

**TWENTY-FOURTH ANNUAL
PACIFIC NORTHWEST
SOFTWARE QUALITY
CONFERENCE**

October 10 - 11, 2006

**Oregon Convention Center
Portland, Oregon**

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

TABLE OF CONTENTS

Welcome	v
Conference Officers/Committee Chairs	vi
Conference Planning Committee	viii
Keynote Address – October 10	
<i>Maintaining Your Competitive Advantage: Strategies to Improve Cognition and Learning</i>	1
Andy Hunt	
Keynote Address – October 11	
<i>Cosmic Truths about Software Quality</i>	11
Karl Wiegers	
Quality in Development Track – October 10	
<i>Planning for Highly Predictable Results with TSP/PSP, Six Sigma & Poka-Yoke</i>	25
Mukesh Jain	
<i>Taking Ownership for Software Development</i>	37
Jim Brosseau	
<i>Making the Most of Community Feedback</i>	45
Dave Liebreich	
<i>Know Your Code: Stay in Control of Your Open Source</i>	53
Douglas Levin, James Berets	
<i>Evolutionary Methods (Evo) at Tektronix: A Case Study</i>	63
Frank Goovaerts, Doug Shiffler	
<i>Smart Result Analysis (SRA) - A Key Competitive Advantage</i>	73
Manoharan Vellalapalayam	
Agile Track – October 10	
<i>Lean Cuisine – How Lean Thinking Bakes in Quality</i>	85
Jean Tabaka	
<i>First to Market or First to Fail – A General Systems Perspective</i>	111
Michael Bolton	
<i>Implementing Kaizen and Six Sigma in Application Development</i>	123
David Anderson	

Managing Testing Track – October 10

<i>Four Behaviors that Hold Testers Back</i>	137
John Lambert	
<i>Step Away from the Tests: Take a Quality Break</i>	145
John Lambert	
<i>Using Social Engineering to Drive Quality Upstream</i>	153
Thomas Gutschmidt	
<i>The Imagination Factor in Testing</i>	167
Mike Roe	
<i>Five Trends Affecting Testing</i>	175
Rex Black	

Measuring Quality Track – October 10

<i>The Challenge of Productivity Measurement</i>	181
David Card	
<i>MAGIQ: A Simple Method to Determine the Overall Quality of a System</i>	191
Dr. James McCaffrey	
<i>Performance Testing: How to Compile, Analyze, and Report Results</i>	203
Karen Johnson	
<i>Benchmarking for the Rest of Us</i>	215
Jim Brosseau	
<i>Key Measurements for Testers</i>	223
Pamela Perrott	
<i>Quantifying Software Quality – Making Informed Decisions</i>	245
Bhushan Gupta, Orhan Beckman	

Quality in Development Track – October 11

<i>Insights into Real Test-Driven Development</i>	261
Peter Zimmerer	
<i>Front End Requirements Traceability for Small Systems</i>	269
Robert Roggio	
<i>The Keyhole Problem</i>	287
Scott Meyers	

Agile Track – October 11

<i>Leading Change: Collaboration and Collaborative Leadership</i>	323
Pollyanna Pixton	
<i>Software Testing in an Agile World</i>	339
Paul Hemson	
<i>Defining Test Data and Data-Centric Application Testing</i>	349
Chris Hetzler	
<i>Test Case Maps in Support of Exploratory Testing</i>	357
Claudia Dencker	
<i>Adopting and Adapting Agile Development Practices in the Real World</i>	367
Don Hanson	

Advanced Testing Track – October 11

<i>Improving Test Code Quality</i>	379
Brian Rogers, John Lambert	
<i>CyberHunters. Diving into the Inner World of Test Engineers</i>	391
John Copp	
<i>Requirements Testing</i>	403
Kelly Whitmill	
<i>Pairwise Testing in Real World. Practical Extensions to Test Case Generators</i>	419
Jacek Czerwonka	
<i>Accelerating Performance Testing – A Team Approach</i>	431
Dawn Haynes	
<i>Leveraging Model-driven Testing Practices to Improve Software Quality at Microsoft</i>	441
Jean Hartmann	

Quality Assurance Track – October 11

<i>Techniques That Inspired Workplace Improvement</i>	451
George Yamamura	
<i>Dynamics of Exploratory Testing</i>	459
Jon Bach	
<i>Myths and Strategies of Defect Causal Analysis</i>	469
David Card	

Proceedings Order Form	last page
-------------------------------------	-----------

Welcome to the 2006 Pacific Northwest Software Quality Conference

I am excited to present to you a fabulous program for the 2006 conference that offers something for everyone.

Our keynote speakers this year will offer you practical insights for stepping back and taking the bigger picture view of your projects. They will also allow you a moment to take a look at yourself and how you learn. This will all be done in a style that will be entertaining as well as rewarding.

Our theme this year is “Quality – A Competitive Advantage.” You will hear practical information enabling you to better assess your situation, improve processes, change your lifecycle, influence people and more.

I think that you will find hearing from some of the best in our field will give you tools and processes that you can use immediately and also inspire you to make the changes necessary to use them. Our Open Space sessions will give you the chance to explore in more depth with experts and peers to further your experience.

Welcome to the 24th annual conference.

Paul Dittman
PNSQC President

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Paul Dittman – PNSQC President and Chair
McAfee, Inc.

Debra Lavell – PNSQC Vice President
Intel Corporation

Diana Larsen – PNSQC Secretary & Open Space Co-Chair
FutureWorks Consulting

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

Arlo Belshee – Technology Chair

David Butt – Program Co-Chair

Esther Derby – Keynote, Invited Speaker & Workshop Chair
Esther Derby Associates

Cynthia Gens – Program Co-Chair
Loui Canz - Louis XV

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

Cindy Oubre – Exhibits Chair

Keith Stobie – Publicity Chair
Microsoft

Patt Thomasson – Communications Chair
McAfee, Inc.

2006 CONFERENCE PLANNING COMMITTEE

Randal Albert

Rick Anderson

Tektronix, Inc.

John Balza

Hewlett Packard

Kit Bradley

Wildblue Consulting

David Chandler

Nationwide Insurance

Albert Dijkstra

James Fudge

McAfee, Inc.

Dennis Ganoe

Optilink Healthcare

Brian Hansen

Jason Kelly

Microsoft

Randy King

Jonathan Morris

Kronos/Unicru, Inc.

Sudarshan Murthy

Portland State University

Ganesh Prabhala

Intel Corporation

Ian Savage

McAfee, Inc.

Eric Schnellman

JanEric Systems

Wolfgang Strigel

QA Labs Inc.

Ruku Tekchandani

Intel Corporation

Richard Vireday

Intel Corporation

Scott Whitmire

Progressive Solutions (USA), Inc.

Yabo Wang

Hewlett Packard

Maintaining Your Competitive Advantage: Strategies to Improve Cognition and Learning

*Andy Hunt
The Pragmatic Programmer*

The raw material of software development is not a language, an IDE, or a tool, it's you. How we learn new technology and acquire new skills is key to our careers. Join Andy Hunt for this presentation that includes a recap of The Dreyfus Model, from his popular talk "Herding Racehorses and Racing Sheep" and a look at how to boost your career by Integrating Brain Modalities, Accelerating Learning, and Managing the Torrent.

Andy Hunt is a programmer turned consultant, author and publisher. He co-authored the best-selling book "The Pragmatic Programmer", was one of the 17 founders of the Agile Alliance, and co-founded the Pragmatic Bookshelf, publishing award-winning and critically acclaimed books for software developers.

Andy started writing software professionally in early 80's across diverse industries such as telecommunications, banking, financial services, utilities, medical imaging, graphic arts, and of course, the now-ubiquitous web.

Maintaining Your Competitive Advantage:

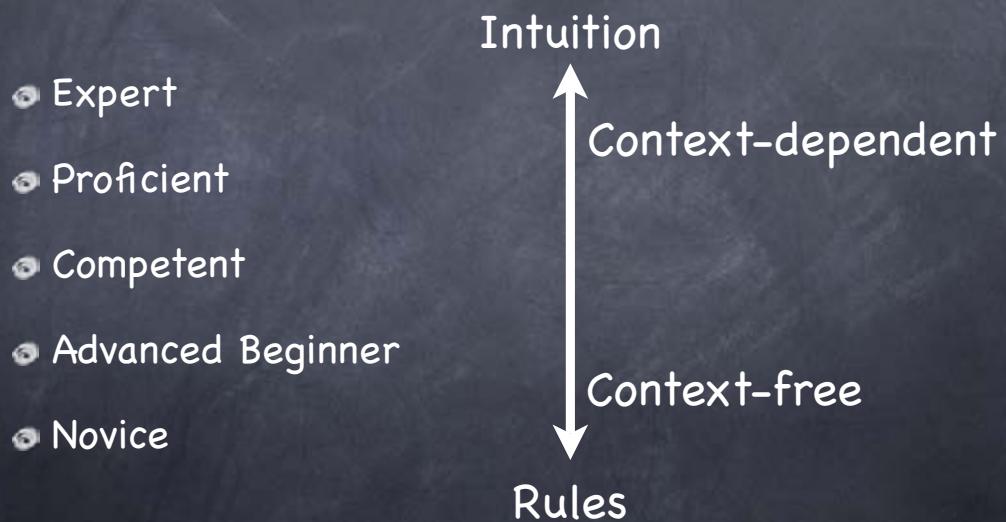
Strategies to Improve Cognition and Learning

Andy Hunt
The Pragmatic Programmers, LLC.
/\ndy
andy@pragprog.com

In this talk:

- What is Wetware, why does it matter?
- Using more of the brain's horsepower
- More effective learning
- Managing the torrent

The Dreyfus Model

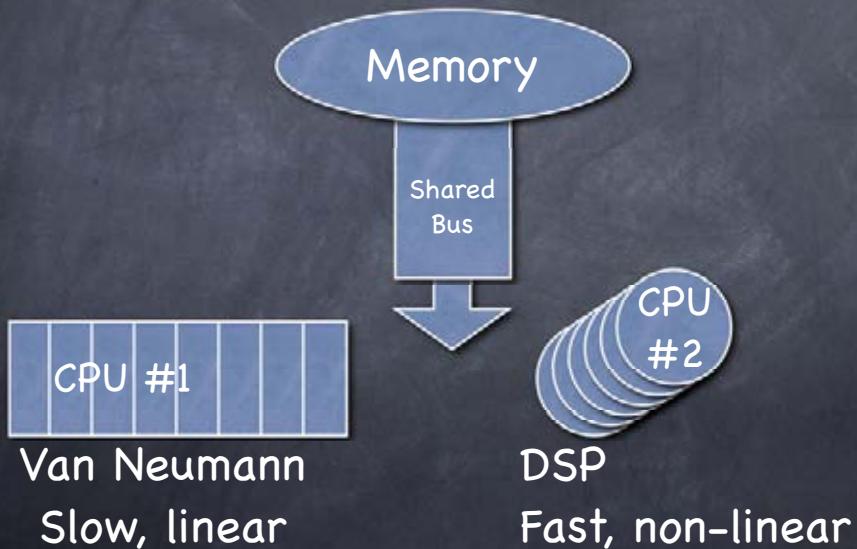


Shocking, but true:

“Mastering a syllabus of knowledge
doesn't increase professional
effectiveness”

Kemp77, Eraut90

This is your brain



Cultivating R-mode

- What does R-mode processing feel like?
- How to engage R-mode more often?
- How to integrate with L-mode processing?

How to engage R-mode?

- Expand sensory involvement
- Tactile enhancement
- Involve imagery, additional senses

Emphasize cross-sensory feedback

- For a given design:
 - write it down
 - draw a picture
 - describe it verbally
 - engage in open discussion

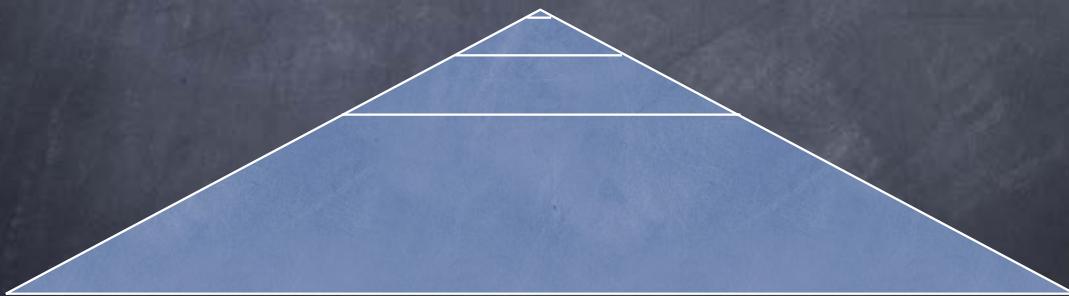
Integrating R-mode with L-mode

- ➊ R-mode to L-mode flow
 - ➋ Experiential immersion first
 - ➋ Multi-sensory, provides context
 - ➋ “I thought I had another foot...”

The magic of an “oracle”

- ➋ Force the pattern matching to reconcile unlike patterns
- ➋ Broadens the scope of material under consideration

Everyone has good ideas



If you don't keep track of great ideas, you'll stop noticing you have them

pragmatic investment plan

- Modeled after financial investment plans:
 - Have a concrete plan
 - Diversify
 - Look for real value
 - Active, not passive investment
 - Regularly (dollar-cost-averaging)

Getting Things Done (GTD)

1. Scan the input queue once and only once
 - ➊ sort it, do it, or delegate it
2. Work each pile in order
 - ➋ avoid context switch
3. Don't keep mental lists
 - ➌ dynamic refresh is too expensive



Reading summary techniques

- ➊ SQ3R:
 - ➋ Survey
 - ➌ Question
 - ➍ Read
 - ➎ Recall
 - ➏ Review

Mindmaps

- Title in center of page
- Draw lines out to major sub-topics
- Recurse
- it's a "2D organic outline"

Situational feedback

- Don't TRY this or DO that, just be aware
- "Notice how..."
- Non-judgmental awareness

Cut down on distractions

- ➊ Email and news sites:
 - ➊ Greedy, rapacious spammers
 - ➋ Chinese space program
 - ➌ Election fraud

/\ndy

andy@pragprog.com

Blog: toolshed.com/blog

Cosmic Truths about Software Quality

Karl Wiegers Process Impact

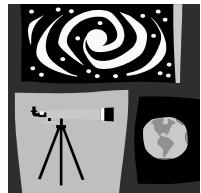
Everyone wants to build and use quality products, but software people debate endlessly the meaning of "quality" and how to achieve it. I have identified ten principles about quality that apply almost universally to software projects. This presentation explains why any software team that cares about quality needs to understand these principles and to choose development approaches consistent with them. The principles are:

1. Quality has many dimensions.
 2. Quality begins with the requirements.
 3. Customer involvement is the greatest determinant of software quality.
 4. Both internal and external quality are important.
 5. Developer discretion has a large influence on quality.
 6. Quality must be made a conscious project priority.
 7. You can pay now, or you can pay a lot more later.
 8. Long-term productivity is the result of high quality.
 9. Iteration is a key to software quality.
10. If you don't design for quality, you won't get it.

Karl Wiegers is Principal Consultant with Process Impact, a software process consulting and education company in Portland, Oregon. Previously, he spent 18 years at Eastman Kodak Company, where he held positions as a photographic research scientist, software developer, software manager, and software process and quality improvement leader. Karl received a Ph.D. in organic chemistry from the University of Illinois.

Karl is the author of the books *Software Requirements*, 2nd Edition, *More About Software Requirements*, *Peer Reviews in Software*, and *Creating a Software Engineering Culture*. He has also written 160 articles on many aspects of software development and management, chemistry, and military history. Karl has served on the Editorial Board for IEEE Software magazine and as a contributing editor for *Software Development* magazine. He is a frequent speaker at software conferences and professional society meetings.

Cosmic Truths About Software Quality



Karl E. Wiegers
PROCESS IMPACT 
www.processimpact.com

Copyright © 2006 Karl E. Wiegers

Cosmic Truth #1



*Quality
has many
dimensions.*

Some Dimensions of Quality



Cosmic Truths About Software Quality

3

Copyright © 2006 Karl E. Wiegers

Cosmic Truth #2

Quality
begins with the
requirements.

Cosmic Truths About Software Quality

4

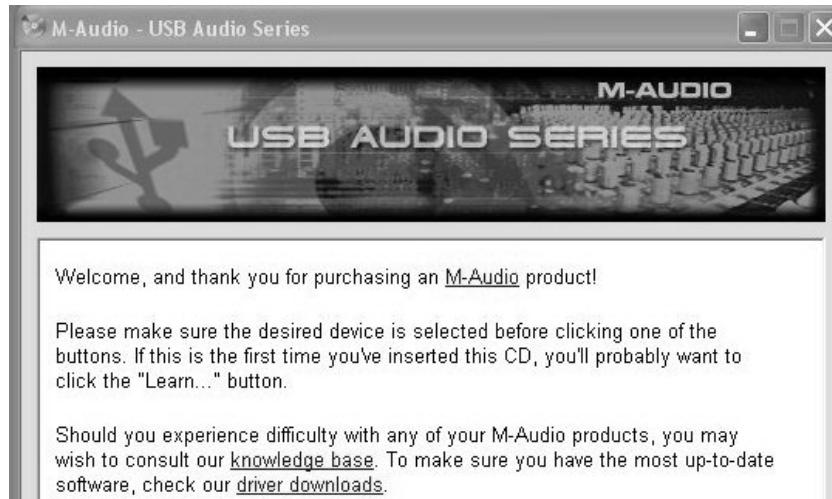
Copyright © 2006 Karl E. Wiegers

Some Quality Attributes

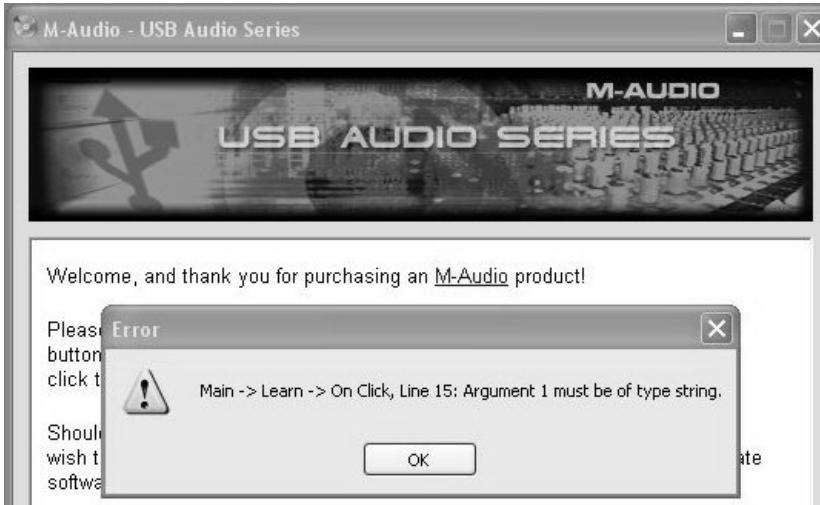
- Availability:** Is it available when and where I need to use it?
- Integrity:** Does it protect against unauthorized access, data loss, and viruses?
- Installability:** How easy is it to install, uninstall, and reinstall?
- Interoperability:** How easily does it interconnect with other systems?
- Performance:** How fast does it respond or execute?
- Reliability:** How long does it run before experiencing a failure?
- Recoverability:** How quickly can the user recover from a failure?
- Robustness:** How well does it respond to unexpected conditions?
- Safety:** How well does it protect against injury or damage?
- Usability:** How easy is it for people to learn or to use?

[Karl E. Wiegers, *Software Requirements*, 2nd Edition, Microsoft Press, 2003]

Installability Example - 1



Installability Example - 2



Advanced Quality Specification: Planguage

TAG: PowerConsumption.Standby

AMBITION: To have the device consume as little power as possible in standby mode.

SCALE: Watts

METER: Average power consumption on 2 units, measured in triplicate for 30 seconds after placing into standby mode.

MUST: 10W ← Power Engineer

GOAL: 5W

WISH: 2.5W

Standby Mode **DEFINED:** The device state in which the firmware ignores all requests except the 'go to running' request and generates no communication toward the host.

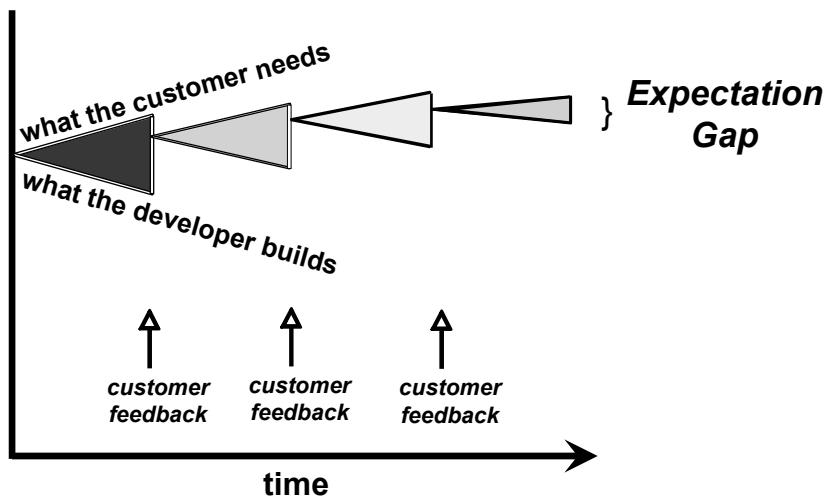
[Tom Gilb, *Competitive Engineering*, Butterworth-Heinemann, 2005]

Cosmic Truth #3



*Customer involvement
is the greatest
determinant of
software quality.*

The Need for Customer Involvement



Obtaining Customer Involvement

- ◆ Identify user classes
- ◆ Select *product champions*
- ◆ Agree on customer rights and responsibilities
- ◆ Have on-site customers
- ◆ Understand how product will be used
 - ✓ use cases, scenarios
 - ✓ operational profile
- ◆ Build and evaluate prototypes



Cosmic Truth #4

*Both external
and internal quality
are important.*



Some Aspects of Internal Quality

- Flexibility:** How easily can new functionality be added?
- Efficiency:** How well does it utilize processor capacity, disk space, memory, network bandwidth, and other resources?
- Maintainability:** How easy is it to correct defects or make changes?
- Portability:** How easily can it be made to work on other platforms?
- Reusability:** How easily can we use components in other systems?
- Scalability:** How easily can more users, servers, or other extensions be added?
- Supportability:** How easy will it be to support after installation?
- Testability:** Can we verify that it was implemented correctly?

Cosmic Truth #5

*Developer
discretion has a
large influence on
product quality.*



Developer Discretion and Quality

- ◆ Specifications can never specify everything.
- ◆ Requirements and designs are never perfect.
- ◆ Internal and external quality must be balanced.
- ◆ Quality results from global optimization.
- ◆ Developers need to know what's most important.
 - ✓ it's impossible to simultaneously optimize everything
 - ✓ use peer reviews to align optimizations
- ◆ Design and coding choices affect internal quality.



[courtesy Scott Meyers, "Programmer Discretion and Software Quality," Rose City SPIN, 1/12/06]

Cosmic Truths About Software Quality

16

Copyright © 2006 Karl E. Wiegers

Cosmic Truth #6

*Quality
must be
made a conscious
project priority.*

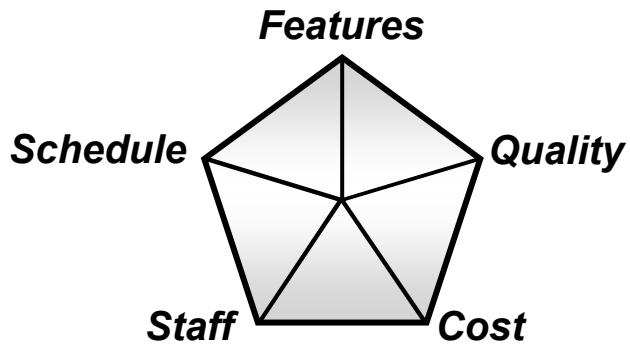


Cosmic Truths About Software Quality

17

Copyright © 2006 Karl E. Wiegers

Five Dimensions of a Software Project



*For a given project, each dimension can be a **CONSTRAINT**, a **DRIVER**, or a **DEGREE OF FREEDOM**.*

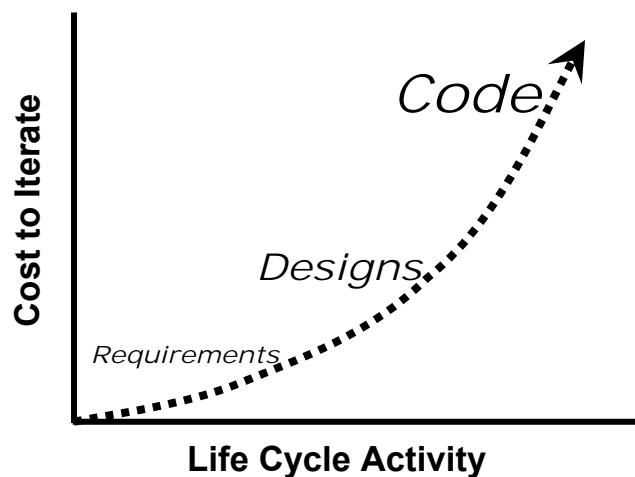
[Karl E. Wiegers, *Creating a Software Engineering Culture*, Dorset House, 1996]

Cosmic Truth #7



Iteration
is a key to
software quality.

Relative Cost of Iteration



Cosmic Truth #8



*Long-term
productivity
is the result of
high quality.*

What's Reducing Your Productivity?

◆ Inadequate knowledge and skills

- ✓ Train the team members



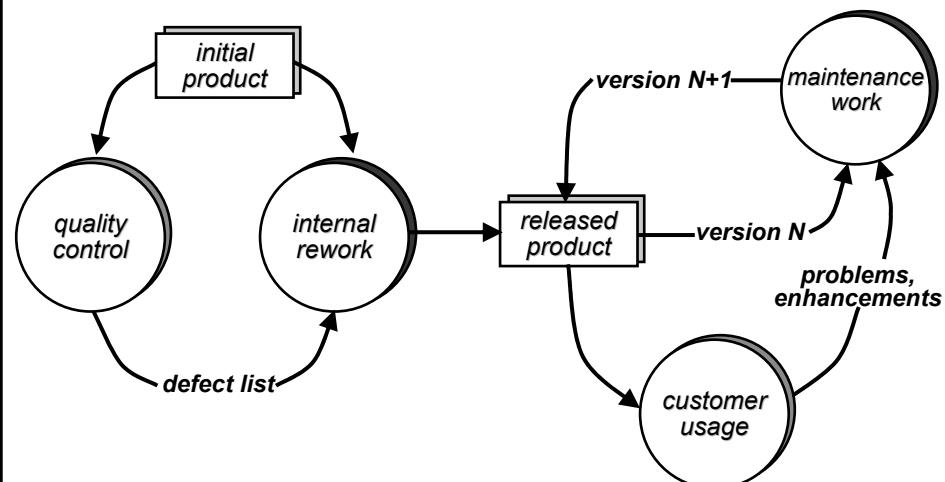
◆ Internal rework

- ✓ 30-40% of total development effort
- ✓ Analyze root causes of rework
- ✓ Focus process and quality improvement there

◆ External (post-release) rework

- ✓ Measure cost of handling customer-reported defects
- ✓ Compare to cost of defects found in system test...
- ✓ ... and by inspection

The Truth About Rework



[Karl E. Wiegers, *Creating a Software Engineering Culture*, Dorset House, 1996]

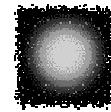
Cosmic Truth #9

You can

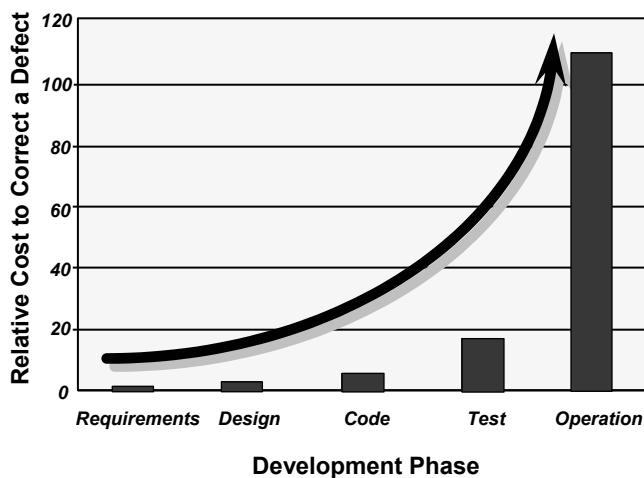
pay now, or

you can pay

a lot more later.



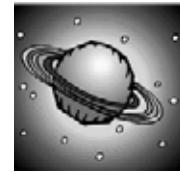
Relative Cost to Fix a Defect



[Robert Grady, "Applications of Software Measurement Conference," 1999]

Cosmic Truth #10

*If you
don't design
for quality, you
won't get it.*



Cosmic Truths About Software Quality

***NO
SURPRISES!***

Planning for highly predictable results with **TSP/PSP, Six Sigma & Poka-Yoke**

Author(s) Names: Mukesh Jain

Abstract

Today, our competitive marketplace demands the best of everything – the best quality, reduced costs and a perfect schedule. A defect-free product delivered on time at minimal cost is the new standard that demanding customers expect, and good suppliers continually strive to meet. It is not an easy task to meet these challenges without compromising high quality levels OR the schedule OR the cost. There is no silver bullet → we can meet all these goals by having a disciplined process and managing with the right set of matrices.

In this presentation, Mukesh will detail how TSP techniques can be followed to plan the right thing, do the right thing and expect the right thing every time.

Often, the projects are planned by Managers, executed by the team and if things go wrong – it becomes the team's failure. It is easy to be reactive at work. It is much harder but very necessary to be proactive and plan your team's work. Sometimes your manager will grab you in the hall and say, "Hey, can you do this project now and finish it in two months?" Or, a senior management planning committee will call you into its meeting and say, "We need this project now. Can you commit to it?" It is very tempting to say "yes." However, saying yes is exactly the wrong thing to do. You can say, "Let me check to see if my previous estimate is still accurate, and I'll get back to you before 5 p.m. today."

Historically, application development is often estimated and tracked based on a gut feeling. A schedule is developed from the rough estimate, and as the team tries to meet schedule, the team struggles to meet deadlines. Corners get cut and often the last phase – stabilization – suffers the full impact of poor planning and estimating. Customers are dissatisfied, teams are exhausted and sometimes quality has been compromised in the drive to meet the promised date.

TSP really shifts the focus from testing as the 'find it and fix it' stage, to each individual engineer acting to prevent defects throughout the project lifecycle. Team members record data during project execution, track key metrics and take corrective action as soon as the project deviates from the plan. Each engineer performs a self-review to ensure the quality of their own output before it goes to the next phase. This brings about a high level of predictability in Schedule, Effort and Quality.

With the paradigm shift, the TSP team plans for their project, tracks all the important metrics on weekly basis and takes corrective action as needed. It becomes the team's project plan and every member is committed to it. Using Six Sigma along with TSP – makes future planning even better. The senior management will start appreciating the project plan and estimates based on historical data (rather than guesstimates).

Mukesh will talk about his experience in implementing TSP in Microsoft India and share the success stories on how the teams had a better work-life balance and delivered very high quality product on schedule and within budget. He will show some examples where some of the teams are using Six Sigma along with TSP to improve the process. Also, he will highlight some common pitfalls to avoid while planning with TSP.

Keywords: TSP/PSP, Six Sigma, Managing with Metrics, project management, Poka-Yoke, Estimation, Mistake-proofing, defect prevention.

Introduction

Delivering the right software on schedule has long been one of the most difficult challenges faced by many businesses. The rate of failure has been high, more than 60% of medium and large software projects are “failures” in the sense that they will be delayed by six months or more, and many will be cancelled.

Many different responses to these problems have evolved in recent years, including those discussed here and many others, such as CMMi, ISO 9000, TSP/PSP, and Six Sigma. In response to these diverse approaches, many organizations find themselves somewhat conflicted and confused as to what is best and what should come first. The goals of this paper include; explaining how these different approaches relate to one another and providing guidance on leveraging the best of all these approaches and how to use them to plan for success.

CMM/CMMi & ISO 9000 models focus on Organizational capability and are typically implemented using a top-down approach. PSP/TSP is more bottom-up approach: where PSP can be leveraged to build individual skills and discipline, and TSP enables teams to build quality products within budget and schedule.

The top-down strategy with the CMMI

- establishes a mature management environment
- develops transition skills
- provides experienced transition resources

The bottom-up strategy with PSP and TSP

- builds development skills
- provides process understanding
- rapidly improves priority projects

The CMMI and TSP

- can be used independently
- work well together

SEI’s research has found high degree of predictability (in schedule, cost and quality) for projects using TSP. Based on the data, it looks like a CMM Level 5 company will probably ship around 1.05 Defects/KLOC and a TSP based project will have around 0.06 defects/KLOC.

Shipping great software on time is a difficult but not impossible task.

Key Challenges in Project Planning

Nobody intends to plan for failure. People do not say, “I want to plan in such a way that the project will fail this time”. We all know the output of the planning process; but do we know what inputs are required for effective planning?

Three Facets of Quality

Any project can be defined in three dimensions. The first dimensioning the product definition is the “feature set”. The second dimension is schedule “shipping on time” and the third is “reliability”. Different people have different views about these facets. The QA team might think “reliability” is the most important thing to do. The Marketing team will talk about “feature set” and the leads/management would think “shipping on time” is very important. So, in the end everybody is focusing on a different priority, which could result in chaos and impact the shipping of the product.

Planning OR Execution

When things go wrong, frequently the team who is executing the project gets the blame – but do we check to see if the planning itself was good? Do we incorporate our experience back to the planning process? Is there a guarantee that we won't end up in same situation next time?

Estimates/Guesstimates

Most of the time, the project planning process and its output are driven by the schedule and past experience is seldom taken into consideration. The Estimates are usually Guesstimates and aren't based on historical data.

Reinventing the wheel

Some people are aware of the inputs required for the planning purposes, but they will always have reasons for not using specific data or a recommended process. I was talking to a project manager about data which I had collected on a particular technology – like LOC, effort, quality etc. The project manager had a strong reason for not using this data for estimation; "our case is different; this data is good – but not useful for me". As a result, we end up reinventing the wheel every time and we were not confident of delivering as per customer expectations on schedule and with quality.

Planning for Success using TSP/PSP, Six Sigma & Poka-Yoke

Often it has been observed that Planning does not fail. The inputs taken for planning are the key for successful planning, i.e. Project scope, Estimates, etc.

Shipping reliable, high-quality software on time is a difficult but not impossible task.

Right set of inputs and outputs

Sometimes your manager will grab you in the hall and say, "Hey, can you do this project now, and finish it in two months?" Or, a senior management planning committee will call you into its meeting and say, "We need this project now. Can you commit to it?"

Ever have a manager give you a brief description of a possible project and ask you for an estimate? After listening to the manager, you give him an estimate, explaining that it is a ball-park estimate based on what you've been told.

To your surprise, you learn a week later that the manager has presented the estimate to a client and puts you in charge of delivering a complete solution based on the estimate you provided. You hurry to explain that the estimate was just a ball-park estimate and you need to learn the detailed requirements before you can provide a final estimate. The manager argues that it is too late for that, saying that the customer has already approved your original estimate and the manager expects you deliver what you initially agreed to.

Try this experiment, come up with a requirement for a software component to be built and ask your team members individually to estimate the time and effort it is going to take – you will be surprised to hear the range of answers. Somebody might give you an estimate of 1 week and another person will say 3 weeks. In any case – none of these people can assure you that you will be able to ship a high quality product within the time frame. (You might think the person giving you the 3 weeks estimate might ship a high quality product.)

Knowing what to deliver and when (*TSP Meeting 1: Management Goals*)

Depending on the market and available technologies, the relative importance of each of the quality dimensions may differ. For example, a low number of defects might be the most important priority in a mature market where competing products are not otherwise differentiated. In another situation, being first to market first may take precedence over reliability. It is an axiom of the personal computer business that the first product to claim a majority market share on a new platform or in a new product may have that position for a

long time. In this case, product feature definition and schedule may be relatively more important than the number of defects, especially if there's no pre-existing installed base. At other times, features might be cut from a product to increase the amount of time available for testing, or to increase the certainty of making a ship date.

Everybody (Management, Marketing, Program Managers, Developers, Testers and Support) needs to decide as a team to make carefully considered trade-offs between the three dimensions of quality. Trade-off decisions must be made throughout planning and until the product ships to the customer.

TSP Launch Meeting 1 ensures that Management & Marketing teams communicate objectives, goals, priorities, the feature set, and business reasons to the development team along with management's minimum and ideal goals for the project and its flexibility around the deliverables – in terms of quality, schedule, resources, and feature set. The team must understand the goals, and ask clarifying questions to make sure all the stakeholders are in agreement.

Knowing who is who

(TSP Meeting 2: Roles)

Things fall through cracks when it is not clear who is doing what and people make assumptions that specific tasks will be taken care by someone else. Overlapping responsibilities will run into conflicts.

TSP Launch Meeting 2 ensures that each of the management stated and implied goals are understood by the team and the roles are allocated among the team members. When all team members consistently meet their role responsibilities, follow the defined process, and work to agreed goals and specifications, the team will be most efficient and effective, will not have team issues, and will successfully deliver as a team.

What are we building and how(TSP Meeting 3: Development strategy)

Once we know what to build – the next important thing is how to build. Usually people do project planning – arranging features and milestones appropriately. Often time, this is not done with the team. To know what we are building, the team needs to start with the conceptual design and have development strategy on how we can build it effectively and then plan for milestones based on that information. This ensures that the team has a clear understanding of the development strategy which is tailored towards their product based on the conceptual design. Also, this is the point where the team identifies if they need any additional process or tools to execute the project effectively.

TSP Launch Meeting 3 ensures that the team produces the product conceptual design, the development strategy, development process, and support plan. The team notes a prioritized list of all the components, elements or features along with each work item's gross size. This list helps the team to ship the right product at the right time.

Estimating

(TSP Meeting 4: Top-Down Planning)

The only time we will have 100% estimation accuracy is after we ship. But, to plan the project, we need to estimate at the start of the project. People can use any method for estimation, but the true value of estimation come from experience and historical data.

How often do people look at historical data for estimation? Do we estimate everything that is needed?

To do effective project planning, we need to estimate the size, effort, bugs – yes, we should estimate bugs also. Usually estimating in traditional project planning is doing on "effort" level only. I.e. if I have to build this feature – I will take 2 days. But with this change of mind set, we should start estimating the size of the component in terms of LOC or FP. This helps us improve the estimation accuracy over time and also the effort required is a function of

component size and not just a gut feel. For the first project, the size estimation accuracy may be +/- 50%. As the team matures, the accuracy tends to improve to within +/- 10%.

$$\text{Effort} = \text{Size} / \text{Productivity}$$
$$\text{Productivity} = F(\text{technology, feature complexity})$$

Available hours – Task Hours

Usually people work 40 hrs / week. But are these people 100% productive for all 40 hrs? Think about it, project meetings, organizational activities, emails, trainings, etc... are non-engineering activities which an employee does in the organization. Planning engineering tasks for all 40 hrs will do no good. The engineer will end up spending 60+ hrs to meet the deliverables planned for 40 hrs. So be realistic and identify non-engineering activities and include them in the project plan. I have seen typically around 27-30 available task hrs for an engineer and around 15-20 hrs for a lead.

TSP Launch Meeting 4 ensures that the team estimates everything – requirements document pages, design classes size, program LOC, test cases, etc. The team looks at historical data for similar technology and features when gauging size and estimating tasks. After the size estimation is done, the productivity data is looked at when considering technology and complexity. The team then looks at their availability (vacation, holidays, trainings, other projects, etc.) and updates the schedule sheet. The overall plan is then generated which gives a good overview of how many task hours are needed and available to complete the project. This ensures that the team has adequate resources and will be able to commit to a project delivery date.

Plan for bugs Injection & detection (TSP Meeting 5: Quality plan)

How often we plan for bugs? We usually have a gut feeling that we will have around 100 bugs during the development cycle for this release and we end up having 10-20 times that. In the next cycle – the same story repeats – the team works to some guesstimate, without documentation of goals and plans, and the actual bug count is way off.

Having disciplined methods of estimating the bug injection rate and detection yield will help the team plan for it. If you are spending 1 hr writing code or requirements/design/etc. – you will inject a particular # of bugs i.e. bugs injected per hour of coding. Then we look at bug detection yield. I.e. how good is the design review process at finding bugs present in the design at that moment? 50%? 70%? What about Unit Testing – can you find 80% of the bugs remaining in the code while doing unit testing? To start with, the team can use the industry data as reference and modify them based on their scenario. Over time, once they collect enough data, they will be in a position to plan effectively and their estimates will be closer to the actual results.

TSP Launch Meeting 5 ensures that the team plans for defects injection and detection; based on their historical data or the industry data. With this they are in a position to predict how many defects probably will be slipped into Acceptance testing and into production and validate their defects yield against the goals. This brings in accountability and over time maturity of the process – which eventually will result in shipping high-quality products on time.

Individual Plans (TSP Meeting 6: Bottom-up plans)

Most of the time, we end up having a Team Project Plan. Are all the team members the same? People are not identical and they usually have different styles of working. Each of the tasks are reviewed to ensure that it is not more than 10 hrs in duration – because if a task is more than 10 hrs – it would be difficult to measure earned value. In some cases by the time a lengthy task is completed it might be too late to respond to an issue.

TSP Launch Meeting 6 ensures that the team project plans are built using individual plans and every team member has got a chance to plan their own tasks after taking into consideration their vacations and other activities. In this meeting each person will come up with a date when

their work will be completed. Load balancing is done to ensure that team members complete their work in the right timeframe to be shipped.

Knowing what can go wrong (TSP Meeting 7: Risk Assessment)

Despite best intentions, crises that conflict with assessment plans often arise.

Doing a conscious analysis of all the risks in a project helps the team know what might go wrong and plan it accordingly.

TSP Launch meeting 7 ensures that all the risks associated with the project's success are identified, discussed, documented and an owner is assigned to track it. This ensures that most risks are identified, tracked, and have a risk mitigation plan.

Getting a "GO" from Management(TSP Meeting 8&9: Management Presentation)

Most of the time, the only thing presented to management is the project schedule. This is not enough. Management should have all the information about the project plan. Full disclosure will enable management to provide valuable feedback. Management needs to know if the team would be able to meet management goals: when is the project end date, what are the risks in the project, and what will be the quality of the product? In case the team thinks they are unable to meet the management goals – the team will present an alternate plan.

TSP Launch meetings 8 & 9 ensure that management is informed about the project plan and the team gets a "GO" to start with the project. This helps both management and the team to work with the same expectations.

Capturing Data for all the activities (PSP | Six Sigma)

When the team members are working on their tasks and activities – lot of data is generated, if we do not capture this data – we are losing something valuable. If we can't measure – we can't manage. In a TSP project, every team member logs their exact time for any activity, they also log interruptions, defects found in the work they have done and the size of work product produced. The data collected gives insight into the project and helps the team to make decisions.

How are we doing? (TSP: Team Status Meeting)

Achieving a relatively accurate view of project status is a very challenging goal.

It is very important for the team to know how they are doing at any moment. This helps them to identify any potential issue and take appropriate action to prevent a major disaster in the project. Data is captured on all the engineering activities done by the team. The data is consolidated at the team level. The consolidated data is compared against the plan. This process gives the team understanding about how they are doing when compared to the plan. TSP's Team Status Meetings ensures the team reviews their project earned value, quality of the work products, hours put in by the team, deviation from the plan, etc.

Schedule slips usually happen a little bit every day. If the tasks are not granular enough, we may not know how much we have slipped until it is too late. So, for any slip in the project, the team should look at the cause of the slip. Is it because the team member(s) did not put in adequate hours OR is the task taking more time. Based on the analysis – we need to watch the parameters and take appropriate actions. If necessary, a new plan needs to be prepared with an adjustment to the ship date or features cut.

Quality problems do not arise overnight. As a team, we need to monitor the quality of the product, defect injection rate, and defect detection yield. For the project to be successful – the team must inject fewer defects and find defects early.

Mistake-Proofing

(Poka-Yoke)

It is impossible to eliminate all errors from any task performed by humans. Nobody wants to make mistakes. People do not say, "I want to make mistakes". No one plans to write a defective code; but Murphy's Law intervenes, and a series of seemingly unavoidable mistakes will introduce defects into the product. (Murphy's Law: "**If anything can possibly go wrong, it will go wrong**"). These mistakes happen so often that we generally spend more time fixing these bugs than creating new programs.

No Finger Pointing - Recognize that it is natural for people to make mistakes. The old way of dealing with human mistake was to scold people, retrain them, and tell them to be more careful. Blaming workers not only discourages them and lowers morale, but also does not solve the problem. You cannot do much to change human nature, and people are going to make mistakes. If you cannot tolerate mistakes – then you should remove the opportunities for making mistakes.

Can we defeat Murphy's Law?

Can we eliminate these costly mistakes?

Can we prevent defects without doing 100% inspection or testing?

People unfamiliar with mistake-proofing techniques usually claim that this is not possible. However, use of simple mistake-proofing techniques can put an end to many repetitive costly mistakes. In order to eliminate mistakes, we need to modify processes so that it is impossible to make them in the first place. Mistakes will not turn into defects if feedback and action takes place at the source of the mistake. Find ways to keep mistakes from becoming defects!

Mistake-proofing is to engineer the process such that "**there is only one way to do it – the right way...**" Mistakes can be prevented or immediately detected and corrected. It emphasizes the creation of processes that can minimize the possibilities of human error. Many repetitive tasks that depend upon the memory of a person are built into the process itself.

Mistake-proofing does not require extensive engineering efforts. These techniques are often inexpensive to develop and implement. A simple idea can be a great way to mistake-proof a problem. Mistake-proofing techniques can be applied wherever there are possibilities of human error. The opportunities for error are identified and error prevention techniques identified to eliminate the potential for error at the source. Using prevention and detection techniques, we could achieve the mistake-proofing solutions in almost everything we do.

Mistake-proofing solutions are best developed in a team environment using a structured problem-solving approach. Ideally, people using the process or responsible for the process should be involved in this.

Learning from Defects

(Six Sigma)

Defects are inevitable

It is a fact that no software can be guaranteed 100% defect-free. When everyone feels that software will have defect no matter what we do, we stop taking action to prevent them from appearing. In fact, defects are inevitable. However, if we think that way, we make it horribly true. Simply fixing the bug is not enough; take a moment to figure out why the bug happened. If we analyze our mistakes and think about the process, we can eliminate most of these mistakes. All processes have the potential for defects. Hence, all processes offer an opportunity for the elimination of defects and the resultant quality improvement.

Whenever you encounter a bug or defect, ask yourself:

What assumptions were wrong?

What rules did I break?

How could I have detected this bug earlier?

How could I have prevented this bug?

Fixing the root cause

Detailed root cause analysis (using Six Sigma methodology) of all defects should be done on a regular basis. (Don't wait for the project postmortem.) Based on the findings of ongoing analysis, appropriate processes need to be updated to ensure these kinds of defects do not occur and if they occur, to catch them early. This will help the team to prevent these defects in the future. When everybody starts sharing this data, all the teams will be enabled to prevent a majority of defects. As a result, estimates will become more accurate, work will be done right the first time and the team will ship high-quality product on time.

Implementing TSP/PSP in organizations

As with any new methodology or model, you need to have a strategy and a plan to roll it out to the organization. The TSP introduction approach must be unique for each organization.

Start by establishing short term and long term goals. A useful immediate goal is to prototype the TSP in your organization. Longer term goals are needed to guide the broader TSP introduction strategy.

The steps are:

- define the overall strategy
- select the introduction approach
- assign responsibilities
- allocate resources
- make a detailed plan
- implement the plan

1. The introduction approach

First Steps

- a. Train a few pilot TSP teams and their managers
- b. Launch the teams
- c. Gather data and evaluate the results
- d. Train internal instructors and coaches

Start with two to four modest-sized pilot projects for TSP.

- a. Select teams with members and managers who are willing to participate.
- b. Pilot team size should be 3 to 15 team members
- c. Pilot project duration 4 to 18 months schedule
- d. Software-intensive new development or enhancement
- e. Representative of the organization's work
- f. Define the prototype goals
- g. Establish responsibilities
- h. Get started

2. Required Training

- a. Executives
 - i. Executive strategy and planning seminar: 1 1/2 days
- b. Project Team members
 - i. (Managers) Management training: 2.5 days
 - ii. (Developers) Two-weeks PSP course for software professionals
 - iii. Two-day personal process course for other professionals
- c. Internal PSP instructor & TSP coach training
 - i. PSP instructor: 5 days (two week PSP course is pre-requisite for this)
 - ii. TSP coach: 5 days (PSP Instructor is pre-requisite for this)

3. Motivate Participation

Developers are reluctant to participate in experimental programs.

- a. Emphasize that management has selected the TSP as a new direction for the organization.
- b. Stress the opportunities to:

- i. Provide TSP experience for the entire organization
- ii. Establish a benchmark for future teams
- iii. Learn a new and exciting technology

There are six principal considerations for a successful transition.

- Name a transition champion
- Define line management transition responsibilities
- Allocate resources
- Establish success criteria
- Sustain the TSP
- Maintain continuing oversight

1. Transition Champion

Appoint a transition champion - Without one, little will get done. The TSP champion is responsible for:

- c. Establishing the transition plan
- d. Negotiating this plan with the product managers
- e. Reviewing the plan with senior management
- f. Monitoring and guiding the transition effort
- g. Leading the dedicated transition resources

2. Line Management Responsibility

- a. The only people who can change the behavior of development groups are the group managers
- b. The transition team can support these changes but cannot make them happen
- c. In periodic performance reviews, regularly ask the line managers for transition status reports

3. Line Management Responsibility

Improvement takes work - Trained and qualified professionals must:

- a. Teach the courses, coach teams, draft policies, and provide guidance and support
- b. Get help in solving problems
- c. Lead the transition effort
- d. Monitor and report on the work
- e. The transition team must have sufficient resources. Small groups (10-15 TSP Projects in a year): 2 TSP coaches. Larger groups: build toward 4% of the professional population

4. Establish Success Criteria

Define the truly successful outcome - Be specific:

- Percent defect free product
- Service and support cost improvements
- Customer satisfaction measures
- Cycle time acceleration

Publish these goals and consider them in reviews and evaluations.

5. Sustaining the Momentum

As with any disciplined activity, the TSP needs continuing reinforcement from management:

- h. Establish regular quarterly management reviews
- i. Review project performance
- j. Examine key process measures
- k. Establish and review benchmark comparisons
- l. Identify, recognize, and reward superior individual, team, and management work

6. Maintain Continuing Oversight

While goals, responsibilities, and resources are essential, they are not enough to sustain a major behavior change. If you delegate transition oversight and appear to lose interest

- m. Progress will be slow
 - n. The effort will likely soon die out
- If you take all of these steps and show continuing interest, TSP transition will be rapid and effective.

Message to remember

The key to TSP introduction is management priority and support. Transition will be rapid and effective if you:

- a. Set and track goals
- b. Establish clear responsibilities
- c. Measure and monitor the work
- d. Recognize and reward superior performance

Conclusion

Delivering a high-quality product on time is not possible unless we have a good process in place and learn from our mistakes and are able to manage effectively.

$$\text{Output } (O) = f(G,I,P,E,M)$$

So, if we expect good (O)output – we should have achievable (G)oals, good (I)nputs, data based (P)lanning, right (E)xecution & facts based (M)anagement. I have found TSP and Six Sigma to be great tools to achieve this consistently.

With the use of TSP/PSP, I have seen the estimation accuracy improving over time. As soon as the team starts collecting data on their projects – you get valuable information and your estimates are more predictable and achievable.

While the relative importance of each quality dimension may change in proportion to the others, the absolute importance of each is high. One cannot succeed unless the capability of the product development group in each of these three areas is very high. The cost of shipping a product with a recall-class defect can be fatal to the profitability of a product group. The cost of fixing a problem in the field is extremely high. Most defects also have an associated cost in customer support. Finally, if a product is late, it may never re-capture the market share it loses to a competitor. And if the product is lacking in imagination, features and functionality, it will have a very short life.

Finding errors at their source and preventing their conversion to defects is the fundamental principle behind mistake-proofing. It is a proactive approach to quality control, since it builds quality into processes. Traditional methods have been reactive, relying on rework and repair as a solution to defects in products. Mistake-proofing eliminates defects rather than relying on inspection and test to detect and correct them. Inspection cannot identify all defects. Inspection is expensive and time consuming and we all know 100% inspection is not practical.

Regular root-cause analysis should be done for the defects found in development and in production. Only if we do this and improve the process – we can move towards the goal of developing nearly bug-free code and shipping high-quality and reliable software.

References

- [1] Humphrey, W. *Introduction to the Team Software Process*. Reading, MA: Addison-Wesley Publishing Company, Inc., 2000.
- [2] Humphrey, W. *Introduction to the Personal Software Process*, Addison-Wesley Publishing Company, Inc.
- [3] Shigeo Shingo, Zero Quality Control: Source Inspection and the Poka-yoke system, 1986, English translation by Andrew P. Dillon, productivity press.

[4] Peter S. Pande, *The Six Sigma Way: How GE, Motorola, and Other Top Companies are Honing Their Performance*, McGraw-Hill.

Author Profile



Mukesh Jain is Quality Program Manager in Microsoft's Windows Live Platform group in Redmond, WA (USA) working on Quality of Service. Until July 2006, he was Quality Manager in Microsoft India, leading & driving TSP, Six Sigma, SDLC, MOF, Process Improvements and Business Intelligence initiatives. He has 11 years of experience with various positions ranging from Developer, QA, Project Manager, to Quality Manager. He has led multinational companies in India with TSP/PSP, ITIL, Six Sigma, ISO 9000 and SEI CMM Level 3-5 implementation and certification. He has worked in Microsoft USA for 5 years in the Business Productivity division before relocating to India in June 2004.

He holds an Engineering degree in Computer Science. He is an SEI Authorized TSP Coach, PSP Instructor, SEI Certified PSP Developer & Engineer, ISO 9000 Internal Auditor, CQA, CQIA, CSQA, CSTE and Master-Microsoft Office Specialist.

He served the American Society for Quality (ASQ) as Secretary (2000-2003). He served the International Society for Performance Improvement (ISPI) as Vice-President (2001). As an executive committee member, he organized TUG Asia 2005 in India in March 2005. He has written several papers on the subject of software quality and project management and presented them to Microsoft and other companies and at international conferences, including IEEE, QAI, ASQ, SPIN, PNSQC, STEP-IN forum, PMI & SEI.

Mukesh Jain

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
Phone: +1-425-421-6193
Mukesh.Jain@Microsoft.com

Taking Ownership for Software Development

Jim Brosseau, Clarrus Consulting Group Inc.
jim.brosseau@clarrus.com

Abstract:

The agelessness of books such as Peopleware and the Mythical Man-Month is a result of both the practicality of the advice provided and the ongoing lack of adoption of the principles espoused therein. We all read these and envision a better work environment, and then lament that our managers aren't taking charge for change. Virginia Satir's note that "Familiarity is more powerful than comfort", is just as relevant in software development as anywhere else. We all need to recognize our role in shaping and enhancing our work environment - it is not simply a management concern. Even as individuals we can significantly improve our productivity, and we are the ones that are best suited to propose and contribute to changes to the team as a whole.

This session brings the Call for Change down to the masses and away from the 'pointy-haired boss'. Individual and team challenges are rampant in the industry, the gains to be made are significant, and we can contribute a great deal to change, even under our bosses' radar. Our motives and attitudes, skills and team relationships are all fair game to explore and expose opportunities for change, and we need to take personal ownership and responsibility for the results. Effective and productive – successful - software teams are more than merely a management concern.

Our current reliance on tools and methodologies is misguided in that it is largely prescriptive, and attempts to remedy this from the Agile camp have gone too far in the other direction. In both cases, the issue is not the original thinking from these camps, but how this is translated to practice by most teams – superficial understanding of the rationale and applicability behind well thought-out approaches drives most teams to implement these changes in a dysfunctional manner. We all need to take responsibility for explicitly, consciously managing our relationships and the approach we use as a successful software team.

Taking Ownership for Software Development

Background

The agelessness of books such as Peopleware and the Mythical Man-Month is a result of both the practicality of the advice provided as well as the ongoing lack of adoption of the principles espoused therein. We all read these and envision a better work environment, and then lament that our managers aren't taking charge for change.

Most of us have heard of and read the CHAOS Report[1] from the Standish Group, a lot of us have read what Steve McConnell[2], Jim Highsmith[3] and many others have to say, we're all involved in a conference here, yet most of us have still been (or currently are) involved with organizations where getting good software out the door is akin to pulling teeth. Why is the industry in such sad shape?

Why does Fred Brooks say that “The gap between the best software engineering practice and the average practice is very wide – perhaps wider than in any other engineering discipline.”, and why is that statement just as relevant today as it was 30 years ago?

Why do so many people still smoke, when we all know that it is likely to lead to premature death? An appropriately similar question, in that in both areas, we know there is a better way, but we fail to break our old habits.

I believe it was Virginia Satir (a family therapist, but the analogy is apt) that suggested that “Familiarity is more powerful than comfort”, and that seems to be the case for the software industry as well. There is some overwhelming inertia holding the industry back from realizing its potential, despite our not being in the comfort zone.

Much of the advice provided through the ages is targeted at management, as the stewards of change in an organization. Unfortunately, it is not clear that this is the place to start. At the top of the food chain, does management really perceive any value in changing the working conditions of the masses?

After all, they are at the top of the food chain – they are where they want to be. The call to ‘leaders’ for change has failed for decades.

There Are Times One Could Even See a Conspiracy

Slavery was abolished in North America in the early 1800’s. While there are surely places around the world where living off the toil of others ranges from being explicitly sanctioned to implicitly tolerated, there has been a general enlightenment in many cultures that mutual benefit derived from reasonable working relationships is a good thing.

Unions and guilds have long been effective in allowing workers to bond together to support their common objectives, with working conditions being a long-standing struggle between the unions and management.

Governments get into the fray as well, and in this province we have the Employment Standards Act. With very clear standards regarding working conditions, it establishes requirements that corporations must comply with to ensure that their employees are treated fairly. So far, so good.

There are exemptions to the act, established to manage the fair and equitable relationship between employer and employee, generally expressed to ensure that the employee is treated fairly. Except in High Technology [4].

For high tech, the act explicitly removes many of the elements that would be considered reasonable work conditions. I would expect there are similar provisions in many, if not all other provinces in Canada and in other areas worldwide.

While I do not know how this legislation came to be, I would guess that tech executives lobbied strenuously to allow for these exclusions so their companies could remain competitive. It seems the perception is that in this tech sector, it is necessary to work under insane conditions to get our products built, and the infrastructure to ensure we are being treated fairly has been largely removed. I'm sure many of us have experienced the result of this to some degree.

The management culture of pushing tech employees hard as a means of getting things done is ludicrous and remains far too common. Attempts to modify the culture are often met with cries of "Get a spine" or "There's a lineup of people waiting to work here", and to some degree, that demand simply fuels the fire. The problem spans the range of safety critical systems to games development, and all spots in between. I was recently speaking with someone from a large global consulting firm, and he indicated there was explicit pressure to put in more hours beyond the standard 40-hour work week – it is an unwritten standard.

For the most part, we have been expecting management to act as stewards of change. How well has this worked?

Typical Solutions and Their Shortcomings

We have a tendency to lean on tools and technology for solutions. From the venerable waterfall model, through spiral and iterative to agile, the models have evolved over the years to acknowledge the need to address change (product change), but they remain a small part of the equation.

We tend to purchase tools to solve our problems – requirements tools that we don't know how to populate, CM tools that we work around or completely ignore, problem tracking tools that don't capture all the issues, project management tools that provide beautiful Gantt charts that are obsolete by the time they are posted. We constantly search for a quick fix that doesn't exist.

We all have a different perspective of what success means, of what role we play in the big picture. These different perspectives are a major source of conflict. We are not coordinated, our emphasis on technology and engineering solutions neglects one minor detail – we are humans trying to collaborate in an innovative fashion to solve difficult problems.

A Thesis

We all need to recognize our role in shaping our work environment - it is not simply a management concern. Even as individuals we can significantly improve our productivity, and we are the ones that are best suited to propose and contribute to changes to the team as a whole.

It has taken close to 25 years of my experience in the software industry to learn some fundamental things: there really is not (and never will be) a silver bullet – it's one thing to read some time ago that there were no silver bullets, another thing to have hopes dashed through trial and error. Many others with broad experience have come to the same conclusion.

The other thing I have come to learn is that we can't expect someone to come along to save the day – it is up to us – each and every one of us – to contribute in a positive way to improving the experience of software development any way we can. Having worked with a number of those that believe themselves to be part of the solution for this industry, I've come to the conclusion that we are all similarly challenged to some degree. Most of the consultants suffer from many of the same issues as the clients they are proposing to help – we all live in our own box, and we all find it difficult to see things from outside our box.

If we are all in our own box, and there is no silver bullet on the horizon – what to do? The solution to our problem is a coordinated team effort to make our lives better. Some of us need to grab oars, some of us need to start baling, and a few might contribute best by staying out of the way if they can't relate to either of the first two options.

We need to drive change ourselves – we can't blame external circumstances forever. We can't rest on the relative youth of the software industry – while medicine is sometimes called the youngest science, it is done in the context of there being a lot to learn and a high rate of change, not as an excuse for our failures.

While the software industry also has a lot to learn and a high rate of change, we are far more passive. We have to get off our collective butts and recognize that we need to drive change for the better – we are the solution we have been waiting for all this time, we have to accept the responsibility to make our software development experiences more effective – and we need to start today.

We all have something to contribute, even if it is simply a new set of eyes to look at a problem from a different perspective. Highly effective teams leverage the combined individual strengths to form a stronger collective.

As Peter Drucker noted,

"Management means, in the last analysis, the substitution of thought for brawn and muscle, of knowledge for folkways and superstition, and of cooperation for force. It means the substitution of responsibility for obedience to rank, and of authority of performance for the authority of rank. Whenever you see a successful business, someone once made a courageous decision." [5]

In this light, we all need to think of ourselves as managers – of our own futures.

The best way to gain security for your future is to take responsibility – to acknowledge that the results you reap will be primarily influenced by your own actions. To look elsewhere is akin to searching for excuses, not solutions.

A Solution Framework

The solution space we have traditionally searched is too narrow – tools, frameworks, methodologies are insufficient – we need to add the elements of different perspectives and other aspects of the human condition, along with transparent communication to the mix to identify lasting solutions that will work for everyone.

Here is one way to look at the range of issues of the software development space:

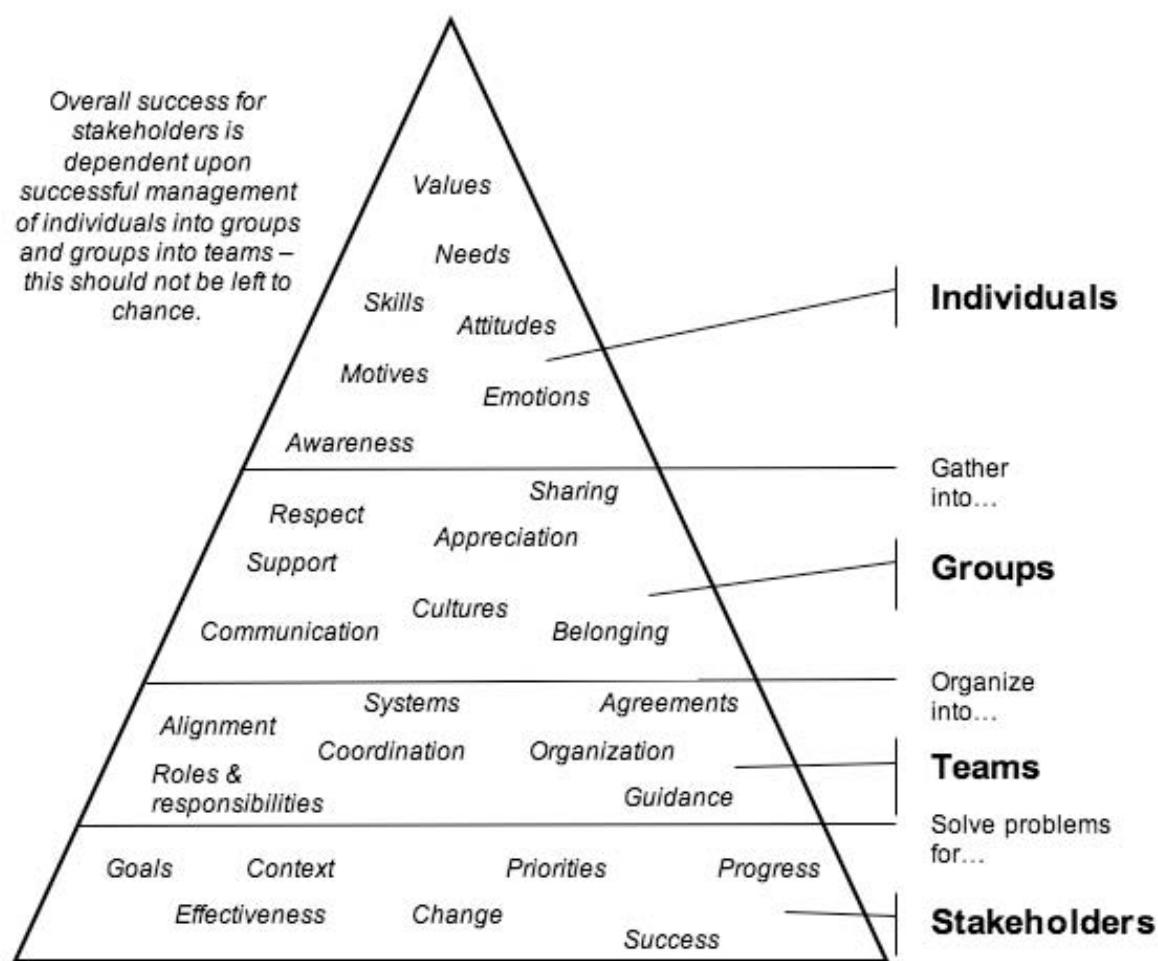


Figure 1: The Range of Issues in Software Development

As with any human endeavor, software development starts with individuals. They each have their unique set of values, motives and attitudes based on their experiences, and a set of skills based on their training and aptitudes. Everyone else's personal environment is as real as yours,

and any affront to that environment, any lack of appreciation for ‘where you are coming from’, is not taken well. Emotions are far more tactical, but just as real and important to consider.

These individuals will gather and interact into groups, where we start to deal with interaction and the forming of relationships. We’ve all been involved with relationships that have been effective and those that are less so, and the ideal is to move into relationships that are positive, with mutual respect and a sense of belonging.

When so inclined, these groups will organize into teams. Here we are dealing with the coordination of a diverse set of individuals. This group leans on systems and guidance in order to prepare them to align towards a common vision – to be ready and able to solve a problem.

With appropriate direction, these teams will work together to solve problems for stakeholders. Not only have they evolved systems for interaction, but they are now working towards a specific goal. There is a notion of progress towards that goal, and eventually an ability to determine whether the endeavor was a success or a failure.

This progression is most effective when taken in this order, with nothing implicitly left to chance. Each piece taken for granted is an invitation to failure, particularly if success and failure is measured in dimensions beyond having shipped on the target date.

When looking at the breadth of issues across this continuum, typical approaches such as the incorporation of a packaged process framework or the purchase of a tool, tend to provide superficial support at best. While they can be seen as critical for structuring and managing huge amounts of data on large projects, all tools depend on the appropriate data actually being provided to the tool. For requirements or design tools and for packaged processes, the overall results will be no better than the quality of the information that is used as input.

Tools, frameworks and methodologies tend to emphasize those aspects of organization that are in the area of teams in the above diagram. Necessary, to be sure, but hardly sufficient. Have you ever been involved in the deployment of a tool or methodology without 100% buy-in, for example? Many purchased solutions of this type are no longer used even a year after purchase.

Our current reliance on tools and methodologies is misguided in that it is largely prescriptive. Attempts to remedy this from the Agile camp have gone too far in the other direction. In both cases, the issue is not the original thinking from these camps, but how this is translated to practice by most teams – superficial understanding of the rationale and applicability behind well thought-out approaches drives most teams to implement these changes in a dysfunctional manner.

Software Is a Team Sport

Our motives and attitudes, skills and team relationships are all fair game to explore and leverage as opportunities for change, and we need to take personal ownership and responsibility for the results of our behaviors.

Effective software teams are more than a management issue, and we cannot equate software development to the assembly-line floor, despite attempts to do so such as the mapping of the Six Sigma manufacturing quality processes to software.

Procedures can serve as a guideline for what to do and provide structure for managing our information, but they are insufficient to guarantee success in all but the most trivial cases. Projects and tactical situations are all different, and a key component of these differences is the evolution of relationships as teams work together. If not carefully managed and nurtured, these relationships will almost surely erode over time.

Software is a creative endeavor done with real humans, with emotions, feelings, needs, and concerns that are both diverse and important. As Maslow suggests in his hierarchy of needs, we cannot rise into the ranks of optimized teams until the physiological and safety needs have been properly dealt with. [6]

What We Can Do

We need to take responsibility for explicitly, consciously managing our relationships in our teams, and to proactively design the approach we use as a team to develop software. This demands participation from all stakeholders involved, and appreciation from all parties of the contribution (technically and emotionally) from everyone at the table.

Communication needs to be clear, open, and transparent. Hidden agendas cannot be tolerated, and we all need to be more effective with the skill of active listening – we need to be capable of empathizing with the positions of others before we pitch our case in discussions. Listening needs to be more than biding time to formulate our counter-argument or waiting for an opportune moment to blurt out our argument or tolerating that noise from the other person's mouth.

Each person needs to respect differences within the group, and be sure to address concerns from these different perspectives. *Vive la difference*, it is these differences that make for unique, innovative solutions, long-lasting and strong relationships, and teams that will succeed now and survive to do so in the future.

We all need to be able to consider the ramifications of our actions and understand the rationale behind what we are being asked to do. If something does not fit well with our mental model of appropriateness, we need to be able to speak up. Silence is not golden, the absence of conflict is too often apathy. If the fit cannot be made, we need to be prepared to talk with our feet.

While this approach is a tougher, far deeper commitment to change, the solution is far longer lasting and significantly more rewarding. We need to stop playing with the symptoms; there really never will be a silver bullet. We need to take charge, work together to build an effective team, and finally take ownership for software development.

References

1. The Standish Group. *The Chaos Report*. The Standish Group, 1994
http://www.standishgroup.com/sample_research/chaos_1994_1.php - the Chaos Report has

been released periodically since 1994, with largely the same damning evidence that we are still challenged in the software industry.

2. McConnell, Steve. *Rapid Development*, Microsoft Press, 1996 – this is possibly the most relevant book from Steve about the issues and solutions for improving software development teams.

3. Highsmith, Jim. *Agile Project Management*, Addison-Wesley, 2004 – As with Steve McConnell, Jim has written a number of important books about the challenges in software development, and provides some proposed solutions.

4. BC Ministry of Labour and Citizens' Services. Employment Standards Regulation – Exclusions for High Technology Professionals (<http://www.labour.gov.bc.ca/esb/hightech/regulat.htm>) - this identifies the exclusions to the labour regulations in the province.

[5] Drucker, Peter. *Management Tasks, Responsibilities, Practices*, Collins, 1993

[6] McConnell, Steve. *Software Project Survival Guide*, Microsoft Press, 1998 - Steve does a good job of mapping Maslow's Hierarchy of Needs to the software world, but it is worth recognizing that the original Hierarchy remains relevant.

Making the Most of Community Feedback

Dave Liebreich, Mozilla Corporation

Various Authors, Mozilla Project

Abstract

The Mozilla open-source software project, which includes the Firefox browser, has a long history of quality-related feedback from its community. From its debut in 1998 when Netscape open-sourced their browser code to the world, many thousands of volunteers ready and willing to contribute to the future of the web have provided direct feedback to the mozilla.org development community. Many millions of users have downloaded and run the mozilla.org applications, and through such use have provided feedback as well.

