses a patch-oriented development process similar to the Linux kernel. Code changes are published in the form of 'diffs' from a baseline kernel release. About once every week or two, the team produces a combined 'patch set' consisting of all of the individual patches in one package. Periodically, the Linux kernel maintainers integrate some or all of the patches into the kernel, allowing the NFSv4 developers to simplify the patch set.

One of the consequences of this approach is that nightly builds cannot be done, as is common in other open source projects. Despite this, it is possible to do per-patch-set builds, which tends to be equally effective at identifying and highlighting compilation issues.

Thus, an early effort was placed into adding the NFSv4 client and server patches for the Linux kernel to OSDL's Patch Lifecycle Manager (PLM). PLM is a system that watches certain web or ftp sites for new kernel patches, downloads them, and runs various cross-compile tests against them [31]. A cross-compiler compiles code for a different architectures than the compiler is running on; often, when a developer is coding for a particular architecture, errors will be introduced that are not seen by that architecture's compiler. Thus, by compiling for a variety of platforms, these sorts of issues are flagged as warnings or errors, and are noted in the compiler output.

For NFSv4, we performed these cross-compile tests on both the base kernel, and the kernel with the new NFSv4 patch applied, and compared the compiler output using diff. This revealed all warnings and errors that the NFSv4 patch would have added to the kernel code.

By collecting this information for each patch, and reporting it to the development mailing list, the issues received attention quickly, and within a month we found that the developers were able to eliminate all warnings and errors on all platforms we were testing. Now, it is rare for a warning or error to last more than a few patch releases before it gets resolved.

Automated Regression Testing

OSDL has a depth of experience with development of automated testing harnesses for the main Linux kernel [32], and thus an obvious contribution was to perform automated regression testing of NFSv4.

The challenges for automating NFS testing compared with regular kernel testing were a) that as a network protocol it required a client/server arrangement rather than running tests on one system under test (SUT), b) that it involved testing of code beyond the kernel, including authentication libraries and network utilities, and c) that it required the ability to vary network conditions. This required creating a new harness able to do network testing, but elements from previous automation harnesses were reused.

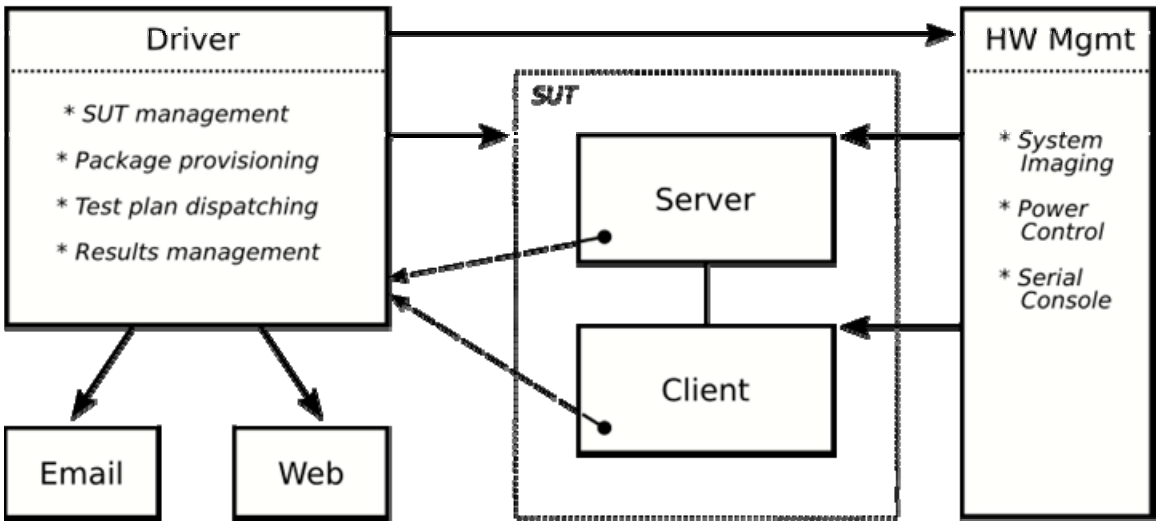


Figure 3: NFSv4 regression test harness

The automated testing system's architecture, as illustrated in Figure 3, involves five different types of machines:

1. SUT Client - This machine acts as an NFS client for testing purposes. In general it is configured with the same software complement as the server, but does not run the NFS daemons.

2. SUT Server - This machine acts as the NFS server, running the rpc.gssd, rpc.svcgssd, rpc.idmapd, rpc.mountd, and rpc.nfsd daemons. The server currently only has one physical client connection, however in theory it could be extended to multiple client and/or multiple server machines if needed.

3. Test Driver - This machine is used to direct the SUTs and oversee the running of the test processes. It also provides an NFSv3 mount point for the SUTs to store their logs to and to share software packages and test scripts.

4. Hardware Manager - This machine provides serial console logging, remote power control, remote imaging, and other hardware-level features. It is kept separate from the Test Driver for security purposes, since this machine also provides these services to other projects in the Lab.

5. Web/Email Server - This machine sits on the OSDL extranet and provides world-readable access to test reports, test logs, tests, and other materials. It also hosts a bug tracker and a wiki.

The NFSv3 mount provided by the Test Driver deserves additional discussion. This mount is not part of the testing process, but rather is used as the primary means of data communication between the Test Driver and the SUTs. The advantage of this design is that it keeps the Driver-SUT interaction simple and straightforward. Since test logs are written directly to the mounted file system, they can be watched remotely even if the SUT is running under heavy load. This also ensures that the SUT gets into a failure state, the logs up to the failure will be captured, and does not depend on the machines continued functionality after its failed.

An obvious risk is that if an issue is introduced with the NFSv4 code that affects NFSv3 behavior, this could introduce a failure into the testing process itself. However, from a testing perspective, this would be a good thing since our goal is to uncover such problems and fits the 'scratching your own itch' aphorism. In practice we have not seen this as a problem, but it forces the tester to give extra consideration to the NFS overhead, performance, and reliability.

The automated testing system is currently used only for performing regression testing. Four test suites are currently being run: Connectathon, PyNFS, LTP, and Iozone. The Connectathon test suite [33] is a standard NFSv3 test developed by Sun for their annual integration testing event held in San Jose. PyNFS [34] is an test suite supporting NFSv4-specific features maintained by the Center for Information Technology Integration (CITI) at the University of Michigan. The Linux Test Project (LTP) [35] is a large test suite collecting many different regression test cases for Linux; for the NFSv4 testing only the NFS-specific tests are executed. Iozone [36] is a commonly used file system stress test. Other tests are planned to be added to the testing framework in the future.