This feedback is a competitive advantage for the project, providing insight into quality-related issues that would be difficult to obtain with a small, dedicated team. But the sheer volume of the feedback can make it difficult to process, identify, and prioritize specific issues to address.

In this paper, the authors present background information about the mozilla.org project, descriptions of the various feedback channels and tools, and walkthroughs of the processes and guidelines project members use to convert this feedback into prioritized tasks.

Bio

Dave Liebreich has almost 20 years experience supporting the development of complex applications and distributed networked systems. He is currently a Test Architect at the Mozilla Corporation, working on improving and extending the testing for mozilla project products such as Firefox and Thunderbird.

Copyright 2006 Mozilla Corporation

Background

The Mozilla open-source software project includes the Firefox web browser, the Thunderbird email client, the Bugzilla bug tracking system, and many other web-based applications and technologies. Firefox, the most popular application in the project, is used by many millions of people throughout the world. Hundreds of thousands of people download Firefox beta releases, while tens of thousands download alpha and other early-milestone releases. Thousands download and test the nightly builds, and hundreds submit code to fix defects and add new features.

Most of the developers and testers are volunteers, devoting a portion of their free time to working on the project. Many large companies, including Novell, Sun, IBM, Oracle, and Google, employ individuals to work full-time on the mozilla.org codebase. Smaller companies whose products incorporate elements of the mozilla.org codebase also contribute feedback, testing, and code.

Feedback from these user, test, and development communities has been crucial in improving the quality of the applications. Translating that feedback into Bugzilla entries, the form used by project members to track work on the code, has been an ongoing challenge.

Feedback Channels

Bugzilla

One of the project norms is that developer work is coordinated in Bugzilla, a web-based defect and work tracking tool.

Early in the project, the development community chose to coordinate all work through Bugzilla. Bugs were filed for all changes to the code, including new features, changes to the build system, changes to the mozilla.org website, and fixes for defects.

Our instance of Bugzilla is almost completely open. A Bugzilla account is not needed to view bugs, so anyone who has a bug number can go look at the bug. They can read through the comments, look at the proposed patches to the code, and see exactly who is working on the issue.

Anyone can create a Bugzilla account for themselves, and the default permissions for new accounts include the ability file new bugs, comment on existing bugs, and add themselves to the list of people notified by email whenever a bug changes.

Community members -- mostly technically experienced software developers themselves -- were encouraged to record their feedback in bugzilla. In order to help prevent duplicate and invalid bugs from being filed, a Netscape QA Engineer (Eli Goldberg) wrote up bug filing guidelines (<http://www.mozilla.org/quality/help/>). These guidelines also help others triage the bugs so developers can better prioritize their work. This document has been maintained by Gervase Markham after Eli left, and links to it have been spread across newsgroup postings and the various mozilla.org websites. Most people who file bugs in Bugzilla have read the guidelines at least once.

Still, Bugzilla remains hard to use. Its complex input form and search mechanisms are discouraging to the less technical members of the community, as well as the more technical members who are in a hurry.

In order to make it easier to obtain feedback, other feedback channels developed and evolved.

With the advent of other feedback channels, the project developed mechanisms and processes to convert community feedback into Bugzilla reports. When bugs are filed based on information from other feedback channels, the source of that information is recorded in the Bugzilla report comments.

Talkback

Talkback, the crash reporting agent distributed with Firefox and Thunderbird, produces the largest number of feedback reports. Talkback was developed by a company named SupportSoft, and the Mozilla Corporation has a license agreement with them to distribute the agent with our products and run a server to collect incoming reports.

The Talkback client is included in official Mozilla builds, which include nightly builds, milestone builds, and release builds on the main development trunk and on the active release branches.

All Mac and Linux users, and 10-20% of Windows users, have Talkback enabled by default. Much like the Windows crash report dialog, Talkback fires up and provides the user with the option of sending in the crash log and adding comments describing what they were doing at the time of the crash. If the user agrees to submit the report, the data is sent to the Talkback server (talkback.mozilla.org).

Reporter

Not all quality issues are crash reports. Often, something just does not work right when viewing or interacting with specific web sites. In order to provide an easy and accurate mechanism to report problems while browsing, Firefox includes a built-in “Report Broken Web Site...” tool.

If a user encounters a problem, he or she can select the Reporter tool from the help menu, answer a few simple questions, and the tool will send the information to the Reporter server.

Discussion Forums

Members of the Mozilla community do a lot of on-line communicating. There are many different channels to support that communication, each of which has its own monitoring mechanisms that guide quality-related feedback to the appropriate system for reporting that information

Web Forums

The MozillaZine Forums are probably the most accessible discussion channels for Mozilla-related topics. Community members often direct new users to these forums for support requests.

The forums are also used by programmers building applications that use the code in the Mozilla codebase, Mozilla developers, and community testers. These more dedicated community members read through the postings and relay the more significant bits on to other areas, including formal bug reporting in Bugzilla.

Newsgroups and Mailing Lists

Mozilla also hosts many newsgroups, all of which have an associated mailing list for participants who prefer email to NNTP (Network News Transfer Protocol) news.

These lists and groups are used to host most of the discussion of the technical issues involved in the Mozilla project.

Many times, a discussion of how to accomplish some technical task reveals a quality issue in the codebase, and one of the discussion participants files a bug.

IRC

A great deal of conversation takes place in IRC (Internet Relay Chat) channels hosted on irc.mozilla.org.

Hendrix

Some users would rather not open another application or log in to the MozillaZine Forums to provide feedback. For them, mozilla.org has set up the Hendrix tool. This tool takes information entered in a simple webpage form, obfuscates the submitter's email address, and posts the information to a public newsgroup.

By scanning through the structured newsgroup postings, mozilla.org community members can more easily identify issues from this otherwise low quality feedback.

Hendrix-based forms are embedded in the default home page for Firefox nightly and candidate builds.

Processing the Feedback

Bugzilla Triage

Hundreds of new Bugzilla bug reports are created each day. Many of them are invalid, or duplicates of existing reports. Some do not have enough information to be of help. Some are work tracking bugs entered by developers. And some are reports of problems that turn out to be high-priority issues.

Developers and testers use the Bugzilla system of products, components, keywords, and tags to search for, identify, and track reports of interest. Very rarely, however, are reports entered with the appropriate settings selected.

A small group of community members performs bug triage, which involves searching for

duplicate reports, marking invalid bugs as such, and classifying the reports using the Bugzilla settings.

One interesting side-effect of an open bug tracking system is that the original bug reporter can view the discussions about the bug and monitor the progress of the effort to resolve it. The original reporter can even see the email address of the developer working on the fix.

Triaging crash reports

The mozilla.org Talkback server receives a lot of crash reports, and the mozilla.org community has developed guidelines and approaches for triaging the incoming information to get the most return on time spent analyzing crashes.

Talkback triage starts at the main Talkback report site. Developers and testers check the list of most frequently reported trunk build crashes around 3 times each week, and branch crashes as often as daily during the release process. The product release teams also use the crash frequency reports in the nomination and approval process for maintenance release fixes.

The people who check the Talkback reports file bug reports in Bugzilla for crashes that contain complete stack signatures, even if there is not enough information to reproduce the crash. Developers and interested community members watch for such bug reports, and investigate the code implicated by the stack signature and other bug reports that might share the same root cause.

For Talkback reports with incomplete or corrupt stack signatures, the mozilla.org community relies on the extra information provided by the end user. A community member usually will include the end-user comments when filing a bug report based on a talkback incident “from the field”, and community members regularly include talkback incident IDs in their bug reports.

Release teams also pay attention to changes in the topcrash lists from release to release. If a topcrash without an explicit fix drops from the list in a new release, members of the team try to figure out which fix might have addressed that crash’s root cause. And new crashes in a new milestone build are given extra attention.

Talkback provides huge value in debugging and isolating stability issues, and it has been used to identify and debug thousands of major crash bugs in Mozilla over the years.

Reporter Triage

The reports on the Reporter server are often used when investigating a problem report from a single user in a different forum. If the site in question is in the “Top 25” list, then the problem is probably not specific to that one user.

Reporter data is also useful when contacting a web site to ask them to change their content so that it will work with the mozilla.org applications. Such evangelism requests tend not to be ignored when they are backed up with a large number of unique reporters of problems.

Discussion Forum Triage

Many experienced community members “hang out” in the various discussion forums. Some actively answer basic questions and help new participants navigate through the wealth of information available about the Mozilla Project.

These people, and other community members who just scan the forums, also take ownership of certain issues, and either file Bugzilla reports, or bring up the issue in a more appropriate (and possibly more technical) forum.

People Providing Feedback

There are long-term mozilla.org community members who have gravitated towards specific product functional areas or product activities.

For example, there are folks who work on layout and rendering issues, finding new bugs and reducing existing test cases. Some other folks work on security, and others work on the build system, and so on.

There are also groups of people who work on testing the nightly builds, looking to be the first ones to report problems.

The Mozilla QA group has organized test days and bugfests to guide new community members in providing feedback.

All of these groups of people make extensive use of the feedback channels listed in this paper.

Nightly Builds

The nightly builds of the mozilla.org products are available to the public. When a patch is committed to the codebase, the original reporter of the associated bug is often asked to check if the fix works for them in the next build.

Summary

The feedback from millions of users is an incredibly valuable asset to the Mozilla Project, and one that requires a significant amount of effort to use effectively. By setting up tools to help users provide feedback, encouraging community members to form a hierarchy of people performing feedback triage, and monitoring feedback reporting sites, the project has been able to convert this feedback into a huge competitive advantage.

Acknowledgements

The following people contributed content and/or provided feedback on this paper:

Asa Dotzler, Gervase Markham, Jay Patel, Eric Shepherd.

Appendices

Mozilla Project “Legal Entities”

- The Mozilla Foundation is a non-profit corporation that provides organizational, legal, and financial support for the Mozilla open-source software project.
- The Mozilla Corporation is a wholly owned subsidiary of the Mozilla Foundation. This corporation employs a small number of people (~50) who dedicate their time to the project. Most of these employees are long-time community members.
- Various non-profit organizations exist to promote, develop, and help deploy Mozilla products in specific regions.

Links to Feedback Channels

- <https://bugzilla.mozilla.org/>
- <http://talkback-public.mozilla.org/search/start.jsp>
- <http://reporter.mozilla.org/app/>
- <http://forums.mozilla.org/>
- <news://news.mozilla.org>
- <https://lists.mozilla.org/listinfo>
- <http://irc.mozilla.org/>

Know Your Code: Stay in Control of Your Open Source

Douglas Levin, James Berets, Black Duck Software, Inc.

dlevin@blackducksoftware.com, jberets@blackducksoftware.com

Abstract

Product developers today are acutely aware of both the opportunities and obligations associated with using open source and third party software. Independent software vendors, embedded systems equipment manufacturers and handheld device makers are all using open source software at unprecedented levels. As members of an active and interconnected global community, combining open source code with their own to make improvements, this process can easily bypass company policies and procedures on software acquisition and licensing review and approval, and the result often is significantly increased business risk.

By combining external components with their proprietary code, companies create a complex mix of intellectual property (IP) that carries licensing obligations with which companies must comply. In a mixed-IP environment, the volume of software licenses to be understood and tracked can quickly become a challenge. Further, those licenses often conflict with one another and can result in software assets with serious intellectual property problems.

To ensure software license compliance, companies need clear visibility into the origins, ownership, and license requirements of each component used. Once companies put in place a system that allows them to know their software code and to remedy any problems with it, they not only reduce their chances of license infringement, which can have business-level implications, they make it possible to increase their use of externally created software in a safe manner.

About the Authors

Doug Levin founded Black Duck Software in 2002 and has been its Chief Executive Officer and President since its inception. Before Black Duck, Doug served as the CEO of MessageMachines and X-Collaboration Software Corporation, two VC-backed companies based in Boston. From 1995 to 1999, he worked as an interim executive or consultant to CMGI Direct, IBM/Lotus Development Corporation, Oracle Software Corporation, Solbright Software, Mosaic Telecommunications, Bright Tiger Technologies, Best!Software and several other software companies. From 1987 to 1995, Doug held various senior management positions with Microsoft Corporation including heading up worldwide licensing for corporate purchases of non-OEM Microsoft software products. Prior to Microsoft, Doug held senior management positions with two startups in California and served as an IT and financial consultant to an overseas development company. Doug is an adjunct lecturer of Entrepreneurship and Management (on leave-of-absence) at the Kenan-Flagler Business School at his alma mater, the University of North Carolina at Chapel Hill. He also holds a certificate in international economics from the College d'Europe in Bruges, Belgium.

Jim Berets is Director of Product Management and Product Marketing at Black Duck Software. He has over 20 years of experience in product management, sales engineering and software development management. Prior to joining Black Duck, as Principal of Beaver Brook Consulting, Jim consulted to technology-industry clients in areas including product strategy and definition, offer development, market and competitive research, and customer acquisition. Previously, Jim was VP of Product Management at CO Space, a data center services company (acquired by Internap Network Services); Director, Sales Engineering, BBN Planet / GTE Internetworking, where he managed the company's worldwide sales engineering team; and a Software Development Manager in BBN Technologies' Distributed Systems Technology Group. Jim holds an SBEE degree from MIT; an MSEE from the University of California, Berkeley; and a CSS in management from the Harvard University Extension School.

© 2006 Black Duck Software, Inc.

Know Your Code: Stay in Control of Your Open Source

Introduction

Product developers today are acutely aware of both the opportunities and obligations associated with using open source and third party software. Independent software vendors, embedded systems equipment manufacturers and handheld device makers are all using open source software at unprecedented levels. As members of an active and interconnected global community, combining open source code with their own to make improvements, this process can easily bypass company policies and procedures on software acquisition and licensing review and approval, and the result often is significantly increased business risk.

Applications using open source are growing dramatically and target both Linux and Windows operating systems. The market research firm IDC recently predicted that the market for packaged software for Linux - including database, application server software, applications, and management tools - was growing at a 44% five-year compound annual growth rate and will reach US\$14B by 2008. These software revenues dwarf revenues for the Linux distributions themselves.

By combining external components with their proprietary code, companies create a complex mix of intellectual property (IP) that carries licensing obligations with which companies must comply. In a mixed-IP environment, the volume of software licenses to be understood and tracked can quickly become a challenge. Further, those licenses often conflict with one another and can result in software assets with serious intellectual property problems. As a result, organizations are left asking themselves:

- How can we maximize the productivity advantages of software reuse, while still assuring that we meet the license obligations of the software?
- How can we identify our intellectual property when we combine it with that of others?
- How can we demonstrate to customers, lawyers, partners, and investors that we have control of our IP?

To ensure software license compliance, companies need clear visibility into the origins, ownership, and license requirements of each component used. Once companies put in place a system that allows them to know their software code and to remedy any problems with it, they not only reduce their chances of license infringement, which can have business-level implications, they make it possible to increase their use of externally created software in a safe manner.

Changing the Development Model

Developers today can easily tap into rich software resources inside their own organizations and from around the world. They obtain software modules and libraries – and even code fragments – from enormous stores of high-quality, re-useable software components. Open source projects are

a consistent source of new, re-usable code, but so too are the growing in-house software repositories and the increasing amounts of modular code from third-parties such as outsourcers.

With such broad availability, developers now focus on selecting the best available components for their projects, and rapidly incorporating them into their own IP to deliver applications that provide optimal functionality, performance and reliability.

The component-based development model has fundamentally changed the software industry. It enables organizations that develop software, either for commercial sale or for in-house use, to accelerate project timelines, improve software quality and reduce development costs. But if not managed properly, the complexity inherent in this new world of ‘mixed-IP’ can pose business and technical risks to an organization. Developers and companies need to avoid the risks and gain the benefits of this promising new approach to software development.

Component-Based Development

A software application’s design, implementation, and maintenance require the investment of precious development personnel, resources and time. Development organizations have long understood the virtues of building new applications by re-using components already built and tested. Indeed, the evolution and rapid adoption of component-based architectures has been driven in part by their effectiveness in promoting economically-significant re-use.

This effectiveness has stimulated the creation of commercial component libraries, which give development teams the option of purchasing pre-built components rather than acquiring the expertise and/or expending the time required to independently create them. In short, the approach can result in faster, less-expensive and more effective software development.

It is therefore appropriate to consider components as intellectual property assets, optimize their usage and protect their integrity. This applies equally to all components, whether they are internally developed or developed externally by a third party.

A development team intent on employing an externally developed commercial component does not typically purchase ownership of that component. Instead, they acquire a license to use that component in a specified manner – perhaps for only a certain number of developers working on a particular project, or with a specified royalty paid for each instance of a shipped product that includes the component.

Thus business judgment is required to ensure that the cost of licensing a commercial component is more than offset by the benefits – the classic make vs. buy tradeoff. Included in the cost analysis must be the effort required to ensure compliance with the license, e.g. limiting the component’s usage to a specific project, or tracking shipments in order to accurately calculate royalty payments.

The Impact of Open Source

Today, open source software has risen to prominence, dramatically increasing the opportunity to re-use existing software. Developers can readily locate potentially useful components from among a wide array of open source outlets. Re-use can take many forms, including bundling independent components, integrating with or using libraries and incorporating source code or source code fragments. In some cases these components can be modified as required to improve functionality, quality, performance or footprint. In many organizations, a developer's skill with Google and SourceForge.net is as important as his or her knowledge of software architecture and implementation.

As with commercial components, the ownership of externally developed open source components and fragments remains with their authors. While most of these authors allow the commercial use of their software without initial payments or royalties, many have chosen to impose other constraints, such as attribution, usage reporting, publication of modifications and improvements, license publication, or in some cases, the obligation to re-license the software as open source.

Such constraints are imposed by means of licenses, some of the most popular of which include:

- The Apache Software License (ASL)
- The Common Public License (CPL)
- The General Public License (GPL)
- The Mozilla Public License (MPL)
- The New BSD License

The Linux operating system, for example, is licensed under the GPL. A more complete listing of open source licenses is provided at <http://www.opensource.org/licenses>.

Open Source License Compliance

Development teams that incorporate open source components or fragments of open source components in their projects must comply with the terms of the licenses associated with those components. This can be challenging for several reasons.

- There is wide variation in the obligations imposed by open source licenses, ranging from the BSD license (which has few obligations) to the GPL license (which has many).
- Some open source licenses are legally complex, introducing constraints whose business implications may not be obvious to a developer choosing to re-use a component. If a company's potential IPO, acquisition or merger value is based on its proprietary product, and that product is tethered to license obligations, the actions may result in the inadvertent dilution of the organization's IP. The licenses of some commercial and open source components are mutually incompatible.
- The origin of a particular source code fragment may be difficult to determine, effectively obscuring its license obligations.

- Discovering the need to comply with a license late in a project's lifecycle can produce disagreeable tradeoffs, e.g. publishing all of the project's source code vs. increasing time-to-market by months while a component is replaced.

In addition to the legal obligations imposed by licenses, developers who incorporate open source components into an organization's projects may either be obligated by the terms of the license or feel a moral obligation to give something back to the open source community.

Management Alternatives

Organizations can react to the challenges of open source software licenses in one of three ways. Some organizations turn a blind eye, ignoring the issue until a catalyzing event or crisis occurs. But the resulting misfortunes – major code rewrites, embarrassing negative publicity, delayed sales and/or failed acquisitions – make this an increasingly unjustifiable approach. This is especially true in the new environment of increased business transparency, executive responsibility and potential shareholder lawsuits.

Other organizations take the Draconian approach of banning all open source software re-use. This strategy is flawed because it is difficult to enforce, decreases productivity and agility compared to organizations that successfully re-use externally developed components, and it often de-motivates development teams by requiring them to apply scarce resources to wheel re-invention rather than to moving forward. Further, both of the above approaches are ineffective because they fail to recognize the reality that open source and component re-use are here to stay.

The third and recommended alternative is to encourage the re-use of both internally developed and externally developed (commercial and open source) components, while establishing controls at critical points in the project lifecycle, for example:

- When components are first added to a project.
- When internally developed components are created or modified.
- At every build.
- At each phase transition in the development process.
- When considering the contribution of a component to an open source project or the transfer of its ownership to another party.
- Before acquiring a significant ownership interest in a software component.

It is important to note that identifying problematic licensing issues early in the development cycle is tantamount to detecting serious software defects: the earlier a problem is detected, the less expensive it is to fix. While IP controls late in the development cycle – during Q&A or release assembly, for example – are better than none, the earlier they happen in the lifecycle the better.

In the remainder of this paper, we will outline several Best Practices that facilitate the management of software IP in the modern development organization. While all of these Best Practices encourage a focus on license compliance throughout the lifecycle, each organization should adopt the subset of Best Practices that meet their business needs. For example, an

organization may wish to give its developers the flexibility to build prototypes using any open source code with no pre-approval, but specify a development phase transition beyond which all externally developed components must be approved. This provides the benefits of speed and efficiency to the developer, with the protection offered by a formal review.

It should be noted that, while all of these processes can be built and executed manually, their adoption and usage will be more effective and efficient when supported with an automated software compliance management system like Black Duck Software's protexIP.

Preparing for IP Management

When beginning to adopt Best Practices, several ‘getting started’ tasks should be considered by the individual or teams responsible for development and licensing. An organization intent on managing its software IP should identify the responsible business, legal and technical individuals who will be involved in the process. The organization also should give them authority to use of software for each active project, and commission them as a group to oversee and manage the planning, implementation and ongoing management of the process.

A good first step for the team is to define the boundary between internally developed and externally developed components (e.g. for business units, contractors, outsourcing organizations). For example, a small organization that prefers taking a conservative approach may deem all of the code developed within its walls to be “internal.” Conversely, that company would view as “external” all software brought in from any outside source (e.g. licensed proprietary, open source, contractors’ work product, etc.).

A more trusting organization might extend its view of “internal” software to include licensed proprietary software and software developed by its contractor and outsource partners. On the other hand, a department of a large corporation may want to consider its department “internal” and consider everyone else, including other departments in the same company to be “external.” Business judgment must be used to determine where the boundary should lie.

Another key task in the preparation process is for the team to identify the development process phase transitions at which component re-use reviews will be conducted. The team also should define criteria for designating an internally developed component as sensitive (due to intellectual property value for example embodiment of trade secrets or patents, or risk), and develop and maintain a list of these sensitive items.

The organization also should consider establishing and maintaining lists of:

- Licenses that are prohibited by the organization.
- Externally developed components that, based on previous reviews, are approved for use in projects, and the situations in which use is approved.
- Internally developed components that, based on previous reviews, are approved for contribution to open source projects or disposition to third parties.

Once these lists have been created, the organization can use them to conduct an initial assessment of its existing code base(s). In this important preparatory step, the organization identifies and establishes the baseline pedigree, licenses and components in use. As with any process improvement, an acceptable alternative approach involves introducing these steps incrementally over time.

Best Practices for Managing Software Intellectual Property

** Re-use existing components wherever appropriate*

Whenever a development team considers adding new features or refining existing functionality in a project, it should explicitly seek internally developed and externally developed components that could accelerate delivery. The team should establish criteria for selecting and procuring these components. As would be expected, any component under consideration that fails to meet functionality, performance, reliability, and maturity or risk requirements should be eliminated.

The team should also eliminate any externally developed component whose license is on the prohibited license list, whose license obligations are financially or legally incompatible with the project's business objectives, or that uses other external components or fragments whose licenses are similarly unacceptable. For example, an organization developing a product that will be delivered under a proprietary license needs to be certain that any open source or proprietary licensed code that is incorporated can be safely included without causing an irreconcilable conflict between licenses. Any components that pass this initial test should be subjected to a make-buy analysis to determine whether or not its acquisition makes sense from a business perspective.

** Track and control changes to internal components*

To protect the organization's critical intellectual property, the creation and modification of all internally developed components should be tracked by recording a timestamp, the names of each author and the applicable objectives and constraints. If a newly-created or modified component is suspected to be sensitive (e.g. a patent is sought, the code embodies algorithms that give the company significant competitive advantage, etc.), the project's legal, business and technical reviewers should be convened. If these reviewers deem the component to be sensitive, they should add it to the organization's list of sensitive internal components.

** Control re-use of sensitive or external components*

By assessing sensitivity and license obligations at the point where a component is first being considered for re-use, decisions can be based on verifiable facts, eliminating last-minute surprises, guesswork, compromises and risk-taking. This dramatically reduces the risk of schedule slippage, cost overruns and damage to the organization's reputation. It also helps prevent the inappropriate re-use of critical intellectual property.

For each component that a project's development team proposes to use within a project, the team should understand:

- The intended use and rationale for inclusion.
- The component's sensitivity.
- How the code will be incorporated.

How the team deals with the component will depend, in part, on whether plans call for that component to be used temporarily or permanently. For example, the intent may be to use that component for a limited amount of time only to speed up prototyping or to advance the early phases of the development cycle, but not be intended to be made a permanent part of the code base.

Another determination the team should make is whether a component will be used only as is, or if modifications will be allowed, and if so, under which approvals.

Development teams should describe the method of integration that will be used to incorporate the component into the project. This is an effective step because different types of integration can create different licensing obligations (e.g., an unmodified copy will be used, a source code fragment will be copied and merged with other source code, an executable will be packaged with the distribution, a statically linked library will be used, a dynamically linked library will be used, etc.).

To achieve greater control over component re-use, teams should also take the following actions:

- Determine whether the component has been previously approved for the proposed form of use (by consulting the approved externally developed components list).
- Choose the component's version and understand its license obligations as well as those of any externally developed components or fragments it contains or depends on. This requires a declaration from the supplier of any externally developed component whose source code is unavailable for direct inspection.
- Understand all potential incompatibilities between the component's license obligations and the license obligations of other externally developed components included in this project.
- Present the above information to the project's legal, technical and business authorizers and request approval to use the component as described.

Approvals should be reflected in the appropriate organization-wide lists:

- If the component is internal and sensitive, the list that covers these items should be updated to note that component's inclusion in this project.
- If the component is externally developed, its metadata and approval details (origin, version, license, license obligations, permitted forms of use, permitted projects, approvers, approval date) should be recorded in the list of approved external components.

** Verify every build and release*

Inspecting the code base on a regular basis decreases the likelihood that unexpected components will be introduced without being noticed. Therefore, at the creation of each project build or at release assembly, the development team should verify that:

- No unapproved sensitive internally or externally developed components or fragments have been added to the project.
- No unapproved changes have been made to sensitive internally developed components.
- No changes have been made to externally developed components whose form of use preclude changes or require that all changes be approved.

Any inappropriate additions or changes discovered during verification should be immediately addressed, either by obtaining approval from the project's legal, technical and business reviewers, or by backing out the offending modification. The root cause of any component misuse should be identified and corrected to ensure no subsequent regression.

By promptly and diligently assessing every build and release, the development team will be able to detect errors when they are least expensive to correct. At the completion of each build or release, the key metadata for all externally developed components should be recorded in the associated bill-of-materials. This enables demonstrable compliance with license obligations, and eliminates any uncertainty caused by changes between project releases by providing a clear audit trail.

** Review compliance at project phase transitions*

As a project completes a major development process phase, its legal, technical and business reviewers should do the following:

- Verify that no unapproved sensitive internal or external components or fragments are used in the project.
- Verify that no unapproved changes were made to sensitive internal components, and that no unapproved or precluded changes were made to external components.
- Review the license obligations of all external components used in the project, and ensure compliance with these obligations.

These phase reviews backstop the development team, and keep the legal, technical, and business reviewers engaged in the management of software re-use. They also verify that changes in the project's objectives have not created legal, technical, or financial inconsistencies with the licenses of components used in the project.

** Control component contribution and disposition*

The rationale for contributing components to an open source project is beyond the scope of this paper, as are the considerations involved in transferring ownership to a third party or creating a

new open source project. However, if a contribution or transfer of a candidate component or fragment is deemed appropriate, the project's legal and business reviewers should:

- Determine whether the candidate component's sensitivity (if internally developed) is an impediment to contribution or transfer.
- Verify the right to contribute or transfer every externally developed component or fragment contained within the candidate.

This helps to ensure that the organization does not inadvertently contribute code that is sensitive or that the organization doesn't have the right to contribute.

* *Assess software components before acquisition*

If an organization is considering an acquisition that would include a significant interest in one or more software components, the designated set of legal, technical and business reviewers should be charged with the following:

- Identifying all included components not owned by the supplier or target.
- Assessing their license obligations with respect to the acquirer's compliance, business objectives, and legal policies.
- Assessing the impact of any required rework or change on cost, revenue, and quality.

Note that this best practice applies to a variety of situations in which financial investments are involved. Such situations include: company mergers and acquisitions, product acquisitions, joint venture formations, venture capital investments, etc.

Conclusion

With open source and other downloadable code freely available, companies must retain control of their product development process. They must manage the complexity of tracking and compliance of the intellectual property used. As development teams become more geographically and organizationally distributed, it is necessary to assure that the process can be managed globally and with disparate and unfamiliar developers. This is becoming increasingly difficult without some form of automated software compliance management system for protecting intellectual property.

By integrating the best practices, whose objective is to encourage development based on component re-use and software assembly in its development processes, organizations will have far greater assurances of compliance with all relevant license obligations and far more effective protection of software intellectual property. Adopting these practices will enable companies to be more aggressive in their use of the software assembly approach. That, in turn, will give companies the benefits and competitive advantage this new development approach promises – including accelerated project timelines, improved software quality and reduced development costs.

Evolutionary Methods (Evo) at Tektronix: A Case Study

Frank Goovaerts, Tektronix Inc., (503) 627-2224, frank.h.goovaerts@exgate.tek.com

Doug Shiffler, Tektronix Inc., (503) 627-7588, douglas.e.shiffler@exgate.tek.com

Biographies

Frank Goovaerts has engineering degrees from the University of Leuven (Belgium) and the University of Cincinnati.

He has worked in the software industry since he graduated in 1981. For the last 16 years he has been with Tektronix in a number of roles varying from developer, to Project Lead, Functional Manager and currently Director of Engineering for the Performance Oscilloscope Product Line. Tektronix is a world leader in test, measurement, and monitoring equipment.

Frank is an advocate for process improvement and continues to drive his organization to create better software, on time.

Doug Shiffler has an engineering degree and a business degree from Brigham Young University. He has worked in the computer and test and measurement industries for the past 23 years in a variety of roles in process, design and quality engineering, program management and is currently a Senior Engineering Manager in the Performance Oscilloscope Product Line.

Doug is a quality advocate and is a believer in the benefits of continual process improvement.

Abstract

This paper documents how we implemented the Evolutionary Method (Evo) at Tektronix, Inc. We start with a brief refresher on this method that dates back to the 1980's (Deming cycle). The method was then evolved by Tom Gilb in 'Principles of Software Management' in the late 80's. We discuss the development methodology used at Tektronix and how the Evo method fits into that methodology. We then detail a case study where we implemented Evo mid project (in a large team) and tracked intermediate results up to the end. We conclude with lessons learned and where we plan to go next.

1 Intro

One cannot argue the fact that the production of high quality products results in a competitive advantage, especially for a company like Tektronix, a world leader in test, measurement and monitoring. In fact, our quality policy states that "Tektronix is committed to providing products and services that consistently meet or exceed our customer's expectations, while continuously improving the effectiveness and efficiency of all aspects of the operation". However, quality comes at a price. There are numerous references (Wiegers, McConnell and others) to the fact that defects take up to 200 times longer to fix later in the lifecycle. Still, we end up fixing those pesky defects at the end of the program, despite the expense (causing both time and budget overruns).

We deployed Evo to help us plan and track milestones and to reduce product development costs by focusing on fixing defects earlier in the product lifecycle.

The case study involves a major platform development effort including software and hardware. One of the strengths of the method is that it can be rolled out across an organization or can target a subset of that organization; it works on small programs as well as larger ones. We initially tried Evo with a small team of software engineers; we then rolled it out to the rest of the software team, and finally deployed it to other functional organizations. Although we cannot claim the success of this program to using the method (after all we didn't run another program where we did not use the method in parallel), we believe that Evo helped us get the product out in time with excellent quality. Most of the team members agree that Evo helps them with day to day prioritization and team communication.

We will explain how we adapted the Evo method to our projects and product development lifecycle. One of the essential strengths of the method is that it forces the project to evaluate frequently and make changes where needed.

2 Evo Explained

Evo is an abbreviation for the development of an **evolutionary** delivery process as a means of program management. Rather than use the traditional waterfall approach to managing product development which emphasizes one large delivery at the end of the complete development cycle, Evo focuses on multiple mini-cycles. Each cycle is concentrated on a subset of the end requirements. It focuses on a prioritized list of what the stakeholder/customer needs are at a given point in time to the level of detail that is required at the time.

Tom Gilb who is an authority in Evo techniques has defined 10 Evolutionary Project Management (Evo) principles. This information and much more can be found on Tom's web site at <http://www.gilb.com>.

Evo Principles (paraphrased from Gilb's web site):

1. Deliver real results, of value to real stakeholders, early and frequently.
2. The next Evo delivery step must be the one that delivers the most stakeholder value possible at that time.
3. Evo steps deliver the specified requirements, evolutionarily. Concentrate on delivering real value to the stakeholder (as defined by the stakeholder) and then measure your results by asking the questions:
 - Exactly what value was delivered?
 - Exactly what did it cost compared to estimates?
 - Was the stakeholder really satisfied?
 - Did new requirements get discovered?
 - Did the technology work as expected?

Evo is about feedback and learning. It is about applying multiple 'Plan-Do-Check-Act' (Deming) cycles concurrently.

One important consequence of this is that the formal requirements are very important. If they are not unambiguously clear, if benefits and costs are not quantified – then we cannot use Evo in the rational engineering mode that is the expected mode of use.

4. We cannot know all the right requirements in advance, but we can discover them more quickly by attempts to deliver real value to real stakeholders.
5. Evo is holistic systems engineering; all necessary aspects of the system must be complete and correct and delivered to a real stakeholder environment. It is not only about programming, it is about customer satisfaction.
6. Evo projects typically require an open architecture, because we are going to change project ideas as often as we need to, in order to really deliver value to our stakeholders.
Open architecture means 'easy to change'. This can include many 'technological enablers', for many types of change. If you want to optimize your product for long term performance (and consequent survival) under conditions of change, you have to be conscious about your product architecture objectives and design specifications. When the stakeholders define a requirement we have to be able to deliver those requirements immediately without compromising system performance. That is what we need the open architecture for. All systems need intelligent open architecture in the long run. Evo projects need it in the short term, during the project.

7. The Evo project team will focus their energy, as a team, towards success in the current Evo step. They will succeed or fail in the current step, together. They will not waste energy on downstream steps until they have mastered current steps successfully.
8. Evo is about learning from hard experience, as fast as we can – what really works, and what really delivers value. Evo is a discipline to make us confront our problems early – but which allows us to progress quickly when we really probably have it right.
9. Evo leads to early, and on-time, product delivery - both because of selected early priority delivery, and because we learn to get things right early.
10. Evo should allow us to prove out new work processes, and get rid of bad ones early.

3 Development Methodology

3.1 Find and Fix Defects Sooner

A key part of our development Methodology is to “Find and Fix Defects Sooner” in the development lifecycle. Ultimately this can lead to defect prevention and eventually defect free code. Our definition of defect free is no defects that will prevent the customer from using the product to its specifications.

The entire industry is in agreement with the fact that having to fix defects later in the life cycle costs a lot more. Multiple resources can be referenced, here are just a few:

- 10-100 times as much to correct once fielded (Steve McConnell, 2001)
- 50-200 times as much to correct once fielded. (Barry Boehm, 1988)
- 15 times as much to correct once fielded (IBM System Sciences Institute)
- 10 times as much to correct during testing (Hughes Aircraft)

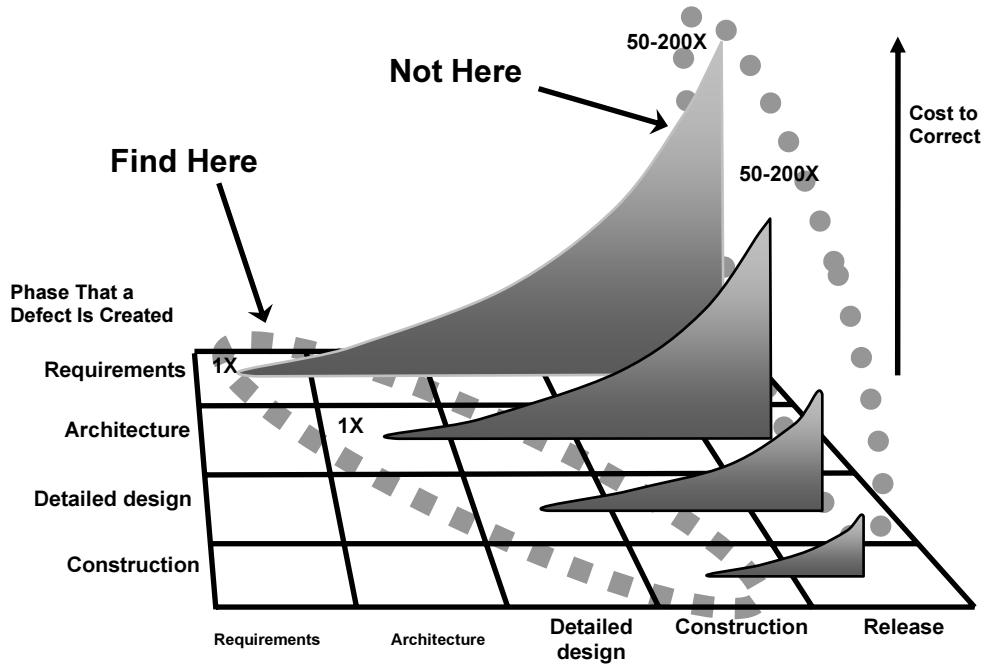
On most software development projects, reworking requirements defects alone costs an inordinate amount of resources:

- 40-50% of the effort (Capers Jones)
- 80% of the effort (Karl Wiegers, 2001)

Finally, up to half of all project resources are used for general ‘rework’, this includes changing requirements, fixing introduced defects, and re-architecting the code.

The following graph is taken from one of Steve McConnell’s presentations. It illustrates how the cost to fix defects can skyrocket during the development lifecycle.

Living with Defect-Cost Increase



Source: Steve McConnell

Phase That a Defect Is Corrected

3.2 Mini-Milestone Development

In support of "Finding and Fixing Defect Sooner", we also institutionalized mini-milestones as part of our Development Methodology.

Frequent deliverables of 'finished' products seems to be the norm in today's Agile world. That doesn't fit for a Test and Measurement company that delivers complex, high quality equipment and software to its customers. To allow us to deliver these quality products (that meet published specs) to our external customers, we follow a stringent waterfall based New Product Development methodology. In other words, we decide what to do, how to do it, do it, test it and ship it. And the 'do it' cycle typically takes the longest.

We tailor the mini-milestones to internal customers. This allows us to release 'finished' products to other organizations which include hardware and software engineering, documentation, qualification and others. The goal is to build excitement for these intermediate releases, both within the development team and the internal customer.

While focusing on mini-milestones, it is important to not lose sight of the final deliverable (i.e. the finished product to our external customers). This is tracked by the program timeline which contains (at a high level) a list of all features and estimated effort (preferably prioritized) that need to be completed to meet the product specifications. Each of these features needs to be planned out with enough details so we can make an accurate timeline, after all we need to feel comfortable that we will deliver a product to our external customers within the defined timeframe.

The deliverables for each mini-milestone need to be clearly defined to allow the team to self measure. The list of deliverables needs to be realistic. Adding deliverables that have no chance of completion adds no value. Ideally, each team member should have one or more deliverable

listed. For each (small) incremental delivery the team can judge how well they did and adjust accordingly. That may include cutting features (better to do that early on, before development starts), adding resources, changing the timeline (if the market allows), or any other proactive (rather than reactive) measure we choose to take.

4 Case Study

4.1 Expectations from this project

This was a multi-year project to develop a new high-end oscilloscope platform. Historically our experience has shown that the software portion of a new product development program is often delivered late. After reviewing the Evo literature and examining the results from others who have used Evo, the authors believed that the implementation of Evo techniques on this program could decrease development time and improve software quality. Evo literature suggested that the use of Evo techniques would result in about 30% schedule improvement as well as higher product quality.

4.2 Background

This program was partially through the engineering development lifecycle when we decided to implement the Evo techniques. The software was developed by a large distributed team of software engineers. The software development was dependant upon the timely delivery of working hardware. Delays in the hardware delivery could affect software development schedule day for day.

At the time of Evo implementation the hardware and software schedule was developed using the traditional waterfall approach in Microsoft Project. A top level schedule was maintained by the program manager and the software schedule (that dovetailed into the program schedule) was maintained by the software project lead. Weekly schedule update sessions were held by the software project team leader and members of the software development team followed by project reconciliation sessions with the program manager.

The challenge for us was to implement a new product development management methodology (mid-project) that would positively improve the schedule and performance of the software team. At that point in time, only the software organization had agreed to use the Evo techniques, the other functional areas chose not to implement Evo.

Evo implementation challenges included:

- Implementation of Evo across a distributed team.
- The software organization was the only functional organization to implement Evo techniques for this program.
- We implemented Evo mid-program rather than at the beginning of the product development cycle.
- Neither management nor members of the development team had prior experience using Evo techniques.
- Almost everyone was skeptical about the implementation of a new process mid-program.
- Many members of the software development team were experienced engineers and therefore used to the traditional ways of planning and executing their software development tasks.

4.3 Short Term Goal

The short term goal was to demonstrate verifiable schedule and quality improvement over historical averages obtained on past product development programs.

The actions taken to achieve this short term goal were:

- Investigate Evo techniques and assess the potential benefits to our software development processes.
- Utilize the knowledge gained from this research to explain and request the support of the General Manager, Program Manager, Software Project lead(s) and the individual engineers.
- Train the software engineering team on Evo techniques with the assistance of Niels Malotaux, an Evo coach based in the Netherlands.
- Tailor the method in such a way that we could adapt it mid-way through the development cycle.
- Determine how to link the Evo weekly scheduling into the master project schedule.
- Determine metrics to use to track progress and adherence to the Evo techniques.
- Minimize the impact on the developers as to not impact current activities.

4.4 Long Term Goal

The long term goal was to realize a 30% improvement in schedule pull-in and quality improvement over the conventional waterfall method.

The actions taken to achieve the long term goal were:

- Demonstrate sufficient benefit to expand the use of the methodology to other functional groups.
- Roll Evo techniques out to other functional groups to realize even greater benefits across the organization.
- Continue to utilize and refine the Evo techniques.

4.5 Evo Implementation Time Line

We implemented Evo over a period of 8 weeks. A detailed chronological report on how this method was introduced in our organization follows.

4.5.1 Week 1

- The general method was explained to the entire organization, not detailing any implementation plans.
- Software engineering management then presented the proposal to stakeholders. Evo was approved for use on a trial basis for the software development portion of the program.
- We held an initial training session with the Beaverton software team.
- Due to the critical nature of this program, the team was given multiple opportunities to decline using this method.
- We started a four week cycle on a trial basis with a subset of the team (8 developers). At that point the team had made a commitment to use the method during this period (reason: it's easy to get overwhelmed by the perceived initial overhead, after three to four weeks people had adapted to using the method and were seeing the benefits).

4.5.2 Week 2 through 5

- The team used Evo on a weekly basis. Niels Malotaux guided us through the initial one-on-one meetings as well as team meetings
- We evaluated the experience of the Beaverton SW development group and decided that we would continue to use the technique
- During these four weeks, we implemented 62 out of 70 features on time!
- The process was rolled out to the Bangalore-India development group for another four week trial run to determine whether Evo would work as well with a remote team. If that effort was successful, we would roll out the method to the entire development team.

4.5.3 Week 6 through 8

- The Beaverton team continued to use the method
- The Bangalore team started to use the method.
- We devised a way to synchronize Bangalore/Beaverton Evo activities by setting up a weekly conversation between the team leader in Bangalore and the liaison in Beaverton.
- Results were favorable

4.5.4 Week 9 through Project End

- We included the entire software team (unconditionally)
- Continued weekly Evo reporting and planning sessions with all members of the development team.
- The team was rather large, so Evo team meetings and reporting were divided into sub-teams. This caused us to need a roll-up meeting.

4.6 Evo Implementation Guidelines

The following were the guidelines that were established for executing the Evo process within the Software team. These guidelines were meant to be a quick reference for members of the team for the most relevant aspects of the process. The team also utilized and referenced the handbook, "*How Quality is Assured by Evolutionary Methods*" (Niels Malotaux),

1. Weekly Evo One-on-one Sessions

These sessions are held between sub-project leaders and the engineers assigned to the sub-team. These sessions are also held between sub-project leaders and the software project lead. Functional managers also attend these one-on-one sessions on a rotating basis to lend support, clear obstacles and to ensure the process was being followed. Guidelines for these sessions included:

- Duration of the one-on-one sessions should be brief (less than 15 minutes).
- Developers should come to the session prepared with a complete list of tasks for the upcoming cycle.
- Developers come prepared with 26 to 30 hours of proposed work (the remaining hours are 'unplanned').
- If a developer cannot identify a full 26 hours of tasks, they should discuss it with their subproject lead prior to the Evo one-on-one session.
- Any issues or action items identified during the session should be noted, addressed, and tracked outside the session.
- Long discussions on design ideas, bug evaluations, etc. should be avoided.

2. Evo Task Planning

- Two thirds of total working hours should be planned (26 hours for a 40 hour cycle).
- Maximum task duration should be six hours.
- Task estimates should include only actual hours spent working on the task (effort hours), not overhead or interruptions.
- Planned absences should be deducted from the total working hours, (e.g. if a developer is working 32 hours in a cycle, plan two thirds of the 32 hours or ~20 hours).
- When an unfamiliar task is encountered, an analysis task should be in addition to the other tasks. After the analysis task has been completed the developer and subproject lead can finish the cycle plan or plan it out in the next cycle.
- Dependencies which affect the ability of an engineer to perform the next cycle's tasks should be noted and confirmed with the individual responsible for delivering the dependency prior to the Evo one-on-one sessions.
- Missed one-on-one sessions should be made up as soon as possible with the subproject lead.
- Each developer should keep a hardcopy of the current cycle's tasks.
- No changes should be made to the Evo database outside of the one-on-one sessions (with the exception of marking a task done, adding details to current tasks and entering ideas for the next cycles(s)). The tool does not allow us to track this automatically, so we rely on people following the process.
- One should schedule the most important task first (as determined by the Project Lead, who in turn gets this input from Core Teams and change control boards).

3. Evo Task Completion

- All tasks for each cycle should be completed as planned.
- If for any reason a planned task cannot be completed during the cycle, the developer should notify the subproject lead as soon as possible. The developer and lead should then re-plan as necessary. Changes to the plan should be recorded on the developer's task sheet.

4. Evo Tracking

- The subproject and project lead must be able to link and track the Evo tasks (hourly/daily resolution) to the master program schedule (weekly resolution) and the milestone checklist.

4.7 Evo Results

4.7.1 Evo near Term Results

Schedule accuracy for this platform development was 50% better than the program average (as measured by program schedule overrun) over the last 5 years, and this product was the fastest time-to-market with the highest quality at introduction of any platform in our group in more than 10 years. The team also won a prestigious Team Award as part of the company's Technical Excellence recognition program.

4.7.2 Evo Sustainability

Based upon the results realized during this program there is across the board support for the use of Evo techniques on the follow-on program both in software and hardware. We believe that having both of these groups utilizing Evo techniques will provide even greater schedule predictability because of the dependencies between the software and hardware schedules. Higher reliability in the hardware schedule will provide higher reliability of the software schedule.

5 Lessons Learned and Next Steps

Following are some of the lessons we have learned after using this method for almost two years now. We intend to use these to fine-tune the process and continue to improve the method.

- Identify early in the project who are the Stakeholders in the Evo process and document expectations.
- Schedule and hold Milestone review meetings in order to gauge the plan to the actual execution of the program.
- The initial implementation of Evo began with the Software Engineering team and we believe that improved results will occur with the inclusion of the Hardware Engineering team in the Evo process during the next development program. This will more clearly highlight as well as track the dependencies between the two functional areas.
- At the start of the Evo process build excitement for stakeholders so that they anticipate and validate the Evo deliverables.
- The first implementation of Evo was implemented mid-way through the product development program, improved results will occur as Evo is implemented at the start of all future development programs within our product line.

5.1 *Evo results we have witnessed and the drivers behind them*

Even though this paper only talks about the implementation of Evo in the software organization of one product line, we have seen many benefits. These include:

- Better quality end-products through improved cross functional integration.
- Constant optimization of resources through continual feedback in the Evo process.
- Reduced rework through emphasis on doing work right the first time (i.e. no defects).
- Completion of the highest leverage tasks first through prioritization (postpone lower priority or less impact tasks). These may include user interface paradigms that we may want to validate, architectural risk reduction activities, manufacturability issues and others.
- Creates a sense of urgency in the team through regular review of performance to plan.
- Feeling of accomplishment as team members are encouraged to have goals and the feeling of accomplishment (or failure at times) as they measure against their goals.
- Timely corrective action through early and continuous feedback on individual contributors to Project Leads and functional managers. (Roadblocks, scheduling inaccuracies become obvious right away).
- Better efficiency through the use of Time-boxed development which keeps people from getting 'stuck' on a problem.
- Improved project planning (Makes it obvious when we have poorly planned providing the motivation to take steps to improve).

The drivers behind these results can be summarized as follows:

- Everyone works on the most important things all the time, priorities are evaluated weekly.
- Promotes the team effort and increases leverage (rather than a group of individuals doing great work).
- Project leads are more closely involved with all aspects of the program.
- Project leads are aware of any issues that arise during any given day.
- Individual productivity is improved because people stay more focused.
- Improved team dynamics during the weekly Evo Team meetings.
- Evo is all about questioning what NOT to do, so that we have more time to do the things we really have to do well.

5.2 Summary Observations

Overall takeaways from using Evo in our organization:

- Evo is helping to bring forth some best practices that will help our engineering development effort.
- Our goal is to take the best parts of Evo and utilize them. This will bring additional best practices to the table.
- During this first implementation of Evo we went through the mechanics of the delivery cycles, not always building the excitement with the team as well as the (internal) customers. In the next use of Evo we need to identify the internal customers up front and ensure we understand and document those items that build excitement.
- We implemented this method mid project (execution phase), we need to adapt this method to the rest of our product development lifecycle, lifecycle (i.e. the definition and design phases) and include the Hardware engineering team in this process.

6 References

Gilb, Tom: Principles of Software Engineering Management, 1988, Addison Wesley, ISBN 0-201-19246-2

Gilb, Tom: Competitive Engineering, 2005, Elsevier, ISBN 0750665076

W.E. Deming: Out of the Crisis. MIT, 1986, ISBN 0911379010

McConnell, Steve: Rapid Development, 1996, Microsoft Press, ISBN 1-55615-900-5

McConnell, Steve: Professional Software Development, 2004, Addison Wesley, ISBN 0-321-19367-9

Malotaux, Niels: "*How Quality is Assured by Evolutionary Methods*",
<http://www.malotaux.nl/nrm/pdf/Booklet2.pdf>

Malotaux, Niels: "*Evolutionary Project Management Methods*",
<http://www.malotaux.nl/nrm/pdf/MxEvo.pdf>

<http://www.gilb.com>

<http://www.malotaux.nl/nrm/Evo>

Smart Result Analysis (SRA) - A Key Competitive Advantage

Manoharan S. Vellalapalayam

Software Architect

Intel Corporation

manoharan.s.vellalapalayam@intel.com

BIO

Manoharan Vellalapalayam is a software architect at Intel with the Information technology group. He has over 15 years of experience in software development and testing. He has created innovative test tool architectures that are widely used for validation of large and complex CAD software used in CPU design projects at Intel. He has chaired validation technical forums and working groups. He has provided consultation and training to various CPU design project teams in US, Israel, India and Russia. He specializes in automated distributed test jobs execution, smart result analysis, GUI testing and cross-platform test architectures. He is currently working on content management and web platform security solution architectures.

Abstract

Modern software applications that are rich in features and large in size need new innovative approaches applied to test result analysis in order to improve time-to-market and quality, which leads to gaining competitive advantage. This paper captures why the conventional method of result analysis to determine test run status is no longer sufficient and is often unreliable. This paper offers a new innovative and practical approach to result analysis that was successfully applied to various large CAD software projects at Intel.

Introduction

Software testing has never been more challenging. Modern software applications are large and increasingly complex. They are interdependent and integrated with various platforms and technologies to provide a rich user experience. Obviously, validating and ensuring high standards of quality can be a huge challenge. This demands a new innovative approach applied to gain the following competitive advantages.

1. Bringing the product to market faster by reducing the development and testing time (TTM).
2. Ensuring software quality as a whole by carefully designed integrated flow testing.
3. Enables non-functional requirements testing by analyzing test results and carefully applying platform(IA64, EM64T, HyperThreading) specific tuning (Quality of Service, QoS).

This paper first captures why the conventional method of result analysis to determine test run status is no longer sufficient and is often unreliable. Next, the process and important components of Smart Results Analysis approach are explained. A detailed background on how to enable test cases for SRA is captured. The XML format and process used for data metrics collection are explained with examples. Result analysis logic and the reports produced to review test execution results are described.

In the applications section the key learning's, challenges and the benefits of using Smart Results Analysis are explained by outlining integrated flow testing and non-functional requirements testing. It also explains why **QoS** is used by the industry to gauge competitive advantage of software products for the benefit of the reader.

Then, this paper describes how Smart Results Analysis (SRA) was used to gain competitive advantages for testing large CAD software tools that are complex and large (with millions of lines of source code), mainly used for Chip design at Intel.

Issues with the conventional approach to test result analysis

In the conventional approach of test result analysis, golden output files are first created with expected results. This is done before running a test case as part of test creation process. The process is repeated for multiple input data sets to record a golden data output file for each input data.

Then these test cases are run during regression testing whenever a new software build is made available. Test result analysis is done by parsing and analyzing the output data file or status log file that were recorded while running a regression test case on the software to be tested. Test Result status is determined by comparing or *diff'ing* the actual output file with the correct expected data file (golden data).

If there were any differences then a test was considered failed, otherwise it passed. This method sounds easy and expected to work. But in practice, this conventional approach was found to be problematic. Below are some of the main reasons why the conventional method of result analysis to determine test run status is no longer sufficient and is often unreliable.

1. This is very unreliable because a harmless change in the output file could trigger false result status. This leads to increased debugging of test failures, re-testing time, resulting in missed deadlines.
2. If the test status was failed, it does not really help developer in root-causing the problem by where exactly or in what stage of testing flow the test failed. This is very crucial in reducing the development time and to save project costs (TTM).
3. Just knowing test run status pass or fail will not be helpful for performance analysis as the data was not collected during the different stages of the test flow. This is very important to fine tune software applications to exploit platform specific capability (QoS).
4. Difficult to find failure patterns when a large number of test runs fails due to problems at the platform level like bad network connection, machine related issues, OS or kernel level problems, changed library versions etc. This causes significant overhead in going through the analysis log file to find the failure patterns, causing delay in the turn around time for the test cycles.

Smart result analysis (SRA) for competitive advantage

What is SRA?

SRA stands for Smart Result Analysis. In short, it is a new smart approach to test result analysis in order to get reliable test results. It helps in bringing product to market faster, supports flow testing and non-functional requirements testing resulting in competitive advantage.

SRA process flow

As a solution to the problems seen in the conventional method of result analysis, Smart Result Analysis uses a simple yet innovative approach to the test case creation, data capturing, output format and reporting phases of a testing cycle as described in the SRA process flow diagram **Figure 1**.

Smart Result Analysis (SRA) process flow begins with **Create TestCase** as the first activity. Test cases are designed with multiple steps or check points in the test flow so as to capture and record feature specific or non-functional requirements metrics. By splitting the test flow into several stages, several metrics can be collected at each stage which can then be used at run time to decide on the flow control or stop the flow itself when a serious error occurred.

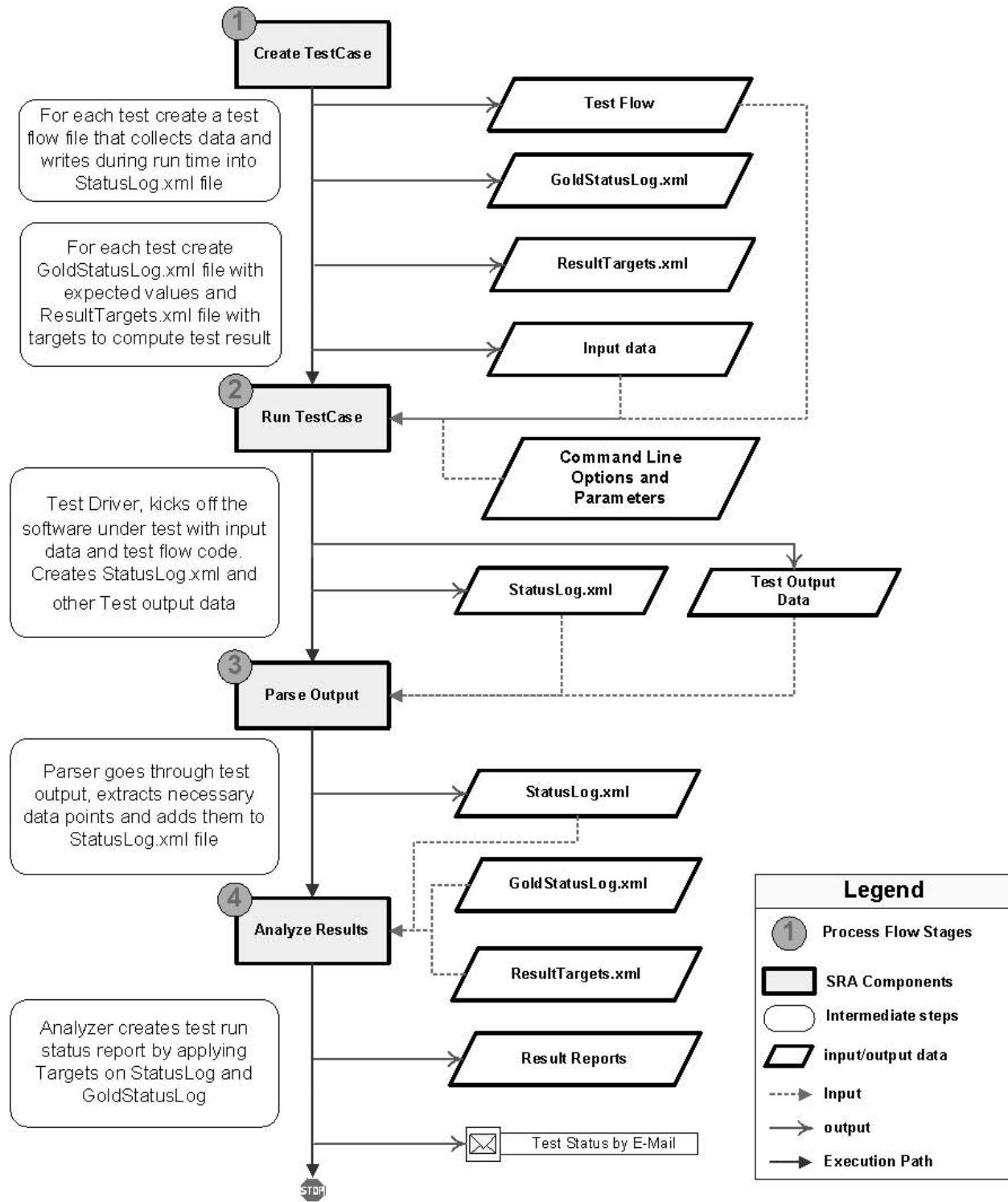


Figure 1

Using this approach, test failures can be easily associated to the failed stage or step of the test flow and also enable developers to quickly root cause the problem, resulting in significant time and cost savings.

During the **Create TestCase** stage, the Test flow file, ResultTargets.xml and GoldStatusLog.xml are important collateral files created as part of test definition. Test flow file is a test source file that includes list of steps to be executed to test product features and collect data metrics. GoldStatusLog.xml file is created in this stage with expected values for data points or metrics that will influence the test result. StatusLog is created by the test driver during runtime. So it contains actual data values or metrics, where as the GoldStatusLog.xml file is created during test creation time with expected data values as shown in ***Listing 2***. These two files are used during test result analysis to determine the test run status.

ResultTargets.xml file as shown in ***Listing 3*** created in this stage, to capture all the target expressions for a test case. Target expressions use values from both StatusLog.xml and GoldStatusLog.xml to compute test result status.

In the next **Run TestCase** stage, the Test Driver component executes the software application by passing the command line parameters and input data as specified in the test case definition. Test output data and other log files are created in this stage. At this stage the test case run is completed and the output recorded. The StatusLog.xml as shown in ***Listing 1*** gets created by the test flow code during the test run time, which then is used by the analyzer to evaluate test results.

In the third **Parse Output** Stage, as opposed to the conventional approach, raw output data files are parsed to extract only the relevant and important data points or metrics that will influence the test result and are added to the StatusLog.xml file created from the previous stage.

In the final **Analyze Results** stage, test results are computed by applying Targets defined in the ResultTargets.xml file on actual metrics and Gold metrics defined in the StatusLog.xml and GoldStatusLog.xml files respectively. Then the test results are formatted for e-mail reports.

Components of SRA

Important components of SRA are described here in detail. Required files, their format with examples and tools used for doing the SRA are also included.

StatusLog Editor

It is a GUI editor to help test authors to create GoldStatusLog.xml file which is part of a test case definition. It hides the complexities of xml file format from the author by providing an intuitive interface to populate the GoldStatusLog.xml file. Normally test authors run the test for the first time to capture the StatusLog.xml file, and then modify it using StatusLog.xml as an input and changes the values for each data metric to the correct expected values.

As shown in ***Listing 1***, each data point or metric is recorded as an element with category, metric, type and unit properties in StatusLog.xml file. Out of these properties category and metric are mandatory, type and unit are optional. For 'type' property 'string' is the default value.

Format for the StatusLog and GoldStatusLog are the same. StatusLog is created by the test driver during runtime. So it contains actual data values or metrics, where as the GoldStatusLog.xml file is created during test creation time with expected data values as shown in ***Listing 2***. These two files are used during test result analysis to determine the test run status.

Status Log:

```
Listing 1 Status Log
<?xml version="1.0" ?>
<!--
    Test Name: AutoRoute
    Date: Jan 13, 2006.

    File name: StatusLog.xml
    Description: Test Run Status Log file
-->
<StatusLog>
<log category="RunTime" Metric="Peak Memory" Type="Number" Unit="MB">3.4</log>
<log category="Error Count" Metric="Source" Type="Number" >4</log>
<log category="File Read Time" Metric="Schematics" Type="Number" Unit="Seconds">0.04</log>
<log category="Routing Design" Metric=" File Read Time" Type="Number" Unit=" Seconds">5</log>
<log category="Routing Design" Metric="Total Net Count" Type="Number">200</log>
</StatusLog>
```

Gold Log:

```
Listing 2 GoldStatusLog
<?xml version="1.0" ?>
<!--
    Test Name: AutoRoute
    Date: Jan 13, 2006.

    File name: GoldStatusLog.xml
    Description: Test Run Gold Status Log file
-->
<GoldStatusLog>
<log category="RunTime" Metric="Peak Memory" Type="Number" Unit="MB">10</log>
<log category="Error Count" Metric="Source" Type="Number" >0</log>
<log category="File Read Time" Metric="Schematics" Type="Number"
Unit="Seconds">1.04</log>
<log category="Routing Design" Metric=" File Read Time" Type="Number" Unit="
Seconds">20</log>
<log category="Routing Design" Metric="Total Net Count" Type="Number">90</log>
</GoldStatusLog>
```

ResultTargets Editor

It is a GUI editor to help test authors to create ResultTargets.xml file which is part of a test case definition. It hides the complexities of xml file format from the author by providing an intuitive interface to populate the ResultTargets.xml file. This GUI tool lets the user create or modify an existing ResultTargets.xml file.

Normally test authors run the test for the first time to capture the StatusLog.xml file and create GoldStatusLog.xml using the StatusLog Editor. Then both StatusLog.xml and GoldStatusLog.xml are opened using ResultTargets Editor, to select important metrics and create target expressions from them into the ResultTargets.xml file as shown in ***Listing 3***.

Listing 3 ResultTargets (Conditions)

```
<?xml version="1.0" ?>
- <!--
    Test Name: AutoRoute
    Date: Jan 13, 2006.

    File name: ResultTargets.xml
    Description: Result Targets file
-->
<ResultTargets>

<target Priority="ShowStopper" PassComment="Memory Usage Check Passed" FailComment="Memory Usage Check Failed">
    <eval>
        <value source="StatusLog" Category="Memory" Metric="Peak" />
        <operator>&lt;=&eq;</operator>
        <value source="GoldStatusLog" Category=" Memory" Metric="Peak" />
    </eval>
</target>
<target Priority="High" PassComment="Total Routed Net Count Passed" FailComment="Total Routed Net Count Failed">
    <eval>
        <value source="StatusLog" Category=" RoutingReport" Metric="TotalRoutedNetCount" />
        <operator>&eq; 1.05 *</operator>
        <value source="GoldStatusLog" Category=" RoutingReport" Metric="TotalRoutedNetCount" />
    </eval>
</target>
<target Priority="ShowStopper" FailComment="Error in the source">
    <eval>
        <value source="StatusLog" Category="Flow Source" Metric="Error Count" />
        <operator>&eq; </operator>
        <value>0</value>
    </eval>
</target>

</ResultTargets>
```

Test driver

Test driver is one of the main components in the testing framework. It invokes the software or application under test (AUT) by passing in the test flow, command line options or parameters and the input data. Basically it prepares the test run logistics like environment variables, temporary and runtime folders, copies the necessary files before launching or invoking the software under test. After the test is run it also makes sure that the entire test flow execution path as captured in the process flow diagram (**Figure 1**) is completed by calling the parser and analyzer.

Many of the CAD tools are huge and have built-in scripting interface support for batch processing. This built-in scripting interface makes test jobs efficient and enables increased scope of test coverage. Using this built-in scripting interface, test flow commands are executed in various stages, collecting data values/points like maximum memory usage, time taken to read an input file, time taken to auto route a cpu design data etc. A simple script library can be used to provide an API to write StatusLog.xml file for data collection, which then can be used for result analysis and benchmarking purposes.

Using this approach benchmark data like performance, memory usage, file sizes are collected over a time and for different product build versions. By plotting the collected values or side-by-side comparison of historical test runs, one can easily deduce how a typical product feature or performance metric performed over the period of time. This brings a powerful data analysis based benchmarking feature with simple investment during test creation.

Parser

Parser is one of the important components in the SRA framework used to extract the relevant and important data points from the product output data and various log files. Using a parser is often the starting point to collect data points or metrics for software under test that do not support script interface. In Black-box testing, parser plays an important role in collecting data points or metrics from test run output and log files into StatusLog.xml file.

Special filters and/or parsers are required for some of the design files that are in binary format. These special filters read the design files and capture the design characteristics like number of nets, cells, terminals and auto route points etc.,

Analyzer

Analyzer is a very critical component in the SRA framework to analyze results of test run using ResultTargets.xml, GoldStatusLog.xml and StatusLog.xml files to determine the test run status. The captured metrics are compared against a predefined behavior specification. If a relevant metric included in the ResultTargets.xml is not found from the StatusLog.xml, then the result for that target is declared as failed. This could happen when there was a serious error even before getting to the reporting point of that metric in the test flow. Typical Input and Outputs are captured in the process flow diagram (Figure #1). The ***Listing 3*** shows an example of ResultTargets.xml file.

As given in the sample ResultTargets.xml file, each target expression is defined as an element at test creation time. Each target xml element has Priority as a mandatory property with PassComment and FailComment as optional properties. PassComment is a readable text that gets printed in the result report when the target passed. FailComment text gets printed in the report when the target failed. These comments should be designed during the test creation time to give meaningful status messages (instead of cryptic target expressions) to report users. By going through these comments in the result report for all test runs, one can easily and quickly **identify a failure pattern** for large regressions.

The ‘eval’ xml element represents a target expression. It can occur multiple times in the same target with different values and can be nested also for complex targets. At run time each ‘eval’ xml element is expanded as an expression that returns a Boolean value by substituting the actual values from both StatusLog and GoldStatusLog.xml files. Malformed eval xml elements are validated by the ResultTargets Editor at test creation time. If any metrics are missing in the StatusLog files or the constructed expression can not be evaluated to Boolean value, the target is declared failed with an appropriate error message by the analyzer tool.

In the SRA approach, the comparison equation is made more powerful compared to the conventional approach by making each target as an **expression**. This enables a test case to have targets that fall in a range with upper and lower bounds, dependency on other data metric and mere existence of other related data metrics to compute pass or fail status. This approach helps to determine the test failure accurately and reliably. A few examples of target expressions are given below:

- Actual Metric 1 = Gold Metric 1 (Boolean)
- Actual Metric 1 = 5 (a hard coded value in the ResultTargets.xml file)
- Actual Metric 1 <= Gold Metric 1 (Lower range with an upper bound)
- Actual Metric 1 >= Gold Metric 1 (Upper range with an lower bound)
- Actual Metric 1 >= Gold Metric 1 and Actual Metric 1 < Gold Metric 2 (a range with upper and lower bounds)

How each target is evaluated to arrive at a result?

The analyzer builds the target eval expression by substituting the actual values for metrics from the StatusLog.xml and GoldStatusLog.xml files. To understand how the targets are evaluated at run time let us look at the example below:

StatusLog.xml file:

```
<log category="RunTime" Metric="Schematic Design File Write Time" Type="Number" Unit="Seconds">0.08</log>
```

GoldStatusLog.xml file:

```
<log category="RunTime" Metric="Schematic Design File Write Time" Type="Number" Unit="Seconds">1.04</log>
```

ResultTargets.xml file:

```
<target Priority="ShowStopper" PassComment=" Schematic Design File Write Time Passed" FailComment=" Schematic Design File Write Time Failed">
  <eval>
    <value source="StatusLog" Category="RunTime" Metric=" Schematic Design File Write Time" />
    <operator>&lt;&eq; 2.0 *</operator>
    <value source="GoldStatusLog" Category="RunTime" Metric=" Schematic Design File Write Time" />
  </eval>
</target>
```

During run time Analyzer expands this target into an expression like this:

*RunTime: Schematic Design File Write Time <= 2.0 * Gold:RunTime: Schematic Design File Write Time*

and substitutes values from the StatusLog files in order to create an expression to be evaluated:

*0.08 <= 2 * 1.04*

Since 0.08 is less than 2.08, the expression is true, and the target result is PASSED.

Reporting Test Results

Analyzer creates the result reports file and also sends out the result report file content as an e-mail report. A sample e-mail report is shown in ***Listing 4***. Many times just knowing a test result passed or failed would not be sufficient. It becomes important to prioritize test failures to know how bad a test failed. For example if a test result depends on 10 different targets or data points or steps to pass, it would be important to know what target failed or in what stage it failed. Even more granularity can be achieved in the test results when priorities like Showstopper, high, medium and low are given to each target. Assigning priorities allow the user to decide which targets are most critical to test success.

So, the SRA approach uses an indicator called **Weighted Failure Count (WFC)** that is assigned for each test run to indicate the status. The **WFC** is the actual Weighted Failure Count computed (of actual number of failed targets) by the analyzer immediately after the test run is completed. Where as **WFC Max** is the maximum possible number of failed targets for a given test run computed based on the targets in the ResultTargets.xml. The **WFC Max** is static and will not change during run time.

The Test status indicator is presented as the ratio of actual number of failed targets (Actual WFC) to a maximum possible number of failed targets (WFC Max) for a given test run.

Test status indicator = Actual WFC / Maximum Possible WFC.

So, when **WFC** for a test run is **0**, it indicates the test run passed. If **WFC** is equal to **WFCMax** it indicates total or complete failure of a test run. An intermediate value **0 < WFC < WFCMax** indicates still a fail status but gives an idea of how many targets failed.

How is WFC or WFC Max Calculated?

WFC is based on weight or priority given to a target.

- Show Stopper = 5,
- High = 3
- Medium = 1
- Low = 0

For example ResultTargets.xml file has the following conditions (called targets):

3 targets as Show Stopper
2 targets as High
7 targets as Low

In this example **WFCMax** (the maximum number that the test could fail) is $(3*5) + (2*3) + (7*0) = 21$. This number is constant for a given test and will not change during test run time.

Include only actual failed targets to compute **WFC** for a test run. This number may change for each test run. In this example, the following is the result from test run:

3 targets as Show Stopper (2 passed and one failed)
2 targets as High (all passed)
7 targets as Low (all failed)

Now **WFC** for this test run is: (including only the actual failures)

$$(1*5) + (7*0) = 5$$

So the test status indicator **WFC/WFCMax** in this example is:

WFC/WFCMax = 5/21

Reading e-mail reports:

In conventional result analysis, quickly finding failure patterns due to network, platform and machine related issues is very difficult and many times not possible. In the SRA method, by looking at the e-mail run report users can quickly find a failure pattern by reading the target failure messages. A sample report is shown in ***Listing 4***

Listing 4 E-Mail Report

```
Subject: MySoftware UserID Build_version_LINUX cadnnn.intel.com
FAILED Started: 05/20/05-13:39:02 Completed: 05/20/05-17:16:46
```

```
Test Results for MySoftware: 05/20/05 (13:39:02) on cadnnn.intel.com
```

```
Regression Status: FAILED
```

```
=====
= Details of Regression Status:
=====
```

```
=
```

```
Test Suite: GUI
```

```
-----
```

```
Number of tests run: 4
Number of tests passed: 2
```

```

Number of tests failed:      2
Number of tests unresolved:  0

Failed Tests Summary:
-----
MySoftware-g_gui-g_gui-t_mytest1-d_design1-null
    Memory Peak Check Failed
    PASSED: 4 FAILED: 1 WFC: 5/25 SS: 1/5
    ShowStopper: RunTime:PeakMemory <=
    GoldStatus:RunTime:PeakMemory
    NO : 129 <= 12
-----
MySoftware-g_gui-g_gui-t_mytest1-d_design2-null
    GUI ReadFile Interface check failed
    PASSED: 4 FAILED: 1 WFC: 5/25 SS: 1/5
    ShowStopper: GUI:ReadFile == GoldStatus:GUI:ReadFile
    NO : 1 == 0

Passed Tests Summary:
-----
MySoftware-g_gui-g_gui-t_checkfilesize-d_design3-null
    Output File design.dat size check passed
    PASSED: 5 FAILED: 0 WFC: 0/25 SS: 0/5
MySoftware-g_gui-g_gui-t_controlpopup-d_empty-null
    PASSED: 5 FAILED: 0 WFC: 0/25 SS: 0/5

```

Applications of SRA

Integrated Flow testing

In the CPU design cycle phases like floor-planning, layout, timing, and routing, many different CAD software tools are used, often utilizing output from one tool as an input to the other. This poses a great challenge to the testing infrastructure to validate the design flow as a whole, not just separately validating individual tool's features. To ensure success and realize competitive advantage, integrated flow test cases are used to validate the entire design flow.

By adopting the SRA approach, integrated test cases were created by assembling individual test cases and running them serially. This also made multiple tool development teams work closely in order to provide competitive advantage to whole flow rather than addressing individual tool problems. Some of the potential bottlenecks, performance issues and data corruption scenarios were easily identified and improved.

Integrated flow testing also made multiple tool development teams adopt standard user interface and common components to provide the end user value as a whole CPU design platform. This reduced CPU designers training time in learning the product, and let them reuse the gained time in the CPU design itself.

Non-Functional Requirements Testing

Lack of bugs is commonly recognized as an indicator of software product quality. This is often measured by number of bugs, defect rate and reliability of product features. Experience reveals that this indicator is not sufficient to ensure competitive advantage or improve the customer satisfaction with a software product.

So, it becomes necessary to gauge the performance of a product feature by how fast it ran, how easy it was to use, how well it worked with other products/features, how much less memory it used, how well it exploited platform features like hyper-threading etc., Non-Functional Requirements testing is used to ensure these quality goals. It is collectively called **Quality of Service** (QoS) in the industry to define software product quality.

Using SRA, test cases were written to capture and test non-functional requirements like memory usage, peak memory usage, garbage collection, parallelism using hyper-threading, time taken for disk I/O operations, CPU cycles, network throughput etc. Competitive products also eventually get the same job done. But how it was done is the key differentiator to gain competitive edge. By adopting SRA in non-functional requirements testing, benchmarking test teams at Intel were able to demonstrate and differentiate the competitive advantage of our tools over the competition. In many cases customers were able to run test cases by themselves to see the results to believe. Developers were able to quickly address performance bottlenecks, carefully apply platform specific efficiencies and fine-tune workflows. This tremendously improved the QoS of CAD tools and increased customer satisfaction.

Results

By adopting the Smart Result Analysis approach in the test infrastructure used by CAD Software tool development and CPU design projects at Intel, competitive advantages were realized in multiple areas:

- SRA was incorporated into the Intel internal testing infrastructure called CERT and used by CAD Software tool development and CPU design projects. Its fast and wide acceptance across various Intel sites worldwide for CPU design projects proved SRA's competitive advantages.
- SRA proved very beneficial in quickly identifying test failure patterns in large regression runs and resolved them very quickly saving significant amount of turn around time. Many of the problems were caused by few bad machines in the pool that attracted many test jobs and failed them. These bad machines were identified quickly with the help of SRA reports and removed from the pool. Bad network connectivity, disk overflow, display issues for GUI testing are some of the other leading problems that got quickly resolved with the help of the SRA approach.
- Developers found the SRA approach very helpful in quickly root-causing the test failures and reproducing them. This was one of the top competitive advantages at the hands of developers for quick turn around time in making fast releases by running the entire suite of regression test cases. Because of this advantage we saw increased participation from the developers in writing well-designed test cases with more emphasis given to well-defined targets.
- Smart result analysis enabled developers to write data collection and benchmarking test cases to test and analyze non-functional requirements like memory usage, output file size and performance.
- Using SRA there are no more false test failures caused by an innocent addition of a print statement in the code or an un-harmful format change to the design file output. Parsers used in the SRA approach were smart enough to ignore those changes, and extract only correct information from the status log and test output data files.
- We realized a reduction in number of test cases by carefully exploiting the power of targets and handling multiple steps in a single test case. Because of the multiple steps more integrated tests were written to effectively improve the quality of the code by testing multiple product features in a single test case. This decreased the maintenance effort of a huge test base and improved the quality of test code itself. This also resulted in less test bugs to fix.
- Using the Smart result analysis, CPU design projects were able to squeeze in project timeline and expand the scope of testing many product features and input combinations. Problems were caught in early stages and fixed before reaching customer land.

Acknowledgements

Author would like to acknowledge reviewers Kit Bradley and Tekchandani Ruku for their valuable inputs to improve the quality and structure of this paper to a great extent. Many Thanks!

Bibliography/References

1. Manoharan S Vellalapalayam, "CERT - A Software Validation System and Regression Infrastructure", Design & Test Technology Conference 2006.
2. Weyuker, E.J., "An empirical study of the complexity of data flow testing", Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop
3. Laski, J, "Testing in top-down program development", Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop
4. DeAbren, B, "Test software design techniques for reuse and portability", AUTOTESTCON Proceedings, IEEE Sept. 2000.
5. Hoe Jin Jeong, "An Integrated Database Benchmark Suite" First International Conference on Semantics, Knowledge and Grid (SKG'06), November 2005

Lean Cuisine – How Lean Thinking Bakes In Quality

***Jean Tabaka* Rally Software Development**

The Lean movement began in the 1950s when Toyota's redefinition of automobile production put the entire manufacturing world on its ear. Although the impact Toyota has had on the world automotive market is well-known, what may be less known is the role quality has played in this explosive paradigm shift. "Lean Cuisine" explores the basics of Lean Thinking and brings it to our quality-tuned ears-how customer value has driven quality; how quality has driven software development; and how can we create value and quality by attacking the 7 wastes of Lean Software Development.

Jean Tabaka is an Agile Coach with Rally Software Development in Boulder Colorado. With over 25 years of experience in the software development industry, she has navigated numerous waterfalls in a variety of crafts (government, IT, consulting) and in a variety of roles (programmer, architect, project manager, and methodologist). Her move to agile software development approaches came in the late 90s as a result of studying DSDM in the UK. Since that time, she has become an agile devotee, consulting with teams of all sizes worldwide wishing to derive more value faster through the adoption of agile principles and practices.

Jean is a Certified ScrumMaster, a Certified ScrumMaster Trainer, and a Certified Professional Facilitator. She holds a Masters in Computer Science from Johns Hopkins University and is the author of *Collaboration Explained: Facilitation Skills for Software Project Leaders* published in the Addison-Wesley Agile Software Development Series.

Lean Cuisine— How Lean Bakes in Quality

Jean Tabaka, Agile Coach

Rally Software Development
Boulder, CO
Jean.tabaka@rallydev.com

Copyright 2003-2006, Rally Software Development Corp

Overview

- Origins of Lean
- The 3—5—7 March
- The Zero Effect—Attacking Waste Creates Quality
- The Lean Diet

The Origins of Lean

Copyright 2003-2006, Rally Software Development Corp

Where did it all begin? Toyota Production System (TPS)



Taiichi Ohno,
Toyota 1950's:
concentrate on
value

Lean
Cuisine

Something had to change dramatically in order to succeed

While it is absolutely true that old ways are easier, it is a mistake to conclude that they are therefore better. People must be given strong encouragement and urged to continue with the improved procedures.

Shigeo Shingo, educator



Shigeo Shingo:
Process analysis,
motion analysis, and
time study analysis

5

Jean Tabaka, Rally Software Development, 2006

Lean
Cuisine

Lean means creating and delivering value for the customer



- Every action, every artifact must create and deliver value

6

Jean Tabaka, Rally Software Development, 2006

Lean
Cuisine

Value means removing waste: don't overburden, smooth out the flow



- Muri – overburden
- Mura – uneven flow
- Muda – waste

**Lean
Cuisine**

7

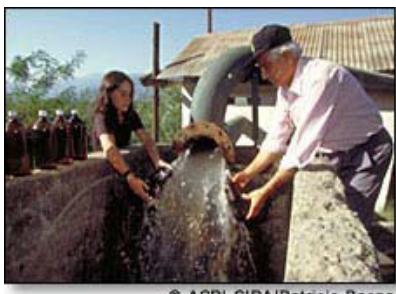
Jean Tabaka, Rally Software Development, 2006

The 3—5—7 March

**Lean
Cuisine**

Copyright 2003-2006, Rally Software Development Corp

The Three: Ohno's Lean Principles define the fundamentals



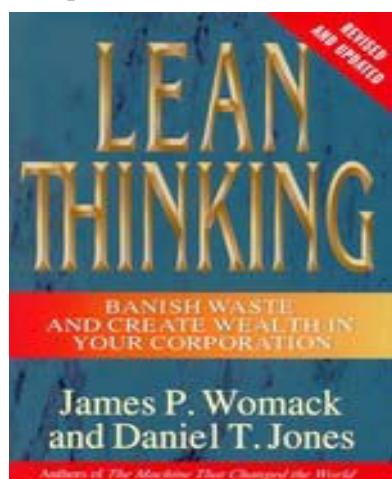
© ACDI-CIDA/Patricio Baeza

- Muri – pay attention to how you load the system
- Mura – remove any bottlenecks that cause uneven flow
- Muda – attack waste ruthlessly

Jean Tabaka, Rally Software Development, 2006

Lean
Cuisine

The Five: Womack and Jones emphasize value and perfection



- Value
- Value Stream
- Flow
- Pull
- Perfection

Jean Tabaka, Rally Software Development, 2006

Lean
Cuisine

10

The Seven: Lean manufacturing establishes seven disciplines



- Eliminate Waste
- Amplify Learning/Increase feedback
- Delay Commitment
- Deliver Fast
- Empower the Team
- Build Integrity In
- See the Whole

**Lean
Cuisine**

11

Jean Tabaka, Rally Software Development, 2006

The Zero Effect—Attacking Waste Creates Quality

**Lean
Cuisine**

Copyright 2003-2006, Rally Software Development Corp

The Seven: Eliminating waste means targeting specific practices



Manufacturing

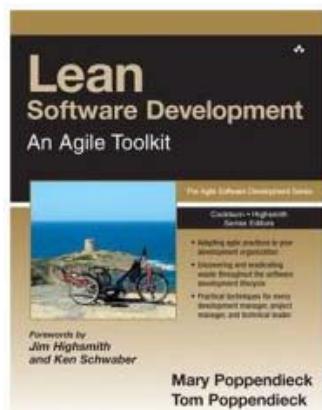
- *Inventory*
- *Extra Processes*
- *Overproduction*
- *Transportation*
- *Waiting*
- *Motion*
- *Defects*

**Lean
Cuisine**

13

Jean Tabaka, Rally Software Development, 2006

The seven wastes in software compare with lean manufacturing



Software Development

- *Partially Done Work*
- *Paperwork*
- *Extra Features*
- *Task Switching*
- *Waiting*
- *Hand-off*
- *Defects*

**Lean
Cuisine**

Lean Software Development, Poppendieck and Poppendieck, Addison-Wesley, 2003

14

Jean Tabaka, Rally Software Development, 2006

Partially done work means value that can't be delivered



- 80% of the components 100% tested versus 100% of the components 80% tested

Lean
Cuisine

15

Jean Tabaka, Rally Software Development, 2006

Paperwork has to be maintained; who is willing to pay for that?



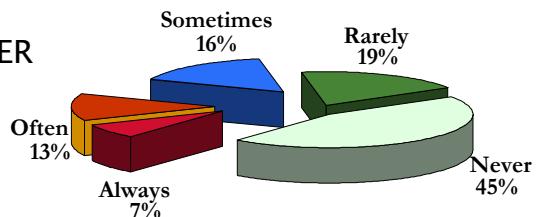
Lean
Cuisine

16

Jean Tabaka, Rally Software Development, 2006

What features are you maintaining that are rarely or never used?

- 20% always or often used
- 64% RARELY or NEVER used



Standish Group Study Reported at XP2002 by Jim Johnson, Chairman
©2004 Poppendieck.LLC

**Lean
Cuisine**

17

Jean Tabaka, Rally Software Development, 2006

How many projects are you switching between at any given time?



**Lean
Cuisine**

18

Jean Tabaka, Rally Software Development, 2006

If you are waiting for functions, or someone is waiting for you, the system is losing value



**Lean
Cuisine**

19

Jean Tabaka, Rally Software Development, 2006

Silo, phase-driven development relies on hand-offs versus collaboration



- What may get dropped?
- How much time does it take?



**Lean
Cuisine**

20

Jean Tabaka, Rally Software Development, 2006

Defects hide technical debt and allow an illusion of “Done”



Lean
Cuisine

21

Jean Tabaka, Rally Software Development, 2006

**The Lean Diet—what to do
more of**

Lean
Cuisine

Copyright 2003-2006, Rally Software Development Corp

Co-locate developers, testers, and product managers



A co-located team eliminates hand-offs and maintains flow

Photo courtesy Jean Tabaka, 2006

**Lean
Cuisine**

23

Jean Tabaka, Rally Software Development, 2006

Load the work through collaborative planning



Developers and Testers:
“What can we get done
in this timeframe?”

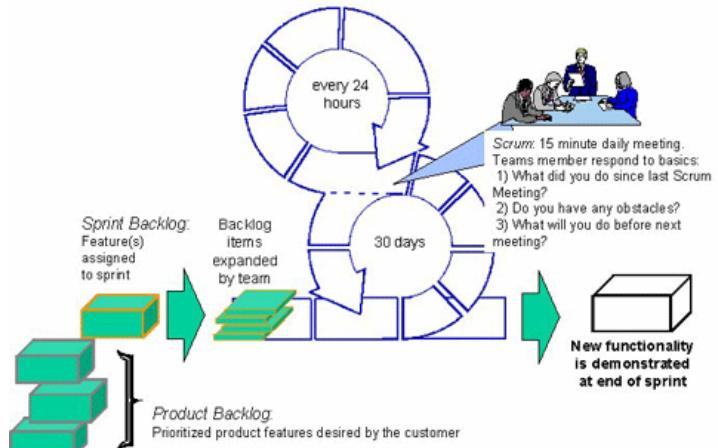
Photo courtesy Jean Tabaka, 2005

**Lean
Cuisine**

24

Jean Tabaka, Rally Software Development, 2006

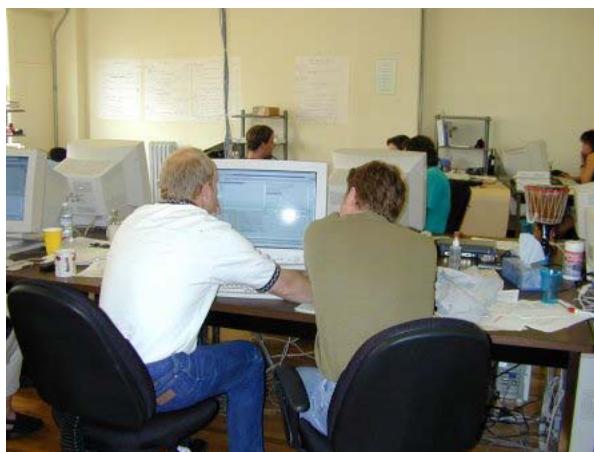
Consider Scrum practices for load and flow; work in small timeboxes



Jean Tabaka, Rally Software Development, 2006

Lean
Cuisine

Pull testing forward: create releasable tested functions (RTFs)



The tester works with the developer and product owner to declare acceptance of a feature

Kent Beck and Eric Evans pair programming. Photo from Industrial Logic: <http://www.industriallogic.com/xp/ra/pp.html>

Lean
Cuisine

26

Jean Tabaka, Rally Software Development, 2006

Maintain real-time automated status to keep everyone on the same page

27

Jean Tabaka, Rally Software Development, 2006

**Lean
Cuisine**

Additional XP discipline around engineering practices can speed up teams



28

Jean Tabaka, Rally Software Development, 2006

**Lean
Cuisine**

Continually inspect and adapt the process and the team practices



Photo courtesy Michele Sliger, 2005

29

Jean Tabaka, Rally Software Development, 2006

Lean
Cuisine

The Lean Diet—what to do less of

Copyright 2003-2006, Rally Software Development Corp

Lean
Cuisine

Don't overstuff the workload— turbulence decreases flow, slack increases it



Flowing traffic in Boston

**Lean
Cuisine**

31

Jean Tabaka, Rally Software Development, 2006

Don't let defects get through: Stop the Line!



**Lean
Cuisine**

32

Jean Tabaka, Rally Software Development, 2006

Don't leave feature acceptance to the end of the lifecycle



**Lean
Cuisine**

33

Jean Tabaka, Rally Software Development, 2006

Don't allow untested features to build up; the customer must be able to pull features



**Lean
Cuisine**

34

Jean Tabaka, Rally Software Development, 2006

Don't let any work break the existing system; the system always runs and builds



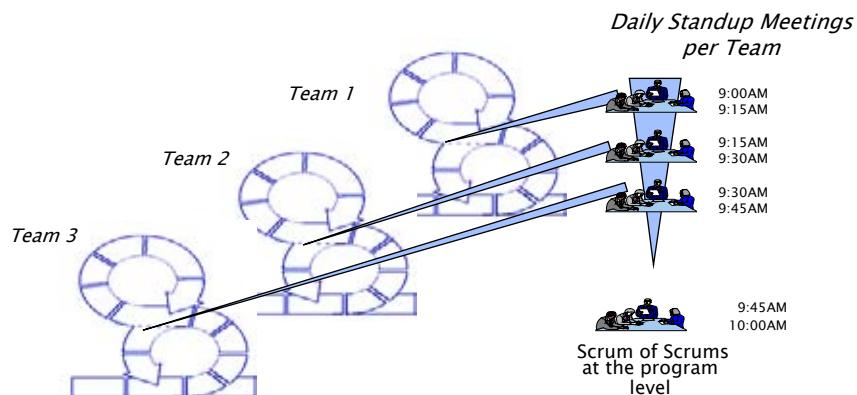
Build status signals

Lean
Cuisine

35

Jean Tabaka, Rally Software Development, 2006

Don't leave status updates for weekly meetings: daily check-ins on quality reduce wait



36

Jean Tabaka, Rally Software Development, 2006

Lean
Cuisine

Don't decrease your effectiveness by being over-matrixed



What are you doing to save time?

<http://www.coping.org/wordauthors/monkey/tasking.jpg>

37

Jean Tabaka, Rally Software Development, 2006

**Lean
Cuisine**

Don't allow organizational silos to impede communication about doneness



38

Jean Tabaka, Rally Software Development, 2006

**Lean
Cuisine**

Stop measuring the wrong things: traditional organizations measure rely on broken feedback loops



"I placed a document in the wrong folder and so received a score of 75% on my process compliance assessment. This impacted my raise for the next year."—paraphrase from recent client

www.pei.literacy.ca/graphics/plain.gif

Jean Tabaka, Rally Software Development, 2006

**Lean
Cuisine**

What Results to Expect

Copyright 2003-2006, Rally Software Development Corp

**Lean
Cuisine**

Productivity can grow and scale

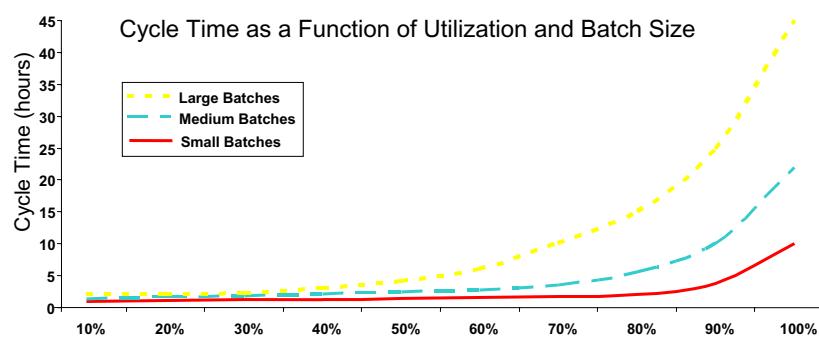
	Waterfall	Project Scrum (Mike Cohn)	Distributed Scrum (SirsIDynix)
Person Months	540	54	827
Lines of Java	58,000	51,000	671,688
Function Points	900	959	12,673
FP per dev/month	2.0	17.8	15.3
FP per dev/month (industry average)	12.5	12.5	3.0

41

Jean Tabaka, Rally Software Development, 2006

Lean
Cuisine

Lean thinking can increase throughput through small batches



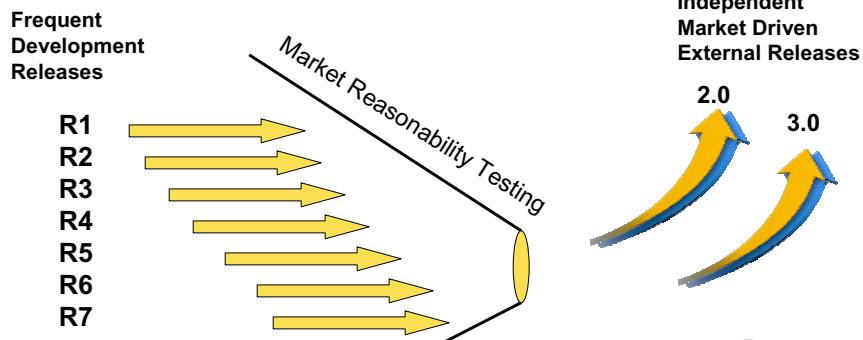
©2004 Poppendieck.LLC

42

Jean Tabaka, Rally Software Development, 2006

Lean
Cuisine

Bottlenecks move outside of development organization; executives manage change with outside groups



Jean Tabaka, Rally Software Development, 2006

The lean organization
is a knowledge-
creating group
focused on
innovation in the
market



Photo courtesy "Collaboration Explained", 2006

Lean
Cuisine

44

Jean Tabaka, Rally Software Development, 2006

Thank You!

jean.tabaka@rallydev.com

Copyright 2003-2006, Rally Software Development Corp



Photo Credits

- Taiichi Ohno: http://www.logistikinside.de/fm/2350/ohno1_400px.jpg
- Shingo quote: http://www.leanaffiliates.com/la_images/mfgquote.gif
- Three Kings:
http://www.bbc.co.uk/languages/spanish/cultural_notes/images/three_kings.jpg
- Trash truck:
http://teamdroid.com/albums/album09/dscn7450_trash_truck.jpg
- Water pipe:[http://www.acdi-cida.gc.ca/INET/IMAGES.NSF/vLUImages/IPS/\\$file/012-WaterSamples.jpg](http://www.acdi-cida.gc.ca/INET/IMAGES.NSF/vLUImages/IPS/$file/012-WaterSamples.jpg)
- Binoculars: <http://www.sylvesterthejester.com/gallery/art/binoculars.jpg>

Jean Tabaka, Rally Software Development, 2006



Photo Credits

- Stop button: <http://www.powellfab.com/images/EmergencyStopLarge.gif>
- Assembly line: <http://www.sahistory.org.za/pages/specialprojects/Luli/Working-Life/Images/5/car-plant.jpg>
- Stacks of paper: <http://www.arc.gov/images/appmag/appmag0101-b2-4.jpg>
- Attic: http://static.flickr.com/28/37664970_bfb90f85b8.jpg
- Busy chicken: http://www.exploratorium.edu/cooking/icooks/images/AS_eggs_.jpg
- Stopwatch: <http://www.udel.edu/learn/dreamweaver/session1site/stopwatch.jpg>
- Baton handoff: <http://www.claybennett.com/images/archivetoons/baton.jpg>
- Magician: <http://www.rainfall.com/posters/images/Theatrical/1683r.jpg>

Jean Tabaka, Rally Software Development, 2006

Lean
Cuisine

First to Market or First to Fail – A General Systems Perspective

Michael Bolton

The first half of the presentation is a report on some links between product marketing, biology, media theory, and data representation, greatly inspired by recent research into general systems thinking, which links these diverse topics. The second half focuses on the role of the tester in projects that are bringing a product to market for the first time.

Traditional thinking in marketing suggests that products that are first in their market category have a tremendous advantage over later arrivals. In his book Crossing the Chasm, first published in 1991 and revised in 2002, Geoffrey Moore identifies several successful and not-so-successful products, and suggests reasons for their successes and failures. In particular, he notes that new products are fundamentally disruptive, and that the majority of consumers naturally develop resistance to disruption. A new product must overcome this resistance.

The arrival of a new product in the market has many similarities to patterns in biology and ecology. New species (similar to new product categories) develop because small but significant numbers of individuals (similar to new product categories) fit into the existing environment and become part of it.

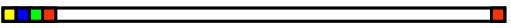
In their 2003 book, McLuhan for Managers, Derrick deKerckhove and Mark Federman outline similar kinds of principles through the lens of Marshall McLuhan's theories on experts and media. They also present tools by which technology companies can analyze the existing environment, or ground, and the impact that a new product—which McLuhan called a *medium*—will have on it. This book makes it clear that McLuhan was a general systems thinker.

A new product will be missing several of the feedback loops that drive innovation, problem fixes, and increased market share. Skilled testers can help to address this problem by heuristic approaches—modeling the users, the product, and the systems in which it works, identifying risks, and providing rapid feedback to management.

Michael Bolton provides training and consulting in Rapid Software Testing all over the world. He is co-author, with James Bach, of the Rapid Software Testing course. Michael has been an invited participant at Cem Kaner and James Bach's Workshops on Teaching Software Testing in Melbourne, Florida, 2003, 2005, and 2006, and was a member of the first Exploratory Testing Research Summit in 2006.

Michael writes about testing and software quality in Better Software Magazine as a regular columnist. He is Program Chair for the Toronto Association of System and Software Quality, a presenter at this year's Amplifying Your Effectiveness conference in Phoenix, and a member of Gerald M. Weinberg's SHAPE Forum.

Michael can be reached via his Web site, <http://www.developsense.com>, or by email at mb@developsense.com.



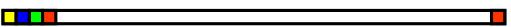
First to Market or First to Fail?

Michael Bolton
DevelopSense
PNSQC
Portland, OR
November 2006



Who I Am

Michael Bolton
DevelopSense, Toronto
Canada
mb@developsense.com
(416) 992-8378
<http://www.developsense.com>



Some Questions

- Is being first to market really an advantage?
- Considering that products that are first to market are often failures, what can testers do to help?
- Are there lessons in marketing that might be important for the mission of advancing tester skill?

What Do These Successful First-To-Market Products Have in Common?

- 
- The Palm Pilot
 - Netscape Navigator
 - The IBM PC
 - Microsoft Windows

Successful First-to-Market Products

- 
- They weren't really first to market
 - They learned from limited successes
 - They integrated with existing systems
 - They were extensible
 - They crossed the chasm

They Crossed the Chasm

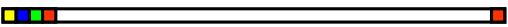
- 
- In Crossing the Chasm, Geoffrey Moore describes
 - the visionaries
 - techno-geeks; technology for tech's sake
 - the early adopters
 - need an edge and are willing to try technology
 - the early majority
 - need to get a job done, and need community
 - One risk for testers: many of us are Moore's visionaries

Summary of the Answers



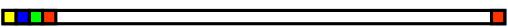
- These products
 - were *good enough*
 - for *enough people*

Technology Marketing



- New products disrupt the existing ground
- Customers develop resistance and immunity to things that might unbalance their systems

Risks for Market Followers



- Some early entries can kill credibility for an entire category
- Example: SoftRAM vs. MagnaRAM

McLuhan and Innovation

- 
- McLuhan defined a *medium* as anything that extends some human capability
 - A medium can be a technology, a product, or an idea
 - Every medium has an impact (or *message*) for the existing state of the art (or *ground*)
 - That is, every *figure* changes the *ground*
 - *A medium is anything from which a change emerges*

Innovation vs. the Expert

- 
- McLuhan contends that experts will resist innovation, because their expertise is invested in the current state of the art
 - "Experts are not only poor at dialogue but are opposed to discoveries."
 - "Breakthroughs discredit experts who are necessarily custodians of the now."
 - "They who know also know 'it can't be done; we've tried everything.'
 - "Specialists treat the disease according to its name, rather than its nature."

Experts vs. Testers

- 
- McLuhan's remarks have cautionary lessons for testers
 - Testers must walk the tightrope between domain expertise on the one hand, and open-mindedness on the other
 - "Testers are people who understand that things can be different."
 - Jerry Weinberg

Parallels in Biology

- Homeostasis
- Evolution
- The Founder Effect

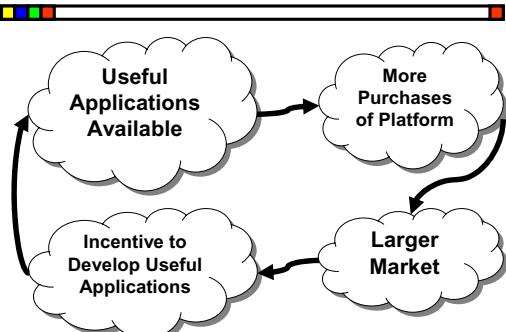
The Founder Effect

- Identified by Ernst Mayr, 1963
- Observed on the island of Surtsey and other volcanic islands
- New populations tend to contain a small number of individuals
- A group of a small number of individuals necessarily shows less diversity than the group from which they were derived... but the new group is distinctly different from the old

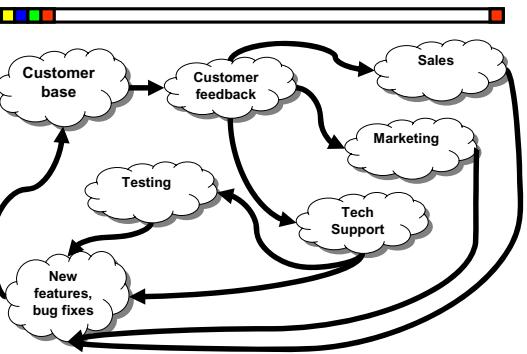
Evolution

- Species survive and evolve through lots of (failing) variations
- Is evolutionary success an adaptive process, or a filtering process?
- General systems principle: your analysis will depend on what you observe
- Species adapt; individuals merely survive (or not)

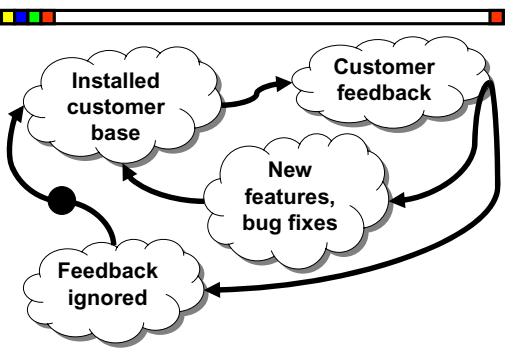
Feedback Loop: Extensible Products



Customer Feedback Loop



Customer Feedback Loop



A Paradox From Tufte

- 
- There are few incentives for meaningful change in a monopoly product (*which a first-to-market product is*)... In a competitive market, producers improve and diversify products; monopolies have the luxury of blaming consumers for poor performances.”
 - emphasis added

Heuristics: Generating Solutions Quickly

- 
- **adjective:**
“serving to discover.”
 - **noun:**
“a fallible method for solving a problem.”

‘Heuristic reasoning is not regarded as final and strict but as provisional and plausible only, whose purpose is to discover the solution to the present problem.’
- George Polya, *How to Solve It*

Types of Heuristics

- 
- **Guideword Heuristics:** Words or labels that help you access the full spectrum of your knowledge and experience as you analyze something.
 - **Trigger Heuristics:** Ideas associated with an event or condition that help you recognize when it may be time to take an action or think a particular way. Like an alarm clock for your mind.
 - **Subtitle Heuristics:** Help you reframe an idea so you can see alternatives and bring out assumptions during a conversation.
 - **Heuristic Model:** A representation of an idea, object, or system that helps you explore, understand, or control it.
 - **Heuristic Procedure or Rule:** A plan of action that may help solve a class of problems.

How Heuristics Differ from Other Procedures or Methods

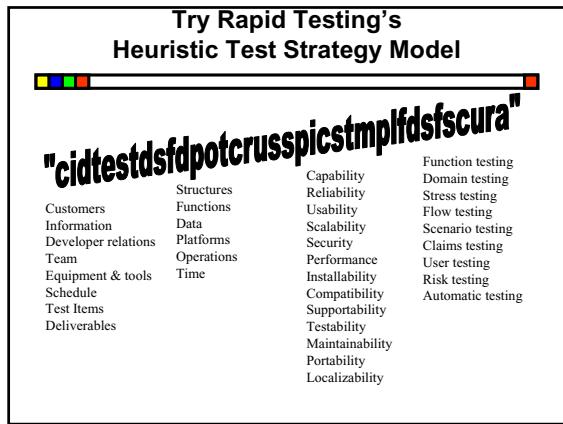
- A heuristic is not an *edict*. Heuristics work implicitly under the control of a competent practitioner.
- Heuristics are presumed fallible and context-dependent.
- Heuristics aid or focus an open-ended solution search.
- Heuristics can substitute for complete and rigorous analysis.

Example of Heuristic Oracles

- We suspect a problem if a product is inconsistent with...
 - History (the past behaviour of the program)
 - Image (of the company or organization)
 - Comparable products
 - Claims (made by anyone who matters)
 - User Needs (as we understand them)
 - Product (internally, within itself)
 - Purpose (the product's intended use)
 - Specification (or standard)
- ...assuming that there is no clear or compelling reason for the inconsistency

Consistency Heuristics with First-to-Market Products

History	✗
Image	?
Comparable Products	✗
Claims	✓
User Needs	?
Product	✗
Purpose	?
Standards	?



Testers and New Products

- Skilled testers can help first-to-market products become more successful
 - heuristic approaches will be necessary
- One role of testers is to think critically about products
- One role of testers is to provide management with information about the product
- This is fundamentally an investigative process

"Plunge in and Quit" Heuristic

Whenever you are called upon to test something very complex or frightening, plunge in!
After a little while, if you are very confused or find yourself stuck, quit!

- This benefits from *disposable time*—that part of your work not scrutinized in detail.
- Plunge in and quit means you can start something without *committing* to finish it successfully, and therefore you *don't need a plan*.
- Several cycles of this is a great way to *discover a plan*.

Try McLuhan's Laws of Media

McLuhan proposed "probes", questions that we can ask about any medium

- What human capabilities or media does the new medium *extend*?
- How might the new medium *reverse* its extension?
- What might the new medium *obsolesce*?
- What obsolete medium does the new medium *retrieve*?

Oblique Strategies: Examples

- Turn it upside down
- Simple subtraction
- Emphasize repetitions
- Use 'unqualified' people
- Emphasize differences
- Do nothing for as long as possible
- Bridges: build, burn
- Short circuit (example: a man eating peas with the idea that they will improve his virility shovels them straight into his lap)

Readings

- Creating a Culture of Innovation
 - Mark Federman
 - <http://individual.utoronto.ca/markfederman/CultureOfInnovation.pdf>
- McLuhan for Managers
 - Mark Federman and Derrick DeKerckhove
- Crossing the Chasm
 - Geoffrey A. Moore
- Quality Software Management, Vol. 1
 - Gerald M. Weinberg
- Rapid Software Testing
 - James Bach and Michael Bolton
 - <http://www.satisfice.com>
- http://en.wikipedia.org/wiki/Founder_effect

Implementing Kaizen and Six Sigma in Application Development

By David J. Anderson, October 2006

Abstract

Many businesses deploy an approach to management, quality assurance and continuous improvement. Two popular approaches are: **Lean/Kaizen** which originated in Japan but was inspired by the teachings of quality assurance experts such as W. Edwards Deming and Joseph Juran; and **Six Sigma** which though most often associated with General Electric originated in the United States at Motorola. It is often true that the information technology departments of businesses using these methods do not have a similar or equivalent scheme. The Software Engineering Institute (SEI) created the Capability Maturity Model Integration (CMMI) [Chrissis 2003] with the intent of creating a framework for a continuous improvement and quality assurance method for software engineering. The author has been responsible for developing **MSF for CMMI Process Improvement**ⁱⁱ at Microsoft Corporation. This paper explains some of the new ideas delivered in MSF for CMMI Process Improvement that enable **Lean/Kaizen** and **Six Sigma** programs for application development projects. Risk management, issue management, project and iteration planning, work item tracking and reporting will be used and mapped against Wheeler's states of control [1992] as a means towards capability analysis and the identification of opportunities for improvement.

Contact Details

david.anderson@microsoft.com

ⁱ For the purposes of this paper the terms Lean and Kaizen are considered synonymous and imply a method of quality assurance and continuous improvement popularized in Japan particularly at Toyota.

ⁱⁱ MSF for CMMI Process Improvement is trademark of Microsoft Corporation. MSF for CMMI Process Improvement is a prescriptive software engineering methodology designed as part of the Microsoft Solutions Framework (MSF) supplied with the Visual Studio Team System product line. It provides guidance, enactment and conformance to process capabilities that implement 17 of the 21 process areas in CMMI model level 3.

Introduction

The author [2003] has shown that software engineering projects can be managed by tracking the flow of customer-valued work at different stages through the application development lifecycle. This approach departs from traditional project management techniques that break work down into tasks and track a graph of dependent task completions using Critical Path Method (Gantt) or Project Evaluation and Review Technique (PERT). Traditional methods typically measure effort expended in comparison to estimates in order to calculate a value for partial work completed. The author's work departs from this and adopts a Lean [Womack 2003] technique of managing the queues of customer-valued work at specialist work functions in the lifecycle. This technique was first described for use in knowledge work problems by Reinertsen [1997]. Tracking the flow of queuing work and managing the queues is inherently simpler and reduces non-value-added work. Time-on-task estimation and any work to track and report actual time-on-task against plan is considered non-value-added and hence waste (muda) from a Lean perspective. The Lean technique of managing with queues is less intrusive and incurs less overhead (waste) freeing up workers to spend more time performing value-added work.

The author's work developed from his experience using the Feature Driven Development methodology [Coad 1999]. Feature Driven Development in turn developed out of Peter Coad's earlier work on object-oriented analysis, The Coad Method [Coad 1995]. This introduced a reliable and repeatable definition for a small piece of customer-valued functionality, known as a *Feature*. *Features* can be tracked throughout the lifecycle of an application development project. Typically, *Features* in development were tracked through 6 stages for application development (figure 1.) [Coad 1999].

This paper expands the author's existing work into a full quality assurance method that enables a continuous improvement approach that is compatible with Lean/Kaizen or Six Sigma programs.

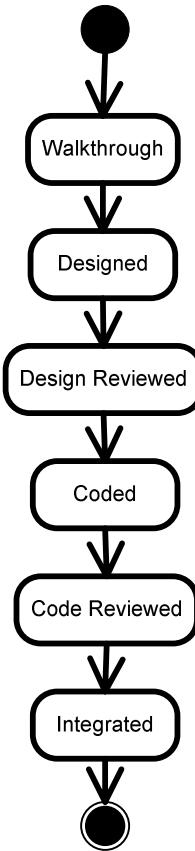


Figure 1. State model for a Feature in Feature-driven Development

Using Cumulative Flow Diagrams

Figure 2 shows a cumulative flow diagram [Reinertsen 1997, Anderson 2003] of a 7-week iteration from a Feature Driven Development project that consisted of a series of similar iterations. In this diagram the designed-reviewed state is not being tracked separately nor is the difference between unit tested and code reviewed tracked separately from integrated into the system build. Hence, the state model is simplified to only 4 states in comparison to the model in figure 1. The cumulative flow data is rich in information. Vertically on any given day, it shows the queue of work items in a given state. The region below the top line shows work not yet started and the darkest section below shows work fully completed. The height in between started and completed shows the amount of work-in-progress. The horizontal distance between started and completed tells us the average time for *Feature* completion (or lead time) at that point in the iteration. The rate of completion at any given step in the lifecycle shows us the production rate (or *velocity*). The size of the queue (determined from the height of different regions in the chart) and the production rate information can be used to predict the future behavior of the team and the iteration. Hence, work-in-progress is a predictor of lead time. Lead time is interesting to the project manager as it directly relates to the project promise and customer expectations for delivery.

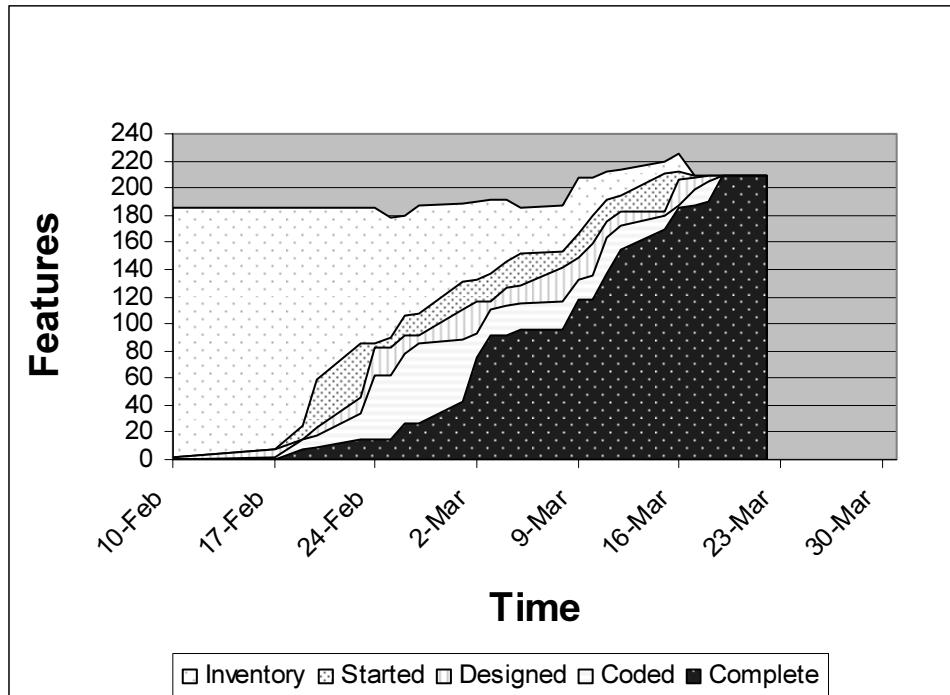


Figure 2. Cumulative Flow Diagram for Feature-Driven Development Project

The production rate, at any given stage in the lifecycle, exhibits variation. The amount of work-in-progress exhibits variation and the queue of work in front of any given stage exhibits variation. This is natural as the production rates of different stages are varying and hence the queue of work will vary accordingly. A growth in the queue of work or the overall work-in-progress can be used to identify bottlenecks.

A bottleneck can be caused in two ways: either there is insufficient capacity to process work items relative to other stages in the lifecycle; or, there is an unusual outage or source of delay reducing capacity or capability. This second cause for a bottleneck is an indication that there is an assignable cause (or uncontrolled) variation [Shewhart 1931].

Control Charts from Cumulative Flow Data

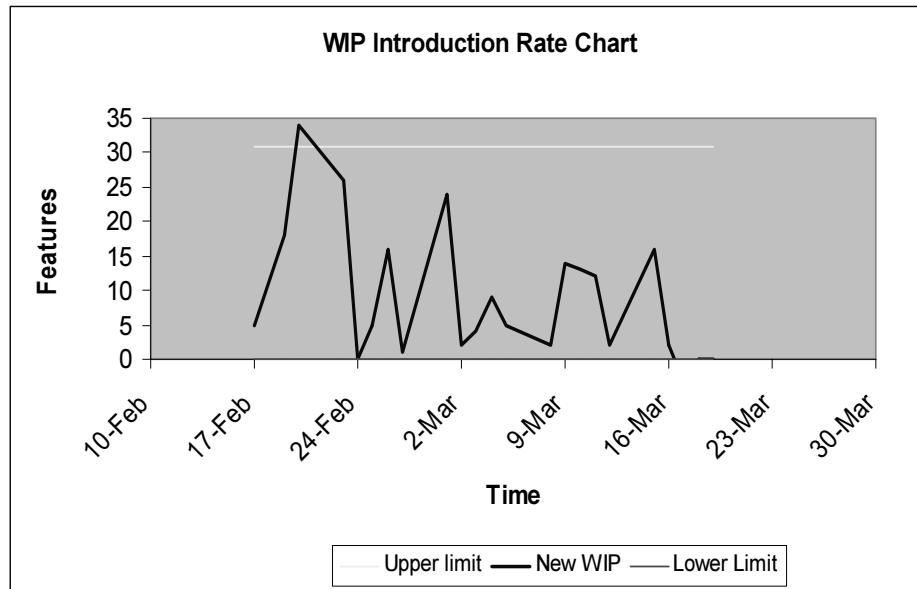


Figure 3. WIP Introduction Range Chart

Figure 3 charts the changes in work-in-process taken from the cumulative flow diagram in Figure 2. The calculation of the control limit on the chart is beyond the scope of this paper. This is the range chart of the WIP Inventory Control Chart, figure 4. It should not be used in isolation.

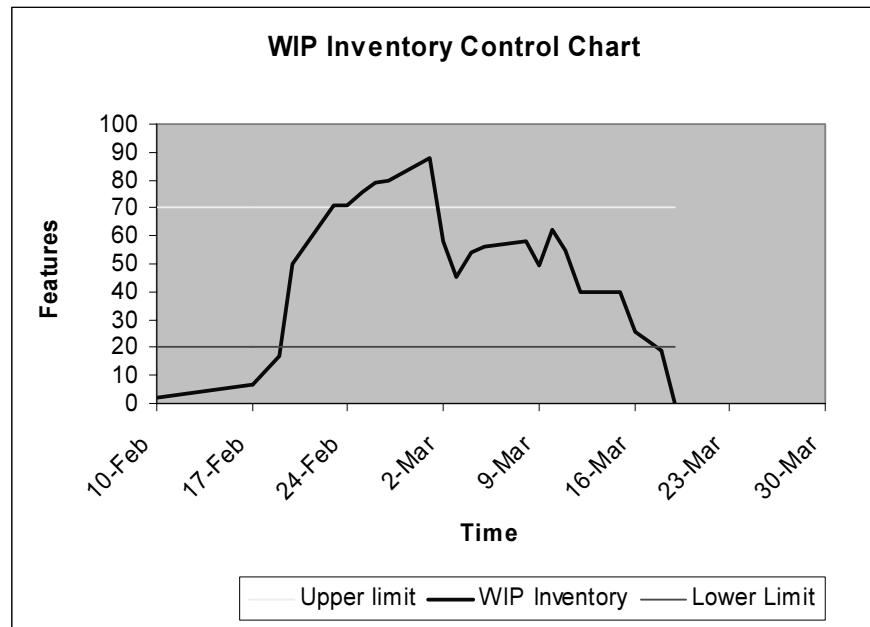


Figure 4. WIP Inventory Control Chart

The author has accumulated significant empirical evidence demonstrating that longer lead times lead to poorer quality. Details are beyond the scope of this paper but a threshold point at around 2 weeks has been observed beyond which defect rates accelerate significantly. A factor of 30 times increase has been observed in initial defect rates between projects averaging 1 week and 3 months for *Feature* lead time. It is perhaps no coincidence that many authors in the agile community recommend iteration lengths of shorter than 2 weeks.

From a management perspective, it is desirable to control the amount of work-in-progress in order to keep lead time from exceeding two weeks. Figure 4 shows the total work-in-progress from the chart in figure 2. February 23rd coincides with an appropriate management intervention requesting that the team make every effort to reduce work-in-progress. There is some lag before the request takes its full effect. Figure 5 shows the lagging trace of lead time for the same iteration. This indicates that the process remained in control within the desired 14 day limit. The project was in fact completed with very low defect rates of around 3 initial escaped defects per 100 *Features*. The Lead Time chart in Figure 5 is less useful because it is lagging. Waiting until lead time exceeds 14 days is likely to cause a much bigger delay in correction because the data is automatically 13 days old when acquired. Hence, the WIP chart (figure 4) is much more useful for management and control, while the Lead Time chart is useful as a report card showing overall project health.

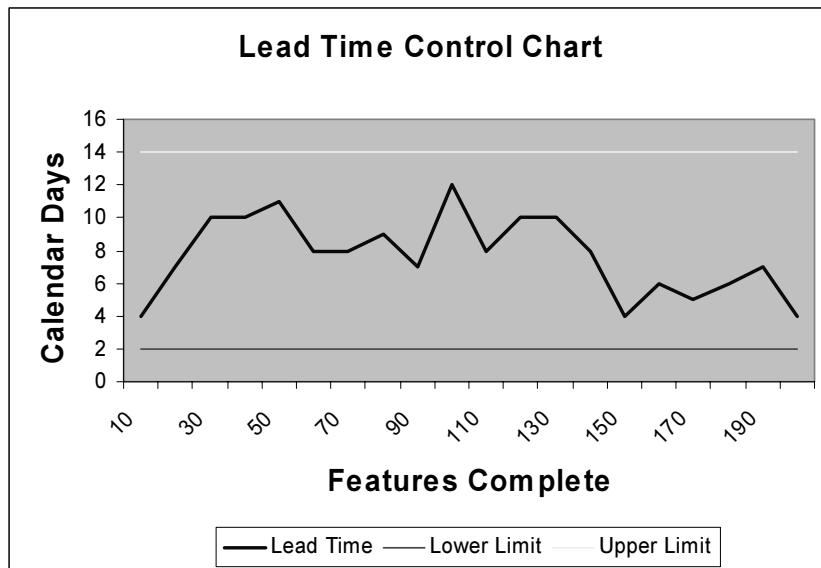


Figure 5. Lead Time Control Chart

Mapping Quality Assurance to Application Development

Wheeler [1992] modeled what he calls the four states of statistical process control as shown in Figure 6.

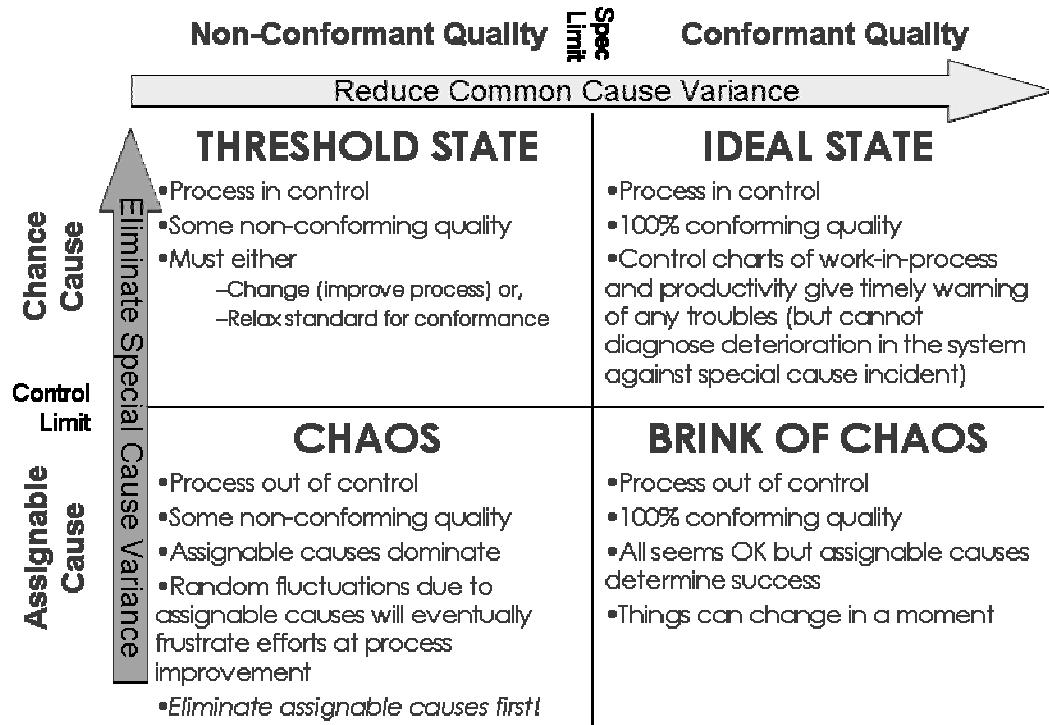


Figure 6. Wheeler's 4 States of Control

Wheeler's model divides the world into two rows – chance cause and assignable cause – and by two columns – conformant and non-conformant quality. The meaning of conformant quality is defined by the market within a particular industry. For application development this is typically defined as “on-time, on-budget, with agreed scope, and within an agreed defect level”. Wheeler states that to move from the left hand column to the right hand column, we must reduce chance cause variation. To do this we can either, improve the system (of application development) or lower the standard – redefine the meaning of conformant quality.

In this context, assignable cause variation implies that the system of application development is not entirely under control. Hence, to move from the lower row of Wheeler's chart to the upper, it is necessary to eliminate (not merely reduce) assignable cause variation. Edwards Deming referred to this as “stabilizing the system” [Deming 2000]. Traditional quality assurance methods follow this pattern. First, assignable cause variation is eliminated and then chance cause variation is reduced. The Software Engineering Institute's Capability Maturity Model Integration (CMMI) [Chrissis 2003] is designed this way. The CMMI model encourages the development of organizational capabilities at model levels 2 through 4 that enable the elimination of assignable cause variation through the use of process areas that focus on gaining control and predictability,

while the capabilities encouraged in the process areas of model level 5 enable the development of a continuous improvement program to reduce chance cause variation.

As both Lean/Kaizen and Six Sigma quality assurance methodologies are designed to drive continuous improvement programs [“Kaizen” literally translates as “continuous improvement”] there is a strong mapping between the CMMI model and the ability to implement Lean or Six Sigma initiatives in application development organizations.

CMMI Approach to Variation

The CMMI accepts that rigorous statistical methods in application development are difficult to achieve because creating an application development team that behaves like a true chance cause system is difficult to achieve. Humans performing knowledge work are involved rather than manufacturing machines. With humans it is all too easy for the skill level of one member of the team to introduce a dominant source of variation. This goes against Shewhart’s original definition for a chance cause system.

An argument could be made that Feature-Driven Development achieves a chance cause system through its use of Feature Teams that work collaboratively on batches of *Features* known as Chief Programmer Work Packages [Palmer 2002]. Because the team works together on each batch of *Features*, the effect of any individual member of the team is lost in the aggregate performance of the team. Hence, there is no single dominant source of variation within the team and as a result a chance cause system results. In a more general approach to application development lifecycle and methodology such as that presented in the Microsoft Solutions Framework and MSF for CMMI Process Improvement [Microsoft 2006], it is unreasonable to assume a chance cause system applies for an application development team. Work is likely to be partitioned and performed by individuals allowing for significant variation in performance across the team, potentially introducing a dominant source of variation. Hence, the CMMI encourages a less statistically rigorous approach to determining chance and assignable cause variation.

Figure 7 shows the Issues and Blocked Work Items chart from MSF for CMMI Process Improvement. In the background of the chart is a bar chart mapping cumulative flow of issues raised during a project. Issues typically identify problems or potential problems in the development of working software. For example, an ambiguous requirement document may be flagged as an issue and an analyst may be tasked with resolving the ambiguity before the developers can write code to implement the requirement. In the event that the issue is not resolved, i.e. the ambiguity is not clarified and corrected in time, then the development of application code may be delayed. This can result in delays to the project schedule. Delayed work-in-progress should be marked as blocked. In the foreground of the chart, a line graph of blocked work items is overlaid on the cumulative flow of issues.

The cumulative flow of issues shows the arrival rate of new issues, the number of currently open issues, and the lead time to resolve issues. If an issue is not closed quickly enough, it will result in work items blocking. Blocking work items imply a growth in work-in-progress and/or a slippage in the schedule. In any event, a spike in blocked work items implies an assignable cause variation. A specific assignable cause may be found in

the issue log. Hence, spikes on the line graph of blocked work items give us a good non-statistical approximation for assignable cause variation happening in the project.

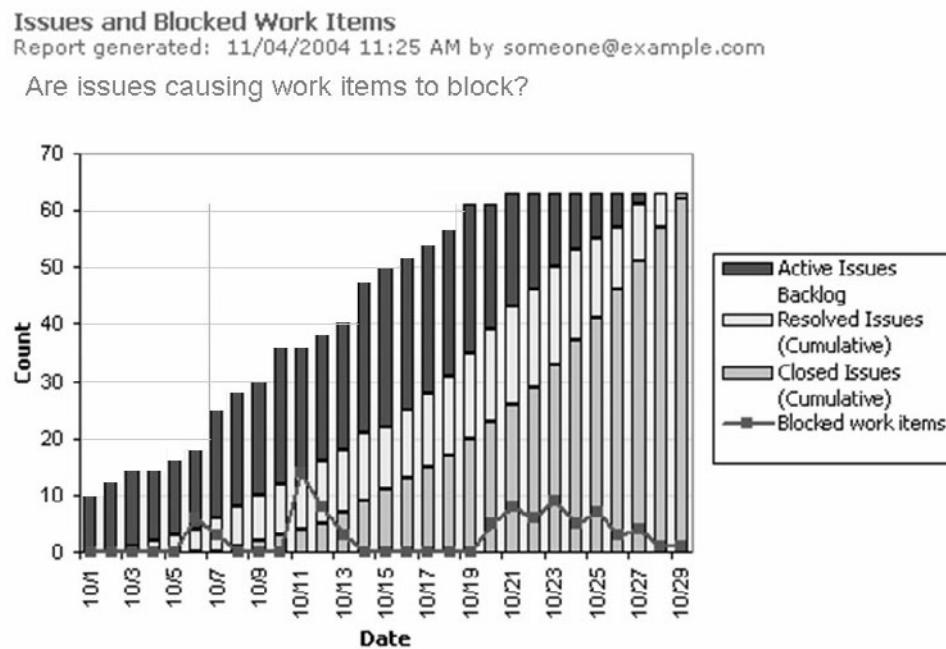


Figure 7. Issues and Blocked Work Items from MSF for CMMI Process Improvement

Chance cause variation might occur in several forms. Figure 8 offers one way of measuring it. It shows the variance in total work items over an iteration within a project. The initial planned scope of work items is shown against the y-axis. The variance through the iteration time period is then shown, breaking out original work items remaining in scope from new work items arriving. There are typically three sources for new work items: change requests – formal requests from the customer to modify the agreed scope; dark matter [Anderson 2003] – items always in scope but omitted from the plan as an oversight during initial analysis; and finally, defects (or bugs) generated during the development of code for the iteration. Defects can be thought of as *waste* from a Lean perspective, while dark matter is simply the result of the imperfection in the analysis technique and the humans performing it. Both can be thought of as chance cause variation. Reduction of this type of variation requires changes to the system (the team of application developers and the methods they use in day-to-day application development.)

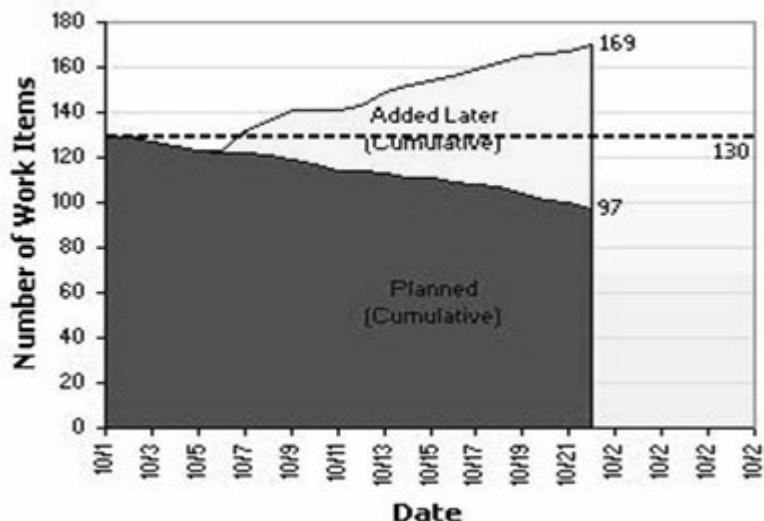


Figure 8. Work Remaining Chart from MSF for CMMI Process Improvement

Hence, through the use of management charts shown in figures 7 and 8, MSF for CMMI Process Improvement provides a non-statistically rigorous method of determining the amount of chance and assignable cause variation affecting a software project. Six Sigma is a process improvement method that seeks to improve processes by objectively studying capabilities and sources of variation in capability. Kaizen is a process improvement method that seeks to identify waste and eliminate it. As variation is a source of waste, Kaizen methods also identify sources of variation and seek to reduce or eliminate them. It is therefore possible to use MSF for CMMI Process Improvement to provide the insight into sources of variation to drive a Six Sigma or Kaizen program.

Six Sigma

Wheeler's chart (figure 6) can be easily mapped to Six Sigma. Both DMAIC (Define, Measure, Analyze, Improve and Control) and DFSS (or Design for Six Sigma) processes relate directly to moving a software development process from right to left on the Wheeler chart. DMAIC can be used to measure variance in estimation against analysis artifacts such as *Features* in FDD and DFSS can be used to insure code written delivers defined requirements.

The Wheeler model also provides a tool for objectively assessing and monitoring the capabilities of different functions in a system of software engineering. Figure 9 shows different projects and an assessment of their capability against a non-statistical approximation of control. The x-axis plots the number of *Features* delivered against the project promise (agreed function) designated by the intersection of the y-axis. Hence, projects falling on the right hand side of the graph delivered more functionality than originally promised and those on the left delivered less than promised. The y-axis designates the amount of variation. Projects exhibiting significant assignable cause variation (many spikes on the chart in figure 7) will appear lower against the y-axis. Projects without assignable cause variation can be plotted above the x-axis. The lower the

percentage of chance cause variation (shown in figure 8) would allow them to plot higher against the y-axis.

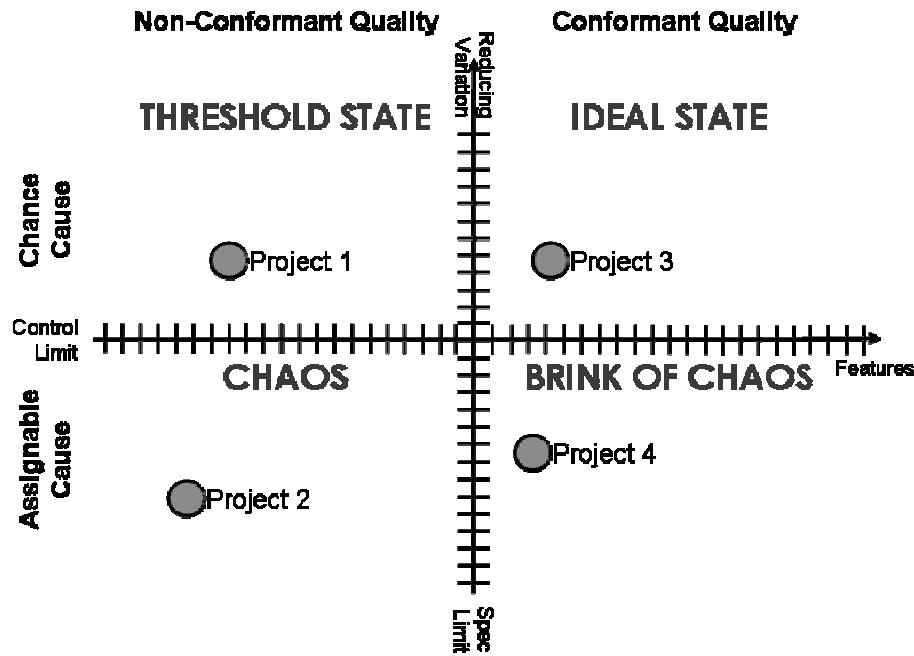


Figure 9. Capabilities Analysis

In Figure 9, a Six Sigma organization would recognize that Project 2 had the worst performance. This project might be singled out for a Black Belt project to examine possible opportunities for improvement to eliminate sources of assignable cause variation and potential reduction of chance cause variation. It is quite likely that there would be a focus on better risk and issue management. The Project 2 team might well be given additional training and encouraged to use tools to better track risks and issues. Management might pay more attention until they are satisfied that the team has issue management under control.

At the same time, a Kaizen program might also look opportunities to drive more productivity and move the project to the right on the chart.

Eliminating assignable cause variation moves the project up on the chart and increased productivity moves it to the right. The best performing project in Figure 9 is Project 3.

However, unlike manufacturing systems where there is generally a very clean separation of assignable cause variation, chance cause variation, and production rate, in knowledge work systems the division is not nearly so clean. In manufacturing problems, Deming and Shewhart recommended that first assignable cause variation was eliminated. Only then should actions be taken to reduce chance cause variation under controlled circumstances. The CMMI model is designed to reflect this approach with model levels 2 through 4 address assignable cause variation first, then model level 5 addressing chance cause variation.

It is very difficult to separate out sources of variation in application development and software engineering. For example, a change to the method of engineering which

produces higher productivity and delivers smaller batches of working code faster is very likely to reduce the number of uncontrolled variations such as change requests. Even changes driven by external forces in the marketplace are reduced as higher productivity reduces time to market and eliminates opportunity for variation. Hence, process improvement in application development is a more difficult problem to manage and control than in manufacturing. Figure 9 is somewhat unsatisfactory as chance cause variation might be reducing while assignable cause variation remains an element in project performance. Figure 10 proposes a 3-dimensional chart that plots assignable cause variation, chance cause variation and productivity on separate axes. Figure 10 allows Kaizen Events and Black Belt projects to be run and attack root cause elements of assignable cause variation or chance cause variation or productivity and track the performance of the changes.

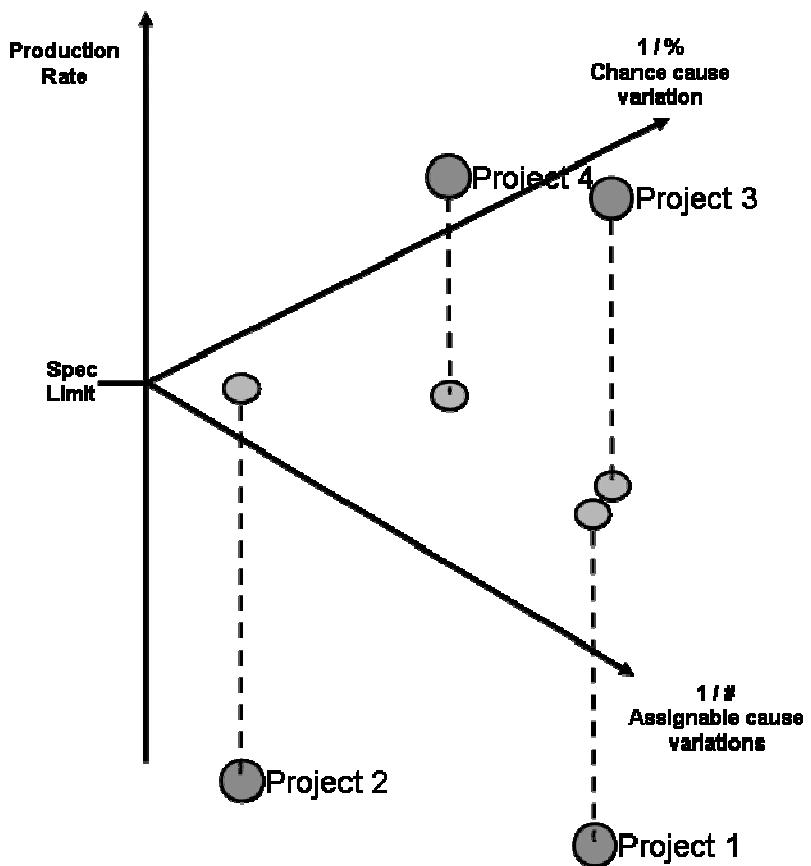


Figure 10. 3-Dimension Capabilities Analysis Chart

Figure 10 translates the data from Figure 9. The best performing project, Project 3, still appears in the top right hand side of the diagram. And the worst performing, Project 2, is still in the bottom left. However, the diagram is now 3-dimensional and productivity or delivery against a project promise (the specification limit) can be tracked

separately from chance cause variation in the engineering process (such as defects introduced.)

Conclusions

Kaizen and Six Sigma quality assurance programs require a process capability to measure chance and assignable cause variation along with productivity and initial quality (defect) data. Modern software development lifecycle methods such as MSF for CMMI Process Improvement and modern work item tracking and configuration management systems such as Microsoft Visual Studio Team System allow for unobtrusive tracking of data relevant to quality assurance. A small set of metrics plotted on only a few (in this case four) charts offer the basis to drive objective process improvement programs.

Six Sigma reduction of variation is supported through the identification of controlled and uncontrolled variation, even when this is done with non-statistically sound methods as shown in figures 7 and 8.

Kaizen events are supported through the collection of cumulative flow data widely used in lean manufacturing. Cumulative flow offers opportunities to manage the queues of work at stages in the lifecycle. It also offers a source of data to drive the study of variation and examine improvement opportunities for waste reduction where the source of waste comes from process variation.

MSF for CMMI Process Improvement is a new application development methodology that includes all the required data collection, reporting and management elements to support Kaizen and Six Sigma quality assurance programs.

References

- [Anderson 2003] Anderson, David J., *Agile Management for Software Engineering – applying the Theory of Constraints for Business Results*, Prentice Hall Professional Technical Reference, 2003
- [Chrissis 2003] Chrissis, Mary Beth, Mike Konrad and Sandy Shrum, *CMMI Guidelines for Process Integration and Product Improvement*, Addison Wesley, 2003
- [Coad 1995] Coad, Peter, Mark Mayfield and Andrew North, *Object Models: Strategies, Patterns and Applications*, Prentice Hall PTR, 1995 (and 2nd edition 1997)
- [Coad 1999] Coad, Peter, Eric Le Febvre and Jeff De Luca, *Java Modeling in Color with UML: Enterprise Components and Process*, Prentice Hall PTR, 1999
- [Deming 1994] Deming W. Edwards, *The New Economics – For Industry, Government and Education* (2nd Edition), MIT Press, 1994
- [Microsoft 2006] Microsoft Corporation, *MSF for CMMI Process Improvement*, <http://msdn.microsoft.com/vstudio/teamsystem/msf/msfcmmi/default.aspx>
- [Palmer 2002] Palmer, Stephen R., John M. Felsing, *A Practical Guide to Feature Driven Development*, Prentice Hall PTR, 2002
- [Reinertsen 1997] Reinertsen, Donald G., *Managing the Design Factory – A Product Developer's Toolkit*, Free Press, 1997
- [Shewhart 1931] Shewhart, Walter A., *Economic Control of Quality of Manufactured Product*, Van Norstrand, 1931; republished by ASQ Quality Press, 1980.
- [Wheeler 1992] Wheeler, Donald J., and David S. Chambers, *Understanding Statistical Process Control*, SPC Press, 1992
- [Womack 2003] Womack, James P. and Daniel T. Jones, *Lean Thinking – Banish Waste and Create Wealth in Your Corporation*, Revised and Updated, Free Press, 2003

Four Behaviors That Hold Testers Back

John Lambert

Microsoft

JLamb@microsoft.com

Abstract

As testers, we sometimes worry too much about what other people think. This paper identifies four behaviors centered around this worry that hold us – and our products – back: asking permission to open bugs; fretting over feature, team, and discipline boundaries; neglecting to interpret data and recognize trends; and commiserating instead of improving.