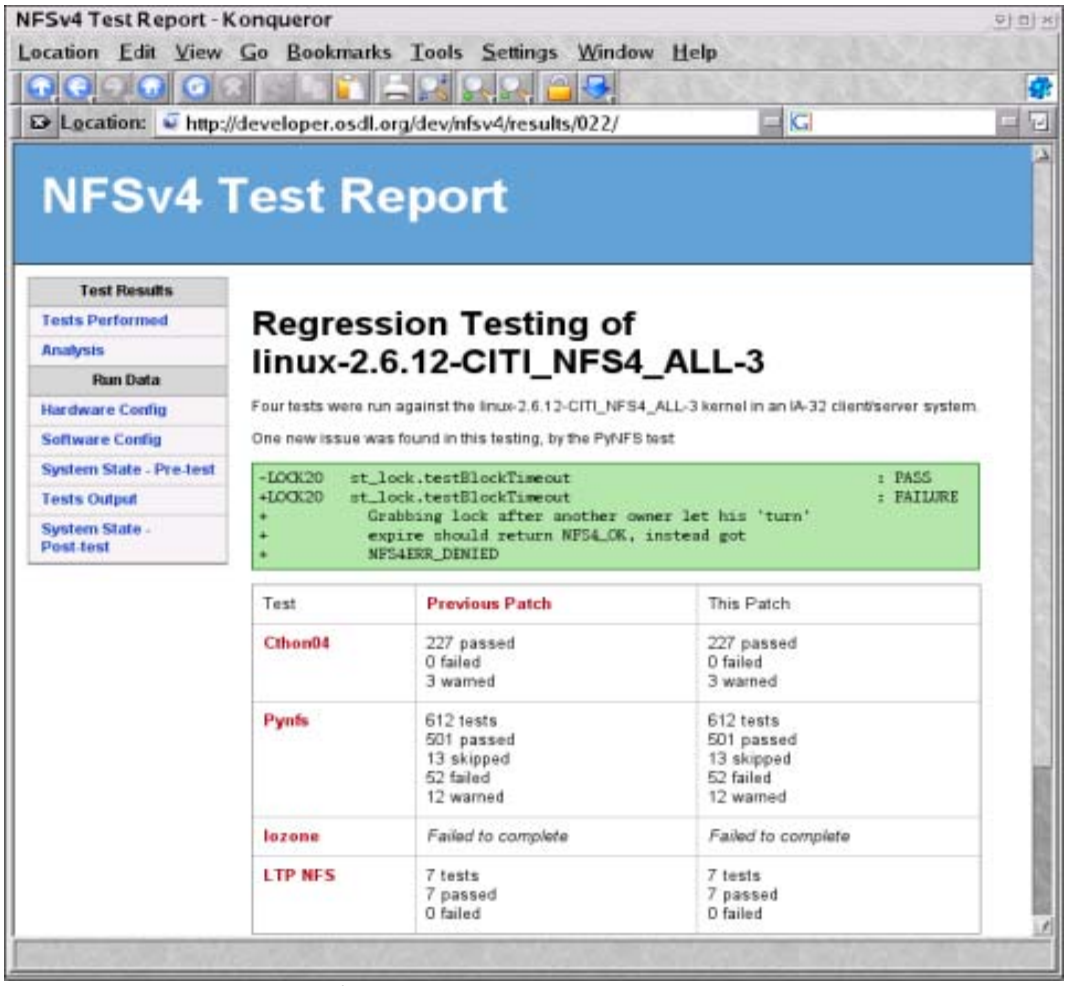


Figure 4: NFSv4 regression test report

The testing system is run on each patch release and results automatically collected. If the results show any change in behavior since the previous run, some investigation and analysis is performed by the tester, to help narrow down problems and identify defects that need to be highlighted. A report is then generated as shown in Figure 4, including both the automatically collected data and the tester's analysis, and submitted to the NFSv4 developers for review.

This testing framework has also been adapted for use with another project, Inkscape. This adaptation implements the SUTs as Xen virtual machines, and thus is able to perform all testing and test management on a single physical

machine. This arrangement is straightforward to set up because Inkscape is a user-space application and does not require rebooting or reimaging between test runs.

The framework is also being adapted for use in testing the Linux Hotplug Memory and Linux Hotplug CPU projects. The code of the harness is open source and reasonably modular, allowing it to be adapted for other uses.

Bug Tracking

One of the consequences of good testing is a stream of defect reports. In open source testing, the volume of defect reports can quickly swell beyond the capabilities of the developers to handle. This can risk burnout or discouragement among the developers. Alternately, the developers could simply ignore defect reports, thus risking alienation and frustration of the users. Thus, having an effective tracking and triage process for defects can be vital for the project's long term success.

One of the first requests OSDL received from its member companies when becoming active with the NFSv4 testing community was to help in establishing a bug tracking capability. Our initial investigation showed some resistance to the idea, due to various reasons:

- Risk of complicating reporting processes
- Risk of extra overhead imposed on developers
- Unfamiliarity with bug tracking best practices
- Previous bad experiences with bug trackers in corporate development settings

Most tellingly, the NFSv4 project had already tried two bug tracking systems previously, and neither "caught on", thus creating a large amount of skepticism about bug tracking in general.

Due to these reservations, our initial attempt at establishing bug tracking began to meet resistance. Rather than push it, and alienate the community, we opted instead to focus on using the approaches the processes the community was already comfortable with. Prior to the bug tracker discussions, defect reports were simply sent to the mailing list for discussion and archiving; we acknowledged that this was an acceptable process for current needs.

But as the months passed, testing efforts began revealing a number of issues, many of which were important but not immediately solvable. The question of bug tracking was raised once again, and this time the development community was much more open to the idea. Even among those who had been skeptical before, there was a willingness to give it a try.

The bug tracker was implemented in June of 2005 and in its first two months of use has seen an increasing amount of activity. It is too early to call it a success, but the level of adoption by the core developers suggests that it has strong potential.

Building a Community-owned Testing Methodology

There are a number of challenges to the community approach. Unlike traditional testing, where a single company owns the process and employs the staff to perform it, in wide, community-driven testing processes it can be difficult to get every area filled.

Also, with open source the distinction between developers and testers is much more blurred. This can sometimes result in more emphasis placed on development than on testing. For NFSv4, a balance must be struck that includes strong emphasis on both testing and development.