Author

John Lambert is a test technical lead at Microsoft. He works on test automation, test methodologies, and penetration testing for the Connected Framework team. He is an inventor or co-inventor on six patents, four of which are related to testing. John graduated with Bachelor of Science and Master of Science degrees in computer science from Case Western Reserve University in Cleveland, OH. John has presented at the Pacific Northwest Software Quality Conference (PNSQC), Seattle Area Software Quality Assurance Group (SASQAG), and STAR.

1 Introduction

Generally, testers are independent and strong. However, we are also human and we care about what others think – sometimes too much. I think there are four behaviors that hold testers – and our products – back when we worry too much about what others think. I will also talk about some of the reasons why we do this. As we correct these behaviors, we must be careful to not go too far.

2 Asking permission to open bugs

Testers do not need to ask for permission to open bugs. When they ask if they should open a bug or if other people think it's a bug, it sets a bad precedent and slows (or prevents) the data from entering the system. (It's great to gather more data and opinions – but that should happen after you open the bug.)

2.1 Examples

Email from Tester to feature team: “Hey, I noticed if I hold down the mouse and drag it around our product, then my music skips. Should I open a bug?”

“Can I open this bug now or should I wait until we’re done with this milestone?”

2.2 Possible explanations

Some testers also think that “by design,” “postponed,” or “won’t fix” bugs are bad and that they should never have any of these types of issues. I think if testers don’t have any of these bugs, then they are probably not pushing hard enough or opening all the bugs they find. (I would be very surprised if every single one of the tester’s bugs were fixed. Is the tester opening only the bugs they know will be fixed and ignoring the other issues? Does the feature team always agree with the tester?)

2.3 Detecting/correcting this behavior

This is somewhat difficult to detect because it usually happens via email or in hallway conversations. If you see this happen, then give feedback as soon as you can. Also, if you see testers opening relatively few bugs, then talk to them about it: they could be focused on different tasks or maybe they are asking permission to open bugs.

It is helpful to remind testers of the “other” purposes of bugs: to document behavior, to provide insight into decisions, to help product support, to record alternatives, to note for V.Next, etc.

Management should have reasonable expectations about the percentage of non-fixed bugs. There should be some of these bugs; driving towards 0% will not be productive.

2.4 Going too far (“I’ll open everything I think of!”)

It is possible to go too far and be overly aggressive about opening bugs.

One example is if you ignore the policies in place – written or not – around “buddy build” or “private build” or “smoke testing.” A developer might ask you to check out a private version to see if he fixed the issue. If you find a regression or that the fix isn’t complete, the practice may be to notify the developer instead of filing another bug. It is important to “play nice” here and maintain a good working relationship.

If you open bugs without thinking them through, then you could be going too far. (A very low fix percentage sometimes indicates this.) Not every fleeting thought should be a bug – although if you are not sure, then file it. Very few bugs should include “see me” or “will add later.”

One issue related to – but separate from – asking permission is asking how to group bugs effectively. When you find a cluster of related issues, it is helpful to ask about the best way to file them – not whether it is okay to open them. For example, if you find thirty spelling mistakes across five different help pages, would the team prefer that you open a bug per mistake (30), a bug per article (5), or one bug for the general issue (1)?¹

3 Fretting over feature, team, and discipline boundaries

Testers should be considerate of – but not beholden to – feature, team, and discipline boundaries. The important thing is to get the right product to the right customer at the right time. The customer does not care if it is your feature or his feature or her feature. It is better to have some overlap in tests than to spend weeks perfectly partitioning your scenarios across teams (and probably dropping some scenarios in the cracks – or preventing the team from adding new scenarios because they are not in the test plan). For disciplines, do not limit yourself based on what your business card says: if the specification is not there and you need it to plan your tests, do not wait for others to do it – write it yourself. Time we spend fretting is time where we aren’t testing and time wasted.

3.1 Examples

Tester on Feature A: “That scenario barely touches my feature! Someone else should cover it.”

Tester: “I can’t do anything because the developer hasn’t written a specification.”

Test lead to test lead on another team: “There’s a lot of overlap between our teams, so let’s complete this template to agree on the responsibility breakdown. It’s only fifteen pages and after that we can checkpoint with a status mail every week.”

3.2 Possible explanations

One reason we do this is we always have less time than we would like for testing and we would prefer to “off-load” as much as possible to other teams.

Another reason is that testers are conservative and would much rather see something in writing than need to rely on someone’s [usually good] intentions.

¹ Which would give a more accurate indicator of overall product quality?

3.3 Detecting/correcting this behavior

To be clear: I think that written breakdowns of test responsibilities are useful when they are at the appropriate level of formality with respect to your project, timelines, and goals. Deciding the level of formality is difficult. If you decide to ignore creating a test “contract,” then you can duplicate tests or miss entire areas. If you decide to go all out with meetings, templates, and signoffs, then you can spend more time on tracking/paperwork than you spend testing. I do not want to discourage doing test planning across boundaries; I do want to discourage excessive worrying about boundaries. At minimum, a test contract should include the names of individuals responsible for testing on each team, the features on each team and who is responsible for testing them, the timelines for each team’s testing, information on how progress will be tracked (preferably an automated method), and the next time that the contract will be updated.

If you find yourself worrying about tests because they involve someone else’s area or spending time trying to convince another team to test something, then you might need to step back and rethink your breakdown – or just let it go and test it yourself. I have seen three testers meet for an hour to decide who would implement a test case that would take one of them less than an hour to implement.

If you “always” or “never” do something with respect to a discipline, then you might be fretting over the boundary. If testers never suggest fixes to developers, then the developers might be missing a useful insight. If testers always close bugs, then testers might be spending time on tasks that other team members could do.²

3.4 Going too far (“Let’s ignore feature/team/discipline boundaries!”)

A different problem arises if you retest features or duplicate test cases. It is okay to “trust no one” but if you open more bugs in their feature than in your own feature, then you have probably crossed the line. Instead, help them understand your scenarios and give them ideas for improving their test coverage.

If your day-to-day job starts to look more like someone else’s job than your own, then you have probably gone too far. If you always find the precise location of the defect or always propose fixes, then you are probably spending too much time doing work that a developer could and should do. You found one bug and did some development work – but you could have found two new bugs if you had focused on testing instead of debugging.³ If you never consult with other teams, you may miss an opportunity to have someone else help you with your testing.

4 Neglecting to interpret data and recognize trends

Testers can be too focused on providing “data without judgment.” It is tempting to stick with “just the facts” and raw numbers – but everyone benefits when testers use their unique perspective and experience to interpret the data. Similarly, testers may notice trends but not recognize or communicate the implications, depriving the team of useful data. It is our job to

² Maybe developers could verify fixes to the developer suites or spelling fixes in the documentation.

³ More cynical: if developers aren’t investigating or fixing bugs, then they’re adding new bugs via new features.

give as much insight on where the project is at now, how the project got there, and where the project is going.

4.1 Examples

Tester email to team: “We opened 103 bugs last week.”

Tester (to self): “None of my bugs have been fixed in the last six weeks: developers are too busy adding new features. Oh well. I should get back to writing tests.”

4.2 Possible explanations

Usually testers are asked for reports, not opinions. Unfortunately, many people do not know the real questions they want answered, they only know about the data that they have seen reported in the past. You will need to interpret that data and infer/discover what they are interested in so you can address it. (For example, some people ask about “the net incoming bug rate” when they really want to know if the developers will be able to fix all the bugs before some deadline.)

When you look under rocks, you sometimes don’t like what you find. If people started yelling at you when you pointed out that the project math does not add up, then you may not be so eager to check the numbers next time.

4.3 Detecting/correcting this behavior

If you find yourself thinking, “People will get upset if I say this,” then maybe you should say it.

For every number you include, ask yourself, “What are two possible explanations for this?”
[Bach]

To correct this, start to interpret the data and extrapolate the trends. What could happen to the project if this pattern continues? What could happen to the project if this pattern stopped? What can you do to help make the pattern get “better”?

Be sure to consider your audience and communication format: the project manager may be a better person to talk to than the entire team and concerns about an individual may be better handled in person with his manager than with an email to them and a CC for their manager.

4.4 Going too far (“Everyone gets my opinion!”)

If you analyze and interpret every single number related to your project, then you may have entered “analysis paralysis,” running the risk of not doing any real work. To correct this, keep the reports simple, the data collection appropriately automated, and the error ranges reasonable.

In addition, if you start by focusing on individuals, then you may have gone too far.⁴ Start with the data and investigate what that tells you about the entire team and the project; do not start with one person and examine all of his data. It is your job to look at the team and the product; it is the

⁴ This is also known as “nitpicking.”

manager's job to assess an individual's performance. (In addition, if you over-focus on a few people, then you can miss what is happening to the entire team.)

Another issue is over-extrapolating based on past data. Even if a tester opened 30 bugs last week, it does not mean that she will always open 30 bugs every single week from now until release.⁵

5 Commiserating instead of improving

Although a negative perspective is characteristic of successful testers, it can also be a limitation when it becomes unproductive griping about the work environment instead of concrete steps towards improving it. It is easy to complain about how hard it is to write a test or how much time it takes to set up a test machine but we instead need to channel our energy into improving things – across feature, team, and discipline boundaries.

5.1 Examples

Tester: "I'll wait to look at the test results: they'll just have to restart the test pass anyway."

Tester: "The lab's down again. I guess I'll just go home early today."

Tester: "These dev suites keep missing bugs."

5.2 Possible explanations

The path of least resistance is not doing anything. Complaining is just slightly more effort. Fixing problems is way up there and preventing problems is even more difficult.

What happened the last time you pointed out a problem to your manager? She probably told you to go fix it. Some people will start to solve problems on their own, presenting their manager with a solved problem. Other people will stop bringing up issues – to their manager, at least.

5.3 Detecting/correcting this behavior

It is important to encourage new ideas – no matter when they come up. One time, our lab was down, the team was blocked, and everyone was on edge. A tester sent an email with a creative idea for improving one of the lab processes but a more senior tester aggressively shot down the idea. Discussion – and suggestions – stopped immediately. After the crisis had passed, the senior tester admitted that it actually was a good idea and agreed to investigate it in future milestones. However, it took a few weeks for the lines of communication to open up again.

Leading by example is one of the best things to do to correct this behavior. Once the team sees one person complain about a problem and then go fix it, they will be more likely to do the same.

⁵ Here are two possible explanations: (1) she ran a spell checker on the documentation and (2) she's about to go on vacation so she worked over the weekend.

5.4 Going too far (“Let’s fix everything!”)

If you find yourself solving every problem that comes up in the best way possible, then you may have gone too far. For example, if your builds have failed, then instead of building a triple-redundant system with automatically restarting worker nodes, it may be easier to have a script that can send email on failures. (As a general point, I think it’s often more cost effective to have computers handle error detection but let humans handle error recovery.)

In addition, if you find yourself fixing (or churning) things that are not broken, then you have gone too far. Testers who over-focus on rewriting test automation and redesigning test infrastructure will spend less time testing the product. Checking for who is not opening product bugs – or who is fixing test bugs – will help detect this.

6 Conclusion

Testers need to find a balance between caring and ignoring what other people think. If we care too much about what people think, we can miss out on bugs, waste time fretting over who does what, miss interpreting project trends, and drag down productivity by complaining. If we don’t care at all, we can fill up the bug database with useless bugs, implement the same test two or more times, upset everyone on the team, or miss bugs while we fix every problem we encounter. By being aware of these four behaviors, we can start to correct them – without going too far. Our testing, our teams, and our products need us to do no less.

7 Acknowledgements

The author would like to thank Randal Albert and Yabo Wang for their insightful review comments.

8 Related work

[Bach] “A New Paradigm for Collecting and Interpreting Bug Metrics.” STAR East 2005.

Step away from the tests: take a quality break

John Lambert

Microsoft

JLamb@microsoft.com

Abstract

Designing, implementing, and running tests is critically important – but sometimes testers need a break. This paper describes four “not testing” techniques that improve quality in a short time slice: watching bugs, helping developers, talking to other testers, and increasing positive interactions.

Author

John Lambert is a test technical lead at Microsoft. He works on test automation, test methodologies, and penetration testing for the Connected Framework team. He is an inventor or co-inventor on six patents, four of which are related to testing. John graduated with Bachelor of Science and Master of Science degrees in computer science from Case Western Reserve University in Cleveland, OH. John has presented at the Pacific Northwest Software Quality Conference (PNSQC), Seattle Area Software Quality Assurance Group (SASQAG), and STAR.

1 Introduction

When testers focus solely on writing and running tests, they detect bugs but they miss opportunities to prevent defects and they are not able to improve their tests and testing skills. Plus, on a long enough timeline, they can get bored with the project and some testers even get burned out on testing as a discipline.

Before stepping away from testing, our test coverage perspective could look like this:

		Filled = using test technique								
Feature (Tester)		1	2	3	4	5	6	7	8	9
	X									
	Y									
	Z									

Every feature is being tested with technique 1, but most of the other techniques are only used on one or two features – and no feature is being tested with technique 8.

By stepping away from the tests and applying the following techniques, we can fill in the gaps and transfer knowledge across the team.

2 Watching bugs

Testers who read bugs in other feature areas are able to identify opportunities for improving product quality. We can automate numbers, trends, charts, and projections, but only a tester can find patterns and suggest improvements to test processes and test coverage.

2.1 General

Because you are doing this as a knowledge transfer/learning exercise for yourself, it is not necessary to be precise: it usually does not matter if it is eight, nine, or ten bugs, or if you start weeks on Monday or Sunday. However, if you are going present to management, then you will need to be more accurate.

2.2 Across the product

One technique is to perform a simple word analysis: as you look at the titles of the bugs across the product, are there any words that seem to occur frequently? For example, you might find that most of the customer-reported bugs refer to an “exception” or an “error.” This suggests that increasing your negative test coverage would help find these issues sooner.

2.3 By team

Another approach is to look at a team’s bugs: what types of bugs have they found recently and what types of bugs are they fixing? If the team is finding many stress issues and fixing spelling errors, it might be a sign that expectations are mismatched. If the team is finding hundreds of bugs each week and fixing tens of bugs, then they might not be on track to meet the next deadline.

2.4 By tester

Pick a tester on the team and look at the bugs she has opened. Is there a template or pattern that she uses? For example, a tester might enter very detailed environment information because she is performing more configuration testing (and finding more issues). Could the same type of bug exist in your feature? Could the bug affect your feature?

2.5 Developers

Look at all the bugs fixed by a certain developer. Are her bugs rooted in the same class of defects, like loop counters, type conversions, or pointer errors? On the overall development team, who has been fixing bugs recently? For example, our team recognizes both “sprint” bug fixers (most bugs fixed in the past week) and “marathon” bug fixers (most bugs fixed in the past month).

The intent is not to punish, but to help predict and learn. A developer fixing a lot of bugs doesn’t necessarily mean that the developer is introducing lots of bugs: maybe the developer finished her high quality code early and so she’s been helping out other developers out with their lower quality code. If you see that a developer fixes lots of race conditions, maybe she would be able to help you debug an intermittent issue.

2.6 What’s not there?

When looking at bugs, also be sure to look at what isn’t there. Perhaps you have been finding a class of issues that no other team has found yet. Can you teach the team how to find those issues?

3 Helping developers

By working closely with developers, you can help prevent defects from being added to the product while opening up the lines of communication.

The key thing here is to offer your assistance, respect their wishes, and meet your commitments. Don’t let them down!

3.1 With the design

Offer to brainstorm, take notes, and review the design. Another way to help is to contact potential customers (without violating their privacy) to discover what they would like to see in your product.

3.2 With testing

Offer to help your developers with their tests. You can review their tests and suggest additional variations or test cases. By educating them on some of your test techniques, you will improve their testing [Marick]. If your test team has a separate infrastructure, you can offer to run their tests in your lab to increase the platform coverage. Share your test harness and test tools for with them so they can benefit from them and write more tests faster.

3.3 With bug investigation

Offer to investigate a bug for the developer. It is important to balance your investigation with your other deliverables: you could be writing more tests or finding new bugs instead. To address this, you might spend two hours on a bug and then report back to the developer with what you tried, what you found, and what you think should happen next.

3.4 With bug contents

Developers usually have preferences for the contents of bugs – as well as pet peeves. Sometimes making a developer happy is as simple as removing WORDS IN ALL CAPS. Other times, you may need to update the database to force or standardize data entry: our “product build number” was optional and developers wasted time trying to identify when tester first discovered the bug.

4 Talking to other testers

When you talk with other testers, you can learn a lot about other parts of the product and other test techniques – both of which will improve your own testing. Reading test documentation on your monitor in your office helps some, but “ad hoc” hallway conversations can be just as – or more – effective: most specifications are out-of-date soon after they are completed, but testers can always tell you about the current state of the world.

4.1 Finding testers

Wandering around the hallways is a great way to find testers, but be sure that you wander around at different times: because of meetings and schedules, not everyone will be around at one in the afternoon, nine in the morning, or six at night.

4.2 What they're testing

Talk to other testers about the parts of the product that they are testing. How do their features interact with your features? For example, you might discover that a tester is responsible for a feature that you used to own a few versions ago, allowing you to share your experience with her.

4.3 How they're testing it

What is their test approach? Are they using different tools? What kinds of test design techniques are they leveraging? You might find that you could benefit from the code they use on their features – and maybe you could both work on common library of test code.

4.4 Bugs they're finding

What kinds of bugs are they seeing in their features? Discussing a bug in person can teach you about how a tester investigates and isolates issues, which will help you with your testing.

4.5 Worry list

What kinds of things are they worried about for the overall project? For example, one tester pointed out that the test lab was going to be down for maintenance two days before the release. We raised the concern to the project management team and they decided that it would be better to postpone the maintenance just in case we needed that time to check any last minute fixes.

4.6 Weekend plans

Although slightly off-topic, asking what they did (or plan to do) over the weekend is a great way to get to know your coworkers better. This helps us see each other as people and to communicate better during stressful times in the project.

4.7 Bugs you'd like to see fixed

You can also use this to tell them about bugs that you have found in their area that you would really like to see fixed. It is in the bug database, but it never hurts to repeat yourself – or say things twice.

4.8 Keep passing it on

Document and share what you learn, giving appropriate credit and maintaining confidentiality. You could mail the technique to everyone, referencing a discussion with tester. Or you could talk to the project manager directly about an anonymous tester's concern. My team has had success with a Wiki page on test techniques: when a tester finds an interesting bug, she can add the technique and a story about the bug so everyone can see.

5 Increasing positive interactions

Because testers get to be both the bearers of bad news and the messenger who is shot, I think it is a good idea to focus on increasing positive interactions within the team. Making other people feel good is a reward unto itself – but the entire team benefits when people feel validated and encouraged, too.

5.1 Neutralize your bug reports

The first step is to stop making people feel bad. In [Black], Rex Black describes this as:

9. Neutralize. Being the bearer of bad news presents the tester with the challenge of delicate presentation. Bug reports should be fair-minded in their wording. Attacking individual developers, criticizing the underlying error, attempting humor, or using sarcasm can create ill will with developers and divert attention from the bigger goal, increasing the quality of the product. The cautious tester confines her bug reports to statements of fact.

This also holds outside of bug reports: emails and communications to the team.

5.2 Food

The quickest way to the team's heart is through their stomachs, so bring in some food and tell the team about the location. Homemade items are always welcomed, and accounting for special dietary considerations is much appreciated.

5.3 Catch them doing something right

It is probably easy to recall mentioning that a specification lacked details. However, when was the last time you wrote a “thank you” for a great specification? It is important to be as detailed in this positive feedback as you would in negative feedback: if you do not explain why the specification was great (level of detail in the user scenarios), the other person may assume it is for a different reason (formatting and layout).

6 Logistics

Integrating these techniques into our projects requires careful attention to logistics: the time we have on a project is finite and there are always more tests to write and run.

6.1 How do you find the time?

With everything else we have to do, how can you find the time to do these tasks?

6.1.1 Schedule it in

You can schedule this work into your planning. However, it is not effective to have everyone do this at the same time: if everyone reviews the same bugs at the same time, the conclusions will roughly be the same.

6.1.2 Cut your least important test

When you look at all your test work, you could probably remove the least important test and do these types of activities instead.

6.1.3 Do it anyway

Another is to just go ahead and do these activities on top of all your other work. This does not scale well but it may force you to better utilize your normal “down” time: the product is setting up, so instead of reading blogs, you could go talk to someone or look at the bug database.

6.2 When do you do what activity?

There are three models for structuring these activities: occasional, role-based, and hat-based.

6.2.1 Occasional

When you need a break, go do something else. The drawback is that this is not very predictable or guaranteed to cover all the activities: some testers naturally gravitate towards certain activities.

6.2.2 Role-based

One way is to assign each activity to a different tester and have them specialize for the rest of the milestone. You have covered all the activities and a tester can become very good at one activity. The drawback is that not all testers will learn how to do each activity.

6.2.3 Hat-based

In “Six Thinking Hats,” [de Bono] describes a model where you wear a metaphorical hat to aim your thinking in a certain direction: you put on your ‘red’ hat and then describe how you feel, etc. This model can also apply for “not testing” activities. The drawback is that it requires a bit more discipline (to do the things you do not want to do) and tracking (to make sure you cover everything). Personally, I use the “hat-based” model: I spend 30 minutes – on a different time and day – on one of the techniques each week.

6.3 How much do you need to prepare?

These activities should be semi-structured, not forced. If you don’t have anything positive to say, don’t make up something – but do think deeply about why you only have negatives. If you’re talking to testers about how they test their features, don’t interrogate them for hours at a time – but do start with a couple of open-ended questions.

6.4 For managers (and those who have managers)

Most managers are accustomed to – and may even prefer – context-switching between tasks and environments. Most individual contributors are not used to it and so managers may need to encourage these testers to step away from their tests, even if it’s somewhat uncomfortable for the tester. There are a few ways that we managers can support them in their growth.

First, we need to explicitly give testers permission, encouragement, and time to perform these activities. We should give everyone this, not just the “senior” testers or those with extra time on their schedule. If only a few people participate, then they will feel out of place and the activities will quickly fade.

Second, we can explicitly remind them that it’s okay – and expected – for “nothing” countable to come out of these activities: they might not find any more bugs [this week] by talking to other testers or reading through the bug database. It’s still important to track and consider the time invested in these activities: someone who spends *all* their time talking to other testers and none of their time writing test cases is probably going too far.

Finally, we can lead by example and do these activities ourselves and share the results with the team. However, we need to be clear that we are doing these activities as individuals, otherwise the team may feel that they don’t need to do them: everyone should be looking at bugs – not just the manager.

7 Conclusion

It takes a bit of “stretching” to step away from your tests. These activities are outside the normal comfort zones of test design and test development. And these activities sometimes don’t even have a clear ending: I start with a general goal (“learn how Kim is testing her feature”) without knowing if I’ll have a pleasant discussion or find a nasty bug farm. But, if each tester makes a conscious decision to improve their test coverage, develop their skills, and advance the test team, then the increase in quality benefits our teams, our products, and our customers.

8 Acknowledgements

Lee Copeland introduced me to the phrase “catch them doing something right.” Ian Savage provided a thorough and thoughtful review.

9 Related work

[de Bono] “Six Thinking Hats.” Back Bay Books. 1999. ISBN 0316178314.

[Black] “The Fine Art of Writing a Good Bug Report.” Quality Week 2000.
<http://www.rexblackconsulting.com/Pages/Library.htm>

[Marick] “Testing for Programmers.” 2000. <http://www.testing.com/writings.html>

Using Social Engineering to Drive Quality Upstream

A study in how simple social practices can force competitive advantage.

Thomas Gutschmidt (thomg@microsoft.com)

Abstract:

The paper highlights some of the struggles test groups face when attempting to create lasting change in corporate environments. From an individual contributor's perspective, this paper specifically targets a tester's individual skills to demonstrate how simple social rules can help push great ideas into implementation. Major topics include:

- Social Engineering – Getting the community onboard, and using both a tester's mindset and the tricks of their trade to push quality upstream.
- Reputation and Courting Attention- Using guerrilla tactics to give visibility to testers in an organization.
- Forecasting – Accurate data collection, and using this data to cost for the future and to push change.

Biography:

Thomas is currently employed as a Software Design Engineer in Test at Microsoft, working for the Unified Communications group on software technologies covering communication and collaboration. Thomas joined Microsoft as a Software Test Engineer in 2003. Previously he worked in various capacities for a small handful of software and technology companies, and also as freelance writer, and social worker.

Introduction

Early in 2005 a small team was put together from Microsoft's Real Time Collaboration Group with a simple charter - to deliver a new, innovative product. This product was very different than ones the group had delivered up to date. From the beginning the team seemed to realize that this was a unique opportunity, but to be successful the rules of the game might need to be changed.

With this realization, the test group researched several development methods and models, and came back to the small team with a proposal to use a method named "Express Development". This paper speaks to the social engineering techniques of persuasion and influencing that the test team used to fight for and integrate these new ideas into the existing product development cycle. Most importantly, it explains how an individual contributor, a tester, can bring processes like these or others to their own group.

The Art of Persuasion

"To succeed in the world it is not enough to be stupid, you must also be well-mannered."
— Voltaire

Why use social engineering, or persuasion? Admittedly either can conjure negative connotations; images of clever telephone hacker tricks, malicious email, and identity thieves.

But put these aside for now, let us say you have a great new idea for improving the efficiency of the software development cycle. Maybe it is THE great new idea, one you so passionately believe in you can hardly contain yourself. This idea, this plan, will augment customer satisfaction, increase product quality, and lower costs to the team. So what do you do with this idea? You go to your leaders, your teammates. You petition the developers and managers in your organization, and you pitch your idea. Then the following happens:

1. **No one seems interested:** You jump up and down on your soapbox, but you receive only blank stares.
2. **You are disregarded:** You are quite ready to accept the lavish praise for your brilliance. Instead? You are ignored
3. **At some point the software ships regardless of the great idea you had:** You are certain that the product is much less of a revenue maker, or weaker, or less reliable, and that customers are much less happy with it because your great idea was never implemented. Yet life goes on.
4. **The process begins again:** Only the next time around, you keep your ideas to yourself, keep quiet, and just do your job.

I'm not saying that this how it is everywhere, I'm just saying that, in my experience, this is a common experience in testing organizations. There seems to be a common

misconception that testers are simply a small roadblock between the product being finished and the product being released to customers.

In general I've found that people working in the software business are extremely bright. I've also discovered that most people in the software business, specifically testers, tend to have multiple ideas on how business can be improved. Sometimes these ideas are simple process improvements, sometimes they are grand thoughts about how products should be designed throughout their lifecycle. Social engineering comes in when a tester has an idea for a process or business improvement, but feels as though they are disenfranchised by their role.

Wikipedia (<http://www.Wikipedia.org>) offers two definitions for Social Engineering. The first is:

“...the practice of obtaining confidential information by manipulation of legitimate users.”

If this were a paper on manipulation, my topics for implementing change in the organization would be:

- Determining psychological weaknesses of people and exploit them.
- Seducing co-workers to your way of thinking.
- Using lay-offs and other threats to keep employees in line.
- Blackmailing and playing on the fears of the population.

If you are expecting me to teach you these sorts of techniques, prepare to be disappointed. Admittedly, there is an art to manipulation as a social engineering technique, but that isn't the focus of this paper. Instead, we turn to the second definition of Social Engineering also from Wikipedia, one that relates more to political science:

“...efforts to influence popular attitudes and social behavior on a large scale, whether by governments or private groups.”

This definition focuses more on the ability to be persuasive or using techniques to persuade and push change. With this more broad definition any law that changes human behavior, for example, can be considered a form of social engineering.

Social Engineering is actually a term that needs to be overcome, because no wants to be “engineered”. No one likes to be coerced, or tricked into doing something, and coercion will eventually create reactions that will work against you. Yet, in order to implement change, you need people to agree with you and to support you.

I'm not proposing coercion and sneaky tricks to implement test-driven change in an organization (well, perhaps a few sneaky tricks). I am proposing methods of persuasion. You can force people to move in a direction of your choice using coercion, but better still

is to make people want to move in a direction you choose. You can do this a number of ways that are not underhanded:

1. **Appreciation:** One of the deepest principles in human nature is the craving to be appreciated. William James (considered the father of American psychology) is probably the first person credited with this idea.
2. **Be Friendly:** Black Hat social engineers use this tact: "*When in doubt, the best way to obtain information in a social engineering attack is just to be friendly.*" (*Sarah Granger - Social Engineering Fundamentals, Part I: Hacker Tactics*)
3. **Use the Truth:** Back up your convictions and desire with strong data.
4. **Unselfishly try to serve:** This is voiced in prose that dates back to the dawn of literacy. The world is full of selfish people. So rare is the exception that they have an enormous advantage.
5. **Give them what they want:** At a fundamental level, many human actions are satisfying some basic need, such as health, hunger, security, the desire for money, or to attain a feeling of importance.

Persuasion, in a nutshell, is a mixture of being appreciative, friendly, honest, unselfish, and giving people what they desire. These are simple principles that I cannot take credit for, instead I have compiled these ideas on persuasion from popular literature on influencing people (please see the listed of useful resources and references in the bibliography at the end of this paper).

Example: Dogfood (Getting others to do the work for you, and taking the credit)

In this paper I use a running example of “dogfooding” to illustrate some of the principles of social engineering. From Wikipidea:

In many development environments, to "eat [one's] own dog food" refers to a point at which a product under development -- early enough in the process to have been previously not-useable -- is delivered, even in its rough state, to all on the project for use... ...In extreme cases, management may issue a dictate that everyone in the organization is to "eat their own dog food" (meaning, for example, "use the latest version of our in-house email program"), as a way of verifying that the product works under real-world conditions. It is often the source of comic-chagrin (such as popularized in the Dilbert cartoon) among workers when such a dictate comes earlier than is practical (if, for example, if the in-house email program can't yet send mail.)

A good example of dogfood might be a car manufacturing company where all the employees own the same make of vehicle that they produce. An example of not dogfooding would be an airline executive who has never flown coach.

Dogfood can be an incredible blessing from a test perspective because it can root out those hard to find issues you find in real-life outside of a test environment. Dogfood can also be a curse, especially in software development, since early versions of the product may contain many bugs, cause crashes, corrupt or lose data, create instability, or otherwise make software unusable. Examples of dogfood issues include:

- Poor quality dogfood releases can actually cripple the business processes that rely on them.
- Unorganized dogfood will create lots of similar issue reports, and sifting through irrelevant issues can tie up the team responsible for the reports (normally the test team).
- Dogfood is a manpower intensive process, using up a large amount of human resources across teams.
- A dogfood process that is poorly implemented will be perceived as randomizing and a waste of time by other teams.

Because of these issues, the biggest hurdle to dogfood adoption in any organization is often community engagement, and the time, skill and commitment it takes to support a community of dogfood users.

Our team's goal with dogfood was to make software quality everyone's job every day. We wanted everyone in the organization, no matter what their role or responsibility, to be involved in the prevention, finding, and the verification of bugs. Our development strategy relied on getting working bits to everyone in the organization early in the development process, and then somehow talking everyone into using it and then taking the time to report any issues or suggestions back to us. In order to make this happen, we had to use a number of persuasion tactics:

Appreciation

Appreciating dogfood participants starts with tracking dogfood users. Know your audience by keeping detailed lists of the people who participated. Once you know who has participated, thank them often and profusely. Appreciation can start with one person, just take one participant using dogfood and make an example of them with some sort of public appreciation for their effort. If it is possible and appropriate, thank their managers as well. Then rotate the appreciation around so it sprinkles across multiple teams and groups.

You can also keep a community engaged with bribery, using self interest as a lever to move people. To make other people come to your dogfood you can use bait and give out awards like team t-shirts, recognition, free food, etc. Giving appreciation to individuals and teams that do participate also helps motivate those who aren't quite onboard yet.

Be Friendly

When you engage a community, engage them in a professional and friendly way. Use polite (not forceful) emails, with some tastefully injected humor (if you are good at that sort of thing), and a sense of teamwork. When you see participants in the hallway who have participated, take some time to say hello, smile and ask them about the experience. Our team has a designated community manger whose job is to be the voice for interfacing with dogfood users, and to advocate for important issues voiced by the dogfooding community if necessary.

Use the Truth

Be honest to the community about the problems and drawbacks of using your dogfood. Publish a FAQ for problems with known workarounds, and make sure dogooders know about any harmful effects of the software before installation. Expose your bug database publicly to dogooders. Beware of unmanaged expectations from your community, such as features that may not be implemented yet, or have been recently cut.

Give them what they want

What dogooders want is really what our customers want; a product that actually helps or entertains them. The onus is essentially on the Test group to deliver to dogfood users a high-quality stable product, be the voice of the customer, and take a reasonable stance before letting something into the workplace that might greatly influence productivity in a negative way.

Unselfishly try to serve

Dogooders are really serving your team, taking extra time out of their already busy day to help you find issues in your software. Because of this, you need to treat dogooders like real customers. Have an established way to escalate issues, and a community forum (newsgroup, distribution list) where people can ask questions. Make sure the whole team (test, developers, managers, etc) are subscribed to any community forums and listening to the pulse of the dogooding community. Come into each interaction with a “You are here to help” attitude.

Provide for your dogooding community members an easy way to report on bugs or how to log product suggestions. Once they’ve logged an issue, give them an easy way to track the issue, so that they can then see how their suggestions and bugs affect the end product. Dogfood issues should always be high priority.

Influence: Gaining Trust and Courting Attention

*I always got more attention than anyone else. If I hadn't, I would have made sure I did.
-Salma Hayek*

Expanding influence involves first gaining the trust of other people in your organization. Trust is a necessary predecessor for initiating change, so if you have trust problems in your organization it will be difficult to initiate any sort of change.

If a tester or test group is trusted, then they are more likely to be engaged early and frequently in the product development process by other stakeholders. If testers are engaged earlier and more frequently, then they in turn can help ensure that products are designed with testability in mind, and the probability of meeting or exceeding product quality goals increases.

Once trust exists between your team and the rest of the organization, you then bring in public relations by promoting visibility to you and your team outside of your immediate sphere.

There are many “tricks” that can help with influence, including flattery, bribes and other sorts of positive reinforcement. I talk about are three basic tactics that are specifically helpful for testers in a development environment; Frequent Delivery, Volunteering, and Community Engagement.

Tactic Number One: Frequent Delivery

One important technique for establishing credibility is to deliver on your commitments. This can mean several things, but to me it is mainly that, when you make a commitment, or any sort of declaration to do work, you make good on that commitment. There is quite a bit of literature devoted to setting goals in business or for career growth, but there are some very specific tricks that have helped me:

1. Work on small tasks, and divide large projects into small tasks. Our team tries to narrow work items down to a day or less. I've seen teams itemize all work into two hour items. Keeping work items small can help contributors reflect over growth consistently and frequently.
2. When creating tasks change "we should..." to "we will...". Make sure your plans can be implemented.
3. Have a well developed criteria for what “finished” is, or a clear definition of “done” (usually called an Exit Criteria). Often this is as simple as a bulleted list of what will be completed, but sometimes listing out what will not be finished is as important.

The idea that a milestone should have established and documented exit criteria before work begins is a common theme among different development paradigms. Using an Exit Criteria, for instance, is one technique you see used in SCRUM methodology:
[\(\[http://en.wikipedia.org/wiki/Scrum_%28development%29\]\(http://en.wikipedia.org/wiki/Scrum_%28development%29\)\)](http://en.wikipedia.org/wiki/Scrum_%28development%29)

Tactic Number Two: Volunteering

Volunteering can be championing a cause in your own group like security, driving a morale event, or being a point of contact for another group. It can also mean helping out in a non-test sort of way. Going above and beyond the call of duty, and delivering above and beyond consistently, really gets peoples attention.

For example, if specification writer is blocking your progress because a specification is not finished, check with the writer and see if you can help. You could assist by authoring part of the specification yourself. Instead of waiting for a developer's check-in, see if you can help out by authoring a data-flow-diagram for them, or assisting with a code review. Our product group encourages this buy allowing blurring between disciplines. It is not unknown for product managers to write code, testers to write low level hooks in the product for automation, or for specs to be based on a Developer's task list.

This kind of discipline blur can be especially beneficial for testers, who are often jack-of-all-trades to begin with, with lots of varied interests and hobbies, and very diverse abilities. They are often ideal candidates to pitch in where there are gaps in other groups.

For testers without diverse skill-sets, volunteering can also be a big part of their growth, helping them to gain new experiences and skill sets.

In addition, the tester's role is one where you have to, by nature, be good at many different sorts of things. For instance, in many groups, testers need to be able to:

- See the product from the perspective of multiple different customers in order to advocate for them.
- Understand the product from the developers point of view, and build a strong technical understanding of a feature in order to thoroughly test it.
- Be able to understand the feature specifications and requirements well enough to write tests.
- Write a test specification that's as detailed as a feature specification or feature design document.
- Often write code as well as a developer at their same level of experience.

In order to just survive in the profession, a tester needs to be able to do many things very well, and since they possess the skill and discipline to wear multiple hats and assume multiple roles they can be ideal volunteers.

There are also some drawbacks to volunteering:

- Testers who volunteer on side projects are taking time away from actual testing, and balancing this is important. If volunteering is taking away from delivering on commitments (frequent delivery), you aren't going to make gains on your sphere of influence.
- With several team-members wearing different hats, the question arises as to whether or not there should be disciplines at all. An organization has to be careful not to lose focus of what each team does best.
- Many contributors fear taking on challenges outside their realm because their boss is tracking them using tools specifically dedicated to track their discipline, and they fear they won't be recognized in their own role. There is some truth to this; if you currently have trouble meeting your daily work load this is not a good tactic.

These drawbacks can be mitigated in a few ways, mainly by showing the value of volunteering. Volunteering can help bring greater visibility to a test team, but it can also help streamline the test effort. Two examples:

1. Helping a developer write low level test hook. Automation support at this level helps the tester understand the code he is testing better, and leaves a sample for the developer to use for further test hooks.
2. Helping a specification writer get a specification released earlier in the schedule means you will be able to get to testing earlier in the schedule.

Tactic Number Three: Community Engagement

You can propel your influence or your team's influence even further by engaging the community outside of the test group, and outside of your project team. Community engagement involves ensuring that your entire organization knows that you are delivering on your commitments and going above and beyond the call of duty.

Community engagement can happen in a number of ways. You can develop a relationship with people outside your team by holding brown-bag sessions, organizing lectures on the technology, giving tech-talks on test automation, collaborate on design docs, working on test patents around the product, or blogging. These are just little things that engage the community, and remind them what it is you are delivering.

Example: Dogfood

Again I bring up the recurring example of dogfood. Here, I outline how Frequent Delivery, Volunteering, and Community engagement can be used to drive successful dogfood..

Frequent Delivery

For dogfood, frequent delivery carries the goal that our group always has a build-able, usable, dogfoodable, and testable product, each every single day. Every build that comes out of development should be of a high enough quality to hand-off to the dogfood community for usage. We have a few processes that help this effort:

- Developers use private branching in the source code tree when product development requires big code changes that may cause disruption.
- Ensuring the feature works before check-in, we do this by testing private builds for features that can be destabilizing.
- Use automation, unit testing, and code coverage analysis to force higher quality check-ins.
- Build the product automated tests along with the product.
- Always test new bits as soon as possible.

I've seen teams with even more rigorous build and check-in systems, where build servers do continuous integration and every check-in to the source code repository triggers a build. Once there is a build the servers run all automation and static analysis before the build is released to the test group for acceptance testing.

Having a rigorous integration check before check-in helps ensure high stability in the product. It allows us to frequently release dogfood builds to show-off new features, and get community support for testing specific areas. We can also release dogfood often, and get bits to stakeholders early. If our stakeholders have bits early, we can retrieve valuable feedback and possibly market data early, creating a chance for the community to drive requirements. These frequent releases help our visibility and help our test team appear competent.

Volunteering

Working with a dogfood community helps to take your work role past testing. Normally in our group Product Managers engage any community. Testers who become engaged in dogfood help ease the work load of the PM. We once had an internal Vice President using our dogfood product who had an issue. After a few emails didn't resolve the issue I met with him along with a developer in off hours (VPs have pretty busy days scheduled) to work though the issue. It turns out that his problem wasn't a technical one, but rather around a design decision the team had made that he did not agree with. Our team turned this issue into a requirement and began working towards incorporating the changes into our product.

Community Engagement

Once dogfood is out, the real work with the community begins. You need to keep dogfood fresh in the minds of the population for continued use of the product.

Our group managed dogfood releases by first releasing to a small set of people, basically a beta only for groups that worked closely with our team. We asked these groups to try the code, and send any bugs and feedback they had to the test organization. Then we made sure the team focused to fix these issues. Every time we added functionality to the build, we increased the scope of the release to a slightly bigger team, and each time, we had a slightly larger group sending requirements, issues, suggestions, and bugs our way.

If you are developing a product that people cannot integrate into their job easily then you should also be supporting dogfooding customers with ideas on how to use the product. Write up scenarios and suggest to dogfooders how they can spend time using the product the way it is meant to be used.

You can also engage the community and control test scenarios by organizing bug bashes against the dogfood bits. These are small, short, organized test scenarios with large groups.

Although “cool” is subjective and varies greatly between organizations, dogfooding a product can also be made cool with a small amount of spin:

- Use Flashy Code Names: teams with names that sound cool (RedTeam, BlackDog, Mojo, etc).
- Exclusivity (ala Google betas): You can start with a select core of people, but advertise to a larger group. Then you slowly roll out to more “friends and family”. Exclusivity breeds a bit of public relations “buzz”.
- Matching t-shirts for those who find issues that get into the product.
- Being publicly labeled an “alpha” user may carry clout in some organizations and groups.

Forecasting: the End is Everything

"I always skate to where I think the puck is going to be." - Wayne Gretzky

Forecasting from a social engineering perspective is about powering change with data. The quickest way to secure peoples minds, and to garner their support, is by demonstrating as simply as possible how an action will effect them. Actions and effects can be predicted to some degree by watching trends in simple things testers have ample access to, like bug databases and product schedules. Forecasting involves scheduling, costing, defect estimation, creating decent documentation, and also being actively engaged in the design phase of the product.

Accurate Costing

It is hard to forecast if you don't know how long it takes to do things. Cost estimation is very difficult to do in software (McConnell talks about this extensively in Rapid Development), especially early in the process. There are some techniques that really help and are common enough that tools exist for software development, three of which, give pretty reasonable results:

- *Constructive Cost Model*: A costing technique designed by Barry Boehm that models software complexity based on program size and estimated lines of code. COCOMO doesn't take human factors into account well - <http://en.wikipedia.org/wiki/Cocomo>.
- *Wideband Delphi*: A cost estimation technique involving bringing a team together, having each make an anonymous estimate, sharing the estimates and assumptions with each other. This cycle is repeated until a consensus number can be reached within the group. As opposed to COCOMO, this technique is very human oriented, but can be manpower intensive and subjective - http://en.wikipedia.org/wiki/Wideband_Delphi.
- *Frequent Estimation*: A known mitigation for poor cost estimation and scheduling is to re-cost and re-schedule frequently. As you travel down through the development cycle, the unknowns become less and less frequent, and costing increases in accuracy. Scrum takes this to the Nth degree and has schedule costing and estimation happen daily (as part of daily meeting). These costs are rolled into a burn down chart that is a warning indicator of common problems and of estimation problem. - http://en.wikipedia.org/wiki/Scrum_%28development%29.

There are also several common defect prediction and estimation techniques:

- Defect Density: Comparing the number of defects per line of code between versions.
- Defect Pooling: The technique is to tracks two arbitrary pools of defects. The total number of defects is equal to PoolA + PoolB minus defects in both A and B. Total projected defects equals PoolA*PoolB/Common Defects.
- Defect Seeding: A technique where a group inserts a set of defects, sees which are found, and uses that to determine the ratio of bugs remaining.
- Belief networks (Bayesian) - http://en.wikipedia.org/wiki/Bayesian_network.

Our team used frequent estimation for common daily feature development, and Wideband Delphi to help predict costs that are further away on the calendar. I also use defect pooling to predict random testing (fuzzing).

Contributing to Design

In order to plan for the future, testers need to be part of the product planning. It is easier to predict the future of a product's development if you are actually involved as part of the design team. During the design phase of a feature a tester can assist greatly by steering the design towards something that is testable. I believe that a testable feature is often also well designed and usable.

Ensuring that testers continue to be a welcome part in the design of a product means first making sure the test group is involved in the design process. Once invited, it means going to design reviews being prepared to contribute. This means finishing all the required reading ahead of time, looking at similar implementations of the feature for hurdles, and perhaps even creating a rough test plan with costs for testing. It is also a good idea to run over bug lists against similar features so you can start thinking about the kinds of bugs that will pop up. Armed with some information, a good tester can actually find bugs in the design before any implementation. Here is a sample checklist of things to consider during product design:

1. Suggest clear separation between UI and Business logic (i.e., can you test the product with automation without using the UI). Ideally the test team will have the ability to exercise all functionality, minimally the ability to exercise core scenarios.
2. For asynchronous request-response components, suggest synchronous versions that wrap the requests, or options to pass context so that test code can match up requests and responses.
3. For the UI itself, suggest hooks into calling the UI elements via automation (if you aren't using standard elements that already provide this).
4. For long running procedural components (setup, data migration), suggest hooks to be able to halt and restart the process.
5. Interfaces should be defined early and change as little as possible, because there is considerable test cost if they change. If interfaces need to change, prefer to deprecate and extend rather than cut and replace so that test code is not immediately broken. Make sure these pieces are well thought out and discussed. This is true for interfaces, UI control IDs, and HTML tags.
6. There may be automation hooks test will need in automation in order to test specific functionality. There may also be hooks that will make testing much easier (ways to switch off features, ways to manipulate data sent to interfaces, etc). Come to a design meeting with ideas on what these could be.
7. For databases a simple way to access data and information from regular test automation. Pre-populated data sets may also be necessary and will most likely need to be generated programmatically.
8. Suggest universal logging across components so that log data can be parsed by a common tool.

9. Design the system/software with a priority to be deployed as soon as possible. In other words, try to tackle design elements that will get working code to customers or dogfooders first, so that you can start getting feedback from a community of users.

Example: Dogfood

Tracking dogfood bugs is a given. It is also important to track dogfood usage. If you can't find a good number of people that think your own product is worth using at your own company it might be a sign you should change a few things. Not only should the use of the product itself be tracked, but it is also important to track the issues and suggestions reported. If we are failing some internal customer expectation, we are most likely failing some external customers.

The final task after implementation dogfood, and engaging the community, is to demonstrate that dogfood, and the corresponding bug-bashes, and betas, are useful. This can be accomplished using data. Data can show how the strategies advance not just test team's needs but also the customers' needs and the product stability and quality. Doing this is as simple as publishing the results of each bash or release, and the bugs logged. When external folks find issues, you can also follow up and show them how and when the issue is fixed to get them invested.

Conclusion

Testers, simply by profession, innately possess the social engineering skills necessary to propagate change, perhaps more so than any other technical discipline. Our profession leaves us well suited to inject our innovation and ideas; all we need is to understand how to exercise the tools given to us.

Useful Resources / References

Rapid Development, Steve McConnell – 1996 - Talks about development best practices, common development issues and problems, team work and team structure in software development

How to Win Friends and Influence People, Dale Carnegie – 1945 - Self help book on improving communicating skills.

PeopleWare: Productive Projects and Teams, 2nd Ed, Tom DeMarco & and Timothy Lister – 1999 - Advice for leadership and running teams in software development.

Zen and the Art of Motorcycle Maintenance, Robert M. Persig – 1974 - essay on the significance of quality and the consequences of quantity over quality.

Characterizing People as Non-Linear, First-Order Components in Software

Development, Alistair A.R. Cockburn – Paper Presented at the 4th International Multi-Conference on Systems, Cybernetics and Informatics in 2000 that talks to the issue of human components in complicated systems.

Sacred Hoops, Phil Jackson – 1995 - Chicago Bull's head coach talks about teamwork and "selfless team play".

The 48 Laws of Power, Robert Greene – 1998 - A compendium of rules that embrace the acquisition of power in society as a game

The Seven Habits of Highly Effective People, Stephen R. Covey – 1989 - Gives advice on improving personal efficiency by adopting specific habits.

Innovation at the Speed of Laughter, John Sweeny – 2004 - Taken from satirical comic writing, outlines techniques to coax innovative ideas from people.

Wikipedia, the free encyclopedia <http://www.Wikipedia.org> – Source for all of the definitions in this paper

Controlling the Human Element of Security, Kevin D. Mitnick, William L. Simon, Steve Wozniak. The Art of Deception - Social engineering handbook from a hacking point of view.

The complete Social Engineering FAQ!, Bernz - Informal Frequently Asked Questions on social engineering as a hacker

<http://packetstorm.decepticons.org/docs/social-engineering/socialen.txt>

Social Engineering Fundamentals, Part I: Hacker Tactics, Sarah Granger – Security Focus article on social engineering and hackers:

<http://www.securityfocus.com/infocus/1527>

The Imagination Factor in Testing

Mike Roe
Symyx Technologies, Inc.
755 SW Bonnett Way
Bend, OR 97702
(541)312-8198
mike.roe@symyx.com

Abstract

The possible existence of software defects must be imagined before the defects can be identified, other than by chance. Stated another way, if testers are not able to imagine the possible existence of software defects, then defects will be identified only by chance. This principle applies to all types of defects, including requirement, code, database, and documentation defects.

This paper describes the importance of imaginative testers and ways to enhance the imagination of the test team. Testers must have and use active imaginations in order to be effective. This seems obvious and is perhaps an age-old truth in the Software Testing discipline, but how does a Test Team put this abstract statement into practice? The audience will be challenged to experiment with some exercises to foster the imagination, with special focus throughout the different phases of a typical software project.

About the Author

Mike Roe is a Senior Quality Assurance (QA) Engineer for Symyx Technologies, a chemical research company based in Santa Clara, CA. For the last three years he has worked with a software development team in Bend, Oregon, that creates scientific electronic lab notebook software (ELN) and the systems in which they are managed. Pharmaceutical and chemical companies use this software to replace paper lab notebooks. Mike has tested software for about eight years in various industries including chemical research, government licensing, accounting, and aerospace.

1.0 IMPORTANCE OF IMAGINATION

When we choose testers to hire, or when we initially judge a Test Team, we look at years of experience, perhaps the breadth of knowledge, the depth of knowledge, and any number of other tangible traits. However, these traits do not completely predict a team's effectiveness. I have tested software for about eight years, with a lead role for about three of those years. Like everyone else, I have experienced my team missing critical defects, even after the software was tested at length by very qualified individuals. In the spirit of learning any lessons that I could, I asked myself why these defects were missed, and always came up with some element of this answer: defects were missed simply because no one imagined the possibility that they might exist.

A software test engineer has many responsibilities, all of which demand an active, well-tuned imagination: 1) Reviewing requirements, 2) Designing test cases, 3) Exploratory testing, and 4) Triaging defect reports. Each of these activities is explored below relative to how active imagination can improve results.

1.1 REQUIREMENTS REVIEW

Among other things, testing requirements includes checking for completeness, as incomplete requirements can result in lots of floundering and reworking of code. Why are requirements incomplete? Do the author and the reviewers intentionally decide that it's just not worth the effort of writing a few more requirements? Perhaps at times this is true; however, I contend that the majority of the time it is because no one imagined the existence of the requirement. Another slant on this is that perhaps no authors or reviewers of the requirements imagined that certain statements could be interpreted in multiple ways.

Example of missing requirements: "This white board pen and eraser must allow a person to create and remove marks on a standard white board." A tester would think of other requirements with a little bit of thought, such as erase-ability and general usage after passage of time, but how about: These products must be fully functional 1) down to 40 feet under water (a means for divers to communicate), 2) under zero-gravity conditions (used on the Space Shuttle perhaps), 3) after driving over the pen with a 4-door sedan on a standard driveway surface, 4) or at temperatures down to 40°F (a walk-in cooler environment).

Example of an ambiguous requirement: "When a user enters an amount value for a material in the form, the total should be updated." Possible interpretations: 1) Does the reader conclude that the user will be allowed to enter an updated total, or that the software should update it without user intervention? 2) Should the software update the total immediately upon the entry of a material amount value, or upon the next screen refresh?

A tester's imagination can improve requirements review by 1) identifying missing requirements, and 2) identifying requirements that can be misinterpreted.

1.2 TEST CASE DESIGN

The step of initially designing test cases is a unique opportunity to imagine problematic scenarios, before being distracted by the product itself. Whether test cases are based on requirements, discussions, or prototypes, the ability to imagine a scenario is critical to creating test cases that may uncover the worst defects.

Example: A tester creates test cases for an application that accepts a string (into a textbox), searches a set of documents in a database, and returns the documents (and metadata) that contain the string. Test case #1: Enter a string, launch the search, and verify that the three documents returned do indeed contain at least one instance of the string. “Great,” the tester says. “What more could the customer want?” Or can we imagine other scenarios to test?

While it doesn’t take much imagination to consider performance-test issues, no-data-returned conditions, and null-value search criteria, the following scenarios may require more thought:

- Enter a string that exists twice in a document. Return set should contain one instance of the document, not two.
- Enter a string that does not match upper/lower case of a string in a document. Should the document be returned?
- Enter multiple words separated by spaces. What should the software do with this?
- Enter a portion of a word that exists in a document. The software may not return documents that contain a substring, only full string. Is this acceptable?
- Enter a string that contains upper ASCII characters. Repeat using kanji (Japanese) or other character sets. Also create and search for documents that have been encoded using different methods.
- Copy/paste a string from the clipboard into the criteria textbox. Repeat using a string that includes format characters such as <tab>, <cr>, or font characters. Repeat with the clipboard containing image content, entire files, and other non-text content.
- Enter a string that contains punctuation or other problematic characters, such as: ,>\$&?#/{}\\.
- Enter a string that will return a number of documents that cannot be displayed on a single page.

1.3 EXPLORATORY TESTING

Exploratory testing could be defined as testing guided only by a tester’s knowledge, their experience, and their imagination, without following documented test cases. Exploratory testing is especially powerful as a first activity to do when developers send the first build of a new software feature to testers. It is a special opportunity because: 1) following documented test cases can actually inhibit the imagination (the tester follows the planned steps rather than the tester’s imagination), and 2) actual implementations of the new feature may not have been

understood when test cases were initially written. There may be testing to do specific to how the developers chose to implement the feature.

Here are some examples of defects that could be discovered using the imagination:

- Saving or checking in a document to the database entailed the user navigating through a wizard (series of steps in a dialog window). Using some imagination, it was realized that the software wrote to the database the document version that existed at the time of the wizard launch. Any editing of the document while the check-in wizard was open did not save; data loss.
- Testers reported that software calculated values as expected using numeric data entered when the computer's regional settings were set to European rather than English (US). Great, but imagine this defect: The software was used to enter data on a computer with European settings, then the data was edited and calculations were run on another computer that had English (US) regional settings. The tester discovered that the calculations were incorrect when the decimal separator character in the numeric data (comma or period) did not match the current PC's decimal separator.
- The software user was able to click a button that launched a wizard to create new user accounts. It performed as expected, but then the tester realized that create-user-account wizards could be launched by clicking on the button multiple times. Attempting to complete wizards with other ones open resulted in a hung application. This wizard was not modal and should have been.

The imagination plays an important role in both test case design and exploratory testing, but in different contexts. When designing test cases, usually before the software is available for test, the tester uses whatever bases for user expectations that are available to imagine possible defects, without any preconceived notions of the actual design or implementation. During exploratory testing, the tester learns the product behavior, the interactions with other components, the designs, the implementations, and the weak areas. This information is used to imagine, uninhibited by scripts (test cases), the most effective ways to break the software. The tester is free to take any number of tangents through the software based on imagined user scenarios and expectations, peculiar observations, possible design flaws, or integration breakdowns.

If no one imagines the many possibilities, then defects will not be identified, unless by accident. Unfortunately, the accidental discoveries may be made by the customer after the product is shipped.

1.4 CHAMPIONING DEFECTS

Cem Kaner published an article with STQE in 2003 about imagining scenario tests to *sell* your defect, to put your defect in a scenario that will result in the stakeholders insisting on the defect getting fixed. Especially for the Marketing and Product Management stakeholders, defect reports must allow the reader to visualize how this defect will upset the end user, particularly if it may result in an end user choosing the software product of a different company. When the tester is able to imagine the right scenario in which the defect will really hurt the end user, his/her defect will get more highly prioritized.

Example bug description: “The client software GUI allows entry and display of up to 30 characters in a textbox, while its corresponding database field has a size of 50 bytes, varchar(50).” As it is described, this defect may seem harmless to the stakeholders; some may question whether it is a bug at all. However, with some imagination the tester would point out that: 1) previous software versions populated the same database used with the software under test and allowed entry of up to 50 characters, 2) other parts of the software populate this same field and allow entry of up to 50 characters, or 3) entering 28 double-byte-English characters (or using certain character sets) into the textbox requires 56 bytes of database storage space and surpasses the size of the field. These conditions result in either some data not being displayed, not fully available for editing, or partial loss of the user’s entered data. Note also that if older database versions have existed with this software under test, the tester would also need to consider its properties and possible needs to migrate into the new database version. The seriousness of these scenarios is such that the stakeholders will likely require a resolution to this issue that may have initially seemed to have no user impact.

The flip side is also true. Knowing that every software fix takes time and carries risk, it is critical that the minimum number of defects get fixed such that end users’ expectations are met. Imaginations can be used to realize workarounds to defects so that fixes may be avoided. For test leads or testers participating in defect reviews, it is important to recognize the opportunity to let the imagination contemplate workarounds that did not occur to the tester. Request that the defect reporter look into possible workarounds.

Example: A critical defect is identified days before the planned release that is found to be specific to Adobe Acrobat Reader® 6. Suggestion: Update Acrobat Reader® to the newest version of 6 and see if the defect no longer exists. Customers may prefer this workaround over a late release, especially if their IT Departments have a policy to keep software updated.

2.0 SUGGESTED PRACTICES TO ENHANCE IMAGINATION

The first step is to recognize the importance of imagination in software testing. But how does a tester advance the power of his or her own imagination and those of co-workers? Similar to becoming an expert fly fisherman or downhill skier, developing an imagination cannot be accomplished by simply reading about it, talking about it, or even thinking about it. It must be practiced. Here are some ideas.

- Team meetings: At team meetings, ask testers to describe a recent case where they used their imagination to think of a missing requirement, resurrect a rejected defect, break the software in a novel way, or exercise functionality in ways that perhaps had not been considered. Teammates will get ideas from each other -- ideas that will increase their ability to think outside the box or create a larger box.
- Schedule some time on a regular basis for your test team to review together a set of requirements, a defect report, or a software feature. Being careful to avoid damaging any

egos, critique the material and brainstorm new possibilities. Expect two potential outcomes during such group discussion: 1) team members' imaginations feed on each other, and 2) the team may discover a new critical issue that avoids lots of wasted time and embarrassment down the road. Along with this activity, the team may choose to read a software book together, such as *Testing Computer Software* by Cem Kaner. In this scenario, everyone reads a chapter before each meeting, and then the chapter is discussed as the material is reviewed.

- Encourage testers to keep asking, "What if ...?" when a defect is identified. When designing test cases, reviewing requirements, and executing test cases, encourage each tester to just stop and think. Sometimes testers fall into that trap where they just need to keep busy, perhaps get through lots of test cases, rather than stop, think, and ask, "what if ...?". When testers observe software behavior during a test case, they should stop and think about what they see and what could go wrong. Sometimes where there is one critical defect, there are many more. Usually there are reasons that a critical defect has reached the test bench; because of these same reasons, other defects may exist in the same area. That initially-discovered defect may be only the tip of the iceberg.
- Encourage testers to take courses on pertinent technology and then present the new knowledge to the group. Encourage them to belong to Test/QA professional groups, attend seminars and conferences, and regularly review websites such as www.stickyminds.com. Exposure to new information feeds the imagination like nothing else can.
- Pairs testing. Consider imaginative ability when selecting individuals to participate in pairs testing. Assign an experienced, less-imaginative tester to work with an imaginative, less-experienced tester. Or perhaps assign a person with significant product and end-user knowledge (SME or product manager) to spend some time with an imaginative tester.
- Another approach that may activate the imagination is a simple reminder of a good tester's initial assumption. A typical user assumes that the software is generally functional and valid, a very appropriate state of mind. A tester needs to assume the opposite: that every letter, calculation, font, color, image, and function of the software under test is wrong, broken, and invalid until proven otherwise. As this approach takes over the thinking process, the mind is opened up to consider many possible defects.

3.0 RESULTS

I have witnessed advancements in both myself and co-workers simply when the significance of imagination is acknowledged as a critical component of software testing. There seems to be fewer instances where defects reached our end users simply because testers failed to imagine their existence. Based on my experience, some of the above-stated practices seem to consistently boost the imagination and testing performance; the effectiveness of other practices seems to be dependent on the situation and the tester.

Attending classes/conferences, reading books/magazines, and keeping up on Test/QA websites (such as www.stickyminds.com) have resulted in a greater ability for our team to imagine new scenarios to test. I would highly encourage participants of these activities to share their newly-

acquired ideas with fellow testers. It is likely that such sharing will increase the activity's value for both the individual and the team.

Pairs testing and brainstorming sessions have also proven to be quite valuable activities, where multiple minds are engaged in the same issue to foster new testing ideas. This activity tends to be most effective when the participants complement each other; one person's strengths fill in another person's weaknesses. A side benefit is increased team camaraderie and growth.

I have seen that periodic challenges from the QA Manager (could be initiated and stated by any member of the Team) tend to increase the aggressiveness and creativity of the defect reports. I have never seen negative results from reminders to question all parts of the software and attempt to break it in as many ways as possible. Note that this practice has very little cost.

The success of discussing interesting defects and/or requirements at team meetings seems to vary from meeting to meeting. For meetings where the conditions are ideal, the imaginations of several individuals routinely uncover issues that would not have otherwise been addressed. Following are some factors that may explain why some meetings have less positive results: 1) attendees' recent test activity, 2) time availability for the meeting, and 3) attendees' ability or willingness to prepare. If a tester is in the late stages of a software project, it may be difficult for that tester to participate in a discussion on requirements review. If team members are working under a tight schedule, then preparation is less likely, and a rush to end the meeting as soon as possible is more likely.

4.0 CONCLUSION

Any Test Team or individual tester can become more effective by making use of and fostering their imaginations. Experiment with ways to accomplish this; success will increase your company's ability to consistently release software products on time with quality that exceeds the competition.

REFERENCES

Kaner, Cem (2003). *The Power of "What if ...and"*. www.Stickyminds.com

Five Trends Affecting Testing

Rex Black, RBCS, Inc.

Five strong winds of change are blowing in the software and systems engineering world. As winds affect a sailboat, these winds of change will affect testing as a field, and testers as a community. Your career as a tester is at stake, and both risks and opportunities abound.

The five trends are:

1. Globalization
2. Test automation
3. Commoditization
4. Compliance, regulation, and tort law
5. Education and certification

In this talk, Rex Black will speak about these five trends and how they affect testing. He will offer cautions about the risks and identify the potential opportunities you face as a tester. For each trend, he will provide references to books and other resources you can use to prepare yourself to sail the ship of your testing career to the destination you desire: professional success.

With almost a quarter-century of software and systems engineering experience, Rex Black is President and Principal Consultant of RBCS, Inc., a leader in software, hardware, and systems testing. For over a dozen years, RBCS has served its worldwide clientele with training, assessment, consulting, staff augmentation, off-site and offshore outsourcing, test automation, and quality assurance. RBCS has over 100 clients spanning twenty countries on five continents, including Adobe (India), ASB Bank, Bank One, Cisco, Comverse, Dell, the US Department of Defense, Hitachi, NDS, and Schlumberger.

His popular first book, *Managing the Testing Process*, now in its second edition, has sold over 22,000 copies around the world, including Japanese, Chinese, and Indian releases. His two other books on testing, *Critical Testing Processes* and *Effective and Efficient Software Testing*, have also sold thousands of copies, including Hebrew, Indian, Japanese and Russian editions. He has written over 20 articles, presented hundreds of papers, workshops, and seminars, and given over a dozen keynote speeches at conferences and events around the world.

Rex is the President of both the International Software Testing Qualifications Board and the American Software Testing Qualifications Board.

Five Trends Affecting Testing... ...And How These Trends Might Affect You



Rex Black
RBCS, Inc.
31520 Beck Road
Bulverde, TX 78163 USA

Phone: +1 (830) 438-4830
Fax: +1 (830) 438-4831
www.rexblackconsulting.com
rex_black@rexblackconsulting.com

Rex Black Consulting Services, Inc.

Technical and Project Management Expertise for Quality

Testing is Blowing in the Wind

- Major changes are happening in software and systems engineering
- Some affect testing
- In the following slides, I'll give some...
 - △ Warnings...
 - ↗ Suggestions...
 - And resources.



Five Trends Affecting Testing

Copyright (c) 2004-2006 Rex Black

Page 2

Five Trends Affecting Testing

1. Globalization of software and systems development
2. Automation of testing, especially early testing
3. Commoditization of information technology and high technology
4. Compliance, regulation, and tort law
5. Education and certification

Five Trends Affecting Testing

Copyright (c) 2004-2006 Rex Black

Page 3

Rex Black Consulting Services, Inc.

Technical and Project Management Expertise for Quality

Trend 1: Globalization

- Falling communications costs and practice with outsourcing make chasing low-cost labor a winning trend
 - △ Your job might go to India – or, worse, the Philippines
 - ↗ Find outsource-proof/outsource-friendly jobs
 - ↗ Work for a company or partnership with outsourcing capability
 - Ed Yourdon's *Outsource* for tips and other resources

Five Trends Affecting Testing

Copyright (c) 2004-2006 Rex Black

Page 4

Rex Black Consulting Services, Inc.
Technical and Project Management Expertise for Quality

Salary of a Typical Software Engineer

Country	Salary	Country	Salary
United States	60	Ireland	40
Germany	56	Mexico	31
United Kingdom	50	Puerto Rico	25
France	42	South Africa	23
Australia	38	Poland	19
Israel	36	Russia	13
Japan	34	India	7
New Zealand	33	Philippines	3

Median annual salary (US\$ thousands) from payscale.com, whose salary survey is statistically problematic, but illustrative.

Five Trends Affecting Testing Copyright (c) 2004-2006 Rex Black Page 5

Rex Black Consulting Services, Inc.
Technical and Project Management Expertise for Quality

Case Studies of Outsource Friendly Jobs

- Facilitating/managing outsource testing
 - Computer vendor based in the US sourcing laptops from Taiwan
 - Computer vendor based in Japan (with US offices) sourcing laptops from Taiwan and Japan
 - Internet appliance vendor based in the US sourcing appliances from Taiwan
 - Bank based in US sourcing system from US company
 - Entertainment company based in Canada sourcing components from various US vendors
- I helped these clients achieve good testing through a combination of on-site and off-site outsourcing

Five Trends Affecting Testing Copyright (c) 2004-2006 Rex Black Page 6

Rex Black Consulting Services, Inc.
Technical and Project Management Expertise for Quality

Trend 2: Test Automation

- Automation has moved beyond regression testing at the graphical user interface
 - Unit testing (e.g., test driven development)
 - Load testing
- △ Testers who can't program can't automate
 - ↗ Learn to program
 - ↗ Learn scripting languages
 - ↗ Learn automated testing tools
 - A beginning C++, Java, or scripting book
 - Test tool list at www.tejasconsulting.com

Five Trends Affecting Testing Copyright (c) 2004-2006 Rex Black Page 7

Rex Black Consulting Services, Inc.
Technical and Project Management Expertise for Quality

Automation Types and Options

Interface	Examples	Program?	Trend
API	JUnit/CppUnit (free) C++-Test/J-Test (pay) Cantata (pay)	Yes, in the language under test	Growing
CLI	Ruby (free) Cygwin (free) TCL (free/pay)	Yes, in a scripting language	Growing
GUI	TestQuest (pay) SilkTest (pay) Perl::GUITest (free)	Yes, in tool language	Saturating

Five Trends Affecting Testing Copyright (c) 2004-2006 Rex Black Page 8

Trend 3: Commoditization

- Will high tech and IT become commodities like electricity and transportation?
- △ High-profit-margin companies might have to learn to live with lower profits
- ↗ Understand commoditization implications
- ↗ Expect increased emphasis on quality, interoperability, usability, etc.
- ↗ Connect high tech/IT with business value
- Nicholas Carr's *Does IT Matter* for analysis

Three Historical Analogies

- Electricity
 - Once a source of strategic advantage
 - Now a commodity input
- Textiles
 - Machine looms and cheap labor eliminated jobs
 - Attempts to resist (Luddites) or find a political solution (tariffs) have failed
- Automobiles
 - Planned obsolescence and low quality through 60s
 - Japanese companies introduced high-quality, low-cost basic automotive transportation

The Computer Hardware Analogy

- As with outsourcing, computer hardware commoditization is ahead of software
- Hardware outsourcing became big in the early 90s, about 10 years before software outsourcing
- Currently, most enterprise application vendors get two-thirds of revenue from maintenance and service
- That was once true of hardware, too, but is certainly not true for most hardware now
- InfoWorld columnist Tom Yager recently wrote, "The yawning sameness of a commodity market is precisely where I wanted the PC to go."
- The leading edge of commoditization is visible in software: Consider Linux and Apache

The Differences, and Why They Matter

- | | |
|--|--|
| <p>Differentiable Goods</p> <ul style="list-style-type: none">● Unique features● More features drive higher prices● Early adopters accept bugs● Constrain users with incompatibilities, etc.● Users must tolerate prickly interfaces | <p>Commodity Goods</p> <ul style="list-style-type: none">● Adequate, consistent quality● Equal features, so vendors compete on price● Later adopters reject bugs● Expected to work with other vendors' offerings● Must be easy to use by non-specialists |
|--|--|

As software and systems become commodities, users will demand cheaper, better-tested, higher-quality products

Trend 4: Compliance, Regulation, Tort Law

- Industry standards, legal regulations, and changing liability standards
 - △ Will your company be sued or barred from the market?
 - ↗ Risks associated with non-compliance and regulatory violations are growing
 - ↗ Testing is a risk-mitigation strategy
 - ↗ Consider adding security to your list of skills
 - www.google.com and www.stickyiminds.com, search for "Sarbanes Oxley" and "computer security"
 - Read the Risks Digest at //catless.ncl.ac.uk/Risks

Three Recent Examples

In the European Union

- The EU took action against Microsoft
- The EU considered bundling of the Windows Media Player with Windows anticompetitive
- Microsoft could find itself barred from the EU market should they fail to comply with this ruling

In the United States

- Healthcare systems must observe HIPAA to protect patient privacy
- This law has significantly affected the workload for test groups in these companies
- Employers must protect employees' information against identity theft

Testing systems for compliance with various standards and laws will continue to grow in importance

Trend 5: Education and Certification

- Education options are wide and varied
- Certification is sweeping the software and systems engineering field
 - △ If your skills fall behind, you become non-competitive—a bad thing in an outsourced world
 - ↗ Self-study, take training, get educated/certified—but be a smart, picky shopper
 - Get certification information on the Internet
 - Check out www.istqb.org

Test Education

- Universities provide some test education
 - In 80s, I had one lecture on testing in my software engineering course
 - In the last five years, four professors have told me they were using my books and materials to teach courses on testing
- Private training companies lead the way
 - I have presented hundreds of trainings around the world
 - Training providers offer testing courses in most software-developing countries
- Nevertheless, most test practitioners remain in the dark on even the most basic techniques
- Unlike programming, testing has not built on the foundations

Certifications for Testers to Consider

Type	Examples	Trend
Testing	ISTQB Foundation, ISTQB Advanced, QAI, ...	Growth
Test tools	Mercury (8), Segue (2), Rational (2), ...	Some growth
Technology	Linux+, RHCE, ... Microsoft (8), Oracle	Some growth
Specialties	CISSP, Security+, ... Certified Usability Analyst...	Big growth

Certification programs establish the essentials of the topic that all competent practitioners must know

What Now?

- Major changes underway for IT/high-tech
- Learn to sail into the wind
- Disruptions create opportunities for those quick enough to seize them
- Consider how the major trends will affect testing and plan your career moves accordingly
- Take control of your career development, and see your employer as only one resource

The Challenge of Productivity Measurement

*David N. Card
Q-Labs, Inc
dca@q-labs.com*

Biography- David N. Card is a fellow of Q-Labs, a subsidiary of Det Norske Veritas. Previous employers include the Software Productivity Consortium, Computer Sciences Corporation, Lockheed Martin, and Litton Bionetics. He spent one year as a Resident Affiliate at the Software Engineering Institute and seven years as a member of the NASA Software Engineering Laboratory research team. Mr. Card is the author of Measuring Software Design Quality (Prentice Hall, 1990), co-author of Practical Software Measurement (Addison Wesley, 2002), and co-editor ISO/IEC Standard 15939: Software Measurement Process (International Organization for Standardization, 2002). Mr. Card also serves as Editor-in-Chief of the Journal of Systems and Software. He is a Senior Member of the American Society for Quality.

Abstract - In an era of tight budgets and increased outsourcing, getting a good measure of an organization's productivity is a persistent management concern. Unfortunately, experience shows that no single productivity measure applies in all situations for all purposes. Instead, organizations must craft productivity measures appropriate to their processes and information needs. This article discusses the key considerations for defining an effective productivity measure. It also explores the relationship between quality and productivity. It does not advocate any specific productivity measure as a general solution.

Introduction

A productivity measure commonly is understood as a ratio of outputs produced to resources consumed. However, the observer has many different choices with respect to the scope and nature of both the outputs and resources considered. For example, outputs might be measured in terms of delivered product or functionality, while resources might be measured in terms of effort or monetary cost. Productivity numbers may be used in many different ways, e.g., for project estimation and process evaluation. An effective productivity measure enables the establishment of a baseline against which performance improvement can be measured. It helps an organization make better decisions about investments in processes, methods, tools, and outsourcing.

In addition to the wide range of possible inputs and outputs to be measured, the interpretation of the resulting productivity measures may be affected by other factors such as requirements changes and quality at delivery. Much of the debate about productivity measurement has focused narrowly on a simplistic choice between function points and lines of code as size measures, ignoring other options as well as many other equally important factors. Despite the complexity of the software engineering environment, some people believe that a single productivity measure can be defined that will work in all circumstances and satisfy all measurement users' needs. This article suggests that productivity must be viewed and measured from multiple perspectives in order to gain a true understanding of it.

International Standards

One might hope to look to the international standards community for guidance on a common industry problem such as productivity measurement. While some help is available from this direction, it is limited. The most relevant resources are as follows:

- IEEE Standard 1045, Software Productivity Measurement [2] describes the calculation of productivity in terms of effort combined with counts of lines of code or function points. It recommends variations to address software re-use and maintenance scenarios. It provides a project characterization form, but does not discuss how different characteristics might lead to different productivity measures.

- ISO/IEC Standard 15939, Software Measurement Process [1]. This standard is the basis for the Measurement and Analysis Process Area of the Capability Maturity Model – Integration [4]. ISO/IEC Standard 15939 contains two key elements: a process model and an information model. The process model identifies the principal activities required for planning and performing measurement. The ISO/IEC information model defines three levels of measures: indicators, base measures, and derived measures. Figure 1 illustrates these different levels of measurement. The counts of inputs and outputs used to compute productivity are base measures in this terminology. Each base measure quantifies a single measurable attribute of an entity (process, product, resource, etc.) Multiple values of base measures are combined mathematically to form derived measures. A base or derived measure with an associated analysis model and decision criteria forms an indicator.
- SEI technical reports discuss how to define effort [12] and size measures [13], but give little guidance on how they can be combined to compute things such as productivity.

Thus, the SEI reports discuss considerations in defining base measures (using the ISO/IEC Standard 15939 terminology), while IEEE Standard 1045 suggests methods of combining base measures to form derived measures of productivity. Note that none of these standards systematically addresses the factors that should be considered in choosing appropriate base measures and constructing indicators of productivity for specific purposes. That is the topic of the rest of this article.

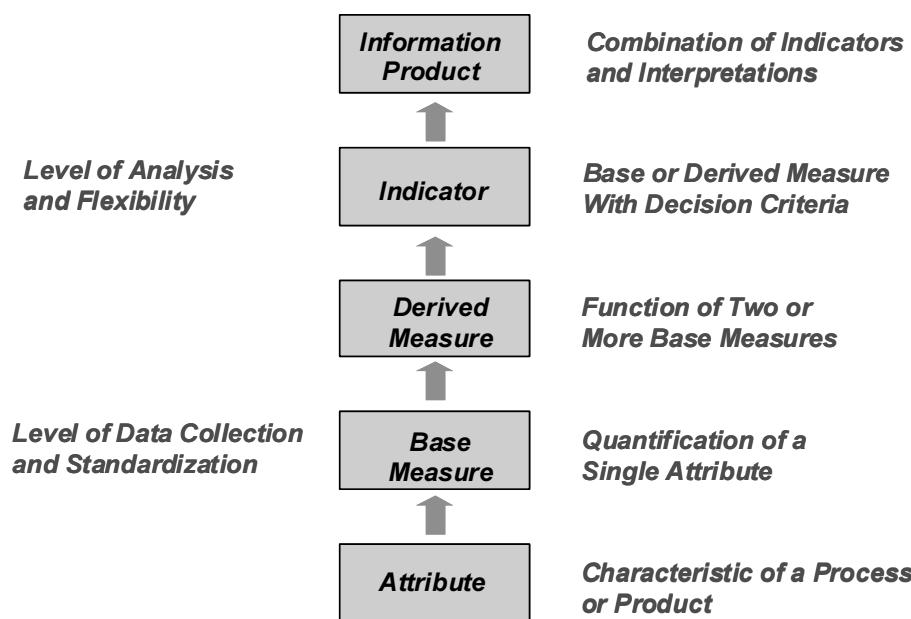


Figure 1. Levels of a Measurement Construct

The Concept of Productivity

Many different approaches to measuring productivity have been adopted by industry for different purposes. This section discusses the common approaches and makes recommendations for their application. The basic equation for productivity is as follows:

$$\text{productivity} = \frac{\text{output_produced}}{\text{resources_consumed}}$$

The simple model of Figure 2 illustrates the principal entities related to the measurement and estimation of productivity. A process converts input into output consuming resources to do so. We can focus on the overall software process or a subprocess (contiguous part of the process) in defining the scope of our concern. The input may be the requirements statement for the overall software process and for the requirements verification subprocess, or the detailed design for the coding process (as another example). Thus (requirements) input may consist of initial product requirements or previous work products provided as input to a subprocess. In this model the “requirements” are relative to the process or subprocess under consideration.

The (product) output may be software or another work product (e.g., documentation). Resources typically have a cost (in the local currency) associated with them. Usually, effort is the primary resource of concern for software development. The output has a value associated with it (typically the price to the customer). The value is a function of capability, timeliness, quality, and price.

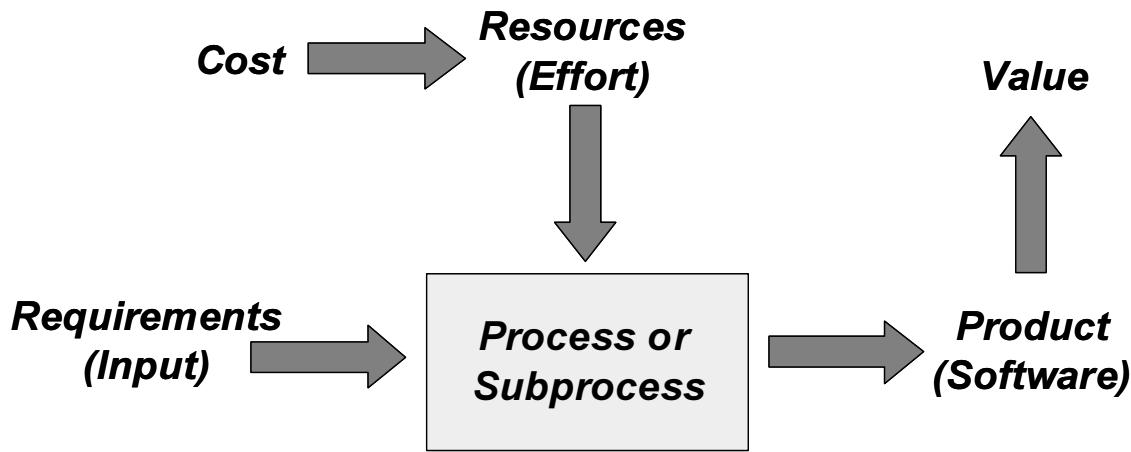


Figure 2. Simple Model of Productivity

Using this model, the numerator of productivity may be the amount of product, volume of requirements, or value of the product (that is, things that flow into the process or subprocess). The denominator of productivity may be the amount or cost of the resources expended. The designer of a productivity measure must define each of the elements of the model in a way that suits the intended use and environment in which the measurement is made.

The product of software development is complex. In addition to code, other artifacts such as data, documentation, and training may be produced. If the resources expended in the production of each artifact can be distinguished, then separate productivity numbers may be computed for each. However, the most common approach is to use a broad size measure (such as lines of code or function points) with a consolidated measure of resources.

Figure 2 is a generic model. The designer of a productivity measure must address the following issues in defining a precise productivity indicator:

- Scope of outputs (product) – which products get counted?
- Scope of resources – which resources get counted?
- Requirements (or other input) churn – what if the target changes during development?
- Quality at delivery – how are differences in quality accounted for?

These issues are discussed in the following sections.

Size Measurement

This section describes the two most common methods for measuring size – the numerator of the productivity equation. These are Function Points and Lines of Code. Function Points is a functional (input) size measure, while Lines of Code is a physical (output) size measure.

The amount of functionality (requirements or input) satisfied usually corresponds to the amount of product delivered. However, the value of a product from the customer perspective often does not track closely to its size. Reuse and code generation tools affect the effort to produce a given quantity software. That is more requirements can be satisfied and more output produced with less effort. Consequently the effects of these technologies must be considered in determining productivity either by weighting the size measures or defining multiple productivity measures for different development scenarios. More than one size measure may be needed to capture all of the information needed about the quantity of product delivered. That is software produced by different methods may need to be counted separately.

Software size, itself, has an effect on productivity. The phenomenon of “Diseconomy of Scale” has long been recognized in software development [9]. This means that the productivity of larger software projects is lower than the productivity of smaller projects, all other factors being equal. Thus, comparisons of projects of different sizes must take this effect into account. This effect is non-linear so that as the range of software sizes increases, the differences in productivity become larger.

Function Points

The Function Point Analysis (FPA) approach has been widely accepted for estimating human-computer interface, transaction processing, and management information systems. FPA involves a detailed examination of the project’s interface description documents and/or prototypes of user interfaces. Usually, these are developed early in the project’s life cycle. When they are not available, similar materials from previous projects may be analyzed to obtain analogous data. The FPA approach rates the complexities of interface elements in a systematic way. Nevertheless, this approach still exhibits a significant element of subjectivity. Many different FPA counting algorithms have been developed. The following discussion is based on the *Function Point Counting Practices Manual* [7]. Different, but similar, measures are required for each of five types of interface elements that are counted. The complexity of each interface element is quantified by considering the presence of three specific attributes. Separate counting rules are applied to each interface type.

FPA was originally developed and promoted as an estimation technique. Because it is based on information that is available relatively early in the life cycle, it can be quantified with greater confidence than physical size measures (such as Lines of Code) during project planning. However, one of the disadvantages of FPA is that even after the project is complete, the measurements of Functions Points still remain subjective. Key contributors to the accuracy of this estimation approach are the skill of the measurement analyst conducting the function point count, as well as the completeness and accuracy of the descriptive materials on which the count is based.

Lines of Code

Perhaps, the most widely used measure of software size is Lines of Code. One of the major weaknesses of Lines of Code is that it can only be determined with confidence at project completion. That makes it a good choice for measuring productivity after the fact, however. The first decision that must be made in measuring Lines of Code is determining what to count. Two major decisions are 1) whether to count commentary or not, and 2) whether to count lines or statements. From the productivity perspective, comments require relatively little effort and add no functionality to the product so they are commonly excluded from consideration.

The choice between lines and statements is not so clear. “Line” refers to a line of print on a source listing. A statement is a logical command interpretable by a compiler or interpreter. Some languages allow multiple logical statements to be placed on one line. Some languages tend to result in long statements that

span multiple lines. These variations can be amplified by coding practices. The most robust measure of Lines of Code is generally agreed to be “non-comment source statements”.

A major factor in understanding productivity, especially in product line development, is taking into account the sources of the software that go into a delivery. Usually, at least three categories are used:

- New – software that is developed specifically for this delivery
- Modified – software that is based on existing software, but that has been modified for this delivery
- Reused – pre-existing software that is incorporated into the delivery without change

The resources required to deliver these classes of software are different. Modified software takes advantage of the existing design, but still requires coding, peer review, and testing. Reused software usually does not require design, coding, or peer review, but does require testing with the other software.

These differences can be taken into consideration by weighting the Lines of Code of each type or by computing separate productivity numbers for each type. The latter approach requires recording resource (effort) data separately for each type of software, so it tends to be less popular. The result of the weighting approach often is described as “Equivalent Source Lines of Code”. Typical values for weighting schemes are as follows:

- New – 100%
- Modified – 40 to 60%
- Reused – 20 to 40%

Ideally the weights are determined by the analysis of historical data from the organization. The concept of Equivalent Source Lines of Code makes it possible to determine the productivity of projects with varying mixes of software sources.

Alternatively, some general adjustment factor can be applied to the software size as a whole to account for reuse and other development strategies. However, that captures the effect less precisely than counting the lines from different sources separately.

Other Size Measures

Many additional size measures have been proposed, especially to address object-oriented development. These include counts of use cases, classes, etc. Card and Scalzo [11] provide a summary of many of them. However, none are widely accepted in practice. That doesn’t mean that they should not be considered. However, their use in productivity measures does not eliminate concern for the factors discussed here.

Resource Measurement

The denominator, resources, is widely recognized and relatively easily determined. Nevertheless, the obvious interpretation of resources (whether effort or monetary units) can be misleading. The calculation of productivity often is performed using only the development costs of software. However, the magnitude of development resources is somewhat arbitrary. The two principal considerations that must be addressed are 1) the categories of cost and effort to include and 2) the period of the project life cycle over which they are counted.

Four categories of labor may be considered in calculating productivity: engineering, testing, management, and support (e.g., controller, quality assurance, and configuration management). Limiting the number of categories of labor included increases the apparent productivity of a project. Calculations of productivity in monetary units may include the costs of labor as well as facilities, licenses, travel, etc. When comparing productivity across organizations it is essential to ensure that resources are measured consistently, or that appropriate adjustments are made.

Requirements Churn and Quality at Delivery

Figures 3a, 3b, and 3c illustrate the effect of the period of measurement on the magnitude of the resource measure. These figures show the resource profile (effort or cost) for a hypothetical project broken into three categories: production, rework, and requirements breakage. Requirements breakage represents work lost due to requirements changes. This may be 10 to 20 percent of the project cost. Rework represents the resources expended by the project in repairing mistakes made by the staff. Rework has been shown to account for 30 to 50 percent of the costs of a typical software project [5]. Usually, rework effort expended prior to delivery of the product is included in the calculation of productivity, while rework after delivery usually is considered “maintenance”. However, this latter rework is necessary to satisfy the customer.

A comparison of Figures 3a and 3b shows the effect of delivery date on productivity. The overall resources required for the two projects to deliver a product that eventually satisfies the customer is assumed to be identical. However, the project in Figure 3a delivers earlier than the project in Figure 3b. Consequently the development effort of the project in Figure 3a seems to be smaller, resulting in apparently higher productivity. However, this project will require more rework during maintenance than the project in Figure 3b.

The project in Figure 3b is similar in every other respect except that it delivered later and had more time to fix the identified problems. This latter project would be judged to have “lower” development productivity, although the total life cycle costs of ownership would be very similar for the two projects. Thus, development cost (and consequently apparent productivity) are affected by the decision on when and under what conditions the software is to be delivered. The true productivity of the two projects is essentially identical.

Comparing Figures 3b and 3c show the impact of requirements churn. While the two projects deliver at the same time, and so exhibit the same apparent productivity, they may experience different amounts of requirements breakage. The project in Figure 3b, with the larger requirements breakage actually has to produce with a higher “real” productivity to deliver the same output as the project in Figure 3c where requirements breakage is lower.

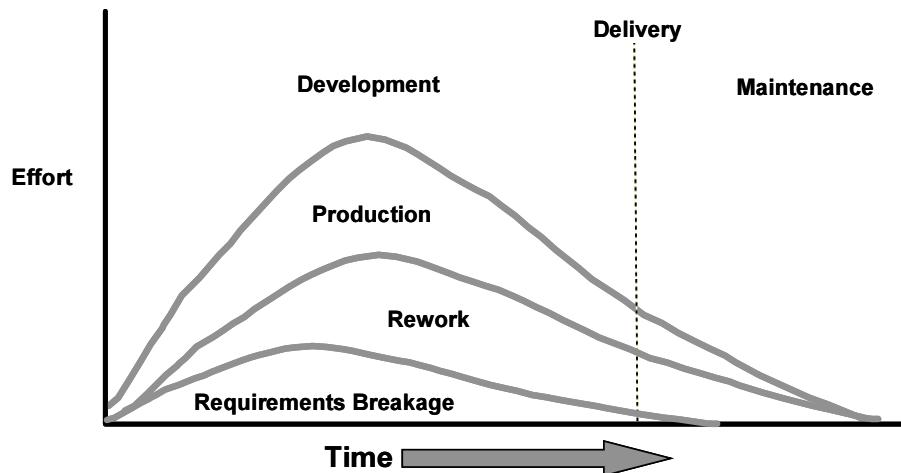


Figure 3a – Apparent “High” Productivity Project

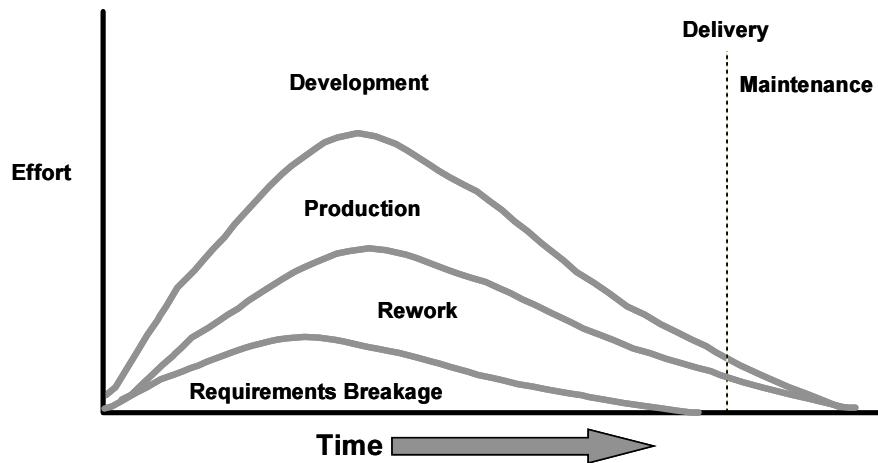


Figure 3b – Apparent “Low” Productivity Project

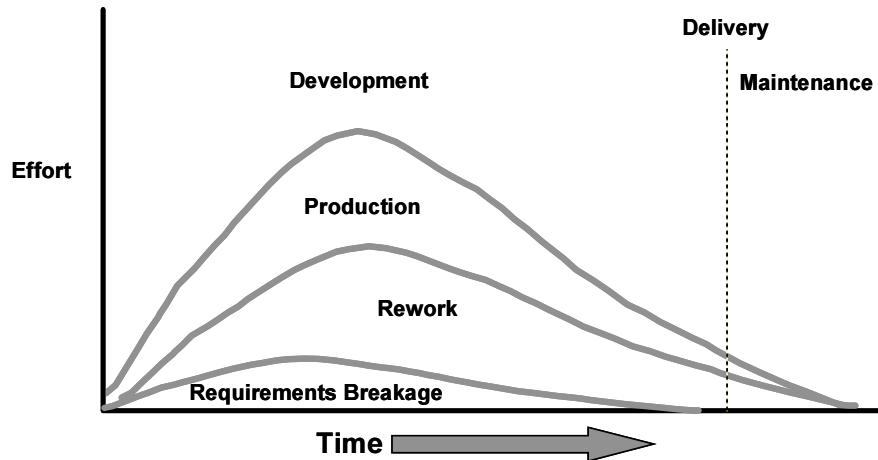


Figure 3c – Highest Effective Effort (Lowest Productivity)

In summary, when determining what value to put in the denominator of the productivity equation, the resources required for rework after delivery should be added to the development cost, while the resources associated with requirements breakage should be subtracted. Obviously, tracking the data necessary to do this is difficult, but such data helps to develop a more precise understanding of productivity.

Typical Productivity Calculations

Measures of size and resources may be combined in many different ways. The three common approaches to defining productivity based on the model of Figure 2 are referred to as physical, functional, and economic productivity. Regardless of the approach selected, adjustments may be needed for the factors of diseconomy of scale, reuse, requirements churn, and quality at delivery.

Physical Productivity

This is a ratio of the amount of product to the resources consumed (usually effort). Product may be measured in lines of code, classes, screens, or any other unit of product. Typically, effort is measured in terms of staff hours, days, or months. The physical size also may be used to estimate software performance factors (e.g., memory utilization as a function of lines of code).

Functional Productivity

This is a ratio of the amount of the functionality delivered to the resources consumed (usually effort). Functionality may be measured in terms of use cases, requirements, features, or function points (as appropriate to the nature of the software and the development method). Typically, effort is measured in terms of staff hours, days, or months. Traditional measures of Function Points work best with information processing systems. The effort involved in embedded and scientific software is likely to be underestimated with these measures, although several variations of Function Points have been developed that attempt to deal with this issue [14].

Economic Productivity

This is a ratio of the value of the product produced to the cost of the resources used to produce it. Economic productivity helps to evaluate the economic efficiency of an organization. Economic productivity usually is not used to predict project cost because the outcome can be affected by many factors outside the control of the project, such as sales volume, inflation, interest rates, and substitutions in resources or materials, as well as all the other factors that affect physical and functional measures of productivity. However, understanding economic productivity is essential to making good decisions about outsourcing and subcontracting. The basic calculation of economic productivity is as follows:

$$\text{Economic Productivity} = \text{Value}/\text{Cost}$$

Cost is relatively easy to determine. The numerator of the equation, value, usually is recognized as a combination of price and functionality. More functionality means a higher price. Isolating the economic contribution of the software component of a system can be difficult. Often, that can be accomplished by comparison with the price of similar software available commercially.

Ideally, the revenue stream resulting from a software product represents its value to the customer. That is, the amount that the customer is willing to pay represents its value. Unfortunately, the amount of revenue can only be known when the product has finished its useful life. Thus, the value must be estimated in order to compute economic productivity, taking into consideration all the factors affecting the customer's decision to buy. Thus,

$$\text{Value} = f(\text{Price}, \text{Time}, \text{Quality}, \text{Functionality})$$

Poor quality may result in warranty and liability costs that neutralize revenue. Similarly, time must be considered when determining the economic value of a product - a product which is delivered late to a market will miss sales opportunities. Thus, the amount of revenue returned by it will be adversely affected. Consequently, the calculation of value for economic productivity must include timeliness and quality, as well as price and functionality.

Note that this definition of economic productivity does not take into consideration the "cost to the developer" of producing the product. Whether or not a product can be produced for a cost less than its value (expected sales), is another important, but different, topic.

Comparing Productivity Numbers

Having chosen a productivity calculation along with appropriate definitions of resource and size measures, productivity numbers can be produced. Comparing productivity numbers from a series of closely related projects (e.g., members of a product line) is straightforward. However, making comparisons across different projects or organizations requires greater care.

Many factors affect the productivity achieved by a project. Most estimation models provide adjustment factors to account for many of these factors. Generally, these influencing factors fall into two categories: controllable and inherent. These two categories of factors must be handled differently when comparing productivity across projects or organizations.

Controllable factors can be changed by management. They are the result of choice, although not always desirable choices. Examples of controllable factors include personnel experience, development environment, and development methods.

Depending on the purpose of the productivity comparison adjustments may be made to account for these factors, especially when using productivity to estimate project effort. However, productivity comparisons of completed projects often are made specifically to evaluate the choices made by an organization, so usually **no adjustments are made for the controllable factors** when comparing productivity results from different projects

Inherent factors are those that are built into the problem that the software developers are trying to solve. Examples of inherent factors (beyond the control of the software development team) include:

- Amount of software developed (diseconomy of scale)
- Application domain (e.g., embedded versus information systems)
- Customer-driven requirements changes

The software development team cannot choose to develop less software than necessary to do the job, build a different application than the customer ordered, or ignore customer change requests in the pursuit of higher productivity. Consequently, **adjustments must be made for the inherent factors** when comparing productivity results from different projects. While quality at delivery is a controllable factor, the eventual quality required by the customer is not, so adjustments also should be made for post delivery repair, too.

Summary

Measures of product size and resources must be carefully selected in deciding upon the construction of a productivity indicator. It is not simply a choice between Function Points, Lines of Code, or another size measure. Many other factors also must be considered. Table 1 summarizes the reported impact of some of the factors previously discussed, as a percent of the project's effort.

Table 1. Factors in Productivity Measurement

Factor	Typical Impact (%)	References
Requirements Changes	10 to 40	8, 9, 10
Diseconomy of Scale	10 to 20	8
Post Delivery Repair	20 to 40	5, 9
Software Reuse	20 to 60	8, 9

Left unaccounted for, the variable effects of these factors on measured productivity can overwhelm any real differences in project performance. Even if an organization finds itself unable to measure all of these factors, they should be excluded consciously, rather than simply ignored.

No single measure of productivity is likely to be able to serve all the different needs of a complex software organization, including project estimation, tracking process performance improvement, benchmarking, and demonstrating value to the customer. Multiple measures of productivity may be needed. Each of these must be carefully designed. This article attempted to identify and discuss some of the most important issues that must be considered. In particular, it did not attempt to define a universal measure of productivity.

References

- [1] ISO/IEC Standard 15939: *Software Measurement Process*, International Organization for Standardization, 2002.
- [2] IEEE Standard for Software Productivity Metrics, IEEE Std. 1045-1992, IEEE Standards Board, 1993
- [3] J. McGarry, D. Card, et al., *Practical Software Measurement*, Addison Wesley, 2002

- [4] M. Crassis, M. Konrad and C. Schrum, *Capability Maturity Model – Integration*, Addison Wesley, 2003
- [5] R. Dion, Process Improvement and the Corporate Balance Sheet, IEEE Software, September 1994
- [6] D. Wheeler and D. Chambers, *Understanding Statistical Process Control*, SPC Press, 1992
- [7] *Function Points Counting Practices, Manual Version 4.0*, International Function Point Users' Group, 1998
- [8] B. Boehm, *Software Engineering Economics*, Prentice Hall, 1981
- [9] M. J. Bassman, F. McGarry, and R. Pajerski, *Software Measurement Guidebook*, NASA Software Engineering Laboratory, 1994
- [10] R.B. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, 1992
- [11] D. N. Card and B. Scalzo, *Estimating and Tracking Object-Oriented Software Development*, Software Productivity Consortium, 1999
- [12] Wolfhart Goethert, Elizabeth Bailey, & Mary Busby, *Software Effort & Schedule Measurement: A Framework for Counting Staff-Hours and Reporting Schedule Information*, Software Engineering Institute, September 1992
- [13] Robert Park, Software Size Measurement: A Framework for Counting Source Statements, Software Engineering Institute, September 1992
- [14] History of Functional Size Measurement, Common Software Measurement International Consortium, www.cosmicon.com/historycs.asp, August 2006

MAGIQ: A Simple Method to Determine the Overall Quality of a System

Presented at the 2006 Pacific Northwest Software Quality Conference
October 9 - 11, 2006
Portland, Oregon

by Dr. James McCaffrey
Volt Information Sciences, Inc.

Abstract

How can you measure the *overall* quality of a software system? Although techniques to measure the quality of individual characteristics of software systems (such as unit testing, performance testing, stress testing, and so on) are quite well known, techniques to measure the quality of software systems as a whole are not widely known. A relatively new and powerful method for software system meta-quality analysis is called the Multi-Attribute Global Inference of Quality (MAGIQ). The four-step procedure for performing a MAGIQ analysis is explained in detail. The MAGIQ technique begins by decomposing the relevant attributes of the software system under analysis into a hierarchical structure. For example, top-level attributes may be API functionality, performance, and security. Each attribute may be decomposed into sub-attributes. For example, the top-level security attribute may be composed of authentication and authorization sub-attributes. Step 2 of the MAGIQ process rates the relative importance of each quality attribute. For example, in one system, performance may be judged more important than user interface, while in another system, the opposite may be true. A mathematical object called a rank order centroid is used to convert attribute ranks (such as 1st, 2nd, and 3rd) into a unit-normalized rating vector (0.6111, 0.2778, 0.1111). In step 3 of the MAGIQ technique, each leaf node attribute of the system under analysis is compared against the corresponding attributes of a baseline system. This step also uses rank order centroids. The last step of the MAGIQ technique combines the quality vectors into a single overall quality metric using a simple weighted sum process. The advantages of the MAGIQ technique compared with other multi-criteria analysis methods such as the analytic hierarchy process (AHP) are: MAGIQ is quick (a complete analysis of a typical Windows-based application takes less than 30 minutes), flexible (MAGIQ can be used to evaluate virtually any type of software system), easy (the computation of rank order centroids uses only simple multiplication and addition), simple to understand (learning MAGIQ generally requires under 20 minutes), and straightforward to interpret results. The primary disadvantage of MAGIQ versus more complex techniques is that MAGIQ generally has lower mathematical sensitivity. Typical scenarios where MAGIQ is valuable include competitive analysis (comparing your software system against an external competitor system), and build analysis (comparing the current build of your system against previous builds).

Author

Dr. James McCaffrey works for Volt Information Sciences, Inc., where he manages technical training for software engineers working at Microsoft's Redmond, Washington campus. He has worked on several Microsoft products including Internet Explorer and MSN Search. James has a doctoral degree from the University of Southern California, an M.S. in Information Systems from Hawaii Pacific University, a B.A. in Mathematics from California State University at Fullerton, and a B.A. in Psychology from the University of California at Irvine. James is the author of ".NET Test Automation Recipes" (Apress, 2006). He can be reached at jmccaffrey@volt.com or v-jammc@microsoft.com.

1.0 Introduction

There are many well-known techniques which evaluate the quality of the individual attributes of a software system. Examples include unit testing, performance analysis, mutation testing, and so on. However, techniques designed to evaluate the *overall* quality of complex software systems are not widely known. These meta-quality techniques are generally called multi-attribute decision models, or multi-criteria decision analyses. This paper presents a recently developed technique called the Multi-Attribute Global Inference of Quality (MAGIQ) which will enable you to quickly and efficiently determine the overall quality of a software system.

The best way to get a feel for MAGIQ is by examining the result of a sample analysis. Table 1 presents the results of a hypothetical MAQIQ analysis. The data in Table 1 show that MAGIQ is being used to evaluate the overall quality of five different versions (Build 5.8.01, Build 5.8.03, and so on) of a software system. The evaluation is based on two top-level attributes, Stability and Functionality. The Stability attribute is composed of sub-attributes CPU Stress, Disk Stress, and RAM Stress. The Functionality attribute is composed of User Interface and Module sub-attributes. In Table 1, the MAGIQ process has assigned relative importance weights to each attribute and sub-attribute of some software system. For example, the CPU Stress, Disk Stress, and RAM Stress sub-attributes have importance weights 0.6111, 0.1111, and 0.2778 respectively. Each software system under analysis receives a relative quality rating on each of the lowest level sub-attributes. For example, for UI Functionality, Build 5.8.04 has the highest (best) rating value of 0.4567, and Build 5.8.03 has the lowest rating value of 0.0400. In the right-most column of Table 1, all the importance weights and quality ratings are aggregated into a final, overall quality metric. In this example, Build 5.8.07 has the best overall quality (0.3058). Build 5.8.04 is second best with a rating of 0.2764, and Build 5.8.06 has the lowest overall quality (0.0938).

	Stability (0.2500)			Functionality (0.7500)		Overall
	CPU Stress (0.6111)	Disk Stress (0.1111)	RAM Stress (0.2778)	User Interface (0.2500)	Module (0.7500)	
Build 5.8.01	0.4567	0.2567	0.4567	0.2567	0.0900	0.2074
Build 5.8.03	0.0900	0.0400	0.0900	0.0400	0.1567	0.1168
Build 5.8.04	0.2567	0.1567	0.0400	0.4567	0.2567	0.2764
Build 5.8.06	0.1567	0.4567	0.2567	0.0900	0.0400	0.0938
Build 5.8.07	0.0400	0.0900	0.1567	0.1567	0.4567	0.3058

<Table 1 - Sample MAGIQ Analysis Results>

The sections of this paper that follow present a brief walkthrough of the sample problem which produced the results shown in Table 1 (to give you a feel for the MAGIQ technique), followed by an explanation of rank order centroids (the fundamental mathematical technique used in MAGIQ), then a detailed look at each of steps used in the MAGIQ technique, a discussion of how to interpret MAGIQ results, and a final section which explains how you can easily adapt and extend MAGIQ to meet your own needs.

2.0 A Sample MAGIQ Analysis

This section presents a brief walkthrough of a sample MAGIQ analysis which produced the results shown in Table 1. The idea is to give you an overview of MAGIQ without going into too many details. Subsequent sections of this paper will delve into the details behind the ideas presented in this section. The first step in a MAGIQ analysis is to set up the problem. This involves determining which systems are going to be compared and which quality attributes are going to be used as the basis for system evaluation. Table 2 presents one possible way to represent a MAGIQ analysis problem.

Problem:	Evaluate overall system quality
Systems under analysis:	Build 5.8.01, Build 5.8.03, Build 5.8.04, Build 5.8.06, Build 5.8.07
Attributes:	Stability
	CPU Stress
	Disk Stress
	RAM Stress
	Functionality
	User Interface
	Module

<Table 2 - Sample Problem Setup>

The second step in a MAGIQ analysis is to assign relative importance weights to each set of comparison attributes. This is done by ranking the attributes (1st, 2nd, etc.) and then converting those ranks into ratings (0.6111, 0.2778, etc.) using a mathematical concept called rank order centroids. This process might yield these results:

Stability (2nd) => 0.2500
Functionality (1st) => 0.7500

CPU Stress (1st) => 0.6111
Disk Stress (3rd) => 0.1111
RAM Stress (2nd) => 0.2778

User Interface (2nd) => 0.2500
Module (1st) => 0.7500

The third step in the MAGIQ analysis is to rank each system under analysis based on each of the lowest level comparison attributes, and then convert those ranks into ratings using rank order centroids. For example, we might get the result shown in Table 3.

	CPU Stress	Disk Stress	RAM Stress	User Interface	Module
Build 5.8.01	0.4567	0.2567	0.4567	0.2567	0.0900
Build 5.8.03	0.0900	0.0400	0.0900	0.0400	0.1567
Build 5.8.04	0.2567	0.1567	0.0400	0.4567	0.2567
Build 5.8.06	0.1567	0.4567	0.2567	0.0900	0.0400
Build 5.8.07	0.0400	0.0900	0.1567	0.1567	0.4567

<Table 3 - Step #3 of a MAGIQ Analysis>

The fourth step in a MAGIQ analysis is to aggregate the quality attribute ratings (from step #2) and the system ratings (from step #3) into final, overall values of system quality. Using the example data above produces:

```
Build 5.8.01 = 0.2074
Build 5.8.03 = 0.1168
Build 5.8.04 = 0.2764
Build 5.8.06 = 0.0938
Build 5.8.07 = 0.3058
```

The last step in a MAGIQ analysis is to summarize the analysis and interpret results. One convenient way to summarize a MAGIQ analysis is to use a table as shown in Table 1. The result data suggests that Build 5.8.07 is best overall but that Build 5.8.04 is a fairly close second. Build 5.8.06 clearly has the lowest overall quality of the systems under analysis.

This brief overview of the MAGIQ technique should give you a rough idea of the process. Now let's take a look at the details.

3.0 Rank Order Centroids

The heart of the MAGIQ technique is a simple mathematical construct called a rank order centroid (ROC). ROCs are essentially mappings from a set of ranks (such as 1st, 2nd, 3rd) to a set of normalized ratings (such as 0.6111, 0.2778, 0.1111). A concrete example will make the idea clear. Suppose you have a set of arbitrary objects, A, B, C and you rank the objects based on some arbitrary comparison attribute: 1st, 2nd, 3rd. You want to assign numerical ratings to the objects in some meaningful way. Let's make an assumption that you want the ratings normalized on a [0, 1] unit interval. This is arbitrary and you can just as easily use [0%, 100%] or any other interval. A naïve approach to map ranks to ratings (but a perfectly valid approach), is to assume that the ratings are equally spaced on the unit interval. This would lead to ratings of 0.75, 0.50, 0.25. However, there are research results which suggest that this equal-distance approach does not reflect true user preferences as well as the ROC approach. See the Barron and Barrett (1996) paper cited in the Reference section for details.

Using rank order centroids, converting ranks 1st, 2nd, and 3rd, into their corresponding ratings looks like this:

```
w1 = (1 + 1/2 + 1/3) / 3 = 0.6111
```

```
w2 = (0 + 1/2 + 1/3) / 3 = 0.2778
```

```
w3 = (0 + 0 + 1/3) / 3 = 0.1111
```

Notice that the rank order centroid values sum to 1.0 (subject to rounding error). A second example should make the calculation pattern clear. The calculations for 1st, 2nd, 3rd, 4th (i.e., four objects) are:

```
w1 = (1 + 1/2 + 1/3 + 1/4) / 4 = 0.5208
```

```
w2 = (0 + 1/2 + 1/3 + 1/4) / 4 = 0.2708
```

```
w3 = (0 + 0 + 1/3 + 1/4) / 4 = 0.1458
```

```
w4 = (0 + 0 + 0 + 1/4) / 4 = 0.0625
```

Expressed in sigma notation, if N is the number of attributes then the weight of the kth attribute is:

$$[\sum_{i=k}^N (1/i)] / N$$

This is very easy to compute by hand or programmatically. And because rank order centroid values are essentially constants (for a given number of objects) ROC values need only be computed once and can then be saved in tables for easy reference. For example,

```
N = 2
-----
w1 = 0.7500
w2 = 0.2500
```

```
N = 3
-----
w1 = 0.6111
w2 = 0.2778
w3 = 0.1111
```

and so on. This approach is particularly useful if you distribute the responsibility for doing MAGIQ analysis among several members of your development team. Listing 1 shows a simple Visual Basic implementation which generates ROC values for N = 5. Figure 1 shows a screenshot of the output of the program.

```
Module Module1
    Sub Main()
        Try
            Dim N As Integer = 5
            Console.WriteLine("Generating rank order centroids for N = " & N)
            Console.WriteLine()
            Console.WriteLine("====")
            Dim k As Integer = 1
            While k <= N
                Console.WriteLine("w" & k & " = " & roc(N, k).ToString("0.0000"))
            End While
        End Try
    End Sub
End Module
```

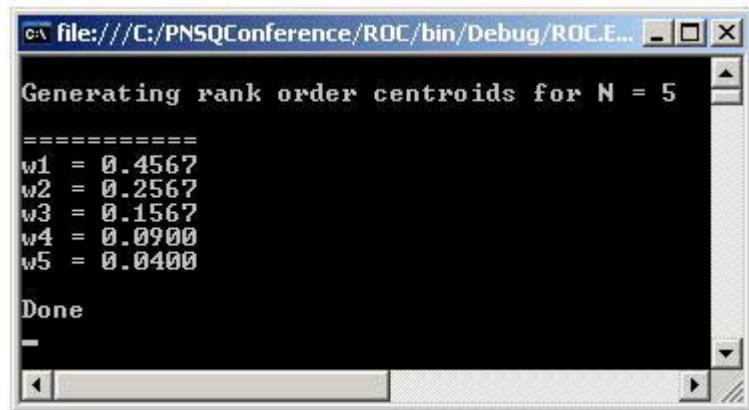
```

        k = k + 1
    End While
    Console.WriteLine(Environment.NewLine & "Done")
    Console.ReadLine()
Catch ex As Exception
    Console.WriteLine("Fatal error: " & ex.Message)
    Console.ReadLine()
End Try
End Sub

Function roc(ByVal N As Integer, ByVal k As Integer) As Double
    Dim result As Double = 0
    Dim i As Integer = k
    While i <= N
        result += (1 / i)
        i = i + 1
    End While
    Return result / N
End Function
End Module

```

<Listing 1 - Computing Rank Order Centroids Programmatically>



<Figure 1 - Output from Program in Listing 1>

A thorough discussion of the rationale behind using rank order centroids to convert ranks to ratings is outside the scope of this paper. However consider this informal argument. Suppose you have just two objects, A and B. You rank them 1st and 2nd respectively. Using the equal-distance approach to assign ratings, you would get $A = 0.6667$ and $B = 0.3333$. Using rank order centroids, you would get $A = (1 + 1/2) / 2 = 0.7500$ and $B = (0 + 1/2) / 2 = 0.2500$. Now because you have ranked A ahead of B, you can assume that A's rating must be somewhere in the interval $[0.5, 1.0]$ and B's rating must be somewhere in the interval $[0.0, 0.5]$. Without any additional information you could assume that the rating of each item is at the midpoint of its possible interval, giving $A = 0.75$ and $B = 0.25$, which are the same ratings as those produced using the rank order centroid technique. However, let me emphasize the argument in this paragraph is just an informal explanation of why using rank order centroids to convert ranks to ratings is a better approach than using the equal-distance approach.

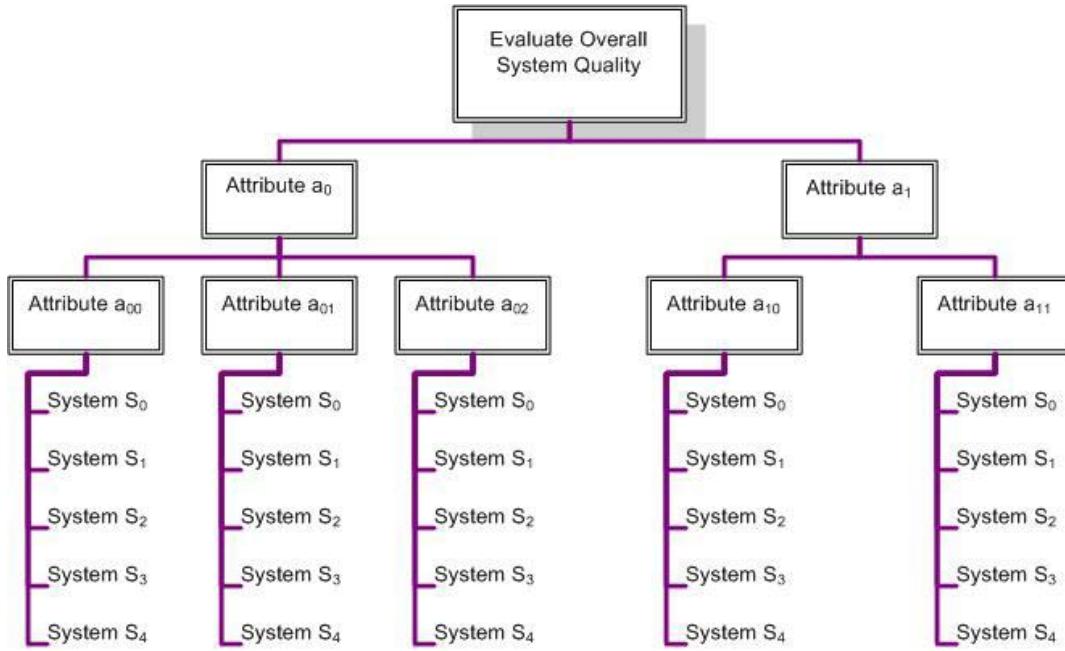
4.0 Setting up a MAGIQ Analysis Problem

The first step when using the MAGIQ technique is to set up the problem space. This involves two tasks: determining which software systems which will be compared by the analysis, and selecting the attributes which will be used as the basis for evaluating the systems under analysis. The number of systems you will choose to analyze with MAGIQ is arbitrary for the most part. If you are using MAGIQ to evaluate different builds of a single software system, as suggested by the hypothetical data in Table 1, using three to seven builds has proven useful in practice. If you are using MAGIQ to compare different but related software systems (such as competitors' systems), the number of systems you use will be determined by the competitive environment. Choosing the quality attributes you are going to use in the MAGIQ analysis depends entirely upon the nature of the software system you are analyzing. Typical top-level quality attributes are accessibility, code coverage, documentation, functionality, internationalization, load, module/unit, performance, security, setup, stability, stress, user interface, and usability. Similarly, decomposing each top-level attribute into sub-attributes is problem domain specific. Experience with MAGIQ suggests that two levels of sub-attributes, each with approximately 3 to 5 quality attributes, is a common decomposition structure.

After determining which systems you are going to compare, and the structure of the quality attributes you are going to use as the basis for comparison, a good approach is to summarize this information in the form of a table or a diagram. This is a common strategy with many multi-criteria decision analysis techniques. For example, Table 4 presents in tabular form a generalized version of the problem presented in Section 2.0 and Figure 2 presents the same problem in diagram form.

Problem:	Evaluate overall system quality
Systems under analysis:	System S ₀ , System S ₁ , System S ₂ , System S ₃ , System S ₄
Attributes:	a ₀
	a ₀₀
	a ₀₁
	a ₀₂
	a ₁
	a ₁₀
	a ₁₁

<Table 4 – Generic Problem Setup in Table Form>



<Figure 2 – Generic Problem Setup in Diagram Form>

Both forms of problem setup representation have advantages. The table form is more concise but the diagram form makes the hierarchical relationship between the comparison attributes more clear. Ultimately, any form of problem setup is acceptable as long as it clearly conveys what the problem is, which systems are being analyzed, and what attributes are being used as the basis for comparison. In the example setup above, there is one top-level set of attributes (Stability and Functionality), and each top-level attribute has one sub-level. As the following sections will make clear, the MAGIQ technique can accommodate any number and structure of sub-attributes.

5.0 - The MAGIQ Process

In this section we'll look at the details of assigning ratings to each comparison attribute, assigning rating to each system under analysis, and aggregating ratings to produce final overall quality values. After you have set up a MAGIQ analysis problem, the next step is to assign relative priority weights to each of the comparison attributes. This is fairly obvious; if, for example, you are evaluating a software system based on performance and functionality, there is no reason to believe that these two attributes are equally important to you. Directly assigning relative importance weight is problematic for several reasons, but assigning relative ranks such as 1st, 2nd, 3rd, etc. is simple. Then using rank order centroids, your ranks can easily be converted to ratings. Similarly, after you have determined relative importance weights of your comparison attributes, there is no obvious way of directly assigning ratings to each system based on each attribute. However, it is very easy to assign ranks to each system based on a particular attribute, and then covert those ranks into ratings using rank order centroids.

When assigning ranks to each system under analysis, you can take advantage of existing numeric test metrics. Suppose you are doing a MAGIQ analysis and are looking at system performance as

one of the comparison attributes. If you already have performance metrics, you can just convert those metrics into ranks and then convert the ranks into rank order centroids. For example, if you have three systems and their performance data is: System A = 1.1111 sec., System B = 3.3333 sec., and System C = 2.2222 sec. Then System A = 1st = 0.6111, System B = 3rd = 0.1111, and System C = 2nd = 0.2778.

Once you have all the attribute priority weights, and system ratings, you must aggregate all those numbers into final, overall measure of system quality. This is done by simply computing a linear combination of the factors which are associated with each system. An example is the best way to explain. Suppose you have the attribute weights and system ratings as shown in Table 5.

	Stability (0.2500)			Functionality (0.7500)	
	CPU Stress (0.6111)	Disk Stress (0.1111)	RAM Stress (0.2778)	User Interface (0.2500)	Module (0.7500)
Build 5.8.01	0.4567	0.2567	0.4567	0.2567	0.0900
Build 5.8.03	0.0900	0.0400	0.0900	0.0400	0.1567
Build 5.8.04	0.2567	0.1567	0.0400	0.4567	0.2567
Build 5.8.06	0.1567	0.4567	0.2567	0.0900	0.0400
Build 5.8.07	0.0400	0.0900	0.1567	0.1567	0.4567

<Table 5 – Attribute Weights and System Ratings>

To compute the overall quality value for the first system, Build 5.8.01, you calculate:

$$\begin{aligned} (.2500) (.6111) (.4567) &+ \\ (.2500) (.1111) (.2567) &+ \\ (.2500) (.2778) (.4567) &+ \\ (.7500) (.2500) (.2567) &+ \\ (.7500) (.7500) (.0900) \end{aligned}$$

$$= 0.2074$$

Similarly the calculations for Build 5.8.03 are:

$$\begin{aligned} (.2500) (.6111) (.0900) &+ \\ (.2500) (.1111) (.0400) &+ \\ (.2500) (.2778) (.0900) &+ \\ (.7500) (.2500) (.0400) &+ \\ (.7500) (.7500) (.1567) \end{aligned}$$

$$= 0.1168$$

And so on. Interpreting the results of a MAGIQ analysis is quite easy; larger values are better. And because MAGIQ results are normalized on a [0,1] interval you can make rough inferences of the relative differences between systems. You should use caution when interpreting the magnitude of MAGIQ quality results. For example in the example above, Build 5.8.01 has an overall quality metric of 0.2074 and Build 5.8.03 has 0.1168. It is mathematically correct to say that, relative to Build 5.8.01, Build 5.8.03 is $(0.2074 - 0.1168) / 0.2074 = 44\%$ worse than Build 5.8.01. However, the magnitudes of the quality metrics depend entirely upon the number of

systems being compared. As a rule of thumb, when comparing four or five software systems, a difference of 0.10 between two systems is significant (in the normal use of the word rather than in the statistical sense).

The 4-decimal precision of the data is somewhat deceiving. Because the original ranking input data is so crude, in general, only the first two decimals are meaningful when interpreting the results. However, you should perform your calculations with four decimals of precision so you don't lose information due to rounding. Additionally, experience has shown that MAGIQ data is best used to monitor system quality trends. A single MAGIQ analysis provides you with valuable information, but a series of MAGIQ analyses over time, provides additional information about the relative quality of your systems under test.

6.0 - Discussion

The Multi-Attribute Global Inference of Quality technique is a close cousin to a multi-attribute technique called the analytic hierarchy process (AHP). The MAGIQ was originally developed as a way to validate AHP results. AHP decomposes a problem comparison attributes exactly like MAGIQ, but AHP uses a pair-wise comparison method instead of rank order centroids to assign ratings. Pair-wise comparisons produce more accurate ratings than rank order centroids but pair-wise comparisons take much more effort. For example, with 7 systems under analysis and 10 comparison attributes, there are 21 comparisons per system times 10 attributes = 210 pair-wise comparisons to perform. It was soon discovered that MAGIQ results correlated almost perfectly with AHP results. Eventually MAGIQ became the primary meta quality analysis technique, and AHP shifted to a secondary role. In essence, the MAGIQ technique is a combination of the hierarchical attribute decomposition used by the AHP technique and rank order centroids. The term "rank order centroid" was coined by F. H. Barron and B. E. Barrett, who also argued for its use in multi-attribute decision problems.

In addition to being quick to perform, another nice characteristic of MAGIQ is that the technique is very easy to understand. With just a few minutes explanation everyone on a development team can learn how to perform a MAGIQ analysis. This allows distribution of responsibility for meta quality analysis, and also allows multiple evaluators. Permitting several people on a development team to perform competitive analysis using MAGIQ often has an unexpected, beneficial side effect. Because the analysis is at a high strategic level, evaluators generally feel they are making a significant contribution to the overall system under development, leading to improved team morale..

Using the MAGIQ technique is not tied to any particular type of software system. MAGIQ is very general and can be applied to virtually any type of system including traditional applications, Web applications, class libraries, and so forth. In practice, there are two common scenarios where you will find MAGIQ useful. The first use-scenario is to perform build analysis, as suggested by the examples in this paper. The second-use scenario is to perform competitive analysis against similar systems. Interestingly, MAGIQ is such a broadly applicable multi-attribute comparison technique, it can also be used in many other problem domains. One interesting use of MAGIQ is to evaluate different job candidates based on attributes such as technical skill, communication skill, and so on.

7.0 - References

Barron, F.H. and Barrett, B. E. "Decision Quality Using Ranked Attribute Weights", Management Science, 42 (1996), pp. 1515-1525.

Edwards, W. and Barron, F.H. "SMARTS and SMARTER: Improved Simple Methods for Multiattribute Utility Measurement", Organizational Behavior and Human Decision Processes, 60, (1994), pp. 306-25.

Jia, Jianmin, Fischer, Gregory W., and Dyer, James S. "Attribute Weighting Methods and Decision Quality in the Presence of Response Error: A Simulation Study", Journal of Behavioral Decision Making, (in press), 1997.

McCaffrey, James. "Test Run: The Analytic Hierarchy Process", MSDN Magazine, June 2005 (Vol. 20, No. 6), pp. 139-144.

8.0 - Acknowledgements

I am grateful to Jason Kelly of the Microsoft Corporation and to Brian Hansen for acting as reviewers of this paper.

Performance Testing: How to compile, analyze, and report results

Karen N. Johnson
1021 Linden Ave
Wilmette, IL 60091

847.644.9789

<http://www.KarenNJohnson.com>

KarenJohnson@acm.org

Author

Karen N. Johnson, consultant and author has 21 years experience in IT. Her experience in software testing has included functional, regression, manual and automated testing, performance, load, and stress testing. Karen has spoken at StarEast, StarWest, and CAST conferences. She has been published in Software Testing & Quality Engineering magazine – now known as Better Software. Karen has attended WOPR – the workshop on performance and reliability, LAWST – the Los Altos Workshop on Software Testing, and AWTA – the Austin Workshops on Test Automation.

Karen is also:

- Director & Member of the Executive Council for the Association for Software Testing.
- Member of CQAA, Chicago Quality Assurance Association
- Member of ACM, Association for Computing Machinery

Abstract

Performance testing an application can involve many test executions; variations on tests and adjustments to runtime executions can result in volumes of output. Compiling the results can be burdensome and yet an important task that requires technical judgment and requires good decision making. What information do you include? What do you exclude? How do you make those decisions? Once your test results are compiled, how do you analyze the results? With all the tests executed, the test results compiled and analyzed – how do you present this information to upper management?

Copyrights and Trademarks

- Mercury LoadRunner TM
- Microsoft® Excel®

1. Introduction

What is performance testing?

In software engineering, performance testing is testing performed to determine some measure of performance such as the time it takes to execute a transaction (or multiple transactions) of a system while the system is under a particular workload.

This definition is my own adaptation of a recent Wikipedia entry [1].

For clarification of this definition and to help define the context of this paper, my assumptions within this definition are:

- The system is a software application.
- A transaction is an activity within the system being tested such as logging into a system.
- A particular workload is the simulation of production usage, for example testing when 500 users are logged in to the application.
- The measurements in performance testing are often measured in time such as seconds.

For anyone who has not worked in the area of performance testing, performance testing can be compared to capturing travel times on a highway. In this analogy the highway is the application and the workload is represented by the number of cars on the highway. Consider the difference in capturing timings on an empty highway versus travel times during rush hour traffic. Timings captured during non-rush hour may be interesting but would not reflect a person's typical commute experience.

One set of questions performance testing hopes to answer is timing questions when the system is under a workload that represents expected system usage. Questions such as:

- How long does it take for a user to log into an application?
- How long does it take for a user to print a report with a predefined set of report criteria?
- How long does it take for a user to get results from a search with a predefined set of search criteria?

The test output might provide answers such as:

- Login takes 5 seconds.
- Generating a report takes 20 seconds.
- Search results are returned in 6 seconds.

Whether performance is fast or slow should be irrelevant to the performance tester. In fact, it is important for a performance tester to maintain an objective unbiased point of view when capturing performance timings. It is *not* the position of a performance tester to begin testing with any particular conclusion or bias towards product performance. It *is* the position of a performance tester to look for the most accurate and consistent performance information that can be generated.

Performance testing requires discipline to control test variables and to execute multiple tests which is necessary in order to gather statistically meaningful results. Multiple tests must be executed before any conclusions can be made. There are multiple test variables that can affect performance timings and it is necessary to execute multiple tests with any changes to any test variables. Depending on the context of the performance testing, here are some of the variables that may need to be controlled:

- How many users are logged in?
- What transactions are the simulated users executing?
- How much think time (*think time* refers to non execution time) is provided between transactions?

If a change is made to a test variable then multiple tests with the same setting must be conducted. To test with different test variables and to execute multiple tests in order to gain sufficient data means that volumes of test output will be generated.

The list of subtopics within performance testing makes performance testing a complex and fascinating topic. Subtopics include (but are not limited to): simulating the live production environment; tools available to simulate users and tools available to capture timings; the labor and investigation needed to determine what transactions the simulated users should be performing; how much think time should be provided; how the ramp up of adding users to the system can affect performance timings. The list can easily be continued.

One aspect of performance testing that I have not seen discussed is: How does a performance tester work with the mounds of test data that is generated? On one project where I worked as the performance tester, the amount of test data generated was so large an additional hard drive was installed for me to store the output. Dozens of tests had been generated; many of the test variables I identified above had been thought about and many different tests had been executed. What I needed was guidance on how to compile the test output and then how to analyze the test data carefully before reporting my conclusions. That is what this paper is focused on. How to compile performance test data in a manner that will help you analyze the output before responding to management's core question: How does the product perform?

The remainder of this paper is organized into three sections: how to compile, how to analyze, and how to present performance test results.

2. How to Compile Performance Test Results

Compiling performance test results requires organizational skills, but the task is more than a clerical matter. In performance testing you need to be knowledgeable about each variable that you can define and set in test execution. You also have to understand how these variables affect the results and to be able to discern variances in the results for deeper analysis. You have to be meticulous about controlling test variables and very careful in managing test results. By carefully organizing test results you can streamline the analysis process and ensure that you are compiling and analyzing the information in a scientific and accurate manner.

The process I recommend to compile performance test results is:

1. Build a test log
2. Identify test executions to include in performance timing analysis
3. Identify test executions to exclude from performance timing analysis
4. Create result folders
5. Create analysis folders

The following sections explain each of these steps.

2.1 Build a Test Log

The only way to obtain statistically meaningful data of an application is to execute multiple performance tests with exactly the same test settings. You should anticipate that performance testing will generate volumes of test execution results. The easiest way to track individual test executions is to use a test log. A test log is a way to record high-level test execution information such as the date and time of the test. You can also develop a test log that helps you distinguish lower level details such as the test variables used such as think time. Develop a test log that makes sense within the context of your performance testing. The key is to record every test execution in an organized way and a test log is a good way to do this.

What should a test log include? A test log should record every test executed including tests that fail and tests that do not run to completion. You will most likely discard some tests from your analysis and reporting but you should not discard any tests from your test log. Some suggested entries for a test log are:

- Test No.
- Date
- Time
- Test Executor's Name (and/or ID)
- Purpose
- Scripts Used
- Scenario Used
- Path to the Results Folder
- Ran to Completion Y/N
- Results Included Y/N

Here's a brief description of each item to record on a test log. You may have additional information you want to add to your test log.

Test No.

Record a sequential number to each test execution. This is just a sequential number with no other test interpretation implied. If you execute tests to gather data for multiple goals such as bandwidth tests or number of concurrent user tests, you may find it helpful to group tests by the test goal and to then number each test execution. (For example: add a column to denote Goal- Bandwidth Tests, and then number each execution for that test goal.)

Date

Record the test date. You might find it helpful to denote whether a test was run overnight.

Time

Record the time of day the test was executed. (You may want to record the test start and stop times.) This entry may be more complex if you want to record the time of day for each PC involved in the tests. For example if the test includes multiple PCs where virtual users are being simulated, you might want to record the time of day for each Host PC. (Host PC meaning each PC that is simulating users.)

Test Executor's Name

Record the name of the person who executed the test. Additionally you might want to record the names of each person involved in the test especially if for each test executed, a team is assembled including network, database, and development staff.

Purpose

Record the purpose of the test execution. If there is a test variable that was changed for a test execution – such as changing the think time identify this test execution detail.

Scripts Used

Record the test script(s) used in execution. Additionally you might want to record what Host PC executed what script. You might also want to record the number of virtual users

Scenario Used

Record the test scenario used in execution. If you used Mercury's LoadRunner application, some test variables are stored in the scenario file in addition to variables that may be included in the test script(s).

Path to the Results Folder

Record the path of the test results. If possible, use a link to the folder. If multiple people will use the test log, it is important to record a universally accessible path. For example, do not record mapped drive letters for network folders. Record the UNC path (server name, share, and path). It is also a good idea to identify what server the data is stored on: Remember, you or another person may need to reference the test data months after execution, and your recordings should be clear.

Ran to Completion

Record (Y/N) whether the test ran to completion. There will likely be tests that encounter errors and tests that do not run to completion. While all tests should be reviewed in the analysis phase, you should not include tests that did not run to completion in the generation of performance statistics.

Results Included

Record (Y/N) whether the test was included in performance statistics. I have found this to be a time-saver when I am going back through result folders looking for specific test details. Sections 2.2 and 2.3 describe the tests I suggest including and excluding in performance statistics.

Additionally, you may also consider logging the Host computer name, number of virtual users, and additional test variable information such as the speed at which virtual users were logged into the system. Again, develop a test log that makes sense within the context of your performance testing. Add to the test log the essential elements of the test executions. When you consider the test log will be a reference pointer to navigate through dozens of test execution folders, you will want to identify key facts to record in a test log.

2.2 Identify test executions to include in performance timings

Use only completed and error-free test executions.

Clean test executions mean tests that ran to completion and tests that did not encounter errors. If a test execution encounters an error it does not mean the test needs to be discarded completely from any of your application reporting but it does mean the test should be discarded from your performance timings. Errors in performance test executions should be reviewed carefully before determining whether any information from the test execution can or should be used. You may have encountered an application error that should be reported. You may have encountered an error that prevents the performance timings to be used but it does not mean that all information from the test should be dismissed.

Use tests using the same criteria and variables.

Suppose you have executed multiple tests to measure performance timings of a handful of transactions. Imagine one of the key transaction timings you wanted to measure was Login. If you executed multiple tests with different ramp up settings, (*ramp up* refers to the number of users added to the system at a time.) you will likely discover that the number of virtual users that login at the same time can make a significant impact in the performance measurements captured. In fact, ramp up time is one variable that can easily affect timings throughout all the transaction timings in a test execution. You should execute tests with your own application to determine if this test variable impacts performance timings. If this variable does affect timings, you must compare results based on segregating the test executions accordingly.

2.3 Identify test executions to exclude in timings

Exclude tests that were not run to completion.

There are numerous reasons a test run may have to be stopped. You may discover you left extended logging on in your test automation tool. You may be told from a database administrator (DBA) that transaction logging was left on during a test execution. You may be told from your network team that there was outside disruption in your network during the test run. I am recommending that these test results not be included as part of the performance timings that you are trying to capture. But I am not recommending that these tests be discarded. Instead, I recommend all aborted tests be held aside from successful clean test executions. All aborted tests must be reviewed for potential important information about your testing tool, application, or test environment.

Exclude tests that encounter test script errors.

Everyone makes mistakes including software testers. Hopefully you allocated time to test the test scripts before your first test execution. Script errors should be investigated and resolved before testing begins. You may still find test script errors after execution begins. You may find flaws in your scripting including not deleting items that are created during execution and then the test script might encounter duplicate object errors on subsequent executions. There are many reasons test scripts encounter errors. Application errors can sometimes be hidden as test script errors – so while you should not include these tests, do not discard them. Review them.

2.4 Create Result Folders

Create a results folder for each test execution. Store all relevant items such as error files, output files, and associated notes within each results folder. See Figure 1. If you did not organize your test output before execution, it is well worth the time and labor to go back and organize test results. In my opinion, it is a requirement to organize test execution output before you can analyze the results. I would find the analysis phase impossible to conduct without organizing the information.

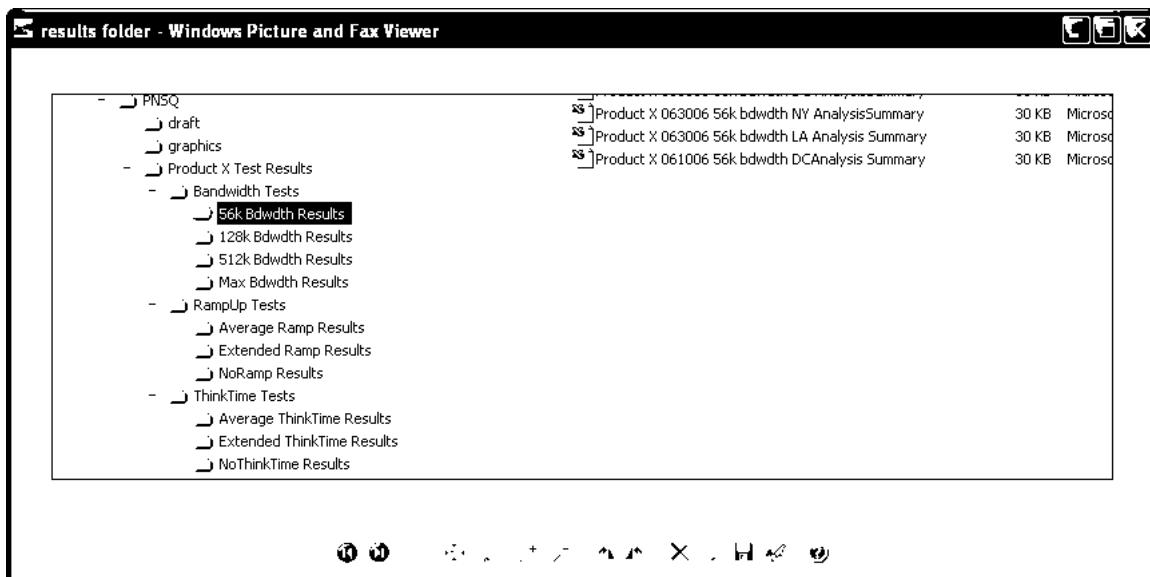


Figure 1. Sample Result Folder Structure

Another method of organizing the details of a test execution is to use a test execution worksheet. A test execution worksheet is used to record details about the test and is primarily used to capture variable settings and nuances of the test execution such as details about the ramp up or think time settings. The test execution sheet is used in addition to the test log. The test log is a brief record of every test executed. The test execution sheet is used to capture details for a particular test.

I found a test execution log on the Internet that must have been modeled after the same version of LoadRunner that I was using on a project. I downloaded the worksheet and used the worksheet to record test details. Regardless of the performance tool you are using, you can create a test execution worksheet and complete a worksheet to record the test execution details while you use a test log to record higher level information about each test execution.

2.5 Create Analysis Folders

Before testing begins, the goals of testing should have been defined. These goals may be represented as questions about the application such as:

- What are the performance timings of the application?
- Where are the performance bottlenecks?
- How does the application perform compared to another version of the same application?

After all the tests are executed and the result folders are built, I recommend creating analysis folders. This will increase the drive space needed as you copy test results into the analysis folders. If you copy the test results, you will use more space but this will also allow you to maintain the original test data in the original test result folders. Unless for some reason you cannot get sufficient drive space, I heavily recommend creating analysis folders. I use these folders to help think through and find my conclusions. Here is the process I use:

1. Create multiple analysis folders. This is not the same as your test result folders. Leave the test execution result folders as they are and create a separate analysis folder for each test goal you need to report on, such as response time, throughput, or process elapsed time. Copy the test execution result folders from each test that will be included into each analysis folder.
2. Next within each analysis folder, compile a single Microsoft Excel file with all the test results for each goal. For example you should have one Excel file for performance timing results. Each worksheet within the analysis Excel file should represent one test execution. Label each worksheet very clearly; I recommend using the test number and date *minimally*. Now you have organized your test output in a way that will enable you to sit back and review the results, look for trends and help you build final conclusions from your testing.
3. For example, suppose I have executed a test 10 times with the same set of variables and test criteria. I copy the result file— the performance timing Excel file— into the analysis folder. I then create one Excel file with each worksheet of the file containing the performance timings of each of the 10 test runs. This eliminates the need to work across folders or files. It is easier to perform operations such as computing averaging and median when you are working within a file.

3. How to Analyze Test Results

Your performance testing team should have included resources outside of QA. Your performance test team may have included a network operations person, a DBA, and a development lead. Each of these persons should have been monitoring different statistics during the test executions. Ask each person to align their statistical information with the test execution information that you have gathered. Share all the test execution information with each person on your performance test. Have each person on the team share the statistical information they captured as well. Once this data is available, you can analyze the results.

In order to learn and *really know* the performance timings for your application, you need to execute the same test multiple times with the same controlled test conditions. You need to be able to discern “normal” performance timings.” Normal in this case means the performance timings that repeatedly fall within the same range. Perhaps normal is not the best term to use, maybe the true term here is *repeated performance timings*. You will not be able to recognize abnormal performance behavior unless you have become familiar with the application’s normal behavior. This desired range is ideally previously determined in performance specifications.

The tester learns the normal range only by repeated execution. This normal range might not be desired or acceptable, and it might change between application versions. That is, the “normal” range might need to be learned and unlearned frequently. You need to be able to detect test executions that are out of range. You should investigate these test executions carefully to determine if there was an issue with the test execution or with the application. Tests with performance timings that are out of range can help you learn information about the application that should be reviewed but these tests should not be included when compiling performance timings. (This paper does not discuss the types of errors that could be found with these tests. This paper focuses on compiling tests for drawing performance test conclusions.)

Once you can recognize “normal” timings, you can begin looking for abnormal results known as *outliers*. I find the following definition of an outlier helpful: a statistical observation that is markedly different in value from the others of the sample [2]. Suppose you execute a test to capture the performance timing of the login transaction and the transaction takes between 4-6 seconds on 10 separate test executions, but on the 11th test run, login takes 20 seconds. The timing of 20 seconds is an example of an outlier.

Hold tests with unusual transaction timings aside for investigation. You may want to calculate final performance timings first with any outliers and then again excluding any outliers and see how outliers affect your results. Test executions where the transaction takes 4 seconds or 7 seconds will be test executions that give meaning to the statement that login takes about 5 seconds to execute. But a test where login takes 20 seconds – can skew the final numbers.

The average and the mean values of transaction timings may not be the only numbers to be calculated and reviewed. Depending on the test tool that you are using, you may have other test data to review as well. For example, in the case of LoadRunner, standard deviation is another value provided in test results and is a value well worth review. Standard deviation can help you identify how wide the range is on a particular transaction. Transactions with wide variances in timings (higher standard deviations) are transactions worth further investigation.

Note that as an objective tester, the goal is to discover an application’s true performance timings and not to focus on the acceptable performance timings that have been defined by the application’s owners, stakeholders or developers. In fact, I try to ignore the defined acceptable and unacceptable range during my testing to avoid subconsciously expecting and looking for performance timings to fall into any range – acceptable or unacceptable. My testing goal is to discover the normal range and only after extensive testing do I compare the timings gathered to the previously defined expected or “hoped for” range.

Use a test log to help you identify which test executions to include and then:

- Review the timings captured.
- Look for consistency in the timings recorded.
- Find the median of the performance timings.

As a non-mathematician I have been challenged to understand how to best compute timings from multiple tests. Not understanding the difference between average and median, I went back to math definitions. See Figure 2 for a computation example.

AVERAGE is the quotient obtained by dividing the sum total of a set of figures by the number of figures.