A third challenge is the sheer complexity of the NFS code stack. In addition to the NFS client and server code in the Linux kernel, there is a surrounding layer of utilities, administrative tools, underlying file systems, add-ons like automounter and cachefs, and authentication services. Most of these components are maintained by people outside the NFSv4 community. Interactions among these pieces and NFS need thorough testing, but with the different versions and configuration settings, have huge numbers of permutations. Opening participation in testing to a wider user community will help distribute the effort and help identify the areas where complexity is an inhibitor.

However, these challenges dovetail with the strengths of corporate-backed testing efforts. Since many of the areas needing assistance happen to be areas that corporate users have a vested interest, those testing efforts will be easily

justifiable as priorities for them. Companies are accustomed to dedicating employees to testing and have developed procedures for organizing large scale test teams. They can scale their contribution to match their business needs, thereby providing an effective way to address complexities - if a given company needs a particular set of interactions tested thoroughly, then the business case will exist to justify funding a testing effort to do so. The Bull contributions to testing NFS is an example of this practice.

Establishing a clear, well-organized, structured testing effort in the open NFSv4 community will enable these organizations to better participate in conducting the testing; they can focus on their own priorities. By encouraging them to share their results openly, NFSv4 as a whole will be improved.

IMPROVING OPEN SOURCE TESTING

Like documentation, testing is an activity that the open source community as a whole continually wrestles with. For example, at the 2005 Ottawa Linux Symposium, David Jones of Red Hat gave a keynote that focused exclusively on the need for more testing and better testing tools. Andrew Morton has also spoken extensively about the need for more testing of the Linux kernel.

The experience seen at the outset of the NFSv4 testing effort seems to echo the situation in many other projects. People recognize that more “testing” is needed, however beyond this it is often unclear what needs to be done. Based on the test needs analysis done for NFSv4, several issues we identified will be relevant to open source testing in general.

Improving Tests

Automated test frameworks are only as good as the tests that they run. Unfortunately, while a variety of general purpose tests exist, many are not actively maintained, or tend to test code that isn't changing very frequently. Jones pointed out that the Linux Test Project (LTP) in particular has not proven to be as useful as it could be, because its tests cover areas of the Linux kernel that no longer change much.

Test automation should not be viewed as something that is completed once the framework has been built; rather, creating the harness is merely the starting point. New test cases and scenarios that exercise new areas of the code must be added with regularity in order to identify new opportunities for improvement. The effort required to create these tests is worth it when the test will be run repeatedly on all future versions of the source code.

Improving Bug Trackers

Jones also pointed out the need for better intercommunication between bug trackers. Many projects do a good job at tracking defects in the project's code, but often a defect will be found to belong to another project. Currently, procedures to transfer or share a defect with another project are very ad hoc [37]. One idea for improving this situation would be to adapt all of the major bug trackers to implement something analogous to Rich Site Summary (RSS) feeds that would provide a publishing/subscription mechanism. This would allow a Linux distribution and an individual open source application to share the defect reports of mutual relevance, or for an application project to migrate defect reports they've isolated to an upstream library to that library's maintainer.

An even more ideal system might encapsulate individual defects in such a way that they could be easily distributed to and shared by any number of projects. A given person should be able to review defects from several sources aggregated into one single bug tracking application, much as one reads multiple mailing lists from one mail client. A company should be able to participate directly and efficiently in all the bug repositories of relevance to their products or services, and track additional information on top of it (such as contact information of clients experiencing the issues).

Another area of potential improvement in bug tracking is in allowing better collaboration on the description of the defect. Oftentimes, a non-trivial defect will accumulate many, many replies, some of which are irrelevant or are superseded by later discoveries. A useful system might allow the description section of the defect report to be freely edited in a wiki-like fashion, to enable multiple people to keep the top level description up to date and correct, so that as new participants join the bug discussion, they will be able to quickly and accurately learn the status, before reviewing the discussion thread.

Improving Testing Tools

Open source testing could also be greatly improved by the creation of better testing tools. There are a number of static analysis tools such as lint, oprofile, gprof, sparse, or valgrind. In many cases these tools need further work to enable them to remain relevant for new programming language features. There are also a number of dynamic analysis tools such as ethereal for network communication analysis, yet there are many aspects of testing for which no dynamic analysis tools yet exist in open source [38]. For example, security testing is an area where tools are particularly desperately needed; most of the good security tools are proprietary and not easily available to open source developers.

Improving Project Metrics

In addition to testing tools, project metrics can be valuable ways to identify problems and to quantify quality improvements in an open source project. A variety of tools exist for performing line count analysis, measuring defect fixing progress, and so forth. Unfortunately, these tools tend not to be available in an integrated fashion, and the overhead of finding the tools and setting them up is often higher than can be afforded by resource-limited projects. Ideally, these tools would be included as part of project hosting services such as SourceForge, or integrated into the project's version control system so they can be run automatically, with results (and graphs) posted on a regular cycle.

Improving Community Involvement in Testing

Of course, the best tests and tools in the world will be no good if they are not used. Thus a critically important need is for increased involvement from the community in performing synthetic testing. As is being seen with NFSv4, synthetic testing is an excellent way to identify classes of problems that users would be unlikely to report. This testing, such as going through an application and trying out every option, or attempting to run the program on a random collection of input files, or pushing the program to its limits, is certainly within the skill of ordinary users, it simply requires some dedicated time to do. As has been found with commercial software development, it can be difficult to recruit people to fill these roles [39].

The need for more testers is quite serious and deserves much more attention than it gets currently. Jonathan Corbet pointed this out in his keynote at the Ottawa Linux Symposium (OLS) 2005, remarking that the past issues of patch management in the Linux kernel have been solved through tools like Bitkeeper and git, enabling an extremely high rate of change in the codebase, and that as a result testing this profusion of code changes is the bottleneck today. Efforts to attract and retain testers, test developers, and toolsmiths are vital for the continued growth of that project, and should serve as important lessons to other open source projects.

Improve “Ecosystem”-level Testing

As mentioned earlier, bugs live in cracks between applications, in the form of interoperability issues, file format incompatibilities, differences in optimization approaches, API changes, and so forth. Many open source projects barely have the resources to test for issues within their codebase, let alone testing against other code “in the wild”. Fortunately, the style of open source user testing is effective at quickly finding these sorts issues as the software is distributed, however this can often be a frustrating experience for users who expect the program to at least install and run within their environment.

At OSDL, we have experimented with automated systems to test a given program against versions of other software, other distros, and so forth. This tends to be a challenging task, since it requires care in managing a variety of configurations, and in that each new “degree of freedom” added ends up multiplying the system complexity.

It may be possible that recent developments in automated provisioning and virtualization such as Xen may give better ways of managing this complexity. More research into using technologies such as these for (semi-)automated testing is needed.

In addition, encouragement of cross-project communication can help mitigate the problems. For instance, two GUI applications that users commonly use together could collaborate on a review of their keyboard shortcuts or menu layout, and identify areas where consistency could be improved. Or upstream/downstream projects, such as an application and its dependent libraries, could arrange automated mechanisms for sharing defect reports or to coordinate API changes.

Improving Testing for Small Projects

From a testing point of view, a systemic problem exists in the open source community in the reality that many projects are far too small and too limited of resources to be able to implement any but the most basic testing practices.

A number of open source programs are maintained by individual developers. Many more do not have an active maintainer. Yet often these small projects fulfill critically important areas in software stacks. This situation can arise due to a wide variety of causes, but they impose a weakness on the open source community because defects fail to get reported and/or fixed, sometimes to the degree that it inhibits the software stack as a whole; a chain is only as strong as its weakest link.

An approach to help address this problem is to provide services that can be quickly and easily applied by small projects to allow them to gain easy test results. For example, a user could paste in a URL of a random piece of software, and the system would download the code, identify and install dependencies, compile the code, run a battery of general purpose tests against it, and then generate and email a summary report. By highly automating this process, it reduces the burden of testing and thus encourages its use.

Another idea for addressing this issue is to encourage larger, more active projects to share access to their testing frameworks with smaller projects that develop tools and libraries that the application depends on.

CONCLUSION

From a testing perspective, open source methodologies are considerably different from traditional, formal methods, yet have been shown to be effective in multiple instances. Even so, they do not obviate the value of performing formal testing as well.

The testing for the new version 4 of NFS is proving to be a great opportunity to tap both open source and traditional styles of testing to strengthen its quality [40]. The use of per-patch cross compile builds and a bug tracker in conjunction with the NFS user community shows the strengths of the open source method for achieving quality improvements. Coupled to that are dedicated testing efforts by Bull and automated testing efforts by OSDL that employ more traditional methods such as direct regression, performance, and robustness test runs against the codebase.