MEDIAN applies to the value that represents the point at which there are as many instances above as there are below.

Averages hide a lot of sins because outliers skew the timings captured. The median may be a better value, especially as the number of test runs increases. Even more important: outliers need follow-up, because they tell you something new about the system. Before reporting results, you may want to spend some time in Excel working with the average and the median of results to see how these calculations affect the results.

Figure 2 shows a sample worksheet created in Excel to demonstrate the difference between the calculations of average and median.

Timings Captured	3
	4
	5
	8
	20
	4
	6
	7
	2
	9
Average	6.8
Median	5.5

Figure 2. Sample Excel File Illustrating the Difference between Average and Median

This sample shows a difference in the compiled values. It is worthwhile to work with the timings from multiple tests and to determine how to calculate your data. It is also important to identify how you calculated the timings in your reporting. I found the definitions in Excel and the examples shown within Excel's Help File to be good references for a non-mathematician. I would recommend searching on the term 'statistical functions' and reading the short definitions and examples provided. You can review the definitions and find the formulas needed to calculate each value.

As you find the median timings for each transaction being measured, the timings may or may not match the desired results. You might discover during testing that the timings you are *learning* to be true of the application do *not* match what has been defined as acceptable.

I have also learned that the more the performance timings discovered do not match the desired results, the more you may need to do your best to ignore desired timings and focus solely on well executed tests. You must be prepared to substantiate your test conclusions. It is preferable that the acceptable range of performance timings have been identified before testing begins. If the acceptable range is defined in advance, then test results are used to determine whether the desired results are met or not. If the acceptable range is not defined in advance, you may discover the test results create a dispute about what is acceptable performance for the application. And when – if you need to – report poor performance timings results, you may find a similarity as when you report functional defects; development may become frustrated or agitated with the test results. The best way to remain the messenger of bad news and not the bad news itself is to have many well-executed tests substantiating your compiled results.

Getting to know your application and its normal behavior will enable you to review test results in a meaningful way - which is essential in the analysis phase. As you execute tests, you need to assess whether each test execution can be included in the overall performance test results. It is likely that one or more test executions encountered errors that will disqualify the test from being included in the final results.

Use the compiled Excel files and begin reviewing the results. Here is the process I recommend:

- Clear your thinking: throw away speculations and assumptions.
- Review each test with an eye towards high-level information on the first review cycle.
- Look for trends and consistency in results.
- Look for outliers.
- Hold aside test results with timings that are unusual for later investigation. (Again these investigations are not covered in this paper.)
- Become familiar and knowledgeable about what “normal” and consistent timings look like.
- Review each test with an eye towards details on a second review cycle.
- Use Excel calculations to build performance timings.
- Build your product conclusions based on Excel results not your speculations.

Be willing to build Excel files, review data, and scrap the information. In other words, you might *think* you have found an important trend. You should pursue each possible revelation by building an Excel file, copying in worksheets, building averages or median values and comparing a handful of test only to find after further review that a theory you have does not hold up. You might stumble down a dead end path but you must be willing to investigate and equally willing to let go of conclusions you cannot quantify with data.

4. How to Present Test Results

Know your audience. What do they need to know? I have spent years learning the craft of performance testing, but over the same period of time, I have learned the *higher* in management I am presenting results to, the *less* time they have and the more important it is to deliver key condensed information.

Upper management appreciates graphs. Simply said, upper management likes pictures and does not like to read long chunks of text. They *don't* want the details but they *do* expect you to know the details. (Details such as how the performance timings compute when using average versus median.) When you enter a meeting room with a short stack of slides, you must have a deep stack of data back on your desk that proves every statement you make. And this is a crucial part of presenting to upper management - know and be prepared to discuss the details behind every statement in your presentation - but only provide details on request.

Remember that upper management is mostly interested in money and all its surrounding activities such as generating, retaining, and protecting cash. Consider the following reality.

As you present the details of performance testing, *your* interests are centered on technical aspects such as:

- The equivalency of the test environment to the production environment
- The design of the tests you created and executed
- The accuracy of the test output

At the same time you're thinking about technical aspects, upper management is thinking about money in the form of:

- Do we need to spend more money on hardware?
- Do we need to hire additional staff or experts to correct a hardware configuration issue?
- Do we need to hire additional staff or experts to correct a SQL performance issue?

Take your technical data and present your material in a manner that answers your audience's questions. Keep your presentation short and simple.

- Present information in small chunks
- Use basic charts or drawings whenever possible
- Build your presentation for short attention spans

When presenting to upper management, be prepared to consolidate your presentation. It is not uncommon for meetings to start late or for a key person to be called out of the meeting for a few minutes. I once presented a stack of information to board room of 15 people and at the end of my well-prepared presentation, the CEO looked up and said: "one question, would *you* ship this application as it stands today?" Know your information well enough that you can be thrown off your planned presentation and yet be able to land on your feet anywhere in your material because you know the information well enough.

5. Summary

A certain amount of scientific and mathematical skills are needed in performance testing. First there is the fact-gathering process in how to simulate a production workload, followed by the need to understand and control test variables during test execution. Scientific skills are needed again with the attention to detail in compiling and analyzing test results. Remain objective during your test and do not be swayed by whatever political pressures or time to market pressures may exist as you capture the most accurate test data you can. Remember to be able to substantiate your concluding performance results. Once you have a solid sense of how your application performs and you can back your findings with substantial and solid data, you need to summarize and present this information in a meaningful way.

References

- [1] Wikipedia, known as the free online encyclopedia that anyone can edit.
http://en.wikipedia.org/wiki/Performance_testing
- [2] Merriam-Webster Online. <http://www.m-w.com>

Benchmarking for the Rest of Us

Jim Brosseau

Abstract

There is no shortage of data to use to try to determine whether your software team is sufficiently productive. Whether it is the often quoted Chaos Report from the Standish group, the quarterly updates from the Software Engineering Institute or the hidden project data behind parametric estimation models such as COCOMO II, it is seductive to hold your own performance against these standards for comparison.

We quickly find with a critical eye, though, that these comparisons bear little relevance for most organizations, especially those that are 'up-and-coming', the growing companies in most need of benchmarks to gauge their performance and progress. This article identifies the challenges with most published information, and enumerates the approaches that we can all use to generate meaningful benchmarking information.

Author Biography

Jim Brosseau has 20 years experience in the software industry in a wide variety of roles, application platforms and domains. A common thread through his experience has been a drive to find a less painful approach to software development. He has worked in Quality Assurance at Canadian Marconi, and was involved in the development and management of the test infrastructure used to support the Canadian Automated Air Traffic System. In the past 9 years he has consulted with numerous organizations worldwide, specifically to improve their development practices.

Jim publishes the Clarrus Compendium, a free weekly e-newsletter with a unique perspective on the software industry. He has been published in PM Network magazine, the PMI GovSIG newsletter, and the SEA Software Journal, and has presented at Comdex West in Vancouver, PSQT North in Minneapolis, the NB SPIN group (via teleconference), Better Software and several local associations. He is principal of the Clarrus Consulting Group in Vancouver, Canada, and may be contacted at jim.brosseau@clarrus.com.

Clarrus Consulting Group Inc.

7770 Elford Avenue, Burnaby, BC Canada V3N 4B7

T. 604.540.6718

To be effective, businesses of all sizes need to understand their performance. While large, established organizations will typically have a solid infrastructure and a constant finger on their pulse, smaller, growing companies often struggle with fundamental issues. This is highly prevalent in technology organizations, where an entrepreneur with the *next great idea* suddenly finds that organizational issues are consuming more and more of his or her precious time. It is wise for any organization to have a clear understanding of how well it is performing, either as an impetus to improve, or as a basis for understanding how much work it should reasonably take on in the future.

What is the best structure for our organization? How productive should we expect to be? What should we be paying our staff? These and many other questions need to be resolved for small growing companies to get beyond their non-technical hurdles to success.

With all these questions to answer and so little time, there is often a rush to quickly find ‘the solution’, whether a general solution really exists for the industry as a whole or not. Among the frequently asked questions are the following:

- **What is the appropriate ratio of software testers to developers?** Companies want to use this number to mold the structure of the development organization, but there is no right answer here. I have worked on significant projects where the developers successfully performed the bulk of the testing of the system, and worked with teams where dedicated testers outnumbered developers almost 2:1 and still could not keep up with the issues that were cropping up.
- **How productive should I expect my team to be (given a variety of factors)?** There is clear benchmark data available indicating average ratios of Function Points to Lines of Code, and productivity in terms of Lines of Code per day, given the type and criticality of application being developed. There is a great deal of data points behind the scenes used to make the information statistically relevant, usually with extremely wide variations. This variability rarely makes it to the surface of the data presented, but is a strong indicator that your mileage may vary – and probably by a large amount, even within your team.
- **How should I compensate my staff?** Industry salary reports have dropped from their staggering heights of several years ago to reflect the changing times. Still, there is significant geographical variation to consider.

For a variety of reasons, many companies turn to externally generated benchmarking data to provide the answers they need. Unfortunately, there is a dark side to the quick and sometimes blind use of external information. Use of benchmarking data needs to be carefully tempered to add the appropriate level of value for organizational improvement.

Benchmarking Data is Alluring

For better or worse, most organizations refer to industry benchmarking data as a means of gauging their performance. Like most people in the industry, I’ve done my fair share of quoting statistics from the Standish Group’s 1994 Chaos Report [1], and have used the quarterly reports from the Software Engineering Institute to describe industry performance in discussions with clients. A number of people and organizations have collected and disseminated a great deal of benchmarking data over the years.

Collected benchmarking data is relatively easy to obtain, as it is for the most part readily available, if for a price. It comes from well established, reputable sources, either published in books or trade journals, or available for purchase from a number of organizations worldwide. It is usually well organized and indexed in a manner that will allow you to quickly arrive at the information you are looking for.

Using data from reputable sources will help you to back up your assertions and can make your arguments much more compelling and defendable. It can be an indication that you have ‘done your homework’.

At times, the allure of using benchmarking data comes from its external sterility. The data provided is based on other people’s performance, and can provide a sanitized look at what the industry is doing. For some organizations, it can become a game to blithely quote industry performance figures while avoiding internal measurement, knowing that the truth can be a difficult pill to swallow.

Benchmark Data: Caveat Emptor

“He uses statistics as a drunken man uses lamp-posts - for support rather than illumination” - Andrew Lang

Imagine a situation where you decide when it is best to leave for work in the morning by observing your neighbor’s departure patterns. Over the course of a month, his average departure time is 7:15, give or take 5 minutes or so. Pretty consistent, so you decide that 7:15 must be appropriate for you as well. Unfortunately, your neighbor works about a mile away, while you have a cross-town trek. Worse yet, you may be on the afternoon shift, or you may work from home. Is that benchmark data worthwhile?

There are a number of problems associated with use of the industry data that we have all turned to on occasion. We need to be extremely careful to drill down past the superficial presentation of the data that usually arrives in the form of a simplified table or graph to determine if it is applicable to our situation at all. Quite often, the data will be presented in a form that may make the chart visually compelling while obfuscating some important elements of information that would otherwise be helpful. With the general availability of spreadsheets and graphics packages, we often find ourselves interested in the superficial presentation rather than the included information.

Wide Variability, Hidden Bias

Beyond the simple data points presented in benchmark data, it is important to recognize that the underlying data may have potential hidden biases or wide variations within the sample space. These attributes, if not clearly understood, can lead one to rely more heavily on the benchmark data than is reasonable.

Parametric estimation models, for example, are essentially the result of curve-fitting exercises based on a broad sample space of thousands of completed software projects, which can make the models compelling to use for early, whole-project estimates. The SLIM parametric estimation model is based on a large number of projects, divided into roughly a dozen different industry

types [2], with the intent to provide a sample space that is relevant to your situation. As you drill deeper, though, you find that the variation within each of these industry types is very wide, and your performance may actually be closer to that of the median of an industry type that does not appear to be close to yours.

The COCOMO II parametric model [3] introduces a bias of another form. While the sample space is much smaller, it is more important to note that many of the projects that have been used for curve-fitting the model are primarily in the defense and aerospace realm, where practices are such that there is a very low correlation with commercial software development or other development types.

Both the SLIM and COCOMO II models have been fit primarily with projects that are fairly large in terms of effort and scope. It would be erroneous to assume that the models could be extrapolated down for use on small projects. To blindly use these models ‘out of the box’ for small projects or projects that have not been calibrated appropriately would be to generate estimates that are falsely defendable. While the data behind the models has been validated, that does not mean that the data cleanly maps to your situation.¹

Sparse, Slanted SEI Data

The SEI’s quarterly *Process Maturity Profile of the Software Community* may suffer from bias problems of its own. According to the August 2002 release [5], the report shows that 19.3% of reporting organizations are performing at Level 1 (the Initial level) of the SEI’s Capability Maturity Model (CMM) scale, which is quite a strong positive indicator for the industry as a whole. The fine print, however, indicates that this is “Based on the most recent assessment, since 1998, of 1124 organizations.”

There are a couple of points to note here. This sample space is extremely small considering the number of software development organizations worldwide. In addition, it is biased not only towards organizations that are aware of the SEI, the CMM, and the suggested best-practices they promote, but also toward organizations that have reported results to the SEI from formal assessments.

In most organizations that I have worked with in the past 4 years, the majority of people were not aware of the SEI, and their practices and performance clearly placed them in the Initial Level of the CMM. For those organizations that I have worked with that have claimed to have attained higher level maturity on the SEI’s scale (i.e., Levels 3-5), most, in my experience, have not been able to perform in accordance with even those goals attributed to the Repeatable Level (Level 2).

¹ Beyond the selection of a specific parametric model for estimation there is the question of which estimation procedure to use? Many organizations will try to take a published procedure, such as that used by the NASA Software Engineering Laboratory [4] and its embedded information (such as uncertainty, phases and approaches) and call it their own. While there are industry-wide principles that their estimation procedure should embrace, there is not a one-size-fits-all solution.

(Ir)Relevance of Annual Data

Often, benchmarking data is provided on an annual basis, which allows you to subscribe and remain current. One must be careful, however, to determine whether time-based trends would be relevant for the information provided. Clearly, annualized reports showing the equivalent of the average results of 1000 coin tosses will yield limited additional insight. While some benchmarking data will benefit from annualized updates (such as new data sectors or evolving trends) there are also some classes of data that do not benefit to the same degree.

It's Not a Diving Rod

There is danger in use of Benchmarking data as a driver for direction in your organization. Industry averages in IT spending, for example, can be extremely revealing if you are on the receiving end of that spending trend, and can be used as one of the drivers for forecasting, especially if historical spending trends have tracked well with your performance in the past. If you are looking at how much your organization should be spending, historical benchmarking data will tell you where the industry has been but will not help you resolve how to best address your organizational needs in the future – budgeting for future spending based on industry trends fails to address what is important for you.

Benchmarking Data's Silver Lining

"All models are wrong, some models are useful" – George Box

All this is not to say that you should never use externally generated Benchmarking data within your organization. There is a great deal of consideration and industry research that has gone into much of the benchmarking data available, and it is important to understand how and whether the data appropriately applies to your situation.

Benchmarking data that is used as a basis for or result of certifications or qualifications - such as ISO Quality standards, SEI Maturity levels, or the Project Management Institute's Project Management Professional (PMP/ designation - provides an indication that the organization or individual has passed a baseline level of performance or understanding. ISO Certified organizations have clearly identified their quality practices and demonstrated that they 'practice what they preach' (although this is not a guarantee that their next project will be a success), and the certification can reasonably be used as part of the criteria in an acquisition process. Individuals with the PMP designation have been assessed to have knowledge of a base set of commonly accepted project management best practices and performed a prescribed amount of work in the project management arena (but this is not a guarantee that they are effective project managers).

For much of the benchmarking data that is available, it is usually the case that the underlying assumptions, variability of the data, and inherent biases can be identified with some digging. The information may be published along with the primary information that has been distilled, may be available from the provided reference information, or may be obtained through deeper discussion with the provider of the data if one is so inclined (and diligent).

The Personalized Solution – Answer Your Own Questions First

A reasonable approach to the use of data is to balance external benchmarking data with that which has been internally derived, with the intent of helping you understand whether or not you are achieving your organizational goals. As Peter Senge noted in The Dance of Change, we need to measure to learn rather than to merely report [6].

With an understanding that the first step is to identify our goals in the measurement process, we can lean on the approach prescribed by Vic Basili's GQM approach, or extend and elaborate on that practice using techniques such as the Balanced Scorecard. Identifying these goals and corresponding model we will use to validate the goals allows us to remove biases from the response, and pull us away from the temptation to simply use data because it is readily available. Our quests become tightly coupled with our culture and organizational needs.

Using this approach, we can then perform our own internal measurements, and make comparisons against industry benchmarks, where it makes sense. With the added internal diligence, we will have a more valuable understanding of the biases that are inherent in the data (and a comfort that the biases are more likely to be working for us rather than against us), as well as the uncertainty or variability in the data set.

It is important to recognize the distinction between statistical variability across industry benchmarking data and individual performance variability that will arise in your own measured data. The former is an indication of the relative applicability of the information to your situation, while the latter is an expected artifact of the measurement approach that needs to be fully appreciated. You need to accept individual variation as a fact of life. Even if you use the information to cull the low-performing individuals (not a recommended practice), you will continue to have variability, and by definition statistically, 50% of the people will always fall below the median of your data set. It is dangerous to fall into the trap of using measures for segregating the team rather than for improving the organization.

Some of your greatest insight will come as you track your own measurements over time and observe the variation and trends that are revealed. This information is not something that can be gained from industry benchmark data, but you can see whether you are tracking towards or away from the industry benchmark data, which will provide a greater indication of the applicability of the benchmark information to your situation.

For this to be effective, you need to be consistent in your measurement approaches within your organization over time. One commonly hears concerns in the industry about inconsistency of measurement, whether it be for histogram categories to collect data, or the approach used (Lines of Code, for example, can be defined as semi-colons, non-blank lines, total number of carriage returns and so on) The bottom line here is that you should select a specific approach, identify that it is the standard and stick with it, to ensure that you are indeed making apples-to-apples comparisons.

Industry Benchmark data indeed has its place in your arsenal of information for making strategic business decisions. Still, it has its limitations that must be overcome with a deep understanding of why you are measuring and balanced with data gathered internally with reasonable

approaches. Taken with a grain of salt, benchmark information can give us the perspective we need to better understand what our internal information is telling us.

References

- [1] The Standish Group, *The CHAOS Report*, 1994
(http://www.standishgroup.com/sample_research/chaos_1994_1.php)
- [2] Putnam, Lawrence & Ware Myers, *Measures for Excellence: Reliable Software on Time, Within Budget*, Yourdon Press, 1992
- [3] Boehm, Barry, et. al. *Cost Models for Future Software Life Cycle Processes: COCOMO 2.0*, Annals of Software Engineering, 1995 (<http://sunset.usc.edu/research/COCOMOII/index.html>)
- [4] National Aeronautics and Space Administration. *The Manager's Handbook for Software Development, Revision 1* (Software Engineering Laboratory Series SEL-84-101), 1990
(<http://sel.gsfc.nasa.gov/website/documents/online-doc/84-101.pdf>)
- [5] Software Engineering Institute. *Process Maturity Profile of the Software Community 2002 Mid-Year Update*, August 2002.
- [6] Senge, Peter et. al. *The Dance of Change: The Challenges to Sustaining Momentum in Learning Organizations*, Currency/Doubleday, 1999.

Key Measurements for Testers

By Pamela Perrott

Construx Software
10900 NE 8th, Suite 1350
Bellevue, WA 09004

Pamela Perrott is a Senior Quality Architect at Construx Software. She has been in the IT industry for 23 years as a programmer, systems programmer, analyst, project manager for tools implementations, and instructor. Pam is a member of the Computer Society of the IEEE, the Association for Computing Machinery, the Data Resource Management Association, the Puget Sound Chapter of the Association for Women in Computing, and the Puget Sound Chapter of the Special Interest Group on the Computer-Human Interface (SIGCHI).

Pam is expert in quality practices, such as implementing inspections, and has a deep knowledge of software process improvement, testing, and software project management. Prior to working at Construx, she integrated new technologies, implemented inspections, and performed complex requirements management at Verizon Wireless.

Pam has an AB from Bryn Mawr College in Biology, an MA from Cambridge University in Biochemistry, a Certificate in Data Processing from North Seattle Community College, and a Master's in Software Engineering from Seattle University. She is also a Certified Function Point Specialist (CFPS).

1. Abstract

The testing function often utilizes very simple measures: how many defects are currently open? How many test cases have we written? How many test cases have we run?

This paper is a survey of typical measurements in test for both projects and entire organizations. By comparing current projects with historical project data, one can better determine the answers to such key questions as ‘how many testing resources do we need? Is the quality good enough? When can we release the product?’

2. Introduction

Most professional software testers know at any given time how many bugs are currently open and how many have been fixed. This paper presents some additional measurements which are useful for testers. Many are easy to implement. Measures are presented which focus on one project and cumulative measures for an entire organization are also presented.

Measurement is increasingly important to the software industry. Both the CMM and the CMMi frameworks require measurement. But what to measure? This paper suggests some practical measurements for the testing function to consider.

3. Background concepts

Measurement of software isn't as easy as measuring many other things we come into contact with every day. It's easy to measure how long a table is, how tall you are, how much you weigh. Some of the measurements in software are much less accurate and are harder to do, but they are still worthwhile.

Many software professionals don't have much experience with measurement principles. One important principle is the difference between precision and accuracy. Precision means how many significant digits the measurement has. $\pi = 4.378383$ is precise, because it has a lot of digits. But it's not accurate! π is 3.14159. And, as humans we tend to believe that if a person reports many digits that they actually know the measurement to that level of precision. So, one thing to remember is not to use any more digits in your numbers than you can really justify.

Accuracy is another concept. Accuracy is how close the measurement is to the true value. I'm 5 feet 5 inches high. Saying I'm 2 meters high is accurate (I'm closer to 2 meters than to 1 meter) but not precise. We'd like our software measurements to be accurate, i.e. to correctly measure the right values of whatever we are measuring.

Humans make assumptions about accuracy based on precision. Thus, if you say 365 days, people assume you know the date the project will be done to the day. If you say 1 year, they don't assume you know the date to the precision of 1 day. If you say 52 weeks, they think you know the end date with the precision of 1 week. So keep this in mind when you decide what units to use when you report your measurements. Early in a project, when there is a large uncertainty to our estimates of when a project will be done, we would probably quote its ending date in quarters: 4th quarter +/- 1 quarter. Then our customers would know we don't know the ending date more precisely than to a quarter.

In deciding what to measure, it's useful to start with our goals. Victor Basili of the University of Maryland¹ proposed a hierarchy where we identify our goals for measuring. Then we can think of the questions, which if we asked them, would tell us if we were reaching our goals. Then we can think of the measurements we need to make so we can answer the questions.

Goals are most useful if they are SMART, i.e.

- Specific
- Measurable or Testable
- Attainable
- Relevant
- Time-Bound

Goals need to be specific so we know exactly what we mean. They need to be specific enough so we can measure or test if we are meeting them or not. They need to be attainable –it needs to be possible to meet them. They need to be relevant to what we are doing or want to measure. And they need to be time-bound, i.e. they need to have a ‘by when’ clause –for example ‘Decrease defects in production by 50% by January 2007’.

Questions may relate to more than one goal, and measures may be useful for answering more than one question. Thus there is a hierarchy, as shown in the diagram below (Figure 1).

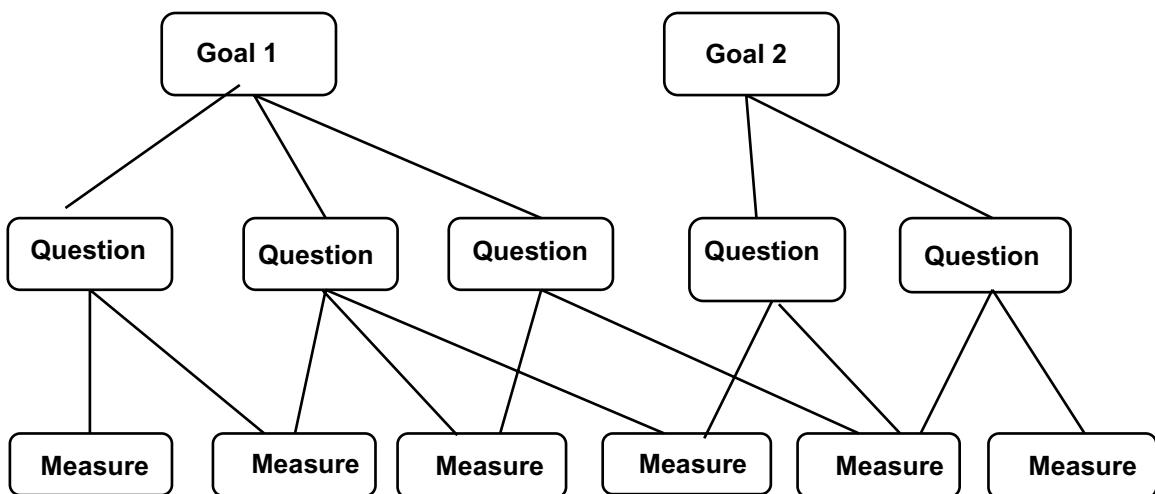


Figure 1. Goal Question Measurement hierarchy

A useful approach to writing a goal is the Purpose, Issue, Object, Viewpoint format, shown in the table below (Table 1)

Goal	Purpose Issue Object (process) Viewpoint	Improve by 10% the timeliness of change request processing from the project manager's viewpoint
Question		What is the current change request processing speed?
Measures		Average cycle time Standard Deviation % cases outside the upper limit
Question		Is the performance of the process improving?
Measures		<u>Current average cycle time</u> * 100 Baseline average cycle time Subjective rating of manager's satisfaction

Table 1. An example of a goal, related questions, and measures

4. Key Measurements for Testers

4.1 Test Planning and Resources

If you are a testing manager, one question you may have is “do we have enough testing resources for the testing we are planning to do?”

How many tests do we plan to run? (You can estimate this if you don't know the answer exactly yet.)

How long does each test case take to design and write?

How long does each test case take to run, on average?

How many full testing cycles, when we get to system testing, do we expect to run?

(Usually system testing will run at least 3 or 4 cycles, finding and fixing bugs in each cycle, before system testing is done.)

How many person-days do we need? (number of tests * time per test (time to write) + number of tests * staff time to run * number of cycles) –if the time to write and run is in minutes, then divide by minutes per day to get number of days.

(Note that most people don't get 8 hours of work done in an 8 hour day. There are meetings, email to read, and other activities that don't contribute to one's main tasks. In some organizations about 50% of the time is taken up with these other activities. It's reasonable to estimate you'll have at least 1/3 of your time in other activities, and thus you would use about 27 hours for a week, or 5.3 hours for a day, in the above calculations.)

How many testing staff do we have?

How long will the testing cycle take, with the staff we have? (# days total / number of testing staff).

(This assumes that equipment is not a critical resource –i.e. that testing staff can test productively during all the time they have. This may not be true in your environment..)

Is the testing phase too long, i.e. our current staff is not sufficient? Do we have to test less or can we add staff?

These calculations are affected by test automation, of course. It takes longer to write an automated test; but running it is shorter. Industry data suggests that the effort to develop an automated test case is about ten times the effort to develop the same test manually. But then the run time may only take a few minutes of setup, or be completely automated and run at night while staff is not at work, so the execution time may be very short in terms of staff time used. For automated testing we'd estimate the time to develop each test case at about 10 times the amount we used in the calculations above, but the time to run tests is very close to negligible. A set of automated tests can be run overnight and the results analyzed the next morning.

For automated testing the formula is, (number of tests * time per test (time to design and to set up/code/script in the automated tool) / minutes per day = person-days we need if we were to automate this number of tests. If the time to analyze the test results after an automated run is significant, it should be allowed for in the calculations, by lowering the minutes per day available for developing new automated tests by the number of minutes per day needed for analyzing the prior runs.

There is a third possibility, where the time of testing is ruled by the machine hours needed. For example, we might have a 120 hour test that has to be run without failure – that may dictate the time required to complete testing.

Of course, we may not try to automate all tests at once, and if so, use both formulas, with the number of manual test cases and the number of automated test cases plugged into each. The total person-days will then be the sum of the person-days needed for manual testing plus the person-days needed for test automation.

For agile projects and manual testing, an approach using test sessions has been developed by Jonathan Bach^{2,3}. Make a list of areas needing testing in advance. Do testing in short sessions, ideally about 90 minutes long, but between 60 minutes and 120 minutes. Set up To Do sheets with one area, testable in 90 minutes, on each sheet. The tester reports the percentage of testing, bug reporting, and setup time within the session in a quick debrief interview at the end of each testing session. A lead or test manager runs the debrief session with the tester.

The calculations look like this: How many perfect sessions of 100% testing does it take to do a testing cycle (say 40)?

How many sessions can a team of 4 testers do in a day? (Say 3 per tester, or 12 per day)

How productive are the sessions? (Say 66% is actual testing)

Estimate of time needed: $40 / (12 * .66) = 5$ days

As testing proceeds and some areas turn out to be larger or harder to test than expected, and thus take more sessions, the calculations can be adjusted accordingly, and new To Do sheets can be written.

4.2 Definition of a Defect

Defects are one of the most often used measures of quality. Clearly quality has many other components, including fitness for use, performance, usability and so on, but the commonest measure is defects.

Exactly what a defect is needs to be defined by your organization. Is it only items found by customers; customers and testers; or customers and testers and developers (unit test defects)? Does it include items found in upstream peer reviews? Does it include only non-trivial items?

Also, are small enhancements ‘defects’? Some organizations choose to categorize enhancements as a type of defect. Others keep enhancements as a separate category. Probably some percentage of enhancements arise due to requirements errors, and one could argue that these should be categorized as defects. It should be possible to categorize enhancements by whether they arose due to a requirements problem (missing, misunderstood, not what the customer wanted) or are genuinely something that did not result from a requirements problem (for example, new government legislation that did not exist when the requirements were written; a brand new idea that did not exist earlier).

4.3 Cumulative Defect Plots

Many testers know how many defects are open and closed at any given time. A graph over time of defects found, fixed and still open is very useful for seeing the progress of testing. Such a graph for an ideal project is shown below in Figure 2. Defects found is a cumulative number that can be gotten from the defect tracking system by seeing how many defects were found at different times from the start of testing. Defects fixed is also cumulative. Defects open is the difference between found and fixed at any given time. One reference which uses this graph is the Manager’s Handbook of for Software Development, available online from NASA ⁴.

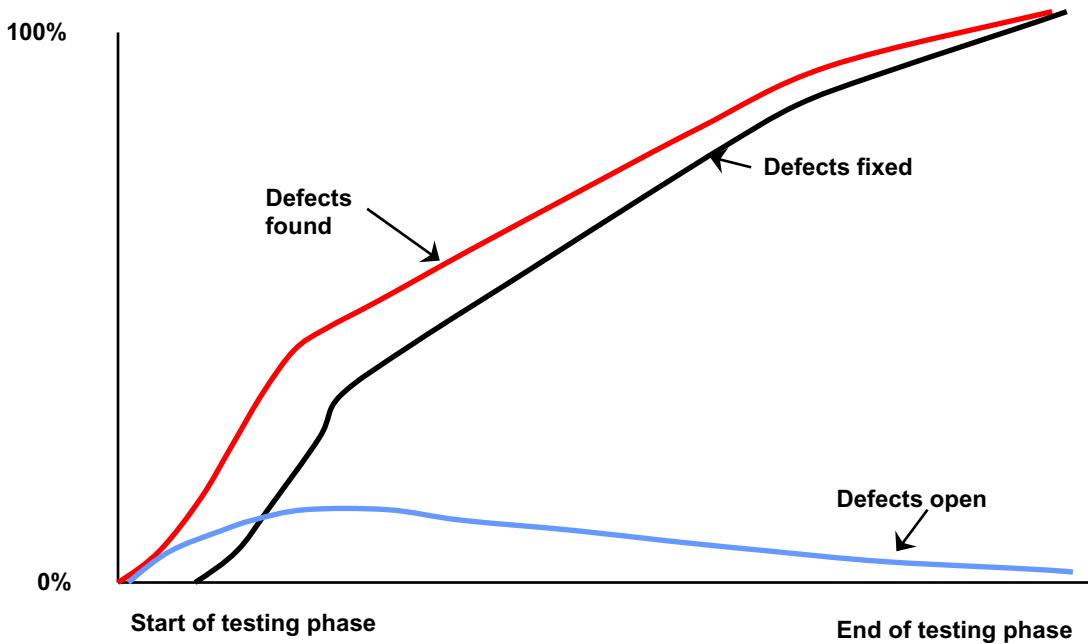


Figure 2. Cumulative Defect Plot

The plot in Figure 2 is an ideal plot. Defects are fixed almost as soon as they are found, and we are fixing defects faster than we are finding them (the time where the slope of the open defect line is negative) from very early in the testing phase. Probably this seldom happens!

A more realistic plot is shown below in Figure 3.

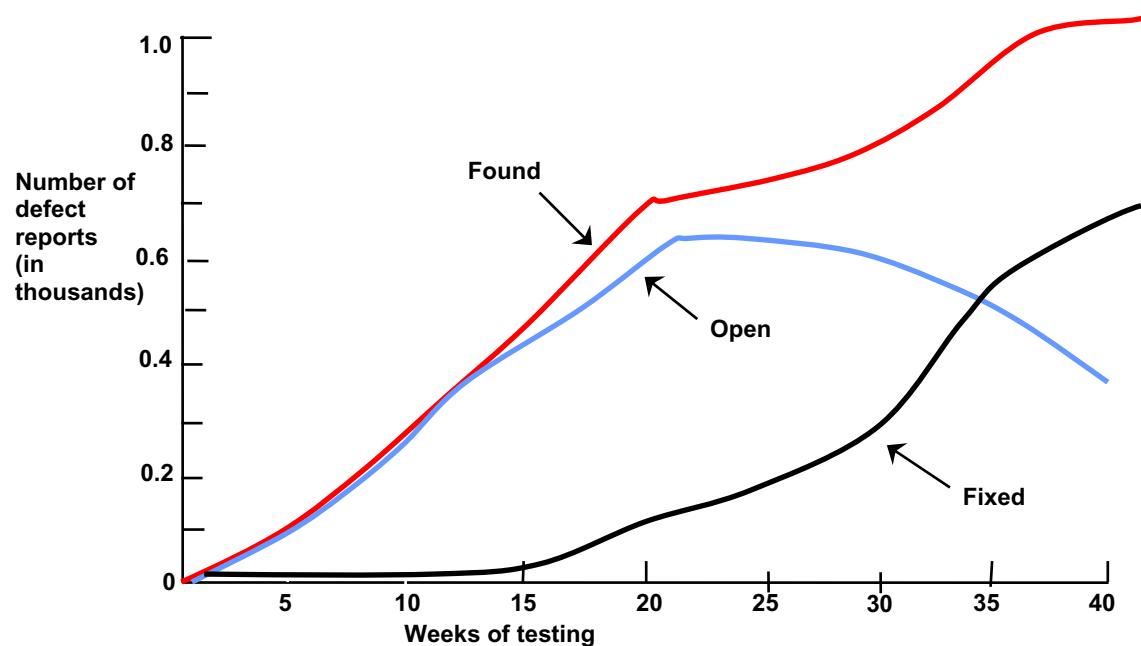


Figure 3. Cumulative Defect Plot from an actual project

Here few defects are fixed until week 20. The number open and the number found are almost the same until week 15. The number fixed isn't greater than the number still open until week 35. Until the slope of the open line becomes negative, you can't predict when the testing phase will end –you are still finding defects faster than you are fixing them. Once the open and fixed lines cross, you've fixed more defects than are still open, which is a cause for celebration. Once you are fixing them faster than you are finding them, you can see the light at the end of the tunnel, and begin to predict when testing will be finished. Of course, it's possible that a new type of testing or a new set of test cases will be run late in the testing phase, and there will be a new peak of finding defects and the number open will rise again, but it's usual for the number open to stay below the number fixed once it finally crosses the fixed line.

This plot is basic, and should be produced by all test teams. The data for it should be already available in the defect tracking system, so it should not require much effort to implement.

Larry Putnam and Ware Myers ⁵ have developed formulas for estimating software projects based on the Rayleigh curve. They have also shown that the rate of finding defects follows a Rayleigh curve, as shown below in Figure 4. They have collected data from thousands of projects. This data shows that if the effort to find 95% of the defects is x , to find 99% takes $1.25x$ more effort and to find 99.9% takes $1.5 x$ more effort, as shown by the lines on the diagram in Figure 4. A batch program typically has to run for about 7-9 hours. When the Mean Time to the next Defect (MTTD) reaches the point where the batch system runs for 7-9 hours without a problem, it was deemed ready for release. The 95% defect removal level often corresponds to a Mean Time to the next Defect (MTTD) of about 8-9 hours. The Rayleigh equation for the error curve can be used to calculate the MTTD. The predicted curve matches the actual defect find rate quite closely, within about 10%. The 99% defect removal level corresponds to a MTTD of about 2 weeks, and 99.9% to a MTTD of about 10 weeks. (see Putnam and Myers pp. 126 –132) Systems which must run 24 hours a day seven days a week, as many systems do at the current time, need 99.9% reliability, which is much more expensive than 95%. The cheapest way to get to the needed reliability, whatever level it is, is to find most of the defects before the testing phase, for example by using Inspections.

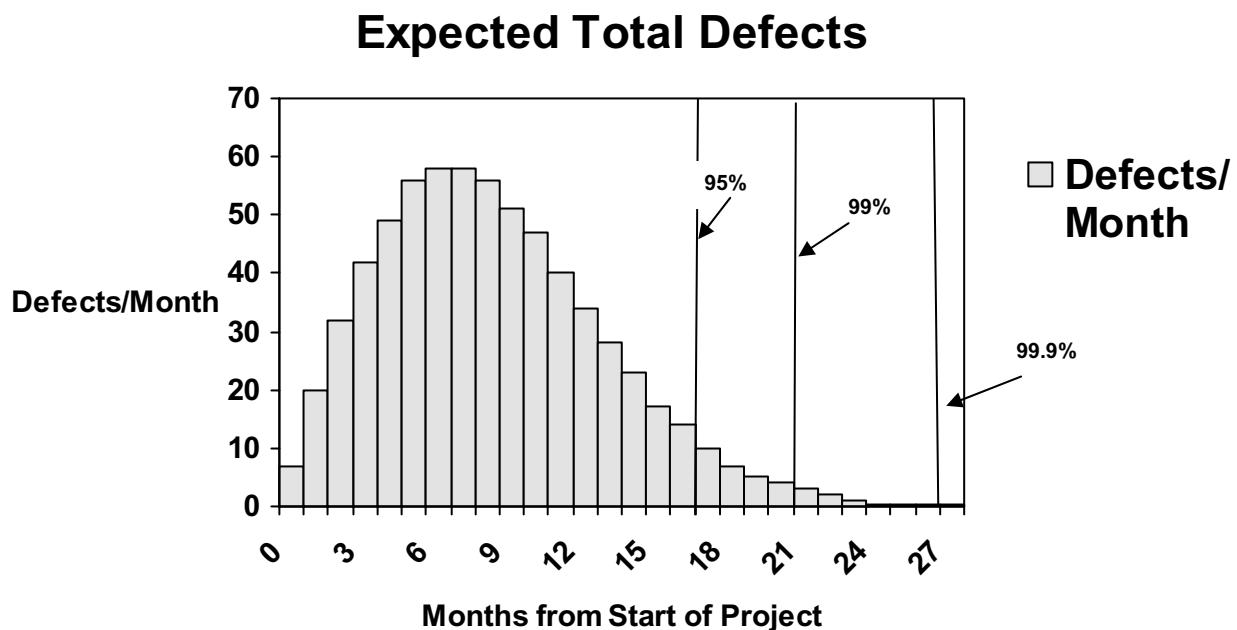


Figure 4. Defect rate follows a Rayleigh Curve

Erik Simmons⁶ has used the fact that the defect arrival rate follows a Rayleigh curve to use weekly measures of defect arrival to predict the future rate, and therefore when testing will be done. It is possible to use the prediction to model

- When a given percentage of the total defects have been found
- When the arrival rate will have dropped below some threshold

4.4 Effort Per Defect

It is useful to record effort to fix defects. Data on the time required to fix defects, categorized by type of defect (originated in requirements, in design, etc), provides a basis for estimating the remaining defect correction work during the last third of a project, i.e. during system testing. This data should be recorded by the developer who fixed the defect as part of the defect close process. The data does not have to be extremely accurate. Estimates such as '2 hours' or '5 hours' are sufficient. Many organizations don't know how much defects cost to fix. Organizations that collect this data often find the cost to fix each defect averages \$1500 or more.

Hewlett Packard⁷ developed a defect fix time model that works in their environment. They found that

- 25% of the defects take 2 hours to fix
- 50% of the defects take 5 hours to fix
- 20% of the defects take 10 hours to fix
- 4% of the defects take 20 hours to fix

- 1% of the defects take 50 hours to fix
- A similar model could be developed in other organizations.

4.5 System Test Profile

Another useful plot, also used by NASA⁸ is the System Test Profile. This plots the tests run and tests passed, with a line for the total tests planned, as shown in Figure 5 below.

The dashed line shows the total tests planned. They may increase during the testing period, but in the example in Figure 5, the number planned remains constant. The solid line shows the number of tests executed and the dash-dot line shows the number of tests passed. Ideally, the number of tests passed line will stay close to the number of tests executed, i.e. the defects causing tests to fail will be fixed quickly, so the tests will soon pass. If the tests passed line falls far below the tests executed line, we are getting behind in fixing defects. The diagram below shows a fairly ideal condition –tests that fail are soon passing.

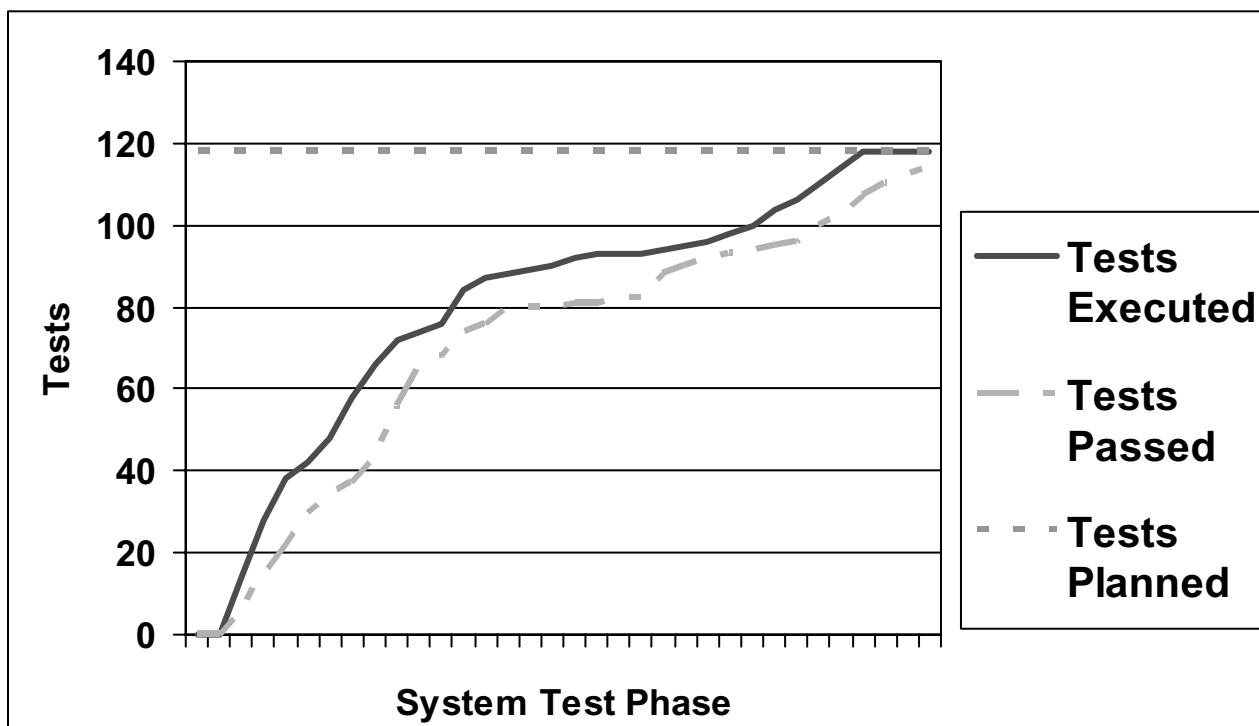


Figure 5. System Test Profile for ideal situation

The prior diagram shows a more or less ideal situation. The following diagram in Figure 6 shows what the diagram would look like if defects are not being fixed quickly, so that tests that fail continue to fail for some time after they are originally executed.

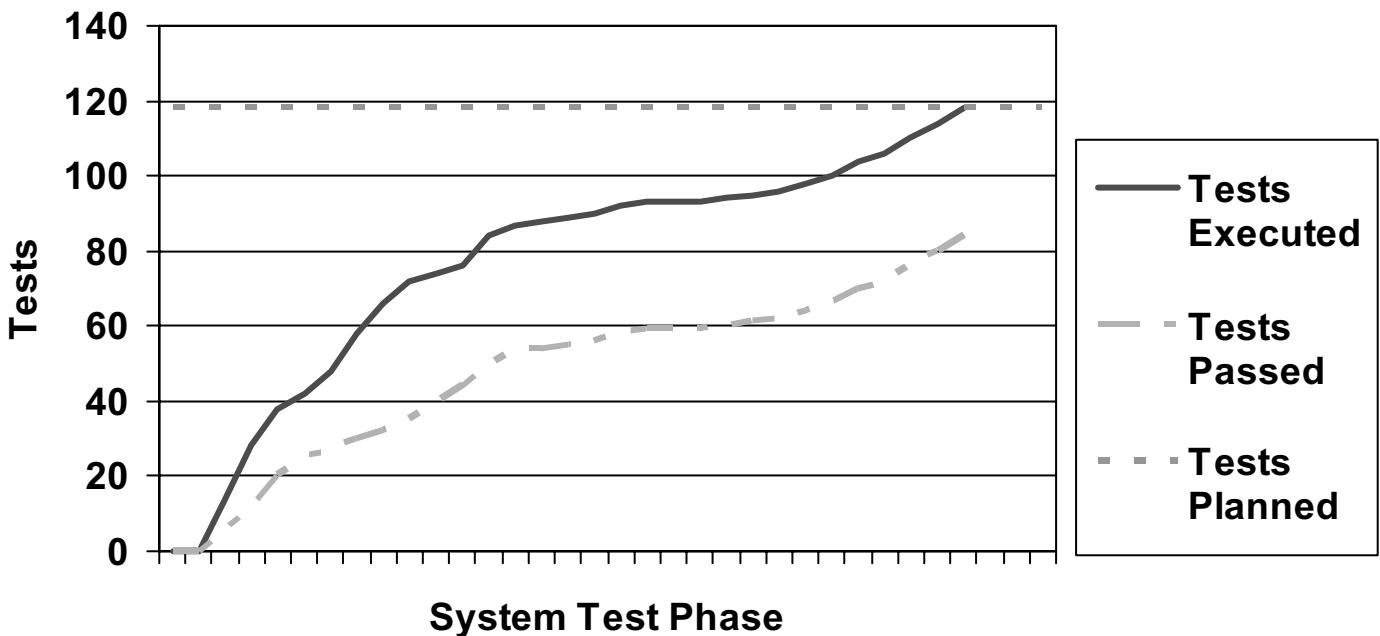


Figure 6. System Test Profile when failing tests are not fixed quickly

4.6 Cumulative Defects and Defects by Phase

Figure 7 shows Cumulative Defects plots from NASA⁴. Here upper and lower limits as well as the expected historical norm are shown. Then actuals are plotted and if the actuals lie between the upper and lower bounds, the project is assumed to be running normally. If the actuals lie outside the upper or lower bounds, then the project manager investigates what the cause might be. The same techniques can be applied to the plots in Figures 2 and 3, i.e. using historical data, the future shape of the plot can be predicted.

NASA consistently finds about 4 defects/KSLOC (thousands of lines of source code) in unit testing, another 2 defects/KSLOC in system testing and 1 defect/KSLOC in acceptance testing, for a total of about 7 defects/KSLOC found in testing. NASA does reviews of requirements and designs before getting to the testing phase, and removes many defects before reaching the testing phase. In an organization which does not do reviews, there could be many more defects remaining to be found in testing.

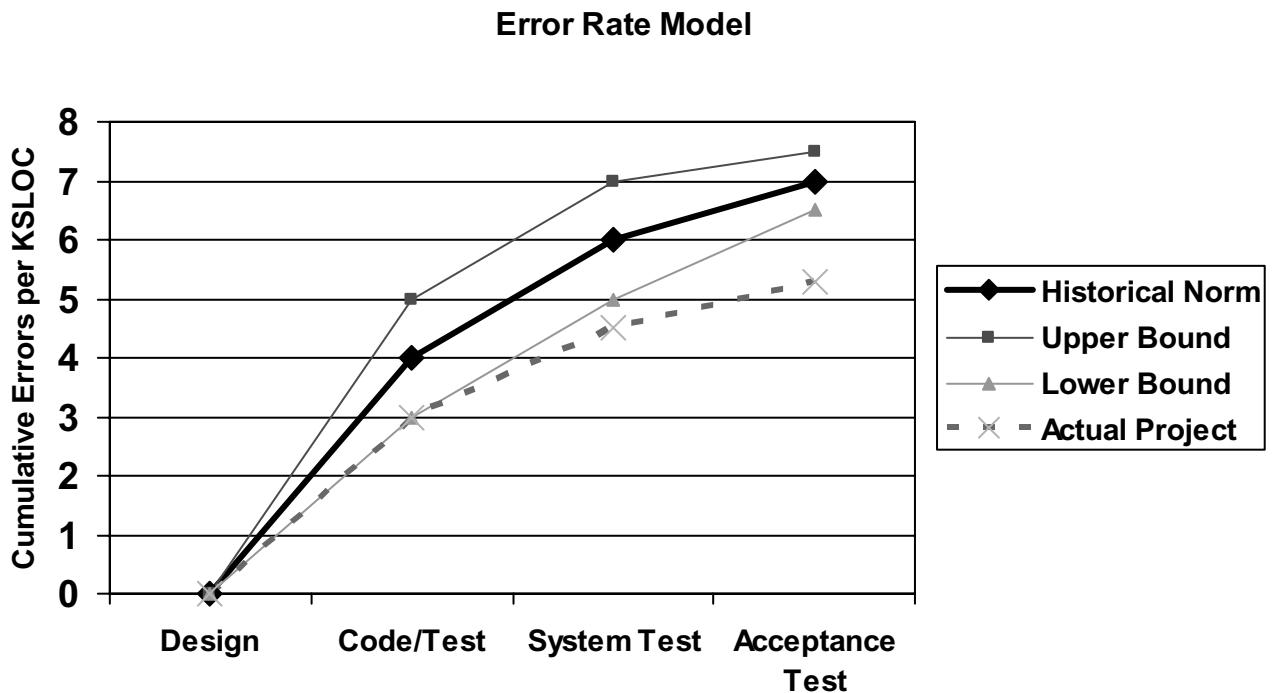


Figure 7. Cumulative Defects Found in Testing –Predicted and Actuals

In Figure 7, the actual errors per KSLOC are less than predicted. This could occur because less testing was done –in which case more thorough testing is needed, or because the software had higher quality than usual. In this case, the software had higher quality (fewer errors) than was usually found.

Putnam and Myers⁵ studied the factors that affect future defect rates. Their findings are shown in the following table.

Increasing Factors	Decreasing Factors
System size	Simplifying the application/problem at hand
Application complexity	
Compressing the schedule	Extending the planned development time Cut in half
More staff	Fewer staff
Lower productivity	Higher productivity

Table 2. Factors that affect future defect rates

Extending the planned development time from the calculated minimum development time (from an estimation equation) can reduce the number of defects expected by 50%, if the extension is significant. For example, deliberately planning a schedule two months longer

than the minimum –15 months instead of 13.1 months, cuts the number of defects expected in half. This may not be possible for most current projects, because schedule seems to be the primary determiner at the current time, more than cost or functionality.

In order to judge whether we've found all the defects for an application, we can estimate its defect density. We need statistics on the defect density (defects/KSLOC or defects/Function Point) of past similar projects. We use the past defect density to predict the defect density on the current project.

For example if our prior projects had a defect density of between 7 and 9.5 defects/KLOC, unless we have made some software process change, we expect the density on the current project to be similar. Thus if our new project has 100,000 lines of code, we expect between 700 and 950 defects, total.

If we've done some testing and we've found 600 defects so far, we're not done. We need to find at least 700 defects and maybe as many as 950, or 100 to 350 more than we've found so far.

It is also useful to record the phase where defects are introduced. Usually, the defects introduced in the earliest phases are the most expensive to fix. If we also record the phase where defects were found, we can see if we find most of our defects in the same phase in which they are introduced. This is the least expensive way. Capers Jones⁹ has reported on the percentage of defects, by severity, for development phases. This is shown below in Figure 8. Note that the most severe defects tend to come from the early phases of requirements and design.

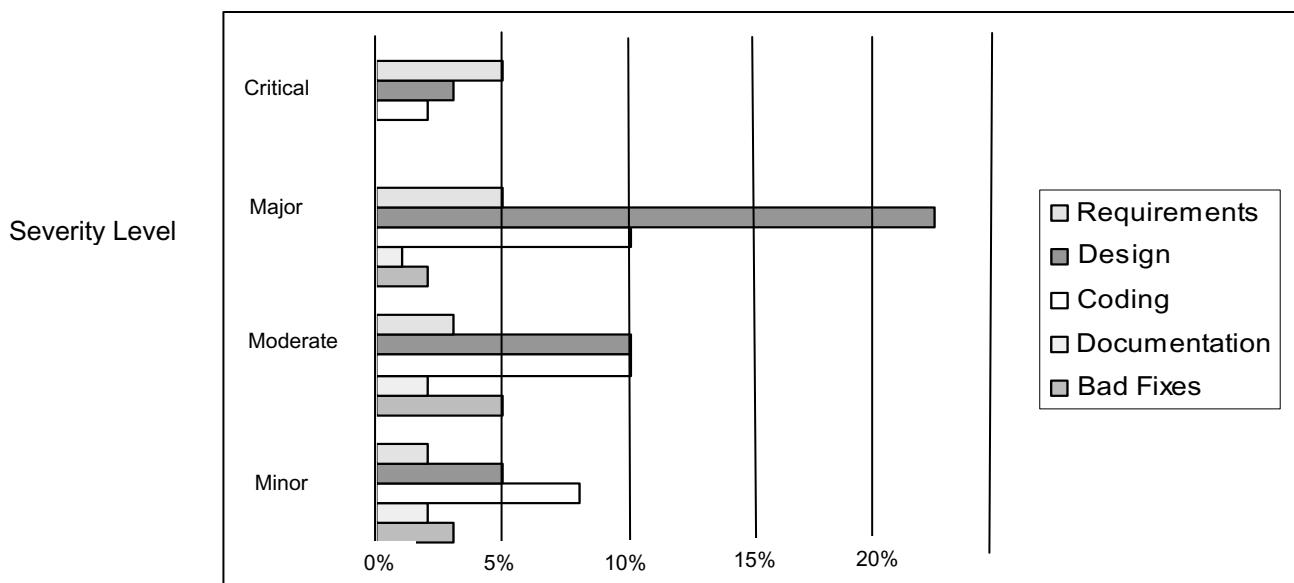


Figure 8. Distribution of Software Defects Origins and Severities

We can model, based on data like the above data collected by Capers Jones, the number of defects expected in requirements, design, construction, etc. His data shows approximately 15% of the defects come from requirements, 40% from design, 30% from coding, 5% from documentation, and 10% from bad fixes. Then we can predict how many requirements, design, etc. defects we expect on our current project.

Another approach, based on data from the SEI, gathered during Personal Software Process (PSP) training, is to model defects based on effort hours. SEI found that during design about 1.76 defects were injected per hour of effort, and during coding about 4.20 defects were injected per hour. The PSP students were given requirements for their programs, so no data was collected on requirements defects per hour of effort. Assuming effort data is collected by phase, this can be used to model the expected defect numbers.

The approach of modeling expected defects by phase and cumulatively was used by Edward Weller at BullHNN¹⁰ and the results are shown in the following figure:

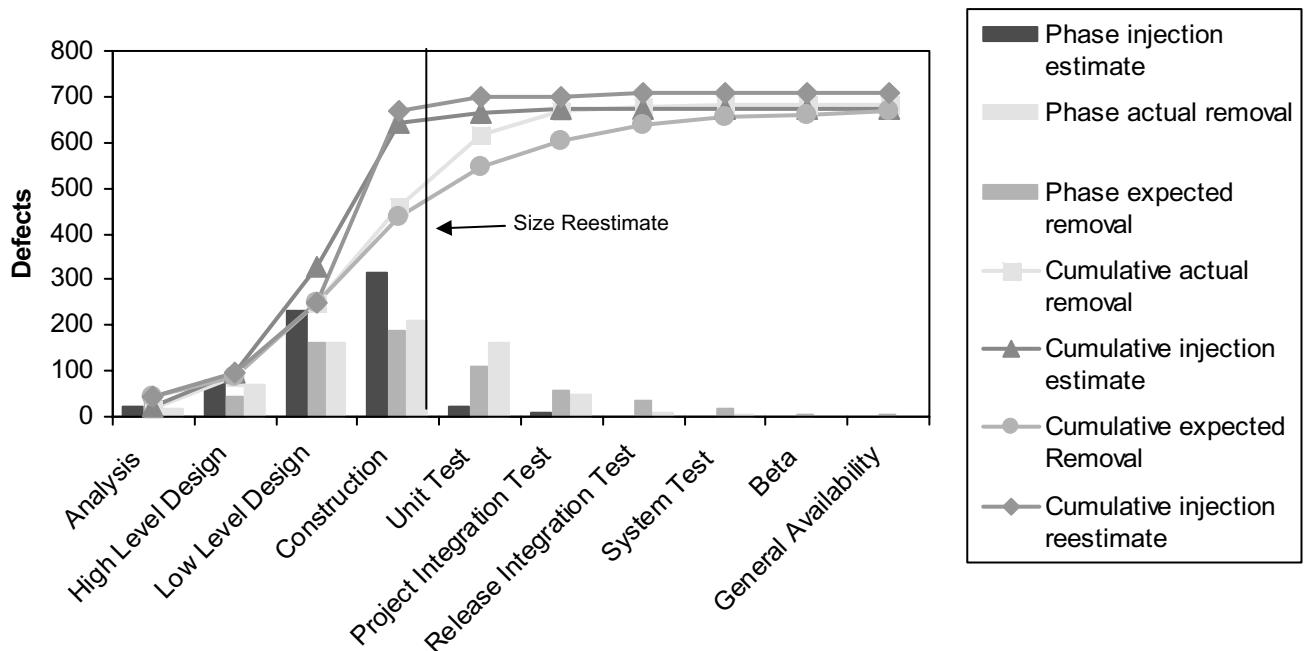


Figure 9. Predicted and Actual Defects found, by phase

This is probably as complex a measurement of defect prediction and actuals as any organization would do. Detailed historical data needs to be collected in order to estimate the number of defects that will be found in analysis, high level design, etc. Most organizations don't collect this data, so this graph would not be easy to implement for those organizations.

What use is it? Well, they found many more defects than they predicted they would, in unit test. They found many fewer in system test and in release integration test. If they hadn't known they'd found more than expected in unit test, they would have been worried that they didn't find enough in system test and release integration test. This can

be seen in the yellow actual removal line, too –the actual removal is higher than expected by quite a bit, starting with unit test; but at the end the expected and actual are almost exactly the same.

4.7 Defect Types

Most organizations record defect severity and/or priority and if the defect is open or closed. It has already been suggested to add the phase the defect was injected and the phase during which it was found. In addition, it is useful to record the type of defect. There are several type classifications. One developed at Hewlett Packard ¹¹ is shown below

Requirements/Specification
Requirements
Functionality
Design
HW interface
SW interface
User interface
Functional Description
Process (interprocess) communications
Data Definition
Module Design
Logic Description
Error Checking
Standards
Code
Logic
Computation
Data Handling
Module Interface/Implementation
Standards
Environmental Support
Test SW
Test HW
Development tools
Integration Software
Documentation
Other

Then, the type is recorded for each defect, and data on the percentage of defects in each type category can be extracted from the defect tracking system. It can be useful to plot this data in a pie chart. What is this data useful for? Well, if a large percentage of defects are due to requirements, for example, it may mean that the organization

needs requirements training, or a better requirements template, or requirements reviews. It points to areas where process improvement is needed.

Here's an example of a pie chart of defects by type. The percentages in the categories can vary a lot between projects, or between divisions in a large company.

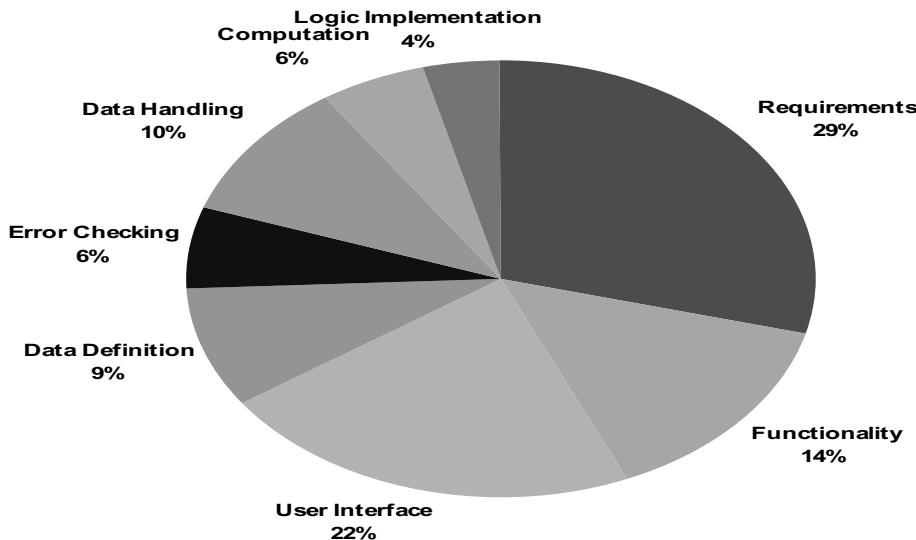


Figure 10. Defects by type category

4.8 Release Criteria

Open defect counts give a quantitative handle on how much work the project team has to do before it can release the software. The cumulative defect graphs as in Figure 2 and 3 can be used to indicate how close the project is to being able to release the software. When the software is nearing release, the number of open defects should be trending downward towards zero, and the fixed defects should be approaching the reported defects line. Also open severity 1 and 2 defects should be approaching zero.

Figure 11 below shows the graph of all defects when near release. Note that the number of open defects isn't zero, but it is getting very low. Figure 12 shows the severity 1 and 2 defects near release. Severity 1 and 2 defects should be zero at release, but some severity 3 and 4 defects may remain.

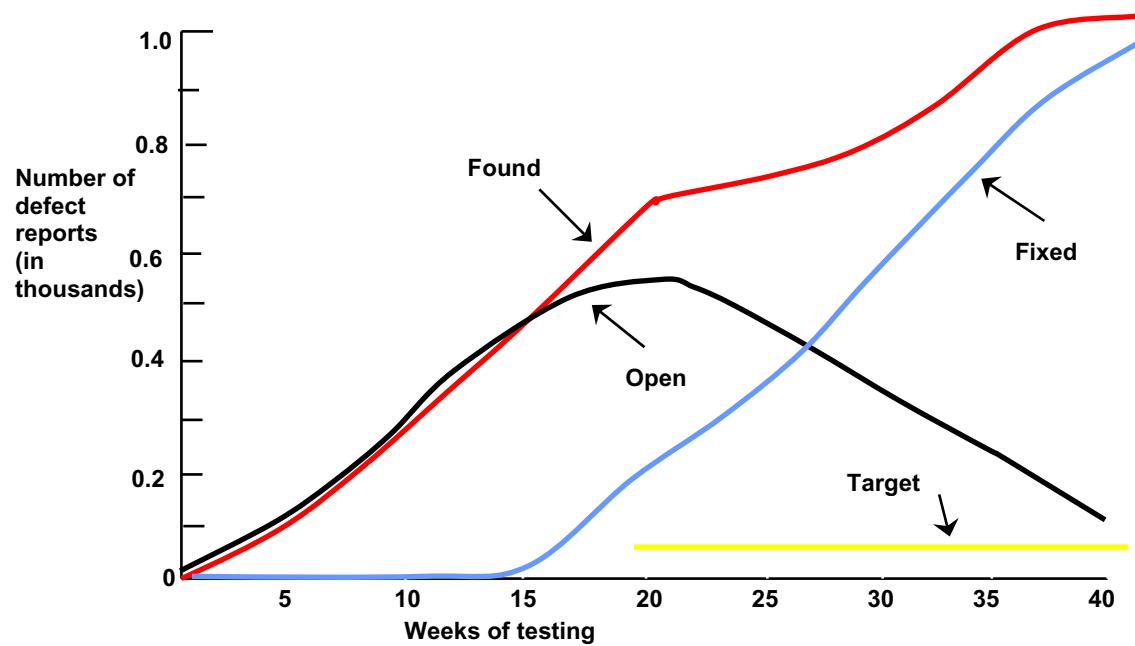


Figure 11. Defect trends near release, all defects

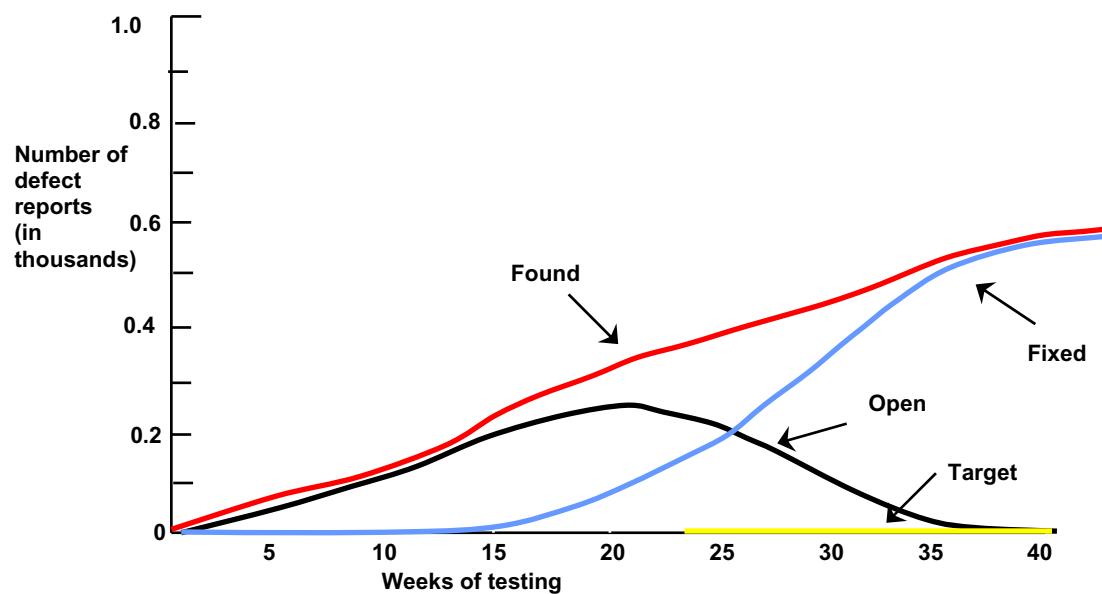


Figure 12. Defects trends near release, severity 1 and 2 defects

Construx Software uses these measurable release criteria:

- Acceptance testing successfully completed
- All open change requests have been incorporated, rejected, or deferred.
- System testing successfully completed
- All requirements implemented, based on the spec
- All review goals have been met
- Declining defect rates are seen
- Declining change rates are seen
- No open Severity 1 defects exist in the database
- Code growth has stabilized

Hewlett Packard¹¹ has also developed a set of measurable release criteria:

- Breadth –testing coverage of user accessible and internal functions
- Depth –branch coverage testing
- Reliability –continuous hours of operation under stress; stability; ability to recover gracefully from defect conditions
- Remaining defect density at release

HP has found a significant decrease in defects for those projects that agreed to meet the release criteria, as shown in the following graph.

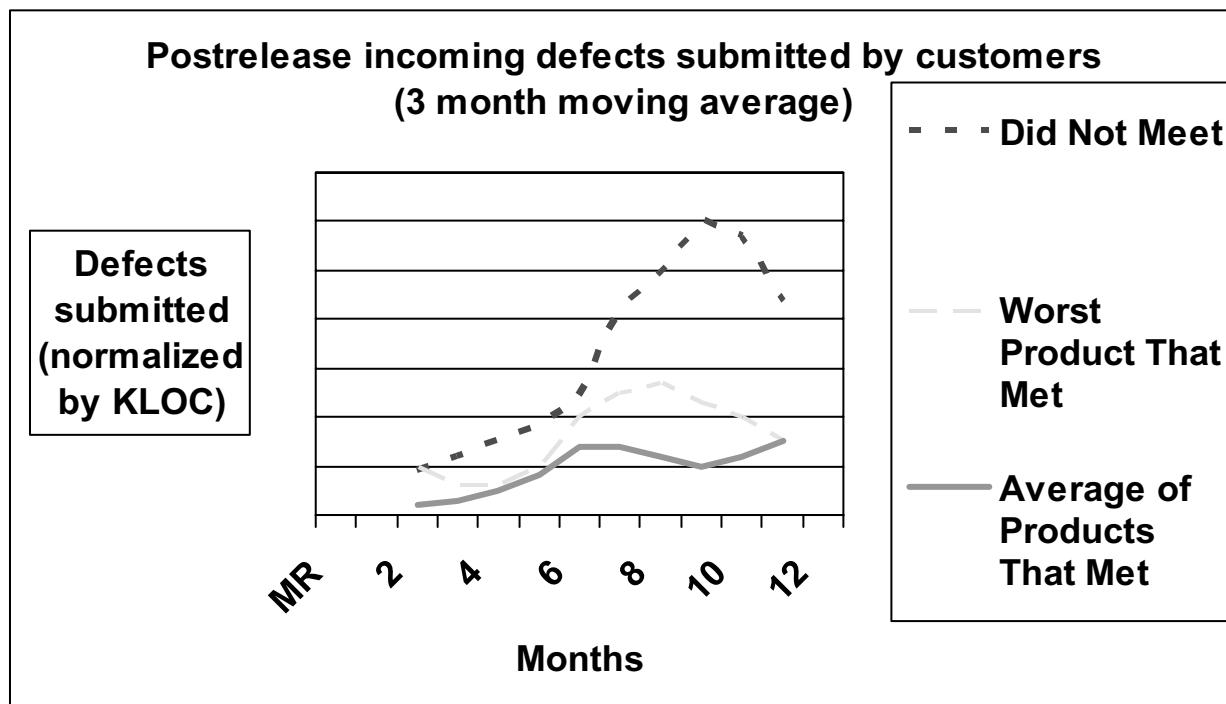


Figure 13. Post Release Defect Density by Whether Met Release Criteria

4.9 Defect Detection Effectiveness

The defect detection effectiveness of several defect detection methods has been estimated by Capers Jones¹², as shown in figure 14. High quality reviews, such as Inspections, have a higher effectiveness than testing. In the figure below, ‘checks’ are one person reviews, where you informally give a document to a colleague to review. ‘Review’ is a multiple person informal review, such as a walkthrough. Inspection is a formal inspection like the process described by Fagan, which has a documented process, checklists, etc. Some authors estimate inspections can be as much as 90% effective.

Capers Jones estimates that testing is between 25% and 50% effective, depending on testing type. These estimates were done some time ago, and it is probable that testing using some method of checking coverage, like the currently available code coverage or branch coverage tools, can be much better than these estimates. Testing effectiveness has been estimated to be as high as 75% in the best circumstances. However, it does show that for a typical testing cycle, in the absence of coverage tools, the effectiveness is not very high. When code coverage tools first were developed, in organizations that thought they had a comprehensive set of test cases, only about 30% of the lines of code were executed by the set of test cases. This may explain the low effectiveness of testing in finding defects. Jones also estimates that for a combination of techniques, the final effectiveness can be quite high, nearing 100%.