The growing adoption of open source technologies by companies and the need for better tested software suggests that some of the things learned in the NFSv4 effort may have broad applicability for other, similar efforts. By making testing easier and more rewarding to do, we all can benefit from having better software to rely on long into the future.

ACKNOWLEDGMENTS

Special thanks to reviewers Richard Vireday, Kees Cook, Craig Thomas, and Mary Edie Meredith, and Eric Schnellman. This work was supported by the Open Source Development Labs.

REFERENCES

1. <http://nfs.sourceforge.net>
2. Thurlow, Rob, et al. "AFS/DFS – Migrating to NFSv4," *NFS Industry Conference*, Sep 2003. <http://www.nfsconf.com/pres03/thurlow.pdf>
3. <http://www.citi.umich.edu/projects/nfsv4/linux/>
4. "Open Source: Open for Business," *Leading Edge Forum Report*, 2004. <http://www.csc.com/features/2004/48.shtml>
5. Raymond, Eric. "The Cathedral and the Bazaar,"2000 <http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/>
6. Massey, Bart. "Why OSS Folks Think SE Folks Are Clue-Impared," *Proc. Workshop on Open-Source Software Engineering, 2003 International Conference on Software Engineering*, Portland, OR, May 2003. <http://www.cs.pdx.edu/~bart/papers/icse-osse.pdf>

7. Schmidt, Douglas C., and Porter, Adam. "Leveraging Open Source Communities to Improve the Quality and Performance of Open Source Software," *1st Workshop on Open Source Software Engineering*, ICSE, Toronto, Canada, May 2001. <http://www.cs.wustl.edu/~schmidt/PDF/skoll.pdf>
8. Erenkrantz, Justin R. "Release Management within Open Source Projects," Institute for Software Research.
9. Wheeler, David A. "Why Open Source Software / Free Software (OSS/FS, FLOSS, or FOSS)? Look at the Numbers!", Mar 2005. <http://www.dwheeler.com/oss_fs_why.html>
10. Miller, Barton P., Fredricksen, Lars, So, Brian. "An Empirical Study of the Robustness of UNIX Utilities," National Science Foundation Technical Report, 1990. <http://www.cs.wisc.edu/~bart/fuzz/fuzz.html>
11. Ge, Li, Scott, Linda, and VanderWiele, Mark, "Putting Linux reliability to the test" Linux Technology Center Technical Report, Dec 2003. <http://www-106.ibm.com/developerworks/linux/library/l-rel/>
12. Vaughan-Nichols, Steven J. "Can you Trust this Penguin? What's Wrong (And Right) With LINUX" ZDNet, Nov 1999. <http://web.archive.org/web/20010606035231/http://www.zdnet.com/sp/stories/issue/0,4537,2387282,00.html>
13. Godden, Frans. "How do Linux and Windows NT measure up in real life?" ID-side, Jan 2000. <http://gnet.dhs.org/stories/bloor.php3>
14. Lemos, Robert. "Security research suggests Linux has fewer flaws," CNET News.com Dec 2004. <http://news.com.com/Security+research+suggests+Linux+has+fewer+flaws/2100-1002_3-5489804.html>
15. "Sites utilizing Microsoft Webserver-Software turn in inferior reliability results in study of Swiss websites," SysControl AG press release, Feb 2000. <http://web.archive.org/web/20011011215009/http://www.syscontrol.ch/e/news/Serversoftware.html>
16. <http://www.dwheeler.com/frozen/top.avg.2001aug3.html>
17. Samoladas, Ioannis, et al. "Open source software development should strive for even greater maintainability", *Communications of the ACM* Volume 47, Number 10 <http://www.acmqueue.com/modules.php?name=Content&pa=showpage&pid=240>
18. Kaven, Oliver. "Performance Tests: File Server Throughput and Response Times," PC Magazine, Nov 2001. <http://www.pcmag.com/article/0,2997,s%253D25068%2526a%253D16554,00.asp>
19. Howorth, Roger and Stevens, Alan. "Samba runs rings around Win20000," VUNet.com, Apr 2002. <http://www.vnunet.com/News/1131114>
20. Dyck, Timothy. "Server Databases Clash," Eweek, Feb 2002. <http://www.eweek.com/article2/0,3959,293,00.asp>
21. Reavis, Jim. " Linux vs. Microsoft: Who Solves Security Problems Faster?" <http://web.archive.org/web/20010608142954/http://securityportal.com/cover/coverstory20000117.html>
22. "New Evans Data Survey Reports Security Breaches Rare in Linux Environment," Evans Data Corp. Press Release, April 2002. <http://www.businesswire.com/cgi-bin/f_headline.cgi?bw.040802/220982285>
23. Pescatore, John. "Commentary: Another worm, more patches," Gartner Viewpoint, 2001. <http://news.cnet.com/news/0-1003-201-7239473-0.html?tag=nbs>
24. Leyden, John. "Zombie PCs spew out 80% of spam," *The Register*, June 2004. <http://www.theregister.co.uk/2004/06/04/trojan_spam_study/>
25. <http://news.netcraft.com/archives/2005/08/01/web_server_survey_turns_10_finds_70_million_sites.html>
26. <http://www.redhat.com/software/rhel/details/>, Aug 2005.
27. <http://www.debian.org/>, Aug 2005.
28. <http://www.freshmeat.net/stats/>, Aug, 2005.
29. Halloran and Scherlis "High Quality and Open Source Software Practices," School of Computer Science, Carnegie Mellon University.
30. <http://nfsv4.bullopensource.org/tools/tests/page24.php>
31. Lebzelter, Judith. "Open Source Testing Framework for Handling a Mulple Product Stack," *Pacific Northwest Software Quality Conference*, 2004.
32. Dabney, Nathan. "The Scalable Test Platform," *Linux Journal*, Nov 2001.
33. Connectathon test suite. <http://www.connectathon.org/nfstests.html>
34. PyNFS test suite. <http://www.citi.umich.edu/projects/nfsv4/pynfs/>
35. LTP test suite <http://ltp.sourceforge.net/>
36. Iozone <http://www.iozone.org>
37. "Bug Report Networks: Varieties, Strategies, and Impacts in a F/OSS Development Community" Sandusky, et al, University of Illinois at Urbana-Champaign.