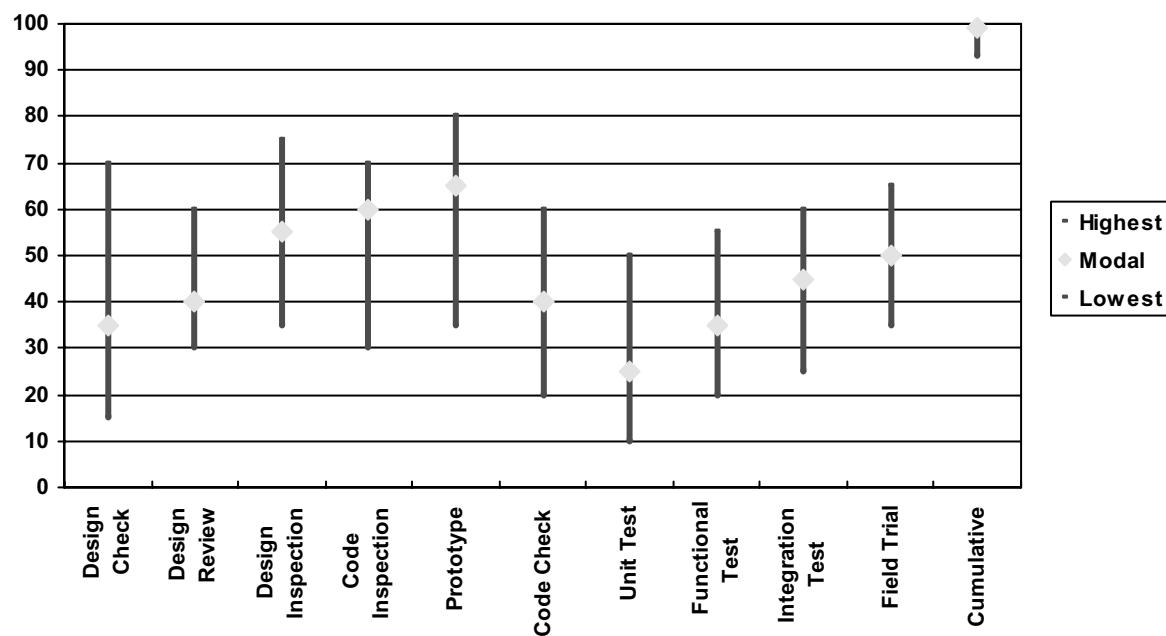


Figure 14. Defect Detection Effectiveness

4.10 Graphs for Executives

The graphs which are useful for top management may be different from the graphs already shown in this paper, which are useful for the test team and the project manager on

an individual project. Executives are likely to want to see rolled up graphs. An example is post release discovered defect density, over time, for all the projects the organization has released, as shown in Figure 15. This organization (HP) had a goal of reducing post release defects, over several years, by 10x.

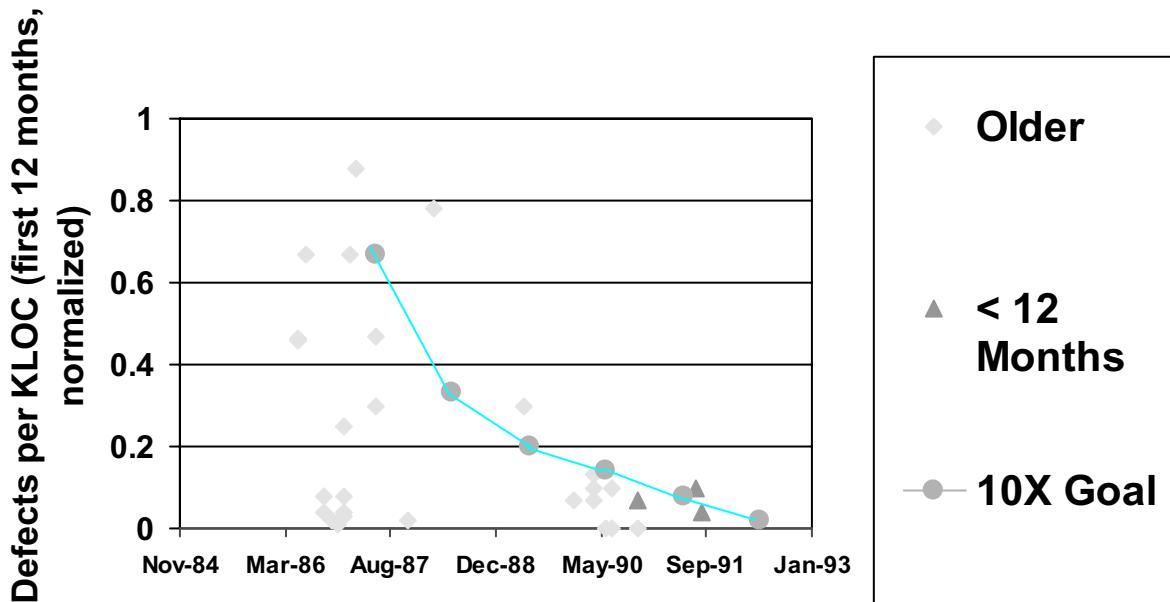


Figure 15. Postrelease Discovered Defect Density

Dots labeled ‘Older’ are from projects with at least 12 months of post release defect data collected, and the triangles are from projects with less than 12 months of post release defect data collected.

5. Conclusions

A number of organizations have used measurements, graphs and trends to track the progress of test case development, test case execution, and defects found, open and closed. Having graphs that show these measurements over time allows tracking and early warning when the project’s graphs are different from historical trends. The project manager or test lead can then investigate the cause of the difference.

Other graphs allow upper management to see if the organization is improving in such areas as post-release defects, defects repaired per month of effort, and the length of time the average defect remains open. Since many organizations are not measuring software development currently, doing so is a competitive advantage to those organizations that are measuring. As they implement process improvements, measurements tell them if their process improvement changes have had a positive effect.

6. References

- 1.** Victor R Basili, Gianluigi Caldiera, and H. Dieter Rombach, ‘The Goal Question Metric Approach’, in Encyclopedia of Software Engineering, Wiley, 1994, available at <http://www.cs.umd.edu/~mvz/handouts/gqm.pdf>
- 2.** Jonathan Bach, ‘Session-Based Test Management’, Software Testing and Quality Engineering, November 2000, at <http://www.satisfice.com/articles/sbtm.pdf>
- 3.** Jonathan Bach, ‘Testing in Session: A Method to Measure Exploratory Testing’, slides of a presentation to Washington Software Association QA SIG, May 13, 2003, available at <http://www.qasig.org/presentations/Session-Based%20Test%20Management.pdf>
- 4.** NASA Software Engineering Laboratory, Manager’s Handbook for Software Development, Revision 1, November 1990, SEL-84-101, available from <http://sel.gsfc.nasa.gov/website/documents/online-doc/84-101.pdf>
- 5.** Lawrence H. Putnam and Ware Myers, Measures for Excellence, Yourdon Press, 1992
- 6.** Erik Simmons, ‘When Will We Be Done Testing? Software Defect Arrival Modeling Using the Weibull Distribution’, Pacific Northwest Software Quality Conference, 2000 at <http://www.pnsrc.org/proceedings/pnsrc00.pdf> - the paper is at pages 194-210 and the slides at pages 211-243 in the proceedings document
- 7.** Robert B. Grady and Deborah L. Caswell, Software Metrics: Establishing a Company-wide program, Prentice Hall, 1987
- 8.** NASA Software Engineering Laboratory, Recommended Approach to Software Development, Revision 3, June 1992, SEL-81-305, available from <http://sel.gsfc.nasa.gov/website/documents/online-doc/81-305new.pdf>
- 9.** T. Capers Jones, Applied Software Measurement, McGraw Hill 1996, page 368
- 10.** Edward F Weller, ‘Practical Applications of Statistical Process Control’, IEEE Software, May/June 2000, pp 48-55, 2000
- 11.** Robert B. Grady, Practical Software Metrics for Project Management and Process Improvement, Prentice Hall PTR, 1992
- 12.** T. Capers Jones, Programming Productivity, McGraw Hill, 1986

Quantifying Software Quality – Making Informed Decisions

Bhushan B. Gupta

Orhan Beckman, Ph.D.

Indigo Publishing Division, Hewlett-Packard

Bhushan.gupta@hp.com

Orhan.Beckman@hp.com

Bhushan B. Gupta has 21 years of experience in software engineering, 11 of which have been in the software industry. Currently a Software Process Architect at Hewlett-Packard, he joined the company as a software quality engineer in 1997. From 1995-97 he worked as a Systems Analyst at Consolidated Freightways where he contributed to the design and development of a Windows based logistic management system. Prior to that Bhushan Gupta was a faculty member at the Oregon Institute of Technology where he developed, planned and taught numerous software engineering courses. He has served PNSQC in various capacities including as a Vice President and a board member. Bhushan Gupta is an advocate for quality and offers workshops in software quality engineering.

Orhan Beckman, Ph.D.

Orhan Beckman is the Human Factors Engineer/Scientist for HP's Indigo Digital Press division. He leads Human Factors research and design for new products and solutions in the commercial printing industry. Dr. Beckman joined Hewlett-Packard in the spring of 1994. He received his Ph.D. in Industrial/Organizational Psychology from Old Dominion University in 1998.

Abstract

Software development and release decisions are often made based on subjective data confined to functionality and defect levels. Software defect levels, although often quantified, do not alone adequately represent the many facets of software quality. The presence, absence, or severity of defects, the authors argue, is not a sufficient measure of subsequent product quality or customer experience. Attributes such as Installability and Learnability can be measured to provide insight into the initial customer experience. Usability can be measured to assess the degree to which the software allows the user to achieve key tasks. Lastly, Performance, Security, Scalability, and Reliability can be measured to better understand how the software will meet critical, but often overlooked, aspects of the customer experience. A product with zero defects can still fail in the end if other key dimensions of software quality are not satisfied.

This paper discusses a method that can be used to characterize and measure software quality in a holistic manner. It describes key quality attributes and techniques that can be used to measure quality on a larger scale. The authors reflect their experiences defining, developing and delivering software into commercial markets. The authors illustrate how these attributes may be graphed as a "Radar Chart" (also known as Spider Diagram) to guide the software development to achieve a desired customer experience. The authors share their experiences on how they knitted these attributes together into a simple but effective method that can be applied to a wide variety of software domains. They present a method to select and quantify key attributes and ultimately express software quality in a holistic manner leading to well defined acceptance criteria. The authors also discuss practices and approaches in the pursuit of the larger quality picture that have yielded desired results. The reader of the paper may benefit from lessons learned, both positive and negative, and internalize practical insights that can be immediately applied.

Introduction

A software product should provide customer value consistently through out its lifecycle. The customer must be able to Install, Learn, Use (accomplish intended tasks), Maintain, and Dispose of the product without sacrificing productivity. These aspects can be used to define the Total Customer Experience (TCE). Depending upon the customer and the value the product provides these aspects may vary in importance. For example, if a software product has to work with an existing hardware product it should not negatively impact the hardware functionality. This would introduce “compatibility” as a new attribute in the system. In order to better evaluate the potential of a software product we can understand and measure these attributes and represent them as components of “Compound Product Quality”.

These attributes have been utilized by various organizations to understand product quality. For example, Hewlett-Packard uses **F**unctionality, **L**ocalization, **U**sability, **R**eliability, **P**erformance, and **S**upportability (FLURPS) to assure adequate product quality [1]. IBM has also used a similar paradigm to achieve the same objective [2]. Intel [3] has also explored these attributes to measure product quality. While the application of these attributes has been broadly understood in building product quality, there are no known attempts to use these attributes as “Compound Product Quality”. The rest of the paper explores a number of these attributes and describes how the authors used these attributes to monitor product quality, support product development and make release decisions.

Background:

This study is based upon experiences acquired from developing application software for the Indigo Digital Press Division of Hewlett-Packard. The application provides the front end services to a variety of HP Digital presses. The development teams are distributed across geographies and time zones. A program management team is responsible to assess the product quality during development and make release decisions based upon an approved acceptance criteria. The group uses agile development methodology. Each software iteration addresses features or customer stories. Each is characterized as a functional component that delivers customer value.

Attributes:

Based upon the frequency of use and customer value the authors have selected a set of prominent attributes for this discussion. Depending upon the application it is advisable to consider additional attributes that may impact the customer value. The set includes **I**nstallability, **L**earnability, **U**sability, **F**unctionality, and **R**eliability. A literature search has yielded the following definitions:

Attribute	Definition	Source
Installability	The capability of the software product to be installed in a specified environment.	http://www.isi.edu/natural-language/mteval/html/222.html
Learnability	Ability to gain knowledge, comprehension, or mastery of through experience or study.	Derived from Learn (Webster Dictionary)
Usability	The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.	ISO 9241-11
Functionality	The capacity of a computer program or application to provide a useful function.	Webster Dictionary
Reliability	The ability of a system or component to perform its required functions under stated conditions for a specified period of time	IEEE Standard Computer Dictionary, 1990

Table 1: Attribute Definitions

Attribute Characteristics and Measurements

The following discussion further characterizes and establishes the measurements for the attributes in Table 1. The authors believe that Installability, Learnability, Usability and Reliability are as important as Functionality.

Installability

Installation is an important event in the product lifecycle because this is often the first contact the customer has with a software product. The installation experience often sets the tone for a user throughout the product lifecycle. There are three stages in the product lifecycle related to installation – first, when a customer installs a product for the first time, second, when the product is updated by the provider and third, the end of life – when the customer does not want to use the software any longer and would like to dispose it. Uninstall is a process that needs to be considered in conjunction with Installation. Although it can be considered separately, the authors have elected to combine these two together as they are often designed and developed in parallel.

Installation characteristics: Simmons [4] has described the important characteristics of installation. Although Simmons explored these characteristics for the Unix environment, they are equally applicable to the Windows environment. The authors have further studied these characteristics and classified them for their better understanding and usage.

Conformity to standard shop policies and practices:

Each shop is governed by a set of security and other standards that should be maintained for its smooth operation. An installer must assure the user that configurations are maintained according to the shop practices. This leads to two aspects of conformity:

- Modification/change in System files such as registry or configuration files
- Modification/change in Existing User Setup files.

Install Area:

An installer should store its code and other necessary files into a single area and minimize broad interactions. The installer should not leave extraneous files, those related only to the install process not to the program itself, in the system as this can result in a lengthy and often manual removal process after installation. Most installers localize these files in a temporary directory.

Componentization:

If a product supports multiple disjoint or loosely related applications, it should provide options to pick and chose the components that the user intends to install. Most installers provide a choice of “typical” or “custom” install. A “custom” install must clearly differentiate between multiple components and provide a simple method to select items. A clear differentiation between multiple components on the installation user interface can help streamline the process for the user.

Installation Tool

The software industry has challenges when it comes to developing and following standards. The confidence level of users increases as they become familiar with a process. User errors can be reduced if the same user interface style is used to install different products.

Updates

Invariably a software product has to be updated as new functionality is added and missed defects are found and fixed. The following factors should be considered when updating existing software:

Ease of installation

A “push” practice has been gaining acceptance in the software industry. When software is updated, users are provided a link to a convenient location on the web where they can download the software. These updates are often free of charge and incremental and it is user’s choice to upgrade. When the upgrade involves major additions or changes the other installation methods, such as shipping a CD to the customer, may be used.

Data Integrity

Any upgrades of software must not compromise the customer data whether it is the data utilized by the software or by other software in the system. If the data needs to be downloaded and then uploaded after an update, it should be automated as much as possible to minimize the potential for human error.

Disposal

This is when a software product is obsolete and must be disposed off. Most products provide an Uninstall utility. A good utility must conform to shop policies and practices as described in the installation section, must not leave any remnants on the system nor compromise any data integrity.

Cancel

Although rarely used, the Cancel operation is critical for all software installations including small utilities, plug-ins and web-based installations. Cancel operation must not compromise system or data integrity. It should leave the system in the same state it was in prior to the start of installation process.

Usability

Usability, as the ISO definition illustrates, is a multidimensional quality attribute. To measure usability a number of independent variables must first be characterized and defined with an appropriate level of specificity [5, 6].

First, define the product. In terms of usability the definition of a product likely consists of more than just the ‘software user interface’. It often includes the hardware and software user interfaces, the documentation, training, packaging, support services and any other component of the system and processes surrounding it that affect the users.

Second, define the user or users of the product. Few products have just one user. Often the person who demonstrates, installs, uses, upgrades and supports the product is not the same. These users can be grouped into user types, for example: Sales, Trainer, Network Administrator, Production Manager, Remote Support, and On-site Support. To foster user perspectives inside and outside the lab, add details to each user type such as level of education, relevant experience and psycho-physiological attributes. A popular method to make the target users of a product appear more real is give each user type a name and a picture. The authors have seen this approach result in engineers and marketing refer to a target user specifically as “Jane” rather “the user” and ask questions such as “Is that something Jane could do?”, or “Will Bill be able to use that design to achieve his specific goals?”

Third, consider and document the goals each user type will try to achieve with the product. User goals can be written as statements that capture what the user is trying to achieve, for example: “Install the product without outside help in under 5 minutes”. To identify user goals we learned about the target customers by discussing customer profiles with marketing and customer understanding and insights from technical support as well as met with actual users of legacy or competitor’s products to understand what goals being (or trying to be met) through the use of the product. Overlap between the different sources increase confidence that a goal is valid. We found limiting the number of goals per user to a handful helps to maintain focus and curtail the subsequent test matrix.

Lastly, define the context in which each user type will be using the product to achieve the specified goals. When defining the context of use include specifics such as the size of (or range of sizes) of the computer monitor, the operating system or systems, the lighting, connectivity types and rates, documentation and support resources. The context of use may be the same or different for different user types.

With these four independent variables defined, a specific product, user, goal and the context of use, the dependent variable usability can be measured. Usability is often measured on five dimensions:

- 1) Assistance: The amount of assistance required to achieve a specific goal
- 2) Success: Whether the user actually achieves the goal
- 3) Time: The amount of time it takes to achieve the goal
- 4) Perception: The user’s subjective rating of the experience achieving the goal
- 5) Efficiency: The amount of resources the user expends achieving the goal

Set a level of acceptability, or acceptance criterion, for each of these five dimensions. Then record target user performance as he or she attempts to complete each task. The usability score can then be scaled from 0 (poor usability) to 5 (high usability) depending on whether each dimension of usability meets the acceptance criterion or not.

Learnability

Learnability can be defined and measured in a manner similar to usability and can be thought of as a subset of usability. Document the independent variable definitions covering product, user type, goals and context. Then decide how much knowledge, comprehension, or mastery through experience or study you expect target users of the product to have before the product can be effectively used. The authors have used five levels to define and communicate more clearly about Learnability throughout development:

- A product is “Intuitive” if users can find and operate the desired functionality upon first encounter as if they were already familiar with the user interface.
- A product is “Discoverable” if with an acceptable amount of exploration, a user can find and operate the desired functionality, also sometimes referred to as “one-trial learning”.
- A product is “Learnable” if with documentation or on-line help, a user can learn how to find and operate the desired functionality.
- A product is “Teachable” if by attending training, in which concepts and organizational principles are explained – and questions are answered, a user can learn how to operate the desired functionality. This level of learnability requires study and practice.
- Lastly a product is “Possible” if with step-by-step personal guidance a user can be led through a task, but without developing more than rote understanding about why a task is accomplished in the peculiar way. This level of learnability requires step-by-step instructions or memorization.

Like usability, it is critical to discuss learnability early in the product’s development. Set a goal describing an acceptable level of learnability of the product. Then work with development partners to ensure the goal is recognized, designed and developed towards to ensure the goal is met or exceeded. The authors have experienced the consequence when learnability is not taken seriously by R&D until late in the development cycle. The end result was a product difficult to train and learn. It also made support of the product in the field more difficult as users often required step-by-step personal guidance to achieve goals using the product. The authors have also seen the positive effects of an early and continual focus on usability and learnability during the development of a software product. This resulted in product that required less training in the field and less support. It also provided R&D with important insights about the appropriateness of previously deployed training methods. Visits to target customers helped us understand why web-based training tools were not appropriate for our target customers working prepress departments supporting offset printing. The hectic pace and noisy environment often prevented the training tools from being effective. As a result of this customer understanding, we investigated and ultimately invested in off-site training methods that provided an environment more conducive to learning.

Functionality

As defined above, functionality is the capacity to provide a useful function. This definition highlights the fact that any function a software product provides must be usable. Software development, no matter how agile, starts with defining the functional requirements. Once defined, each function must be complete and validated that it will work as expected in the target environment(s). Two aspects that quantify the functionality are completeness and the number and severity of defects. The following is an in-depth discussion of these two aspects.

Completeness

A simple measure of completeness is the percentage of planned functionality complete at any instant during the development. Chunking is a practice used to manage overall functionality in logical units. Agile Development characterizes these units as “features” (Feature Driven Development) or “user stories” (XP) and delivers a predetermined number of units per iteration (or release). It uses features/stories not only as a measure of completeness but also a measure of velocity. Since each iteration delivers releasable code the delivered feature is either complete or not. Agile methods also facilitate iteration planning as a part of release planning thus focusing on what feature (or story) will be built in a particular iteration and the long term vision of a product. This is a convenient way to measure completeness. This measurement may not serve well where a “waterfall” development methodology prevails. In that environment completeness is subjective to the judgment of the development team.

Defects – Number and severity of defects

While completeness is a relatively simple criterion, defects need careful consideration when measuring completeness. The presence of defects degrades quality. Developing a defect free product is often expensive and time consuming. Therefore products are often delivered with some number of known and unknown defects. The definition of defect severity is based upon the customer impact and probability. Once the severity is known the overall product quality can be quantified based upon the number of defects. Acceptance criteria often specify “No High Severity Defects” as these are the defects with a high customer impact and high frequency of occurrence. It is therefore advised to monitor the number of severe (critical) defects throughout the testing cycle.

The number of defects found is a function of number of tests successfully executed. A successful execution includes end-to-end execution of test cases. An alternative is that a test executed partially meaning it did not proceed due to a failure or missing functionality. These tests are often called “Blocked tests”. Other scenarios may prevent test execution and result in blocked tests. A system under test will either pass, meaning it behaves as expected, or fail. The following data should be collected in order to validate the functionality of a system:

- Total number of tests to be executed
- Number of tests completely executed
- Number of tests passed
- Number of test failed
- Number of test blocked

In situations when all the test cases are not established, tracking the number of test cases to be developed will also be necessary.

Reliability

Reliability is a concept that has long existed in the hardware world and has been studied in software over a period of time [7]. To discuss the notion of reliability we need to first define failure. A system failure is defined as a departure in system behavior in execution from user requirements and is a user oriented concept [7]. In digital printing one goal is to print at the press (maximum) speed. Therefore anytime the press is not printing at engine speed due to software failures it represents a departure from the user's expected system behavior. An equally important criterion is the amount of time the user is unable to continue as planned due to system reliability problems. In printing environments a printed page is the natural unit and the number of pages printed in a specified time is a measure of reliability. Since print job characteristics vary with respect to image type and size, customers often prefer to define natural unit as the number of jobs printed. They both can be represented on a time scale, the amount of time the system has been working. Based upon these aspects, the following elements measure the reliability of software in a printing environment:

Availability:

Musa [7] defines software availability as the fraction of time during which the system is behaving acceptably. It is the ratio of uptime to the sum of uptime and down time. Once again the operating conditions must stay the same. This means that the print jobs characteristics are the same while measurements are taken. The other metric that can be derived from these data is the Mean Time Between Failure (MTBF), also a measure of availability. For MTBF the measurements are taken for a predetermined time using the normal operational conditions. In a specified time, the uptime and down time are measured and the availability is computed. The time can be measured as the execution time (processor time) or natural time.

Recovery Time:

Recovery time is the time it takes for the system to start running again after a failure. It is desirable that the system start executing at the same level prior to stopping. In a printing environment recovery time is increased if a print job has to be restarted. In production printing environments where the goal is to run the presses at engine speed, long recovery times are expensive to the business. Establishing acceptable recovery time helps set the right expectation for the developers to meet the needs of the end customer.

Performance:

In the printing environment, Performance is expressed as the number of pages printed per minute. Performance is an important attribute especially when software is intended to support hardware. It is a potential candidate for a bottleneck and consequently should be monitored closely. Its impact on perceived quality can be large. For commercial markets it should be treated with the same importance as other core quality attributes. The authors have treated it as an element of reliability since testing constraints, job type and size apply to both Reliability and Performance.

Case Study

Measurements Techniques

We are building a second generation product using state of the art technology deploying formal measurement tools and methodology. We used the following methods to measure and gather the pertinent data for each attribute:

Installability: The data for instability was gathered by conducting an interview with the developer. Being the most intimately involved actor, the developer discussed characteristics of Installability defined in the attribute characteristics section. His knowledge and understanding of the installer design and development provided a reasonable assessment of Installability.

Usability and Learnability – We used standard measurement techniques as defined in the Attributes Characteristics and Measurements techniques.

Functionality: The application functionality was divided into features that were scheduled to be delivered in various iterations. Since there is only a single delivery to the customer, the entire functionality only comes together and is completely qualified towards the end. However, as features were developed and tested at the end of each iteration, each was counted as complete. This is a deviation from the ideal agile development where product is released to the customer at the end of each iteration.

The test execution was supported by a proprietary tool that stored all the test cases. The tool provided querying capabilities to gather data on passed, failed and blocked tests.

Reliability: The reliability data was gathered by conducting a well defined set of tests. The availability tests were started and monitored continuously by a test engineer. The operation continued over the weekends and both the uptime and downtime was determined to compute availability. The Recovery time was not an integral part of the Availability measurements and was recorded as discrete events. The unit of Performance was the number of pages that were ready to be printed in a minute. Since this was an assessment of the software only we did not consider the actual printing time. The Performance tests were run using simulators, not the actual hardware.

Results

The following discussion presents the state of quality attributes as measured along the development. The results are expressed as a percentage of ultimate goal for each attribute in an iteration. In the case of Usability and Learnability we have compared the two generations of the product due to the lack of availability of data. Measurements must be taken at regularly defined intervals to understand the progress of product development at a regular basis.

Installability

The chart below shows the current status of the Installability at the final iteration of the product, the point in time in the development when the product is ready formal qualification. The measurement is based upon the grading scale 1-5 (equivalent of academic grading system A-F), 5 being the ultimate goal. If desired it can also be expressed as a percentage of goal. Clearly, Install Area, Cancel, and Componentization elements of Installability still need improvement. The product was a conglomerate of various functionalities; consequently the Install Area attribute was difficult to achieve. Although Componentization is often not apparent to the customers, improving it often helps with satisfying the Install Area attribute. Cancel attribute impacts the customer experience and is high on the list of improvements. Overall we have done significantly better than our 1st generation product.

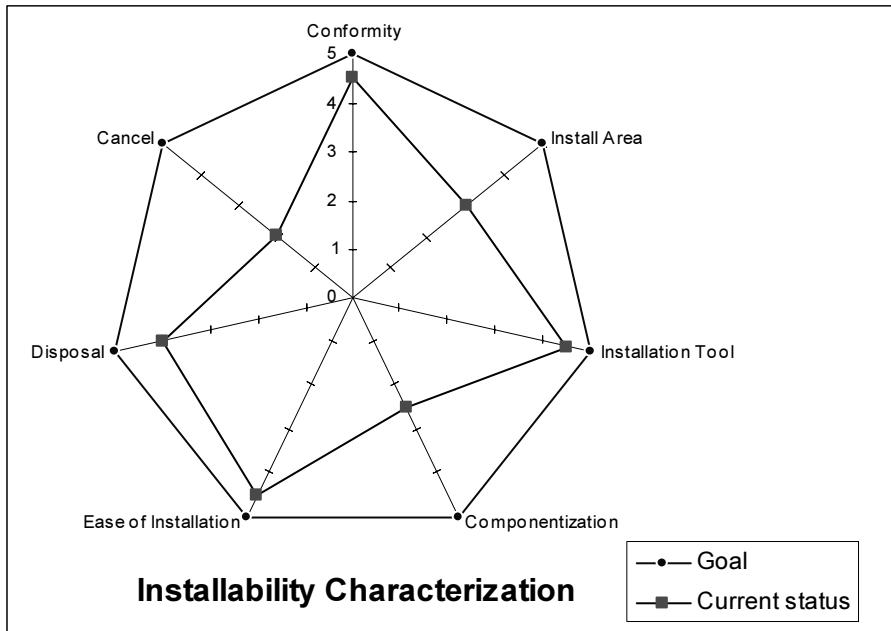


Figure 1. Installation Radar Chart using 5 Point Grading scale

Usability

We defined with partners inside and outside the lab a representative set of goals the customer should be able to reach using the product, and mapped refined those into task statements. We assessed the usability of the first generation product and the current product using the same methodology. User performance was measured on the five usability vectors defined above. A score for usability on each vector for each product was obtained by examining the average results for the representative set of tasks. The results are summarized below. Significant increases were found on each usability dimension. We attribute these changes in large part to the lab's increased attention to usability early and throughout the product development cycle. This early and continual focus on approaching development from a customer perspective did not achieve all that we had hoped but certainly moved the product in the right direction in terms of usability as this graph illustrates.

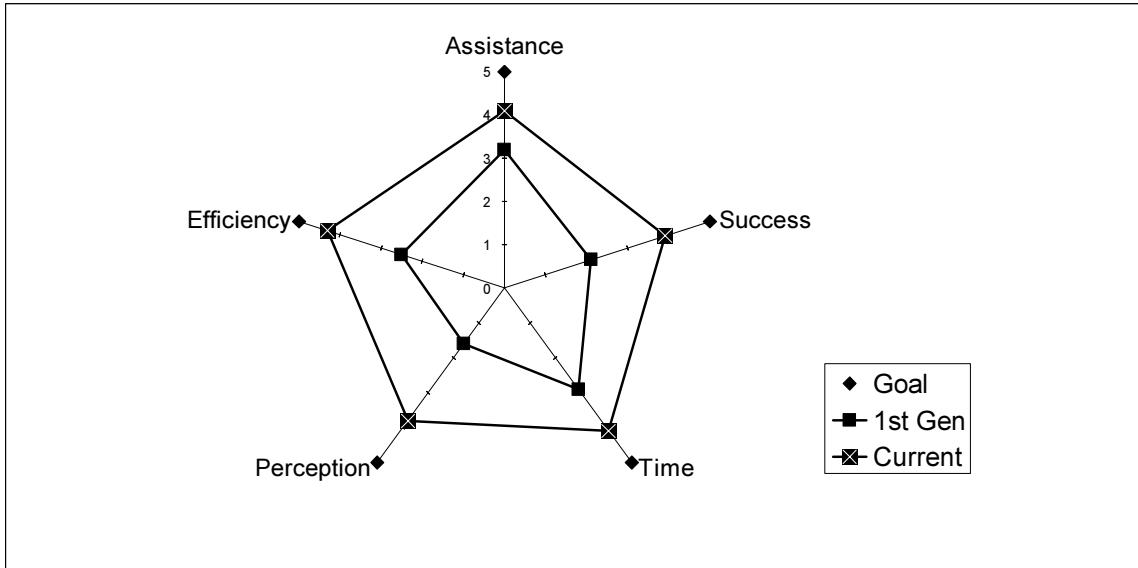


Figure 2. Usability Radar Chart for Two Generations of a Product

Learnability

Learnability was measured following a similar methodology as usability. The results were summarized to a single score for each product. The scale below maps to the 5 levels of learnability defined above: 5 = Intuitive, 4 = Discoverable, 3 = Learnable, 2 = Teachable and 1 = Possible. The learnability data may be examined in a number of different ways controlling different independent variables such as task, training level and level of previous experience beyond training. Further analysis was used to identify where improvement was needed. We found it useful to have a single score for communicating to a wider audience.

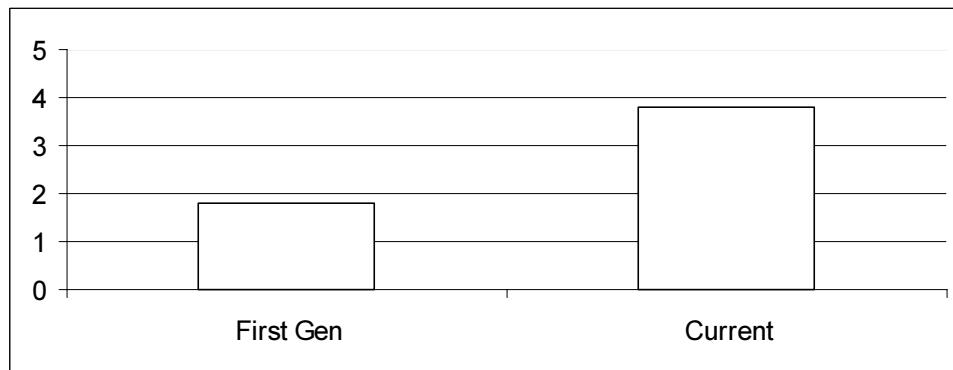


Figure 3. Learnability Comparison for Two Generations of a Product

Functionality

The functionality snapshot varies over the period of development. More and more features are complete as we move along the development especially in an agile environment with a single product release. The current Functionality status has been expressed as a percentage of ultimate goals. Figures 4 and 5 show the status at the end of the 2nd iteration and the 3rd (final) iteration. The percent defects represent the percentage of defects still open as measured against the goal. Only high severity defects have been considered here. As we get closer to the end of the

development, the number of open defects reflects a downward trend. It is possible that the development team decides to take an aggressive approach about driving down the open defects and set a new goal. If that happens both the goal and the status move towards the center of the polygon while other elements move away from the center. The reader should be aware that the percentage test development and execution refers to the tests that were needed to test the integrated functionality and not the overall testing efforts.

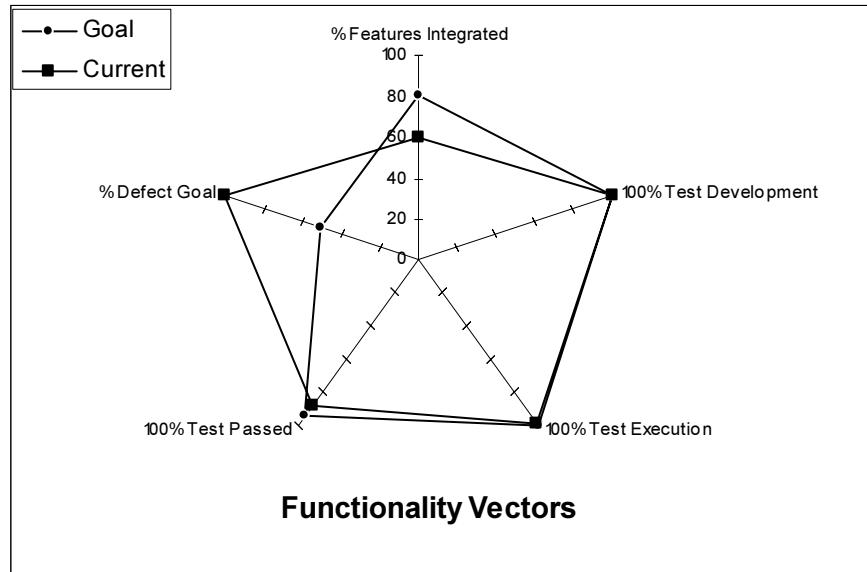


Figure 4. Functionality Status at Completion of 2nd Iteration

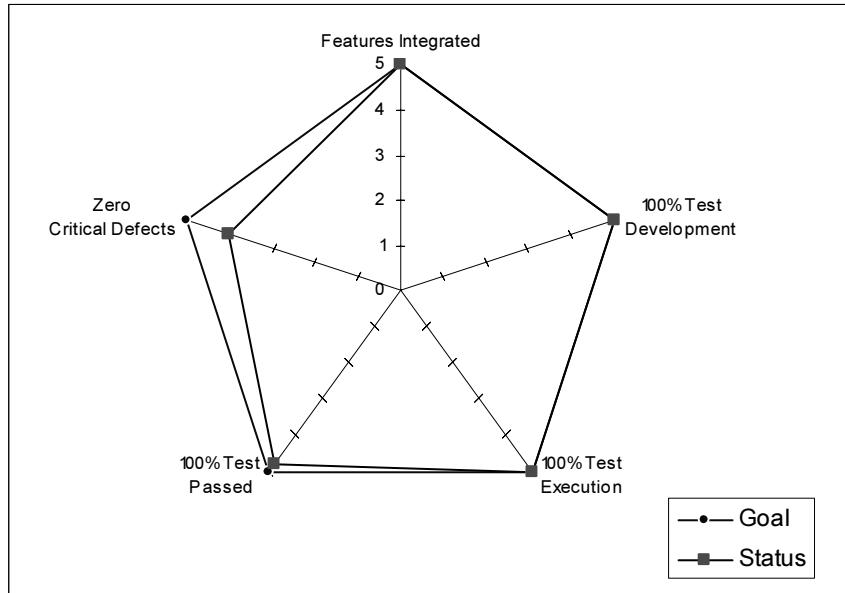


Figure 5. Functionality Status at the Completion of 3rd (final) Iteration

Reliability:

The measures we had collected in this study are Performance, Availability, and Recovery time. A radar chart of these attributes is shown below:

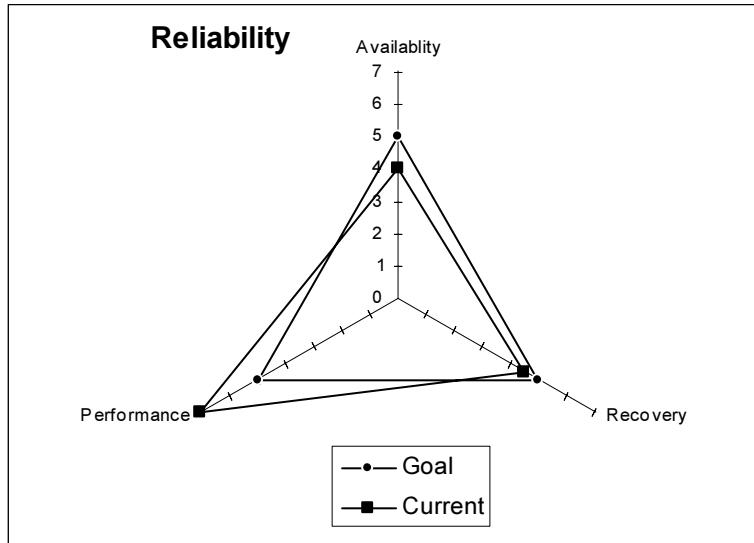


Figure 6. Reliability Status at the last Iteration

Compound Quality

To make significant decisions such as going to Beta and customer release, the management has to review the overall quality of the product taking into consideration the quality level of each attribute. Once the significant attributes of quality are established and measured, they can be distilled into a single value by normalizing and averaging them. It is also possible to derive a weighted average and generating a value if any particular attribute is more important than others.

The following table shows a snap shot of average values for each attribute for our product. The values have been normalized on a scale of 1-5, 5 being the desired value for each attribute. These values have been derived by normalizing the ranking and percentile scales.

Installability	Usability	Learnability	Functionality	Reliability
3.57	4	3.5	4.5	3.75

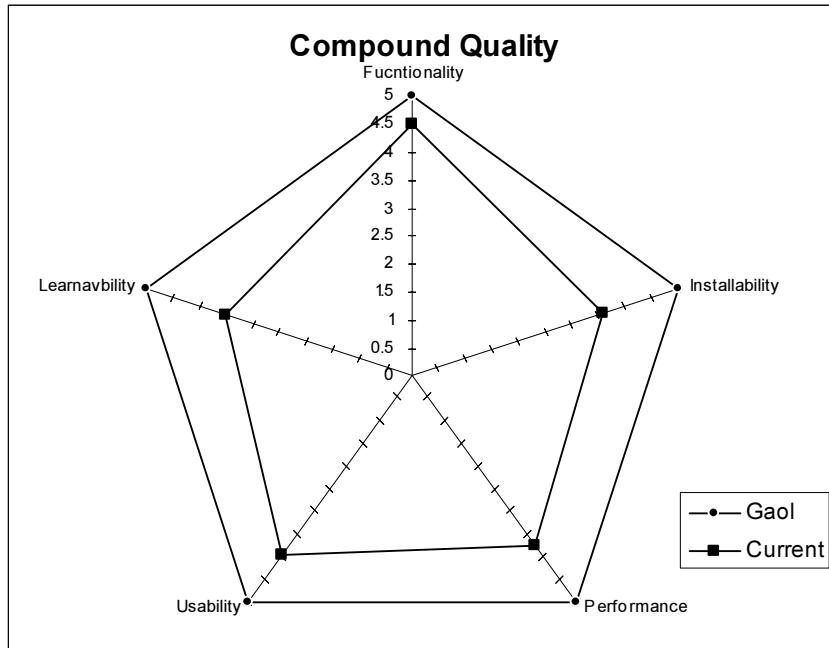


Figure 7. Compound Quality Snapshot

The management team can study each attribute with reference to its value to the customer and make an informed decision.

This concept can be further extended to a notion of quality index where we can associate an index to the overall quality. If each of these attributes is equally important, we have a quality index of 7.8 on a scale of 10. If we discriminate between the attributes and assign a weight to each attribute (Installability = 2.0, Learnability = 1.5, Usability = 2, Functionality = 2.5, Reliability = 2.0, a Total of 10), then our overall quality index will be 7.9. (Sum of weight x status for each attribute, for example Functionality = $4/5 * 2.5$). We observe a higher quality index due to the fact that functionality now has a higher contribution.

Conclusions:

Software quality has multiple facets that all play an important role in building products that provides high customer value. Not all attributes contribute at the same level to product quality. Management should evaluate the impact of each for the target market and users and choose the right set to help make better quality-based product decisions. Extending the definition of quality to include attributes other than just functionality can provide a shorter more manageable path to a high level of overall product quality. The measures discussed in this paper provide insight to many facets of software quality and indicate how these measures can support product decisions. Compound quality concepts, tools and measures can be used achieve to a high level of product quality, customer value and ultimately customer satisfaction.

References:

1. Grady, Robert B., Practical Software Metrics for Project Management and Process Improvement, PP. 32, Prentice Hall, 1992.

2. Kahn, Stephen H., Metrics and Models in Software Quality Engineering, Addison Wesley, 1994.
3. Gatlin, Manny and Hannon, Greg, Managing Software Quality, 18th Annual Pacific Northwest Quality Conference, Portland, Oregon, October 2000.
4. Simmons, Steve, Design for Installability, Proceedings of USENIX Applications Development, Symposium, April 25-28, 1994, Toronto, Ontario, Canada.
5. Lim, K.Y. and Long, J.B. (1992). Facilitating human factors input to system design. In Proceedings of the CHI '92 Conference (ACM, New York). 549-556.
6. Smith, S.L. and Mosier, J.N. (1986). Guidelines for Designing User Interface. Software. Bedford, MA: Mitre Corporation. Report 7 MTR-10090, ESD-TR-86-278.
7. Musa, John, Software Reliability Engineering, McGraw Hill, 1998.

Insight into Real Test-Driven Development

Peter Zimmerer
Siemens AG
Otto-Hahn-Ring 6
D-81739 Munich, Germany
+49 89 636 42155
peter.zimmerer@siemens.com

Peter Zimmerer is a Senior Principal Engineer at Siemens AG, Corporate Technology, in Munich, Germany. He studied Computer Science at the University of Stuttgart, Germany and received his M.Sc. degree (Diplominformatiker) in 1991. He is an ISTQB™ Certified Tester Full Advanced Level.

For more than 15 years he has been working in the field of software testing and quality engineering for object-oriented (C++, Java), distributed, component-based, and embedded software. He was also involved in the design and development of different Siemens in-house testing tools for component and integration testing.

At Siemens he performs consulting on testing strategies, testing methods, testing processes, test automation, and testing tools in real-world projects and is responsible for the research activities in this area. He is co-author of several journal and conference contributions and regular speaker at international testing conferences, e.g. at Conference on Quality Engineering in Software Technology (CONQUEST), SIGS-DATACOM OOP, GI-TAV, STEV Austria, Dr. Dobb's Software Development Best Practices, Conference on Testing Computer Software, Quality Week, Conference of the Association for Software Testing (CAST), PSQT/PSTT, QAI's Software Testing Conference, EuroSTAR, and STARWEST.

Abstract

Today, many people talk about test-driven development (TDD) and there is some hype to perform test-driven development in software projects. In this situation when many people tell you to do TDD it cannot be bad to ask the question: What is really behind TDD?

Test-driven development (TDD) is an approach to software construction in which developers write automated unit tests before writing code. These automated tests are rerun after changes to code. Proponents of the TDD approach assert that it delivers software that is easier to maintain and of higher quality than using traditional development approaches.

In this paper I share my view of TDD's advantages and limitations and how the TDD concept can be extended to all levels of testing. Based on experiences gained from real-world projects employing TDD, I explain how to use TDD practices to support preventive testing throughout the development process which results in new ways of cooperation between developers and testers. This approach helps us to see new aspects of test-driven development and to get a better understanding how it fits into the big picture of software testing and development overall.

Keywords

Test-driven development (TDD), test-first development, eXtreme Programming (XP), preventive testing

1. INTRODUCTION

To find a description of test-driven development (TDD) is not difficult today – there are even many books which contain TDD in their title. Kent Beck characterizes test-driven development (TDD) [1] by two simple rules

- never write a single line of code unless you have a failing automated test
- eliminate duplication

which practically results in performing the following three steps in development:

- write a test that fails
- write necessary code to pass the test
- refactor the code.

A similar description is given by Scott W. Ambler [2]:

Test-driven development (TDD) is an evolutionary approach to development which combines test-first development where you write a test before you write just enough production code to fulfill that test and refactoring.

TDD is also a core practice of eXtreme programming (XP). Among others it avoids the problem that creating unit tests after coding is often quite difficult because of the low testability of the implementation.

Some people (e.g. Kent Beck [3] and Ward Cunningham [4]) state that test-first development is not a testing technique but rather a way of specification, design, and development. Perhaps that is more a philosophical statement, but to me test-first development is testing because testing is not only test execution at the end, but it includes activities such as creating test requirements, test specifications, test design, and test cases.

After a short introduction of the typical usage of TDD I want to explain five lessons learned and some of the limitations of TDD based on my professional experience. These lessons are important to take into account, because as with any testing approach TDD is not a silver bullet. That is, using TDD requires adequate skill and judgment in the selection and application dependent on the context.

Understanding these lessons learned and limitations of TDD can result in a more comprehensive view of TDD. In addition, they can provide some guidelines on the implications of TDD and the ability to apply TDD to all levels of testing not only unit and acceptance testing.

2. TYPICAL USAGE OF TDD

When looking into practice we can see the following characteristic usages of TDD.

2.1 Unit testing

Typically unit testing is done by the programmer. This is supported by different testing frameworks and tools, particularly by the different xUnit tools [9] such as JUnit for Java [10] and NUnit for C# [11]. These frameworks and tools are quite popular in the industry.

2.2 Acceptance testing

Acceptance testing is supported by different testing frameworks and tools to bring customers, developers, and testers together. For example in the acceptance testing framework FIT (Framework for Integrated Test [12]) by Ward Cunningham and others, tests are specified as tables, and fixtures act as the glue between the written tests and the application's code. This is ideal for data-centric tests where each test does the same kind of thing to different kinds of data.

In addition, there is FitNesse [13], a fully integrated standalone wiki and acceptance testing framework based on FIT by Robert C. Martin and Micah D. Martin. There are also tools that focus on web testing over HTTP (see [9]).

These frameworks and tools are suitable for specific kinds of systems and domains (for example web and Java applications), but are not used in the industry so much up to now at least compared to the xUnit tools like JUnit or NUnit.

3. SOME BASICS OF TESTING

So far, we can see that TDD has a focus on unit testing and acceptance testing as well as a focus on the detailed implementation of tests. Now, look at some basics of testing which testers have known for many years but which are still worth to emphasize:

- Finding bugs during test execution is *not* the optimum. The goal must always be to find bugs early during test specification, not late during test execution!
- Testers experience every day that specification artifacts like requirements, use cases, models, architecture, and design must be corrected, improved, extended and completed for testing which results in changes. Thus during the creation of a test specification and test design which includes building a test model or defining abstract (i.e. non-executable, more high-level) test cases we can already find and prevent bugs. That means *creating a test specification is testing*. Furthermore, these designed tests represent a set of executable specifications.
- Design for testability (which can be shortly characterized as visibility/observability and control) must be built in from the start. That is not a private affair of the testers and will have benefits for the whole project team.

Altogether, these basic facts of testing lead to the following question: *Why should we restrict TDD to only unit and acceptance testing (see section 2) with executable test cases implemented in detail?*

3.1 Preventive testing

When looking back for many years we can see that the idea of TDD is not new. One origin of the TDD idea has been already given by Bill Hetzel [5], [6] more than 15 years ago by using the term preventive testing. He said:

Preventive Testing is built upon the observation that one of the most effective ways of specifying something is to describe (in detail) how you would accept (test) it if someone gave it to you.

The means if we use testing to discover, identify, create, specify, influence and control requirements, architecture, design, implementation, deployment, and maintenance artifacts then *testing really drives development* and then *creation of testware artifacts leads software development*.

That is, the idea of TDD is nothing new, but rather an old idea (see also reports and references given in [4]). What's new is that these things are used more and more often in projects today. That is the real big benefit of the TDD-hype brought to us by XP and agile methods.

4. LESSONS LEARNED

4.1 Lesson 1 –Preventive testing

These facts result in my first lessons learned of TDD motivated by the idea of preventive testing. I see TDD composed of three parts:

- test-first design
- test-first implementation and
- refactoring

It is very beneficial to use the TDD approach (test-first design) already to create, evolve, and improve the requirements, design, architecture, etc. This includes early creation of abstract test cases (i.e. describe the expected behavior as a test and specify how you would test a requirement or a high-level design) as well as executable test cases implemented in detail (test-first implementation). TDD is possible and highly recommended on every test level, not only for unit and acceptance testing, and it is driven by the idea of preventive testing! This emphasizes the importance and benefits of early testing activities like building a test model, creating abstract test cases which then represent a set of executable specifications, and proactive design for testability.

4.2 Lesson 2 –Test-first implementation

Looking in more detail on the implementation aspects of TDD, i.e., on test-first implementation, in real world projects sometimes things are not so easy to do in practice as described in some nice little demo examples. It is not that test-first implementation is impossible, but it can have high costs. This is especially important for example in the areas of testing GUIs, web applications, distributed objects on application servers, event-based reactive systems, and embedded systems.

For GUI testing there are also different testing frameworks and tools available (for example, JFCUnit [14] and Jemmy [15]) which in principle support the TDD approach for GUI applications as well.

Another approach is to divide the code into appropriate components that can be built, tested, and deployed separately. Most of the functionality (i.e. the business logic) is built outside the context of the user interface code (i.e. presentation layer) using TDD. And the user interface code is just a very thin layer on top of rigorously tested code. That is, as much functionality as possible is built outside the GUI. This good basic architectural style of a clear separation between business logic and presentation layer is not new and is already well known for a long time, for example, a 3-tier architecture as given in the figure aside and the model-view-controller (MVC) design pattern.

Presentation

Business Logic

Persistence

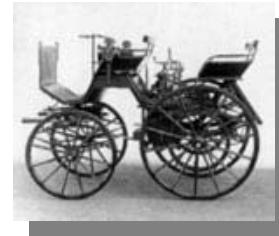
4.3 Lesson 3 –TDD and innovation

Much of the software we build today is innovative and may include some kind of invention as well. These circumstances can also make TDD more difficult to realize. To explain that I will use the following engineering example from history:

In 1886 the first car was invented by the Germans Carl Benz and Gottlieb Daimler, but only about 70 years later, Mercedes-Benz invented the first crash tests (which are systematically done since 1959).

But, what would *real* TDD mean for this example from the history in engineering? We can draw the following conclusion:

Using the TDD approach to full extent would mean that somebody first invents the crash test and then afterwards the car to pass this crash test.



This procedure is really difficult to do in practice. Rather I think that this is almost impossible to be the usual way of developing such a highly innovative product like a car!

Now, we can try to map this example from engineering to our software world. Innovations in software can appear on many different levels such as for example “some innovative lines of code”, “an innovative framework (especially the xUnit frameworks!)” up to “a new software technology”. At the same time it is very difficult to quantify the degree of innovation and invention in a specific piece of software and to determine the influence of the development method on the innovation.

On a more abstract level, when looking back in the history of software engineering we can see that in most cases (or always?) testing came after development and not vice versa. For example, innovative software technologies such as object-orientation, component-based software, web technologies, aspect-oriented programming, and grid computing were first envisioned; testing strategies, testing methods, and testing tools for them followed later.

Regarding architectural and design patterns which claim to contain innovative approved best practices, here again the patterns have been invented first (or some have been reinvented based on already known knowledge). Later people started to think about how to test a specific design pattern, i.e. also design patterns have not been developed in a core TDD manner.

And what's about all the innovative testing tools we get from the commercial testing tool industry? Do you think these tools are usually developed in a TDD manner? I don't ...

Altogether, we should not restrict ourselves in our own innovation and inventiveness by performing the TDD approach too rigorously and by expecting that we can always define the test first. To get innovation in software is not an easy task at all and starting first with a test for a really new thing is even more difficult because TDD adds some kind of indirectness and an additional level of abstraction. We need creativity in our testing and development work to make our software more innovative and to do not limit the next small or big steps in software evolution.

4.4 Lesson 4 –TDD and non-functional requirements

TDD should also be used to discover, identify, create, specify, influence and control non-functional requirements. Vague, low-quality requirements like “Our architecture shall be very flexible” are not testable. Instead, using test cases to create requirements and describe how you would test it (i.e. doing preventive testing) will result in finding bugs, improving the quality of the requirements and increasing testability. These test cases can be abstract and non-executable or concrete and implemented in detail dependent on the level of abstraction of the requirements. In the same way it is very helpful to develop performance requirements in a TDD manner

On the other side there are some characteristics of non-functional requirements which limit the usage of the TDD approach. For example regarding security most of the security bugs are not in the area of the intended and known behavior as well explained in Figure 1.

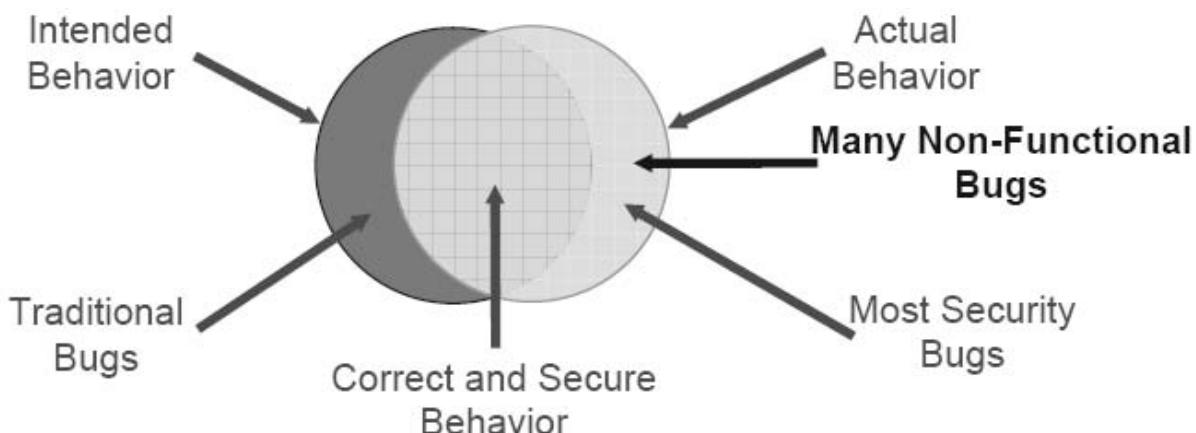


Figure 1: Intended vs. actual application behavior (see [7]).

But, how can a test be created upfront for an unintended, unknown behavior? This example shows some limitations of TDD in the area of non-functional requirements as well.

4.5 Lessons 5 –Cost efficiency and predictability

Testers often complain that they are involved too late in development projects. I completely agree with that, but, in a highly innovative, iterative, agile project there is the possibility that a lot of rework in test design and test implementation must be done because of continuously changing early requirements and architectural prototypes. So, it is important to find the right balance in the projects for that when using the TDD approach especially when doing a lot of test implementation work at the beginning.

Furthermore TDD is not sufficient in the real world because test cases created using a test-first approach or generated from (always incomplete) requirements and design are never enough and must be always extended and completed. We cannot predict everything, so we have to use approaches like exploratory testing (see [8]) as well. In addition, decisions made during implementation won't be well tested by tests exclusively created up front.

Altogether, doing test-first only is not enough but we additionally need some “afterwards” testing which I call *test-second*.

5. EXPERIENCES

In my experience TDD strongly increases visibility and importance of testing in the whole project. And as case studies have shown it delivers benefits concerning quality improvement and productivity (see [5]). Although TDD is often used in the context of an iterative/incremental, agile development process (so do I) it should not be restricted to it. That means TDD and the idea of preventive testing is useful and applicable even in an old-fashioned waterfall process. Figure 2 gives an example for a workflow visualizing TDD in an agile project using Scrum.

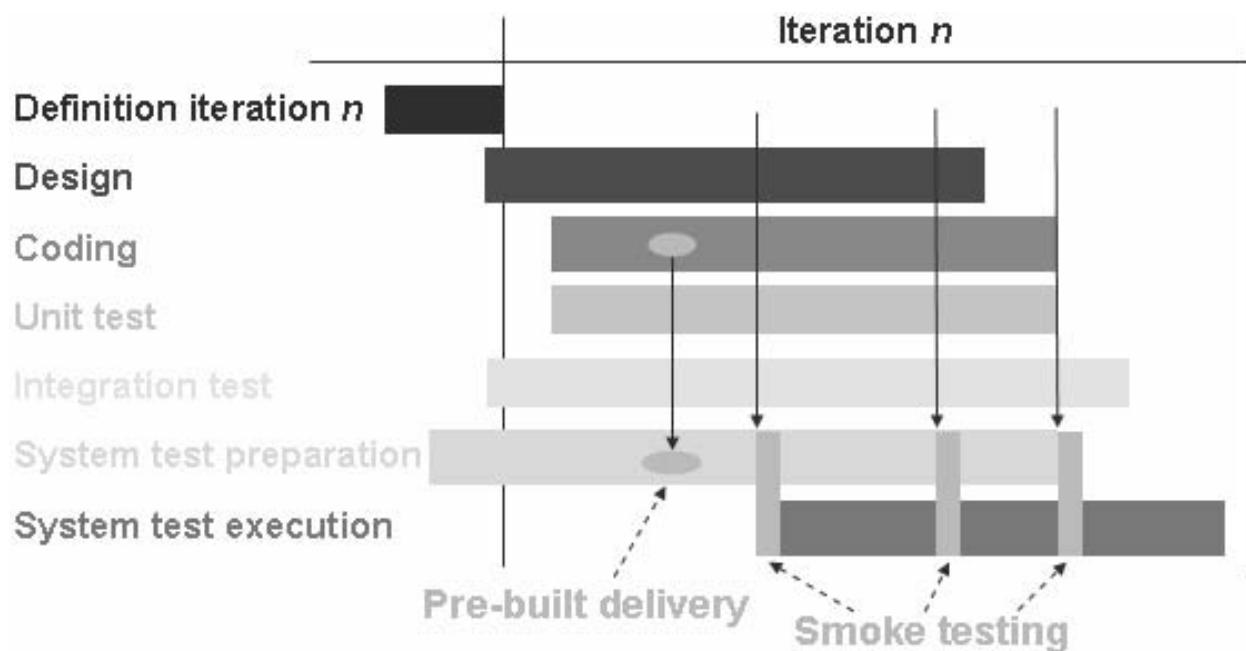


Figure 2: Workflow visualizing TDD in an agile project

TDD needs changes in the development concerning process, people (including management!), and tooling. And TDD results in a closer cooperation between testers and developers, which produces big benefit on its own.

When doing TDD sometimes it is not easy to find out how much of the implemented test code is really developed in a TDD way or which is developed more conventionally "afterwards". I myself was confronted with this issue in a project when the developers asked me: "How do I test private member functions?" At first, from a technical point of view there are different answers and possibilities dependent on the used programming language (for example friends in C++, reflection in Java). But when I thought about that in more detail I came to the following interesting conclusion (which might be a bit too strict in general, but ...):

In core TDD this question is not allowed, i.e. it does not make sense, because in TDD we first have the designed and implemented test and then do some implementation to pass this test. I.e. the details of the implementation do not matter so much.

Here, I recommend using these kinds of indicators to see whether and to what extent the TDD approach is really used in a project.

6. SUMMARY: TEST-FIRST + *TEST-SECOND*

To summarize my understanding of TDD and the different lessons learned and limitations of TDD which I have described in this paper I recommend taking the following items into account when realizing and enhancing your own TDD approach:

- TDD = Test-first design + test-first implementation + refactoring
- Do not miss the benefits of the more conceptual usage of the TDD approach during creation and improvement of requirements, design, architecture, etc.
- TDD is possible and strictly recommended on every test level, not only for unit and acceptance testing (preventive testing): let testing drive your development and maintenance at all!
- The right project specific balance is the key for cost efficiency.
- TDD needs changes in development: process, people, and tooling.
- TDD results in a closer cooperation of testers and developers.
- TDD is neither 100% possible nor sufficient in real-world projects.
- TDD does not completely replace conventional afterwards software testing: so, do test-first as well as *test-second*.

Apart from this, if you have not already used and practiced TDD then I highly recommend to start with TDD tomorrow! You will see you will deliver better software with higher quality, you will have less pain in your projects, and in the mid-term you will also speed up your project! Furthermore be aware of some circumstances and constraints which require a flexible and adjustable testing approach. But, to select and adapt your own approach that's for what you are paid in your job as a test engineer, test lead, test manager or test consultant. Good luck!

7. REFERENCES

- [1] Kent Beck: Test-Driven Development by Example, Addison-Wesley Professional, 2002
- [2] Scott W. Ambler: Introduction to Test Driven Development (TDD),
<http://www.agiledata.org/essays/tdd.html>
- [3] Kent Beck: Aim, Fire, IEEE Software, Volume 18, Issue 5, Sep/Oct 2001, pp. 87-89
- [4] D. Janzen and H. Saiedian,
Test-Driven Development: Concepts, Taxonomy, and Future Direction, IEEE Computer, Vol. 38, No. 9, September 2005, pp. 43-50
- [5] R. Craig, S.P. Jaskiel: Systematic Software Testing, Artech House, 2002
- [6] D. Gelperin, B. Hetzel, The Growth of Software Testing,
Communications of the ACM, Vol. 31, Issue 6, June 1988, pp. 687-695
- [7] James A. Whittaker, Herbert H. Thompson, Herbert Thompson: How to Break Software Security, Addison Wesley, 2003; <http://www.sisecure.com/chart.htm>
- [8] Exploratory testing: Cem Kaner <http://www.kaner.com/articles.html>, James Bach <http://www.satisfice.com/>; Brian Marick <http://www.testingcraft.com/exploratory.html>
- [9] XProgramming.com: <http://www.xprogramming.com/software.htm/>
- [10] JUnit: <http://www.junit.org/>
- [11] NUnit: <http://www.nunit.org/>
- [12] Framework for Integrated Test (Fit): <http://fit.c2.com/>
- [13] Fitnesse: <http://fitnesse.org/>
- [14] JFCUnit: <http://jfcunit.sourceforge.net/>
- [15] Jemmy: <http://jemmy.netbeans.org/>
- [16] Testdriven.com: <http://www.testdriven.com/>

Front End Requirements Traceability for Small Systems

Robert F. Roggio
Professor

Department of Computer and Information Sciences
University of North Florida
Jacksonville, FL 32224
broggio@unf.edu

Abstract

One of the most controversial topics software development practitioners debate is the area of requirements traceability. While there are some who question the value and the cost of requirements traceability in application development, most practitioners assert that effective traceability of requirements can lead to a higher quality application that is more likely to meet or exceed stakeholder expectations. Yet, for many the salient issue is exactly how do we effectively trace requirements that is both low in cost, high in value, and easy to implement? Simply put, how much effort is needed? What is an effective mechanism to support traceability? This paper presents a case study using experiences and data captured from a real application development undertaken by two teams of graduate software engineering students. These teams were charged with the specification, design, development, and implementation of a non-trivial medical information system envisioned to support a small medical clinic whose mission is to serve the working, medically un-insured in Jacksonville, Florida, a metropolitan community of about 1.2 million people. The Volunteers in Medicine (VIM) [5] pro bono project was undertaken in response to a pressing need by this volunteer group. While the functional requirements were ever-changing, the non-functional requirements, as the VIM is staffed with almost all volunteers many of whom are senior citizens facing an automated system designed to replace accustomed hard copy record keeping, scheduling and reporting. Each team identified a single individual tasked with ensuring functional requirements traceability, which was part of each of the eleven deliverables. This successful approach uses forward and backward traceability matrices starting with careful elicitation and capture of Needs, the filtering of Needs to Features, and the elaboration of Features to Use Cases – all one to many relationships. Traceability was extended to include an Analysis Model. The paper discusses traceability beyond the analysis model extending into design. It is also pointed out that the traceability matrix approach can be readily applied to tracing features to non-functional requirements as well as tracing both functional and non-functional requirements to specific test scenarios.

1. Introduction to Traceability

In order to discuss traceability issues in particular, it is important to build a common framework, and this can be done by establishing accepted definitions related to traceability in the context of this article.

1.1 What is Traceability?

Traceability may be defined a number of ways, but in the context of requirements traceability (RT) it may be defined as “the ability to describe and follow the life of a requirement, in both a forward and backward direction from elicitation of stakeholder needs through to deployment of the application. This tracing includes capturing and modeling the requirements via specifications, through analysis, design, testing, through periods of refinement and change, and ultimately to implementation. RT involves tracing the life of a requirement; that is, “...something that a computer application must do for its users. It is a specific function, feature, quality, or principle that the system must provide in order for it to merit its existence.” [2]

1.2 Why Do We Undertake Requirements Traceability?

Scott Ambler, [3], presents some compelling arguments for the undertaking of requirements traceability. He addresses the motivation to do so from two perspectives: (1) from a software engineering perspective, and (2) a business perspective. The benefits he outlines from a software development point of view appear to be abundantly clear and offer significant benefits to

all stakeholders, while the potential benefits attributable to a business perspective may be more arguable, hence the controversy surrounding requirements traceability.

From the software development perspective, requirements traceability can significantly assist in ensuring we are developing software that the company actually needs. Traceability assists us in aligning changing business needs with the software the organization has developed. We want to ensure our development people are indeed developing software reflective of current needs. Secondly, requirements traceability assists the company in constantly capturing and documenting current business knowledge that could be lost through employee attrition. By having current sets of requirements, the impact of proposed changes are more readily assessed. With a number of changes proposed, a company may choose to invest in the changes that most favorably provide a better payoff with reduced risk and effort. And, of course, requirements that are traceable assist in understanding the company's development process itself and perhaps to identify areas of potential process improvement.

From the business perspective, however, the benefits are oftentimes questionable. Perhaps the most significant benefit is that many contracts now require companies to provide evidence of requirements traceability. While this may be oftentimes satisfied in a very haphazard manner and may involve square-filling, the results are usually not designed to improve the software itself or the process itself. As Ambler points out, "...and developers are often completely soured on the concept of requirements traceability." [4] Further, traceability is also an apparent defensive one - to cover oneself from liability lawsuits or criticism. Thus, while producing requirements traceability may satisfy these immediate and sometimes legal goals, it is less than glamorous to work for a company that takes measures to protect itself by undertaking traceability exercises specifically for protection.

2. The Volunteers in Medicine (VIM) Project

2.1 Project Description.

This project started in August 2005 with two teams of graduate computer and information science students. The teams were formed were charged in defining, specifying, designing, and implementing a new medical information system for VIM, an organization in Jacksonville that provides pro bono services to the working uninsured. Almost everyone associated with the VIM is a volunteer: they have doctors, nurses, nurse practitioners, and a host of other office-related volunteers. Much of their funding is from philanthropy and state or government grants based on usage statistics on office visits, patient needs, logged appointments, laboratory equipment needs, etc. The developed application supports all volunteer information, patient information, and provider information with a user-friendly, learnable interface designed to support VIM operations. While the project was completed in May 2006, four graduate students wished to extend the project into the summer in order to add enhancements and other features that exceeded the original scope of this application. This effort is now ongoing.

From a practical perspective, this effort continues to offer a tremendous opportunity for students to gain very real life experiences with a number of the realities of software development, such as the management of requirements, change management, working in a team, working with state of the art software tools and methodology, all with the motivation that – unlike many academic software development projects – this one will be used and used heavily. A somewhat unique challenge to young college-age students is the interface, and significant emphasis was placed on this component of the application. Most users of this application are senior citizens and accustomed to paper and pencil and in some cases eschew computers and modern technologies. So, of particular importance was the design of a non-intimidating, user-friendly, very clear interface - with additional on-screen notes to assure end-users that all is well.

2.2 Team Organization and Methodology.

While the two teams of eight students used the IBM-Rational suite of tools (namely Rational Rose and Requisite Pro), it is arguable that a different methodology could have been used.

Nevertheless, this ‘lighter’ heavy-weight methodology was selected to support development, as there is widespread use of the Rational Unified Process (RUP) ® in the Jacksonville, FL area. The two teams selected their own project manager and attempted to apportion specific roles to various members, such as use-case specifier, tester, software architect, database designer, and more. Lastly, each team received the same requirements via interviews, emails, and personal contacts. But each team proceeded independently into their own design and implementation.

Two students had previous exposure to the RUP and software development in previous academic work, so each were assigned to different teams to provide for project oversight such as the development of an iteration plan, serving as a resource for Rose and Requisite Pro, and, most appropriately, to be responsible for all requirements traceability in the project.

3. Needs, Features, and Use-Cases

According to Laganiere and Lethbridge [5], a requirement is a statement about what a proposed system will do that all stakeholders agree must be made true for the customer’s problem to be adequately solved. Further, these requirements may be typically captured as functional and non-functional requirements together constituting what is often called a Software Requirements Specification (SRS). Functional requirements are the services directly provided to the users that are provided by the system and answers questions such as what does the system do for me, what values does it provide to me, and how does it help me do my job. Non-functional requirements, not at all less important, are oftentimes considered ‘constraints’ or sometimes ‘quality metrics’ associated with a system. Failure to satisfy these requirements can completely render a system useless. Satisfying these requirements often limit resources that can be used, bound aspects of quality, restrict design decisions, and constrain development environments. These often impact efficiency, performance, maintainability, portability, scalability, distribution, security, legacy dependencies, and other practical design and implementation considerations.

While the overall quality of applications developed appears to be improving with somewhat fewer industry-wide cases of development failure, it appears additional emphasis continues to be focused on the management of requirements, which most practitioners feel is the most fundamental root cause of software failure. Most senior practitioners tenaciously assert that development problems are less likely to arise from selection of a language, methodology, or environment, or other tools than in getting the right requirements and getting the requirements right. Knowing that the functional requirements and non-functional requirements are correctly captured and have been accurately traced through a series of requirements artifacts (abstract to concrete) and that the specifications do, in fact, represent stakeholder needs goes a long way in ensuring that the correct application is developed and deployed.

Leffingwell and Widrig [7] suggest this notion of requirements traceability oftentimes centers on cost and return on investment. Where applications simply cannot fail (business-critical, medical-critical, safety-critical applications) developers are less apt to argue the need for traceability. Where software failure will not result in loss of life, property, or other major assets, the need and expenses associated with requirements traceability may be argued well into the future. .

More and more developers now openly subscribe to a traceability approach that starts with identifying and capturing needs, mapping these stakeholder needs into application features, and then mapping these features into use-cases with a strict traceability exercise in each mapping. Thus, we are asserting tracing requirements artifacts to other requirements artifacts in an effort to develop a comprehensive picture of requirements understandable by all stakeholders.