38. “Adopting Open Source Software Engineering (OSSE) Practices by Adopting OSSE Tools” Jason Robbins, University fo California, Irvine.
39. “Modeling Recruitment and Role Migration Processes in OSSD Projects” Chris Jensen and Walt Scacchi, Institute for Software Research, Bren School of Information and Computer Sciences, University of California, Irvine.
40. De la Torre, Lynne and Harrington, Bryce. “NFSv4 Testing: Preparing for Enterprise Deployment,” *LinuxWorld Magazine*, May 2005. <http://linux.sys-con.com/read/86018.htm>

APPENDIX A: NEW FEATURES IN NFS VERSION 4

- Tracks file state. Unlike prior versions of NFS, in NFSv4 file state (locking, reading, writing) is tracked between the client and server
- Permits lease-based locking. Allows the client to take ownership of a file for a period of time; it must contact the server to extend this lease
- Allows file delegation. NFSv4 servers can allow NFSv4 clients to modify cached files without contact to the server, until the server notes that another client needs it and issues a 'callback'
- Implements compound RPCs (Remote Procedure Calls). Multiple NFS operations (LOOKUP, OPEN, READ, etc.) can be combined into a single RPC request, thereby minimizing network round trips and thus improving latency
- Supports security flavors. A number of sophisticated security mechanisms including Kerberos 5 and SPKM3 are implemented, and APIs are available for adding new security mechanisms down the road
- Supports ACLs. On POSIX systems and Windows, NFSv4 standardizes how ACLs are used. Named attributes are also added, allowing user and group names to be accessed as strings, not just numeric Ids.
- Combines several distinct NFS protocols (stat, NLM, mount, ACL, and NFS) into a single protocol specification to allow better compatibility with network firewalls
- Supports file migration and replication

Proceedings Order Form
Pacific Northwest Software Quality Conference

Printed Proceedings are available for the following years.
Circle year for the Proceedings that you would like to order.

2005 2004 2003 2002 2001 2000 1999

To order a copy of the Proceedings please send a check in the amount of $35.00 each to:

PNSQC/Pacific Agenda
PO Box 10733
Portland, OR 97296-0733

Name_____

Affiliate_____

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.

Download PDF – 2004, 2003, 2002, 2001, 2000, 1999, 1998, 1997

The 2005 Proceedings will be posted on our website in November 2005

Copyright