Initial stakeholder needs tend to oftentimes be very abstract and may include more than just desired features of a proposed application. For example, a need might be: “The system will allow students to register for courses and change their registration as simply and rapidly as possible. It will help students achieve their personal goals of obtaining their degree in the

shortest reasonable time while taking courses that they find most interesting and fulfilling.” [6] As is clear, this statement of need may be quite meaningful to one stakeholder, but it is quite ‘high-level’ and abstract.

Features are usually considered the functional requirements. However, they too may oftentimes remain abstract, not have any context, and be horribly boring to read. These may include statements of inputs, desired outputs, statements regarding timing, quality, and data. These are often expressed as, “The system shall..... The system shall....” Naturally, these are often supplemented with tables, flowcharts, possibly screen shots, examples of computations, formulas, and more. But features are mapped to from stakeholder needs. Similarly, features must be traceable back to stakeholder needs.

In [7], Dean Leffingwell and Don Widrig claim that different kinds of projects produce different kinds of requirements artifacts. These artifacts can be organized and managed in a number of ways. But regardless of the type of project, a model that traces requirements may be represented in Figure 1, as shown below.

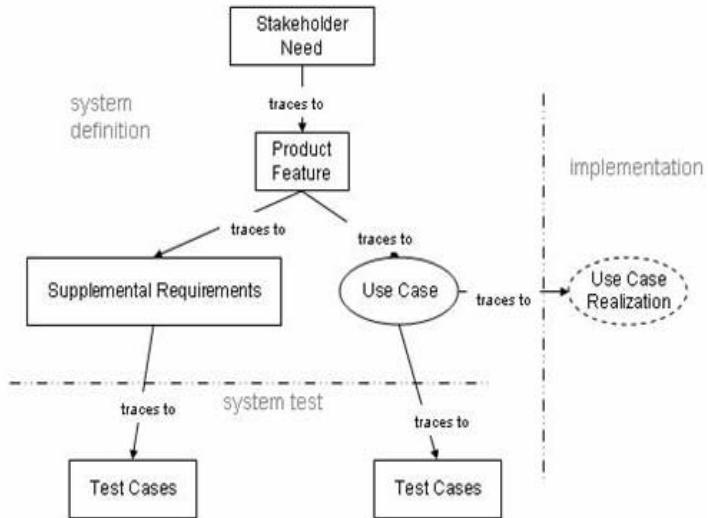


Figure 1. Overall Traceability Flow [7]

The call for software development usually arises from stakeholder needs, where a stakeholder may be anyone associated or having some kind of vested interest with a potential application. But ‘needs’ tend to be quite abstract and may well include desires that are not intended / cannot be accommodated by a system. Nevertheless, the captured needs must be mapped into features that can be accommodated by a system. Since a comprehensive set of needs is critical to support specification, design, and implementation, great care must be exercised so that all appropriate needs map to features that can be satisfied by the system. This is only the first part.

4. Mapping Using Traceability Matrices

While there is a huge number of traceability tools available to assist in a variety of traceability exercises [4], Leffingwell [7] suggests simple matrices for tracing artifacts starting with stakeholder needs. The model shows the tracing of requirements from the problem space into a solution space; from requirements to design and implementation.

Leffingwell and Widrig go on to suggest similar mechanisms for backwards traceability (features to needs) and other scenarios, such as features to use-cases (forward and backward) and more. We selected this mechanism to support traceability efforts in the VIM project.

4.1 Needs to Features

Using a number of personal interviews with the business manager and several different end-users of the application, the business needs were captured and were encapsulated into the following traceability matrix:

ID	NEED	FORWARD TRACEABILITY
N1	Record Repository	F1, F2, F5, F32
N2	Scheduling	F6, F7, F9, F11, F12, F13, F31
N3	Tracking and Report	F14, F15, F16, F17, F18, F20, F22, F32
N4	Remote Access	F23
N5	Provider Access to Patient Records	F 24, F25
N6	Mass Communication	F26
N7	Central Control for Setting Up Authorized Access	F33
N8	Protection Against Unauthorized Access	F29, F30

Figure 2. The Basic Business Needs for the VIM Organization

Needs were simply numbered N1 through N8 with a textual expansion in the second column. While only a several-word description is shown in the Need column, the analysis of each need and the resulting required system features are shown in the Forward Traceability column. Again, the Features are merely numbered in ascending order. As might be expected, each user need translated into one or more system features – the real requirements, albeit abstract, of the application to be developed.

In order to verify that all Needs are mapped to one or more Features and that each Feature is a requirement based on at least one need, it was essential to develop and backward traceability matrix, as shown in Figure 3.

ID	FEATURE	BACKWARD TRACEABILITY
F1	The system will allow the user to add, delete, and update patient records	N1
F2	The system will allow the user to add, delete, and update volunteer records	N1
F5	The system shall enable approval of personal profile changes by the business administrator of volunteer coordinator	N1
F6	The system shall enable the insertion and deletion of volunteers into the schedule	N2
F7	The system shall enable the insertion of deletion of providers into the schedule	N2
F9	The system shall enable the insertion of patient qualifying and examination appointments into the work schedule	N2
F11	From the schedule, the user will be able to access information about a particular appointment. This in	N2
F12	The system shall allow the rescheduling of appointments missed by a patient	N2
F13	The system will allow the scheduling of grant timeframes and associated proposal and reporting due dates	N2
F14	The system shall enable the business director to create statistical reports based on patient race.	N3

ID	FEATURE	BACKWARD TRACEABILITY
F15	The system shall enable the business director to create statistical reports based on patient age.	N3
F16	The system shall enable the clinical director to create statistical reports based on patient occupation.	N3
F17	The system shall enable the business director to create statistical reports based on provider and volunteer hours. These reports will assign a dollar value of these hours based on provider and volunteer status	N3
F18	The system shall enable the business director to create statistical reports based on patient diagnosis	N3
F20	The system shall track the hours of individual volunteers and providers based on their status (nurse practitioner, doctor, UNF student)	N3
F22	The system shall track the dates and amounts of donations	N3
F23	The system shall allow remote access to VIM-Jax information resources to authorized individuals.	N4
F24	The system shall allow providers the ability to search for patient records	N5
F25	The system shall allow providers the ability to update patient records. This includes recording diagnosis and treatment of patients.	N5
F26	The system shall allow the distribution of information to a selected group of stakeholders via email.	N6
F29	The system shall authenticate users for certain parts of the application based on user ID	N8
F30	The system shall not allow users to access system resources until the user has "logged in".	N8
F31	The system shall enable the insertion of patient qualifying appointments into the work schedule	N2
F32	Maintain chronic medical conditions treatment. The system will allow an authorized user system administration capabilities such as creating and deleting users and reset forgotten user passwords.	N1, N3
F33		N7

Figure 3. Backward Traceability Matrix for VIM Project

The team found this exercise essential, for features that could not be mapped back to user-needs were very suspect and resulted in additional inquiry and verification. All known need-feature relationships were recorded. Similarly, it was noted that some features seemed to satisfy more than one need. The bottom line is that the development team did not want identified Needs to have required Features that were not captured nor to have developed Features that were not reverse traceable to specific Need(s). Because there was not a large number of needs, the effort here was not substantial.

Features represent the functional requirements and continue to be captured in the accustomed boring manner as, “the system shall...” or “the system must be able to”. This format has been one of the many motivations for the use of use-cases. [2] Now as important as it is to trace needs to features, it is equally important to trace features to use-cases. Use-cases represent a

different and oftentimes much clearer requirements artifact that can be used to drive an entire development process.

4.2 Features to Use-Cases

It is important to stress that a number of modern processes are use-case driven and not merely the RUP. But since the RUP was the underlying methodology in the VIM project, the development of the use-case specification was essential. Use-cases support follow-on analysis activities in support of developing an analysis model, support the design model via use-case realizations, support the development of the interface prototype, support the development of a test strategy, support end-user demonstrations, and much more. Given this backdrop, then, the team proceeded to carefully examine the requirements captured as system features to develop the use case model. A use-case index citing use-case names and descriptive data followed by the features-use-case traceability matrix are shown respectively in Figures 5 and 6.

USE CASE NO:	TITLE	ACTORS	LEVEL MATURITY	DESCRIPTION	LAST MOD
<u>UC-01</u>	Schedule Patients	Clinical Director, Volunteer, Relational Data Base Management System (RDBMS)	Façade	This use case is started by the Clinical Director or a Volunteer and it allows them to schedule patient appointments.	10/21/05
<u>UC-02</u>	Schedule Volunteers	Volunteer Coordinator, RDBMS	Façade	This use case is started by the Volunteer Coordinator to schedule volunteer work hours.	10/21/05
<u>UC-03</u>	Schedule Qualification Consultation	Clinical Director, Volunteer, RDBMS	Façade	This use case is started by the Clinical Director or a Volunteer to schedule a <i>qualification consultation</i> appointment.	10/21/05
<u>UC-04</u>	Maintain Patient Records	Office Coordinator, Volunteer Coordinator, Volunteer, RDBMS	Façade	This use case is started by either, the Office Coordinator, Volunteer Coordinator or a Volunteer to create, update, and delete patient records.	10/22/05
<u>UC-05</u>	Maintain Volunteer Records	Volunteer Coordinator, RDBMS	Façade	This use case is started by the Volunteer Coordinator and it allows her to edit volunteer records.	10/21/05
<u>UC-06</u>	Research Statistics	Business Administrator, RDBMS	Façade	This use case is triggered by the Business Administrator to look up, compile and print out organizational statistics.	10/21/05

USE CASE NO:	TITLE	ACTORS	LEVEL MATURITY	DESCRIPTION	LAST MOD
<u>UC-07</u>	Send Group Email	Business Administrator, RDBMS	Façade	This use case is started by the Business Administrator and it allows him to mass email all the volunteers in the database.	10/21/05
<u>UC-08</u>	Authorize User	All Users of the System, RDBMS	Façade	This use case is started by any user of the system and it allows them to gain access on site or remotely, and check authentication.	10/22/05
<u>UC-09</u>	Perform Administrative Tasks	Business Administrator, RDBMS	Façade	This use case is started by the Business Administrator and it allows him to create and delete users, and to reset user passwords.	10/22/05

Figure 4. Use-Case Index – VIM Project

ID	FEATURE	FORWARD TRACEABILITY
F1	The system will allow the user to add, delete, and update patient records	UC-04
F2	The system will allow the user to add, delete, and update volunteer records	UC-05
F5	The system shall enable approval of personal profile changes by the business administrator or volunteer coordinator	UC-04, UC-05
F6	The system shall enable the insertion or deletion of volunteers into the schedule	UC-02
F7	The system shall enable the insertion or deletion of providers into the schedule	UC-02
F9	The system shall enable the insertion of patient examination appointments into the work schedule	UC-01
F11	From the schedule, the user will be able to access information about a particular appointment. This in	UC-01
F12	The system shall allow the rescheduling of appointments missed by a patient	UC-01
F13	The system will allow the scheduling of grant timeframes and associated proposal and reporting due dates	UC-06
F14	The system shall enable the business director to create statistical reports based on patient race.	UC-06

ID	FEATURE	FORWARD TRACEABILITY
F15	The system shall enable the business director to create statistical reports based on patient age.	UC-06
F16	The system shall enable the clinical director to create statistical reports based on patient occupation.	UC-06
F17	The system shall enable the business director to create statistical reports based on provider and volunteer hours. These reports will assign a dollar value of these hours based on provider and volunteer status	UC-06
F18	The system shall enable the business director to create statistical reports based on patient diagnosis	UC-06
F20	The system shall track the hours of individual volunteers and providers based on their status (nurse practitioner, doctor, UNF student)	UC-06
F22	The system shall track the dates and amounts of donations	UC-06
F23	The system shall allow remote access to VIM-Jax information resources to authorized individuals.	UC-09
F24	The system shall allow providers the ability to search for patient records	UC-04
F25	The system shall allow providers the ability to update patient records. This includes recording diagnosis and treatment of patients.	UC-04
F26	The system shall allow the distribution of information to a selected group of stakeholders via email.	UC-07
F29	The system shall authenticate users for certain parts of the application based on user ID	UC-08
F30	The system shall not allow users to access system resources until the user has "clocked" in.	UC-08
F31	The system shall enable the insertion of patient qualifying appointments into the work schedule	UC-03
F32	Maintain chronic medical conditions treatment.	UC-04
F33	The system will allow an authorized user system administration capabilities such as creating and deleting users and reset forgotten user passwords.	UC-09

Figure 5. Features to Use-Case - Forward Traceability Matrix - VIM Project

And, of course, equally important, the backward traceability mapping matrix:

ID	USECASE	BACKWARD TRACEABILITY
UC-01	This use case is started by the Clinical Director or a Volunteer and it allows them to schedule patient appointments.	F9, F11, F12
UC-02	This use case is started by the Clinical Director or Volunteer Coordinator to schedule volunteer work hours.	F6, F7
UC-03	This use case is started by the Clinical Director or a Volunteer to schedule a <i>qualification consultation</i> appointment.	F31
UC-04	This use case is started by either, the Office Coordinator, Volunteer Coordinator or a Volunteer to create, update, and delete patient records.	F1, F5, F24, F25, F32
UC-05	This use case is started by the Volunteer Coordinator and it allows her to edit volunteer records.	F2, F5
UC-06	This use case is triggered by the Business Administrator to look up, compile and print out organizational statistics.	F13, F14, F15, F16, F17, F18, F19, F20
UC-07	This use case is started by the Business Administrator and it allows him to mass email all the volunteers in the database.	F26
UC-08	This use case is started by any user of the system and it allows them to gain access on site or remotely, and check authentication.	F29, F30
UC-09	This use case is started by the Business Administrator and it allows him to create and delete users, and to reset user passwords.	F33

Figure 6. Use-Case to Features - Backward Traceability Matrix – VIM Project

What is also important to note in this approach, as is true for the needs to features traceability matrix, is the multiplicity between relationships. Although features are indeed statements of requirements, they tend to be rather abstract. Use cases consist of requirements communicated via stories of end-user – system interactions. Thus, they tend to be much more understandable and specific. Because of this, we often have a 1:n relationship between features and use-cases that capture the same ‘requirements.’ Although some practitioners feel that the ‘n’ may be one or at least a very small integer, empirical data from the VIM project indicates that a use-case may implement more than a single feature.

In retrospect, it is also an imperative that as we construct and examine the use-cases, we ensure that as developers we have gleaned all the meaning out of the features and have appropriately captured those requirements in the use-case scenarios, albeit as a more concrete level of detail. Use-Cases were then run back to the stakeholders to ensure no required features were omitted or inappropriately mapped into the use-case specification.

4.3 Use-Cases to Analysis Classes

Given a set of use-cases that all stakeholders felt captured the essential features of the desired application, the development team proceeded to undertake use-case analysis, wherein the use-case specifications are mapped into ‘areas of concern.’ In particular, the use-case analysis undertaken by the VIM development teams needed to create a small model that separates interfacing, control, and business entity concerns from each other. Recognizing that the development of such an Analysis Model does not directly morph into a Design Model with implementation classes, the exercise nevertheless provides class diagrams and interaction diagrams using analysis artifacts. This development greatly assisted in the follow-on design

model, as team members could then concentrate on their areas of the software architecture, such as those artifacts necessary to implement a presentation layer; those concerned with the application domain itself; those concerned with the business domain; those addressing technical services, etc. The resulting development of analysis classes is shown in Figure 8.

ID	USECASE	CLASS TYPE	FORWARD TRACEABILITY
UC-01	Schedule Patients: This use case is started by the Clinical Director or a Volunteer and it allows them to schedule patient appointments.	BOUNDARY: VolunteerHome AppointmentCalender AppointmentScheduleOptions AppointmentOptions AppointmentDetails RDBMSBoundary CONTROL: SchedulePatientsControl ENTITY: Appointment Providers PatientRecord Scheduel	VolunteerHome AppointmentCalender AppointmentScheduleOptions AppointmentOptions AppointmentDetails RDBMSBoundary SchedulePatientsControl Appointment Providers PatientRecord Scheduel
UC-02	Schedule Volunteers This use case is started by the Clinical Director or Volunteer Coordinator to schedule volunteer work hours.	BOUNDARY: VolunteerCoordinatorHome AppointmentScheduleOptions WorkCalendar WorkScheduleOptions WorkScheduleDetail RDBMSBoundary CONTROL: ScheduleVolunteerHoursControl ENTITY: User Schedule VolunteerHome AppointmentCalendar AppointmentOptions	VolunteerCoordinatorHome AppointmentScheduleOptions WorkCalendar WorkScheduleOptions WorkScheduleDetail RDBMSBoundary ScheduleVolunteerHoursControl User Schedule VolunteerHome AppointmentCalendar AppointmentOptions
UC-03	Schedule Qualification Consultation This use case is started by the Clinical Director or a Volunteer to schedule a <i>qualification consultation</i> appointment.	BOUNDARY: QualificationOptions QualificationDetails RDBMSBoundary CONTROL: ScheduleQualificationConsultationControl ENTITY: PatientRecord Appointment VolunteerCoordinatorHome RecordKeeping SearchForPatient	QualificationOptions QualificationDetails RDBMSBoundary ScheduleQualificationConsultationControl PatientRecord Appointment VolunteerCoordinatorHome RecordKeeping SearchForPatient
UC-04	Maintain Patient Records This use case is started by either, the Office Coordinator, Volunteer Coordinator or a Volunteer to create, update, and delete patient records.	BOUNDARY: PatientRecordOptions AppointmentSelection EditAppointmentDetails RDBMSBoundary CONTROL: MaintainPatientRecordsControl ENTITY: PatientRecord Appointment VolunteerCoordinatorHome RecordKeeping	PatientRecordOptions AppointmentSelection EditAppointmentDetails RDBMSBoundary MaintainPatientRecordsControl PatientRecord Appointment VolunteerCoordinatorHome RecordKeeping
UC-05	Maintain Volunteer Records This use case is started by the Volunteer Coordinator and it allows her to edit volunteer records.	BOUNDARY: VolunteerSearch EditVolunteerRecords RDBMSBoundary CONTROL: MaintainVolunteerRecordsControl ENTITY: Person User Providers Volunteer VolunteerGroup	VolunteerSearch EditVolunteerRecords RDBMSBoundary MaintainVolunteerRecordsControl Person User Providers Volunteer VolunteerGroup
UC-06	Research Statistics This use case is triggered	BOUNDARY: AdminHome ReportOptions	AdminHome ReportOptions

	by the Business Administrator to look up, compile and print out organizational statistics.	CONTROL: RDBMSBoundary ENTITY: ResearchStatisticsControl Person Providers
UC 07	Send Group-Email This use case is started by the Business Administrator and it allows him to mass email all the volunteers in the database.	BOUNDARY: PatientRecord Volunteer Appointment AdminHome AdminTools EmailForm RDBMSBoundary CONTROL: SendGroupEmailcontrol Person Providers ENTITY: User Volunteer VolunteerGroup Login RDBMSBoundary CONTROL: N/A
UC 08	Authorize User Use case is started by any user of the system; allows them to gain access on site or remotely, and check authentication.	ENTITY: User BOUNDARY: AdminHome Admintools AddUser CONTROL: PerformAdminTasksControl ENTITY: User
UC-09	Perform Administrative Tasks This use case is started by the Business Administrator and it allows him to create and delete users, and to reset user passwords.	 CONTROL: PerformAdminTasksControl ENTITY: Person Provider Volunteer User

Figure 7. Use-Cases – Analysis Classes Traceability Matrix – VIM Project

4.4 Traceability of Non-functional Requirements Using Traceability Matrices

Normally non-functional behaviors are not captured in use-cases. Satisfying these requirements may be as important and in some case more important than meeting some individual functional requirements. Projects can be implemented and distributed with known flaws that will be addressed. Sometimes, failure to meet non-functional requirements may well prevent release. Projects may fail by not satisfying these in some instances.

Non-functional requirements may be often captured as a feature or need in a vision document. According to Leffingwell [7], this capture may be realized in a table as illustrated in figure 8. Using this mechanism, we are mapping general system features or needs into specific supplementary requirements. These do not generally find their way into use-cases because they are not unique to a use-case. Non-functional requirements typically thread themselves through use-cases. And, while they may be mentioned within the context of a use-case (and this is good), they must be accommodated and traced elsewhere – hence the matrix mechanism.

The VIM project's most pressing non-functional requirements were two: the clarity of the user interface and the security of the medical records (In that the system is quite small by industry standards, there were no significant constraints on performance, legacy considerations, distribution, etc.) The care needed on the interface was so dominant that it was consistently in the foreground for all interface considerations, prototyping, and demonstrations. It was not documented separately. Similarly, the security of medical records access was particularly critical, but easily addressed. Only the business manager or clinical provider (physician or Nurse Practitioner) will have access to medical records. These are handled by password. All developers signed a Confidentiality and Commitment Statement. But because the VIM is a

volunteer clinic, they are not strictly bound legally to adhere to HIPAA – although they do their best just to be on the safe side.

4.5 Tracing Requirements beyond Requirements Artifacts

There are a number of authors who suggest that extending traceability to a very detailed level stop at this point; that is, we have traced functionality from one requirements artifact to another requirements artifact. As we embark upon design, it is claimed that additional traceability is unnecessary, because we actually have the mapping already in place: a one-to-one mapping of a use-case to its use-case realization: a requirements artifact to its corresponding design artifact. It is further asserted that the real purpose of the use-case realization is to provide an implementation approach for the requirements captured in a use-case. In essence, traceability, it is sometimes claimed, is greatly simplified.

	Supplementary Req 1	Supplementary Req 2	...	Supplementary Req n
Feature or System Requirement #1	X			X
Feature or System Requirement #2		X		X
Feature or System Requirement #m		X		X

Figure 8. System Requirements or Features and Supplementary Requirements [7]

4.6 Tracing Requirements into Use-Case Realizations

The literature contains many instances of tracing requirements from statements of needs through various named artifacts into generally more concrete additional requirements artifacts. The artifacts take on a variety of formats and each is accompanied by claims of goodness and wide applicability. This process may be referred to as pre-Requirements Specifications. [4] It is important to note, however, while many authors acknowledge the importance of requirements traceability, most emphasize this activity localized in the front-end of development. This may be attributed to widespread feelings that the root causes of software failure often center on inadequate problem analysis and the failure to properly manage requirements and not necessarily the tracing of them once properly analyzed and captured.

Where traceability to code is desired, tracing requirements into the components of the use-case realizations is required. This can be a very daunting task, even for a small system. For starters, developing interaction diagrams for each nontrivial scenario to discover component behaviors must be accomplished. Then, the structural relationships of collaborating objects (associations, dependencies, etc.) can be identified.

Using use-cases to drive traceability into design components of design particularly for a non-trivial system can result in thousands of objects with their myriad associations, aggregations, compositions, inheritance, dependencies and other relationships. Tracing a requirement to this level may well not be worth the cost and effort. Even with appropriate tools (without which tracing may be impossible) tracing to this level of detail would be an intensive undertaking – and may be likened to the accustomed phrase: analysis paralysis.

Perhaps one approach to tracing functionality into design without being driven by the details of use-cases is to revert to another form of requirements, namely the features, or some higher level of abstraction. From this vantage, features, such as ‘The system must provide...’ and “The relay must close when...” may be more readily ‘traced’ to some kind of collaboration of, perhaps, classes (perhaps spanning more than one use-case), the collaboration of which realizes a finer level of design. Such collaborations may be named and tracked. Further, such collaborations may be mapped / traced into code that realizes the collaboration.

4.7 Tracing Requirements into Testing

Tracing functionality into testing for small systems can once again be supported very well using the matrix tool approach. Using our use-case driving approach to testing, at an abstract level, a use-case must be tested by a suite of test cases. Yet we clearly know that use-cases consist of a number of scenarios (happy path, alternatives, exceptions, etc.) each of which should undergo independent verification. Further, it is this tracing to testing that is necessary to support any comprehensive traceability strategy. Sample templates are shown in Figure 9. The VIM development teams did not use this matrix approach to trace use-cases to scenarios to test cases. They did, however, use the use-cases to develop test case development by ensuring each scenario had tests to verify compliance

Use Case	Scenario Number	Originating Flow	Alternate Flow	Next Alternate	Next Alternate	Use Case	Scenario Number	...	Test Case Id
Use Case Name #1	1	Basic Flow				Use Case #1	1		1.1
	2	Basic Flow	Alternate Flow 1				2		2.1
	3	Basic Flow	Alternate Flow 1	Alternate Flow 2			3		3.1
	4	Basic Flow	Alternate Flow 3				4		4.1
	5	Basic Flow	Alternate Flow 3	Alternate Flow 1			4		4.2
	6	Basic Flow	Alternate Flow 3	Alternate Flow 1	Alternate Flow 2		4		4.3
	7	Basic Flow	Alternate Flow 4				5		5.1
	8	Basic Flow	Alternate Flow 3	Alternate Flow 4			6		6.1
	1	Basic Flow					7		7.1
Use Case Name #2							7		7.2
							8		8.1
						Use Case #2	1		1.1

Figure 9 Traceability from Use-Cases to Scenarios and to Test Cases [7]

5. Assessment of the Traceability Process using Matrices

So what is the practical assessment of using traceability matrices in the VIM development effort? Was a ‘better’ application produced? What was the effort? What was the cost? Was it worth it?

5.1 General Comments - Testing

Using the traceability matrix approach on the front end was very successful from a number of perspectives. Tracing using the matrix approach was not explicitly undertaken in testing, although the use-cases did drive the testing process. Without a formal mechanism to ensure comprehensive testing was done, there is not a strong assurance that sufficient numbers of test cases were, in fact, developed and run. This was a shortcoming in our process. In practice, the testers on the development team ran tests based parameters gleaned from the use-case scenarios. When shortcomings appeared, such as an attribute in the database not updated when the interface indicated the attribute was updated, the testers reported these errors via an internal defect management application to the analysts/programmers who investigated the problems, made changes, performed unit testing, and resubmitted for more comprehensive verification by the testers. Ultimately, the customer undertook acceptance testing.

5.2 General Comments - Design

In extending the traceability matrix approach into design, this approach became very quickly almost unmanageable, in our opinion, and not terribly practical – even for this small system. Figure 10 shows a traceability matrix from use-cases to design classes (DC) and design subsystems (DS) (all unnamed in this matrix).

ID	Backward Traceability	Forward Traceability	Forward Traceability 2
F1	N1	UC-03	DS3, DC11.01, DS5, DC16.01, DC16.02, DS7, DC20.01, DC20.02, DC20.03, DC20.04, DC21.01, DC21.02, DC21.03, DC21.04, DC21.05, DC21.06, DC22.01, DC22.02, DC22.03 DS3, DC11.01, DS5, DC16.01, DC16.02, DS7, DC20.01, DC20.02, DC20.03, DC20.04, DC21.01, DC21.02, DC21.03, DC21.04, DC21.05, DC21.06, DC22.01, DC22.02, DC22.03
F2	N1	UC-03	DS3, DC11.01, DS5, DC16.01, DC16.02, DS7, DC20.01, DC20.02, DC20.03, DC20.04, DC21.01, DC21.02, DC21.03, DC21.04, DC21.05, DC21.06, DC22.01, DC22.02, DC22.03 DS3, DC11.01, DS5, DC16.01, DC16.02, DS7, DC20.01, DC20.02, DC20.03, DC20.04, DC21.01, DC21.02, DC21.03, DC21.04, DC21.05, DC21.06, DC22.01, DC22.02, DC22.03
F3	N1	UC-03	DS3, DC11.01, DS5, DC16.01, DC16.02, DS7, DC20.01, DC20.02, DC20.03, DC20.04, DC21.01, DC21.02, DC21.03, DC21.04, DC21.05, DC21.06, DC22.01, DC22.02, DC22.03 DS3, DC11.01, DS5, DC16.01, DC16.02, DS7, DC20.01, DC20.02, DC20.03, DC20.04, DC21.01, DC21.02, DC21.03, DC21.04, DC21.05, DC21.06, DC22.01, DC22.02, DC22.03
F4	N1	UC-03	DS3, DC11.01, DS5, DC16.01, DC16.02, DS7, DC20.01, DC20.02, DC20.03, DC20.04, DC21.01, DC21.02, DC21.03, DC21.04, DC21.05, DC21.06, DC22.01, DC22.02, DC22.03 DS3, DC11.01, DS5, DC16.01, DC16.02, DS7, DC20.01, DC20.02, DC20.03, DC20.04, DC21.01, DC21.02, DC21.03, DC21.04, DC21.05, DC21.06, DC22.01, DC22.02, DC22.03
F5	N1	UC-03	DS3, DC11.01, DS5, DC16.01, DC16.02, DS7, DC20.01, DC20.02, DC20.03, DC20.04, DC21.01, DC21.02, DC21.03, DC21.04, DC21.05, DC21.06, DC22.01, DC22.02, DC22.03 DS2, DC6.01, DC6.02, DC6.03, DC6.04, DC6.05, DC7.01, DC7.02, DC7.03, DC7.04, DC7.05, DC7.06, DC8.01, DC8.02, DC8.03, DS5, DC15.01, DC15.02
F6	N2	UC-01	DS2, DC6.01, DC6.02, DC6.03, DC6.04, DC6.05, DC7.01, DC7.02, DC7.03, DC7.04, DC7.05, DC7.06, DC8.01, DC8.02, DC8.03, DS5, DC15.01, DC15.02 DS3, DC9.01, DC9.02, DC10.01, DC10.02, DS6, DC17.01, DC17.02, DC17.03, DC18.01, DC18.02, DC18.03, DC18.04, DC18.05, DC19.01, DC19.02, DC19.03 DS3, DC9.01, DC9.02, DC10.01, DC10.02, DS6, DC17.01, DC17.02, DC17.03, DC18.01, DC18.02, DC18.03, DC18.04, DC18.05, DC19.01, DC19.02, DC19.03
F7	N2	UC-01	DS3, DC9.01, DC9.02, DC10.01, DC10.02, DS6, DC17.01, DC17.02, DC17.03, DC18.01, DC18.02, DC18.03, DC18.04, DC18.05, DC19.01, DC19.02, DC19.03 DS3, DC9.01, DC9.02, DC10.01, DC10.02, DS6, DC17.01, DC17.02, DC17.03, DC18.01, DC18.02, DC18.03, DC18.04, DC18.05, DC19.01, DC19.02, DC19.03
F8	N3	UC-04	DS3, DC9.01, DC9.02, DC10.01, DC10.02, DS6, DC17.01, DC17.02, DC17.03, DC18.01, DC18.02, DC18.03, DC18.04, DC18.05, DC19.01, DC19.02, DC19.03 DS1, DC1.01, DC1.02, DC1.03, DC1.04, DC1.05, DC2.01, DC2.02, DC2.03, DC3.01, DC3.02, DC3.03, DC3.04, DC3.05, DC4.01, DC4.02, DC4.03, DC5.01, DC5.02, DC5.03 DS1, DC1.01, DC1.02, DC1.03, DC1.04, DC1.05, DC2.01, DC2.02, DC2.03, DC3.01, DC3.02, DC3.03, DC3.04, DC3.05, DC4.01, DC4.02, DC4.03, DC5.01, DC5.02, DC5.03
F9	N3	UC-04	DS1, DC1.01, DC1.02, DC1.03, DC1.04, DC1.05, DC2.01, DC2.02, DC2.03, DC3.01, DC3.02, DC3.03, DC3.04, DC3.05, DC4.01, DC4.02, DC4.03, DC5.01, DC5.02, DC5.03 DS1, DC1.01, DC1.02, DC1.03, DC1.04, DC1.05, DC2.01, DC2.02, DC2.03, DC3.01, DC3.02, DC3.03, DC3.04, DC3.05, DC4.01, DC4.02, DC4.03, DC5.01, DC5.02, DC5.03
F10	N4	UC-02	DS1, DC1.01, DC1.02, DC1.03, DC1.04, DC1.05, DC2.01, DC2.02, DC2.03, DC3.01, DC3.02, DC3.03, DC3.04, DC3.05, DC4.01, DC4.02, DC4.03, DC5.01, DC5.02, DC5.03 DS1, DC1.01, DC1.02, DC1.03, DC1.04, DC1.05, DC2.01, DC2.02, DC2.03, DC3.01, DC3.02, DC3.03, DC3.04, DC3.05, DC4.01, DC4.02, DC4.03, DC5.01, DC5.02, DC5.03
F11	N4	UC-02	DS1, DC1.01, DC1.02, DC1.03, DC1.04, DC1.05, DC2.01, DC2.02, DC2.03, DC3.01, DC3.02, DC3.03, DC3.04, DC3.05, DC4.01, DC4.02, DC4.03, DC5.01, DC5.02, DC5.03 DS1, DC1.01, DC1.02, DC1.03, DC1.04, DC1.05, DC2.01, DC2.02, DC2.03, DC3.01, DC3.02, DC3.03, DC3.04, DC3.05, DC4.01, DC4.02, DC4.03, DC5.01, DC5.02, DC5.03
F12	N4	UC-02	DS1, DC1.01, DC1.02, DC1.03, DC1.04, DC1.05, DC2.01, DC2.02, DC2.03, DC3.01, DC3.02, DC3.03, DC3.04, DC3.05, DC4.01, DC4.02, DC4.03, DC5.01, DC5.02, DC5.03 DS1, DC1.01, DC1.02, DC1.03, DC1.04, DC1.05, DC2.01, DC2.02, DC2.03, DC3.01, DC3.02, DC3.03, DC3.04, DC3.05, DC4.01, DC4.02, DC4.03, DC5.01, DC5.02, DC5.03
F13	N4	UC-02	DS1, DC1.01, DC1.02, DC1.03, DC1.04, DC1.05, DC2.01, DC2.02, DC2.03, DC3.01, DC3.02, DC3.03, DC3.04, DC3.05, DC4.01, DC4.02, DC4.03, DC5.01, DC5.02, DC5.03 DS1, DC1.01, DC1.02, DC1.03, DC1.04, DC1.05, DC2.01, DC2.02, DC2.03, DC3.01, DC3.02, DC3.03, DC3.04, DC3.05, DC4.01, DC4.02, DC4.03, DC5.01, DC5.02, DC5.03
F14	N5	UC-05	DS5, DC13.01, DC13.02, DC14.01 DS5, DC13.01, DC13.02, DC14.01
F15	N5	UC-05	DS5, DC13.01, DC13.02, DC14.01 DS5, DC13.01, DC13.02, DC14.01
F16	N5	UC-05	DS5, DC13.01, DC13.02, DC14.01 DS5, DC13.01, DC13.02, DC14.01
F17	N6	UC-06	DS4, DC12.01, DC12.02, DC12.03

Figure 10. Partial Traceability Matrix: Use-Cases to Design Classes in Use-Case Realization

At this point, the teams felt that the payoff in tracing functionality into specific design classes was not going to be worth the effort. Had this been a safety-critical system or similar system, then it might have been worth the cost to pursue this (but likely not using a matrix approach). But such was not the case in this development. It is, however, interesting to note that the iteration plan did address the implementation of a number of design classes in each iteration. For example, one iteration implemented: DC9.01, DC9.02, DC10.01, DC10.02, DC11.01, DC17.01, DC17.02, DC17.03, DC18.01, DC18.02, DC18.03, DC18.04, DC18.05, DC19.01, DC19.02, DC19.03. Another iteration implemented the following design classes: DC11.01, DC16.01, DC16.02, DC20.01, DC20.02, DC20.03, DC20.04, DC20.05, DC21.01, DC21.02, DC21.03, DC21.04, DC21.05, DC21.06, DC22.01, DC22.02, DC22.03.

Requirements were continuously updated throughout the development. During each iteration, stakeholders made modifications to the requirements and the development team made adjustments accordingly. Each update resulted in a thoroughly documented and updated specification. Traceability was modified to reflect these changes.

5.3 General Comments – End-User Involvement

It is important to note that the development teams had near constant access to the end-users. Even though considerable unit testing and verification testing was undertaken by team members, it was the rapid availability of the end-user comments that provided for a most productive development environment. When end-user testing did not coincide with end-user expectations, a flag was raised and rework was undertaken to bring the current effort back in line with the use-case specifications. Scope creep was a major problem that was addressed with the end-user.

5.4 General Comments – Cost Effectiveness

So, from a practitioner's perspective, were the costs worth the benefits? What, if anything, was saved? Was money saved? Was any money made? While quantification is important, it is difficult to offer only numbers without presenting concomitant qualification factors.

The entire deployed project (I must confess that at the time of this writing, the team is undertaking 'Phase II') required about 8000 lines of code. The total time spent on developing the requirements traceability matrices was approximately fifteen hours. So, this equates to one hour of time per 533 lines of deployed code.

A few members of the development teams were employed in downtown Jacksonville at major corporations, such as Blue Cross Blue Shield of Florida, Bell South, CSX Transportation, and a major government contractor. In several of these organizations, the development cost ranges from \$71 to upper 80s. Arbitrarily, let's use \$75/hour. Given this, the total 15-hour cost of developing and maintaining the traceability matrices was \$1125. This sum is distributed over the development cycle of the project, or about nine months – about \$125/month. (Please recall, these are students taking a number of additional courses as part of their graduate studies; most have full or near-full-time IT jobs and families as well. Hence the nine months.)

The project required about 2000 hours of effort. This includes a good deal of 'start up' time, as the developers were required to learn a number of new technologies used during development with which they were unfamiliar, such as use-cases, UML modeling, Rational Rose® and Requisite Pro®, VB.Net, ASP.Net, and others. Extrapolating, the project is thus costed out at about \$150,000 (2000 hours @ \$75/hour).

A number of benefits resulted from the use of traceability matrices.

The number of errors per use-case was few. Developers estimate that there were about five errors per use-case that centered on a misunderstood requirement, rather than lighter programmer errors. Since the customer was available, these were very quickly reworked and resolved. Customer access was a great help. The team felt that the process of carefully mapping needs to features to use-cases resulted in very reliable use-cases, such that when properly implemented, errors were both few in number and reasonably minor in complexity. Average time to repair an error based on misunderstood or misstated requirement via use-case scenarios (MTTR) was 2.0 hours.

Because the use-cases were so carefully traced-to, the requirements were more reliable and rework was minimized. Since the RUP is a use-case driven process, and since the use-cases drive testing, errors uncovered during testing were often refinements of requirement(s) and generally easily repaired.

6. Conclusions

Of all the techniques that are available to trace requirements [4] from needs to features to use-cases to an analysis model, it was clear to the two teams that the process of assuring traceability using a matrix mechanism was a very worthwhile exercise. A single member on each team had responsibility for ensuring traceability (as well as other duties). Each deliverable (there were eleven) included an updated report on traceability as well as a presentation, within which traceability was included.

The teams felt it important to ensure that their effort in developing the specifications, the analysis model, the design and implementation were spent on features that were needed. There were no frills. The assurance that all efforts undertaken were, in fact, traceable to stakeholder needs was very important. The cost was minimal in that no specific commercially available tools were used. Further, because the application developed was small in a number of ways, the matrix approach to requirements traceability was an effective tool – at least for the front end.

One of the difficult tasks experienced was the determination as to just how much traceability and effort expended on developing the traceability strategy were enough. When the teams entered into serious design, as implied by Figure 10, attempts to manage traceability via matrices became unwieldy and very cumbersome. It was at that time that the use of traceability matrices was going to cost more time / effort than the nature of the project required. Yet, in not pursuing

traceability into the design classes and the services that each provided was unsettling at times. The traditionally small analysis model allowed team members to feel comfortable with traceability to that point, but the sheer size of the design model and this lack of assurance that all desired functionality was captured in the participating classes proved to be of some concern. The teams did not attempt to group and trace 'features' into collaborations, as presented earlier; however, great care was exercised in the development of test cases.

The teams made a process error in not returning to the traceability matrices to trace use-cases to test suites. While great care was exercised in developing test cases directly from the use-case specifications, a formal mechanism would have provided additional assurance of sufficient test coverage was accomplished.

Our conclusions are that for small systems, the matrix approach for requirements traceability is an approach that supports a more reliable development process resulting in fewer development errors due to the tracking of requirements into technologies (use-cases here) that drive user-interface development, functional development, testing and a number of other related activities. The development team felt that requirements traceability should be integrated into the very fabric of the development process. The extent of requirements traceability, however, should be governed by the nature of the application and the criticality of specific requirements.

It is important to have everyone 'on board' if the team is to undertake viable requirements traceability. First of all, the importance of requirements traceability was clearly articulated from the beginning of the project. In discussions addressing the best practices of software engineering, the management of requirements and the failure to embrace changing requirements receive top billing. The advantages of an iterative development process the notion of time-boxed iterations, and several other development fundamentals underpinned by the RUP were stressed prior to even embarking on the project. Not impaired by experience, the students did not question the cost effectiveness of the traceability exercises and readily saw the value. The strategy was clear; traceability was integral to our process; and the understanding of the importance was shared equally by all team members.

Despite many advances in the area of requirements traceability, failure to trace requirements remains a serious problem today. Many feel that infusing a traceability culture into the organization helps to mature the organization with the concomitant improvement in productivity. Tracing the life of a requirement may be an onerous undertaking, and there is no single solution to this noble goal. According to Scott Ambler, traceability is difficult, but a mature approach to requirements traceability may be the difference between organizations that are successful at developing software and those that are not. [3] We have concluded that for small systems, use of the traceability matrix is an effective mechanism to support traceability that is of low cost, high value, and reasonably easy to implement.

References

- [1] Turbit, Nevile, "Requirements Traceability," Project Perfect – Project Management Information, http://www.projectperfect.com.au/info_requirements_traceability.php.
- [2] Kulak, Daryl and Eamonn Guiney, *Use Cases – Requirements in Context*, Addison-Wesley, Second Edition, 2004. ISBN 0-321-15498-3
- [3] Ambler, Scott, "Tracing Your Design," Software Development Magazine, www.sdmagazine.com. April 1999
- [4] Gotel, Orlena C. Z. and Anthony C.W.Finkelstein, "An Analysis of the Requirements Traceability Problem", oczg; acwf@doc.ic.ac.uk
- [5] <http://www.vim-jax.org>
- [6] Lethbridge, Timothy and Robert Laganiere, "Object-Oriented Software Engineering" McGraw-Hill, 2001, ISBN: 0-07- 709761-0
- [7] Leffingwell, Dean, Don Widrig, "The Role of Requirements Traceability in Software Development," http://www.therationaledge.com/content/sep_02/m_requirementsTraceability_dl.jsp

The Keyhole Problem

Scott Meyers
www.aristeia.com

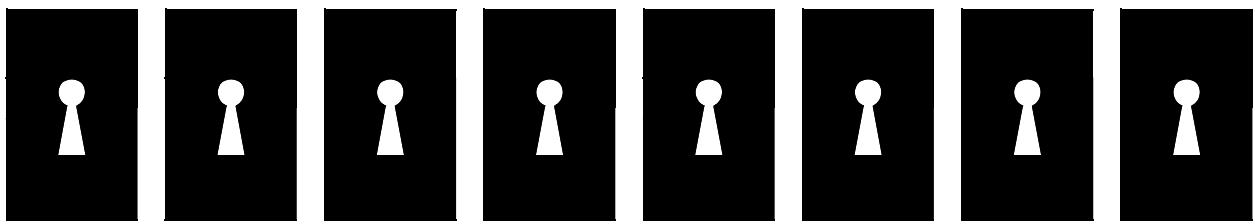
Software too often imposes gratuitous restrictions on how we see or interact with the world - forcing us to experience the world through the keyhole of a door. The Keyhole Problem arises every time software artificially restricts something you want to see or something you want to express.

If you want to see an image, but your image-viewing software artificially restricts how much of that image you can see at a time-that's the keyhole problem. If you want to specify a password of a particular length, but your software says it is too long-that's the keyhole problem. If you want to type in your U.S. telephone number, but your software refuses to let you punctuate it in the conventional manner with a dash between the three-digit prefix and the four-digit exchange-that's the keyhole problem.

This talk, which applies to software written in any language, discusses what "keyholes" are, why they are worth caring about, and suggests ways to design and implement software containing as few keyholes as possible.

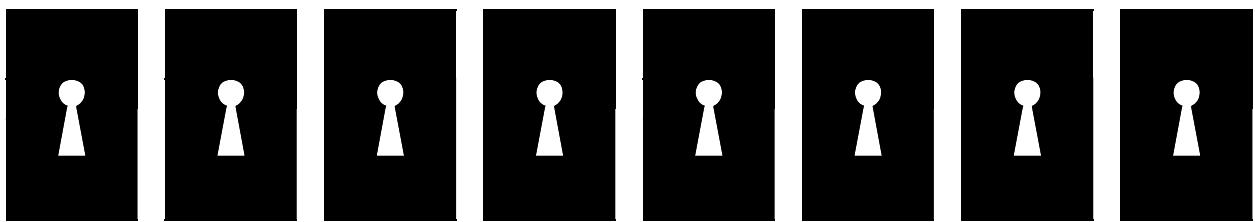
Scott Meyers is an independent author and consultant with over three decades of experience in software development practice and research. His perennially best-selling Effective C++ books (Effective C++, More Effective C++, and Effective STL) defined a new genre in technical publishing, and his Effective C++ CD introduced several innovations in the web-based presentation of technical material.

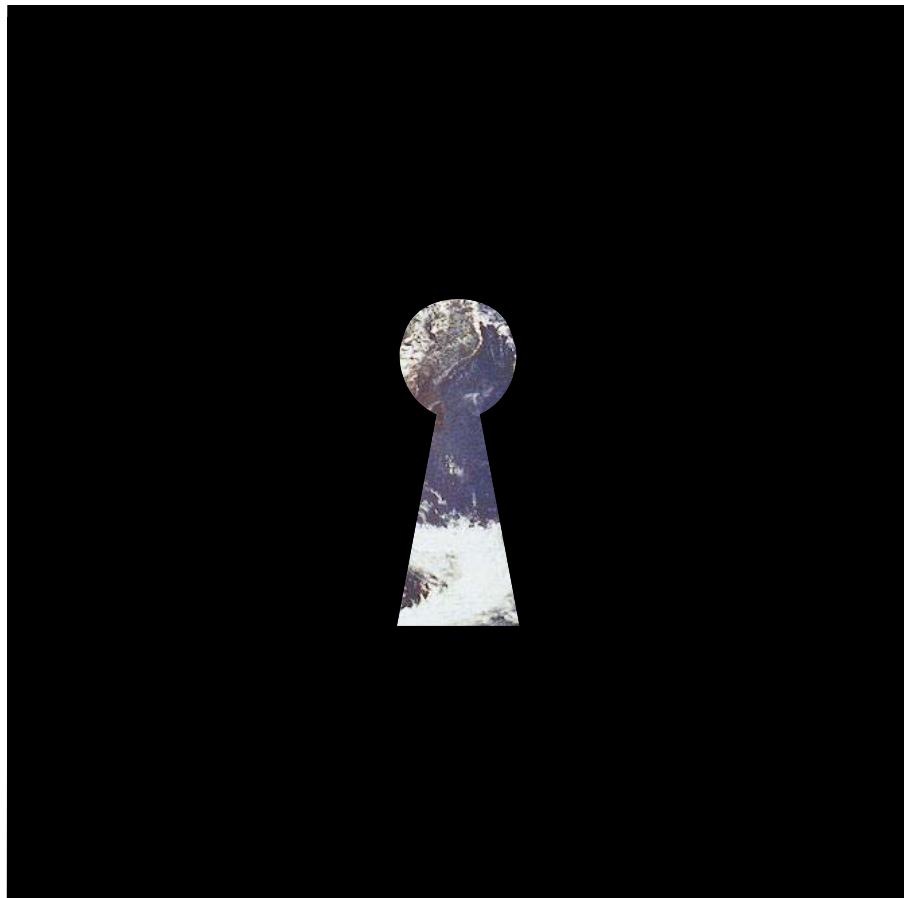
Scott is Consulting Editor for Addison Wesley's Effective Software Development Series and a member of the technical advisory boards of the online journal, The C++ Source, and Software Development magazine. He received his PhD in Computer Science from Brown University.



The Keyhole Problem

Scott Meyers
www.aristeia.com





Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 3

Topics

- What are “keyholes?”
- Why are they important?
- How can we eliminate or avoid them?
 - ⇒ In some cases, elimination (from current software) isn’t worth it, but avoidance (for future software) is.
 - ⇒ Due to time restrictions, I’ll treat this aspect superficially.

Almost everything that follows is something I experienced personally.

- I don’t seek out keyholes.
- They’re unavoidable.

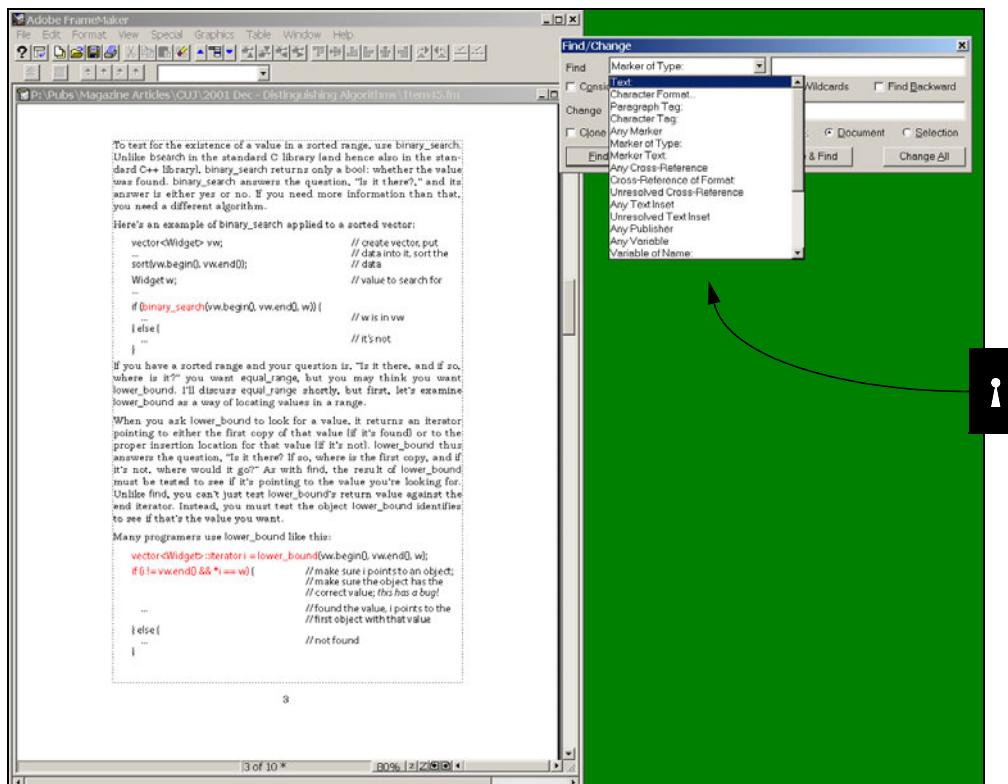
Visible Keyholes (GUI Components)

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 5

The Vertical ListBox/ComboBox Keyhole

From Adobe's FrameMaker 6:

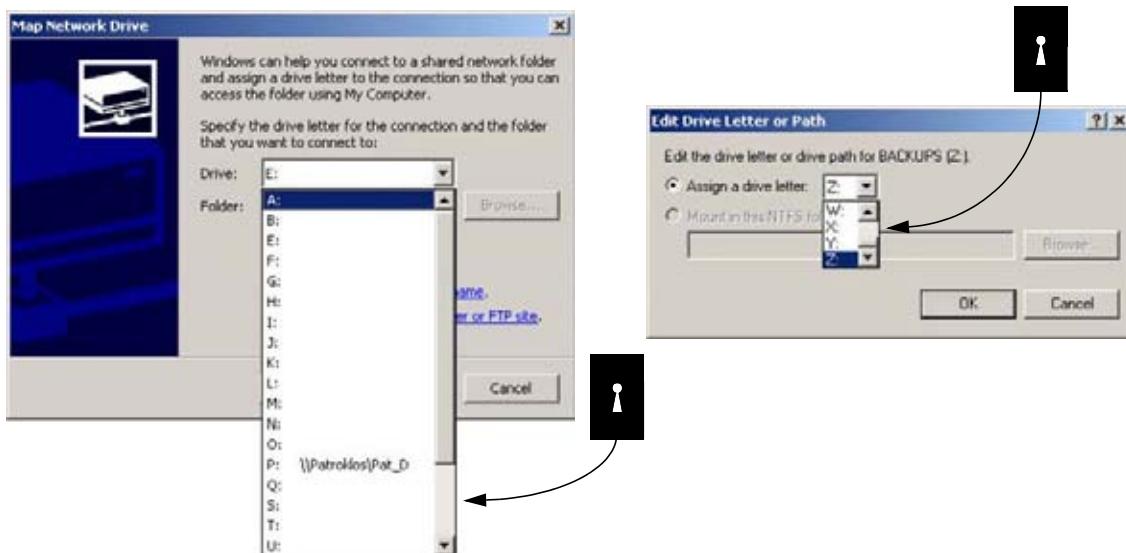


Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 6

The Vertical ListBox/ComboBox Keyhole

Microsoft's Windows 2000 demonstrates that keyholes lead to inconsistency:

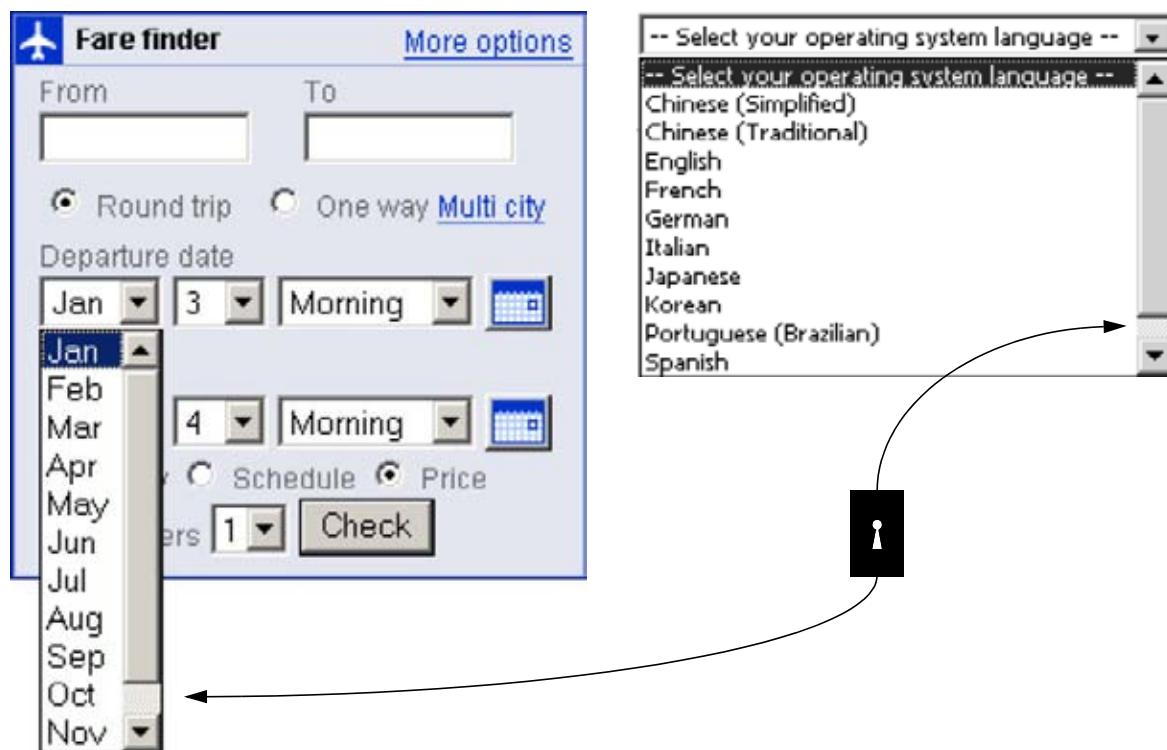


Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 7

The Off-By-One ListBox/ComboBox Keyhole

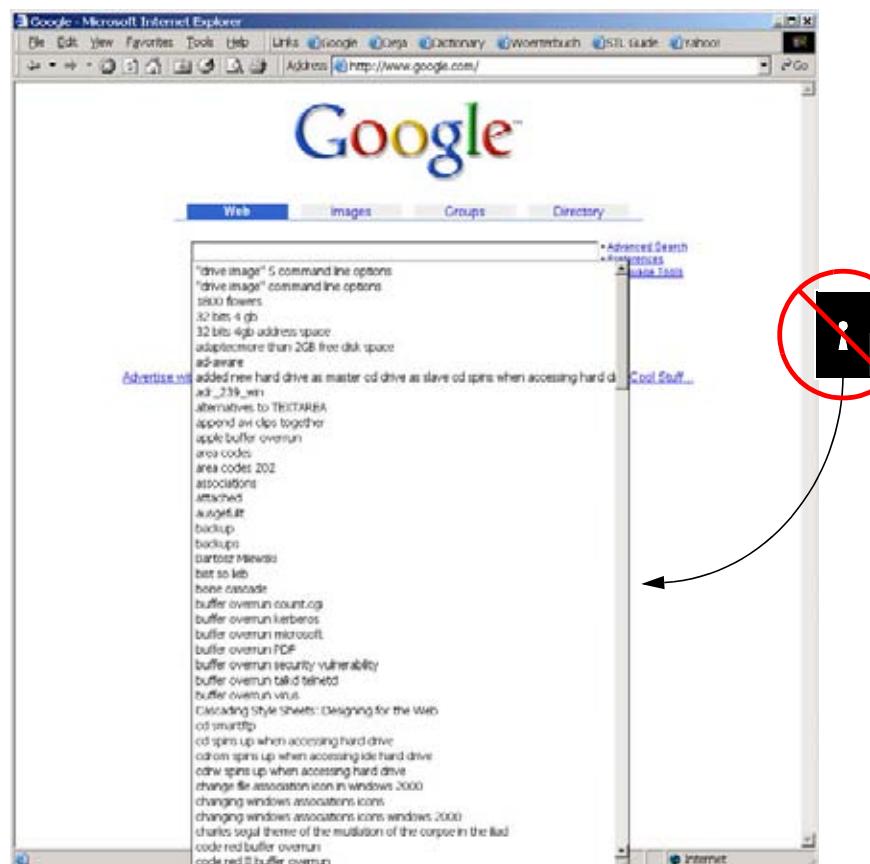
Murphy strikes United Airlines' and Microsoft's web sites:



Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 8

It Doesn't Have to be This Way



Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 9

The Horizontal ListBox/ComboBox Keyhole

From Microsoft's web site for an older version of Windows Media Player:

Windows Media Player 7.1

Getting Started:

Thanks for using Windows Media™ Player 7.1. This guide is designed to help you get the most of our newest player. Windows Media Player 7.1 is designed to offer the best consumer experience for the discovery, download, personalization, and playback of high quality Windows Media audio and video, and the popular MP3 format. Windows Media™ Player 7.1 represents a significant enhancement to the Windows Media consumer experience and lets you play the most popular streaming and local audio and video formats, organize and catalog your digital music collection, then easily transfer albums or custom playlists to a portable device. If you're looking for new music, listen to a multitude of Internet radio stations worldwide, or find information about thousands of new artists in the Media Guide. You can also personalize the look and feel of your Windows Media™ Player into everything from a shark's head to a television.

The best part is that using Windows Media™ Player is a breeze. Here are some steps to help you get started and become productive right away:

- **Now Playing** plays video content, or displays track information and visualizations (splashes of color and geometric shapes that change with the music beat) during audio playback.

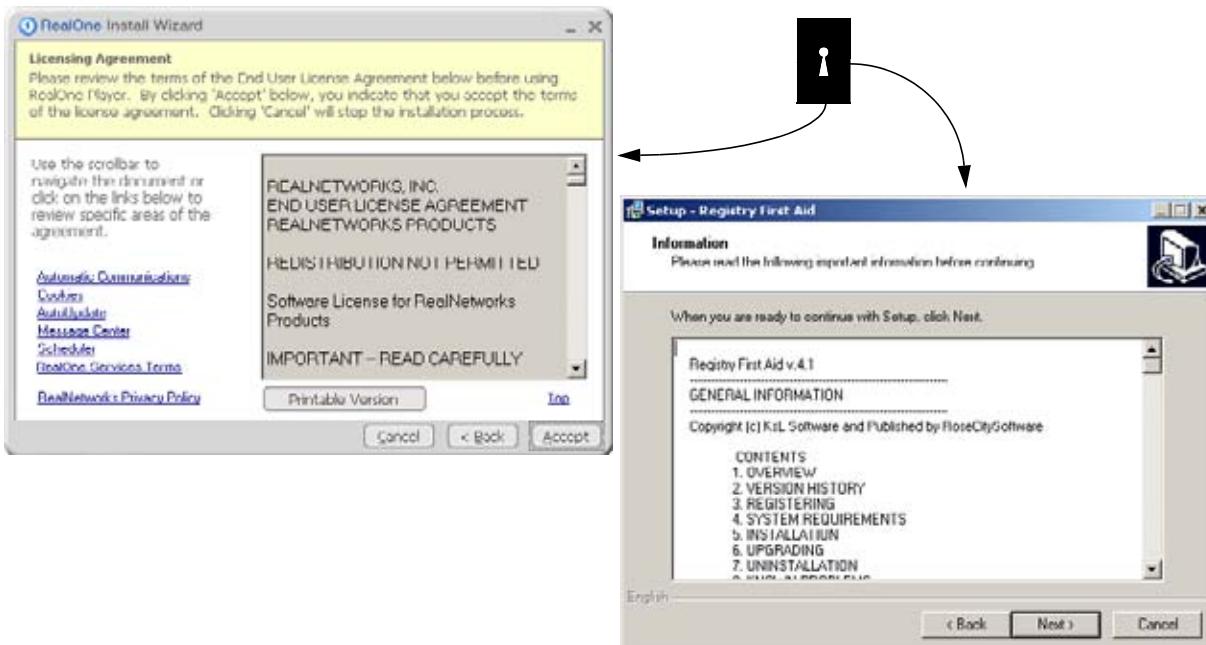
- Note that this is also a vertical listbox keyhole.
- “Burn your own au” makes me nervous.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 10

The Fixed Size Window Keyhole

Popular during program installation. From Real Networks and Rose City Software:



Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 11

The Fixed Size Window Keyhole

Application windows can also be fixed in size. From PC Tools' Registry Mechanic:



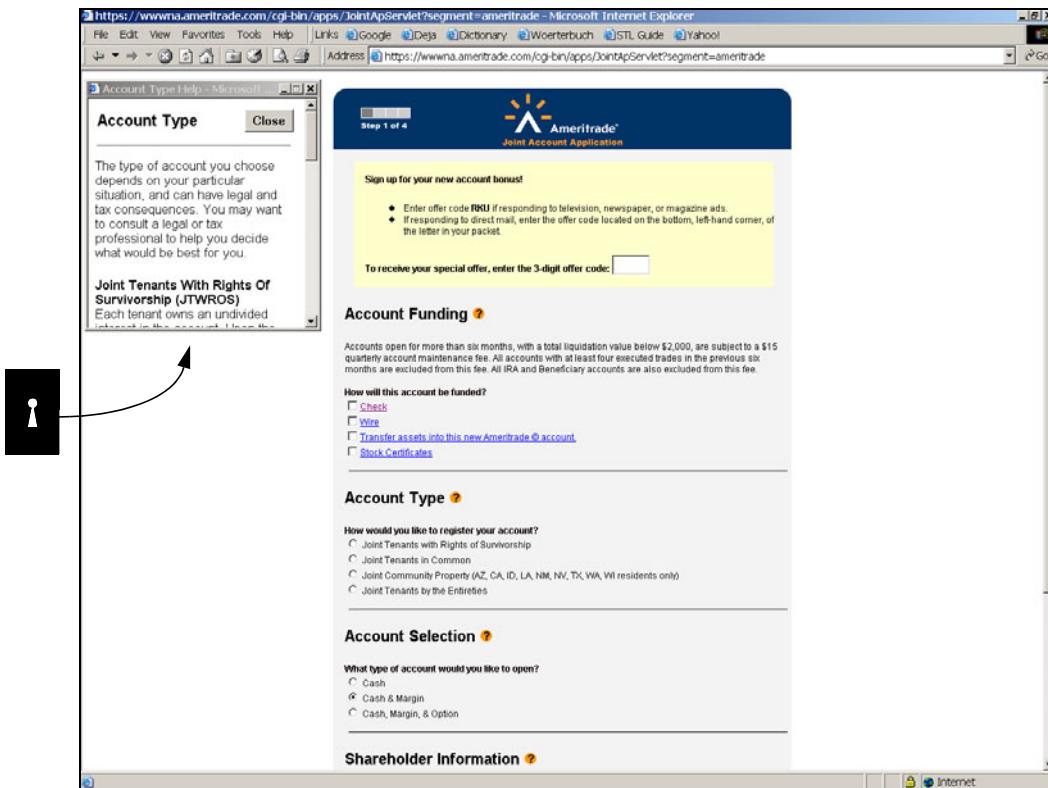
- This window is fixed at 800x600. This allows 5 of 457 problems to be shown.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 12

The Fixed Size Window Keyhole

Web sites like fixed-sized windows, too. From Ameritrade's web site:

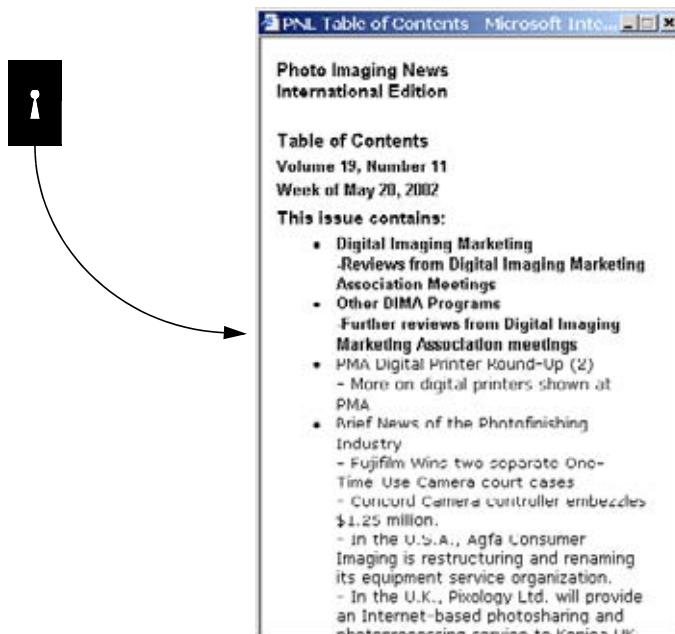


Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 13

The Fixed Size Window Keyhole

An interesting twist: an unscrollable window from Photo Imaging News' web site:

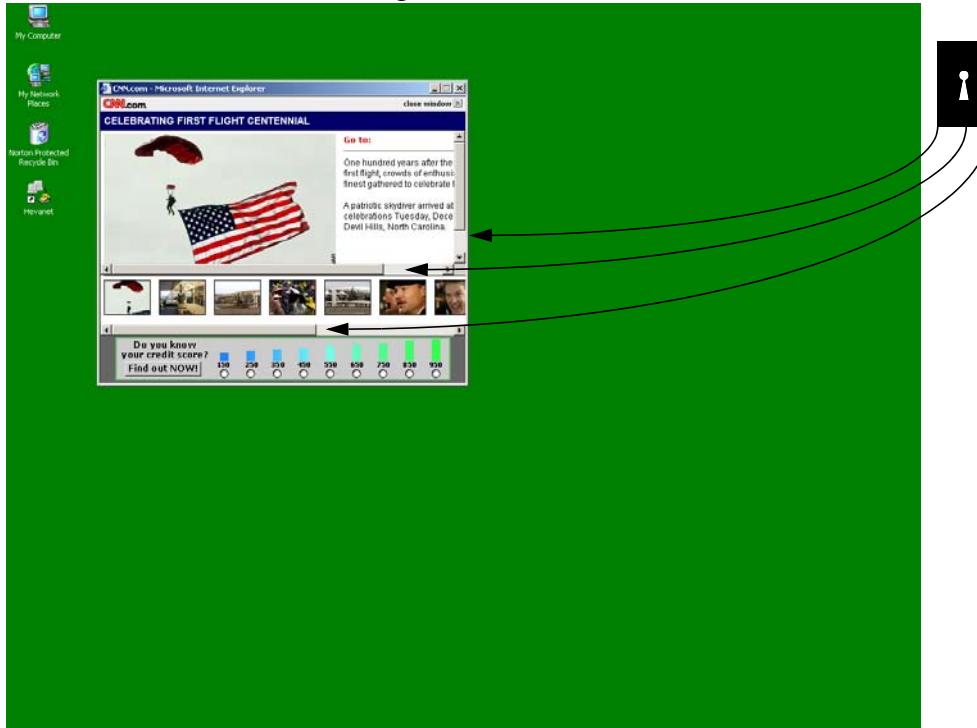


Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 14

The Fixed Size Window Keyhole

CNN's web site has no trouble offering scroll bars:



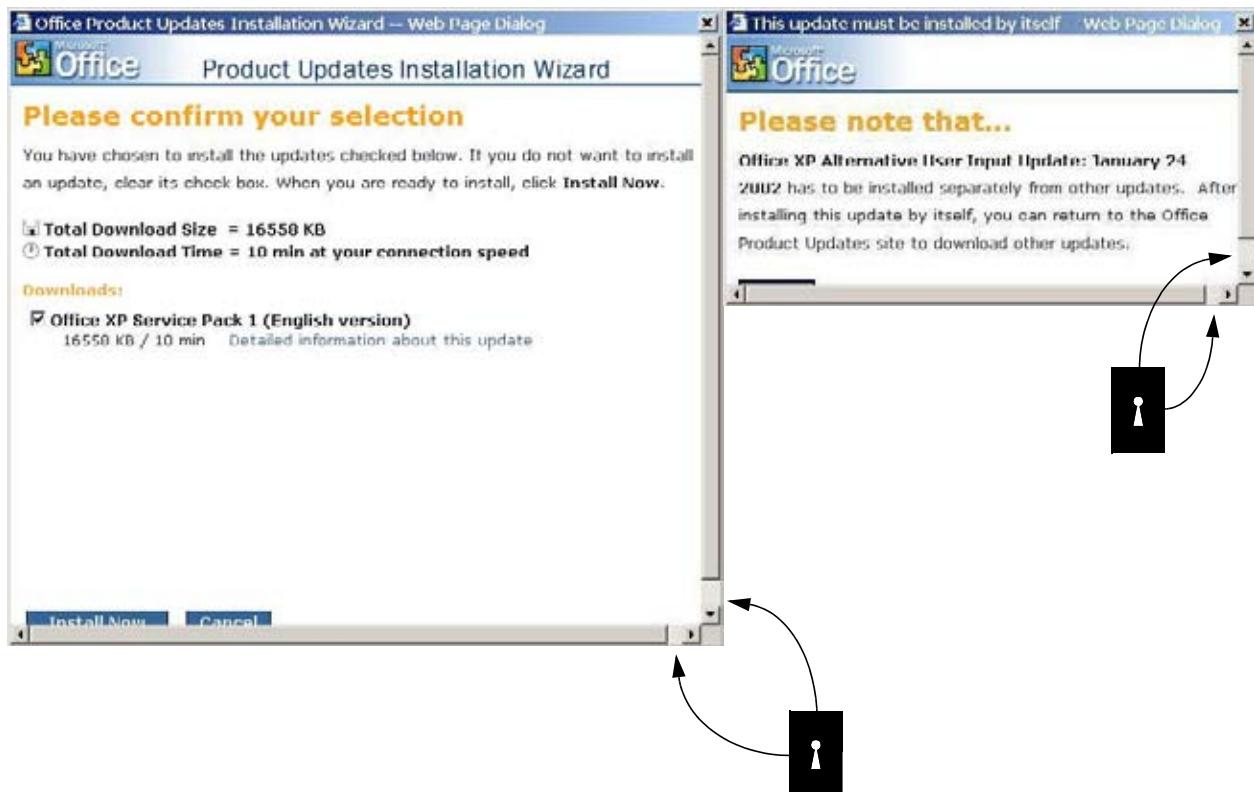
- Interestingly, the window size is fixed in Internet Explorer, but not in Firefox.

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 15

The Fixed Size Window Keyhole

Microsoft's Office Update web site has trouble only when...can you guess?



Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 16

The Fixed Size Window Keyhole

For web pages, fixing these problems is trivial.

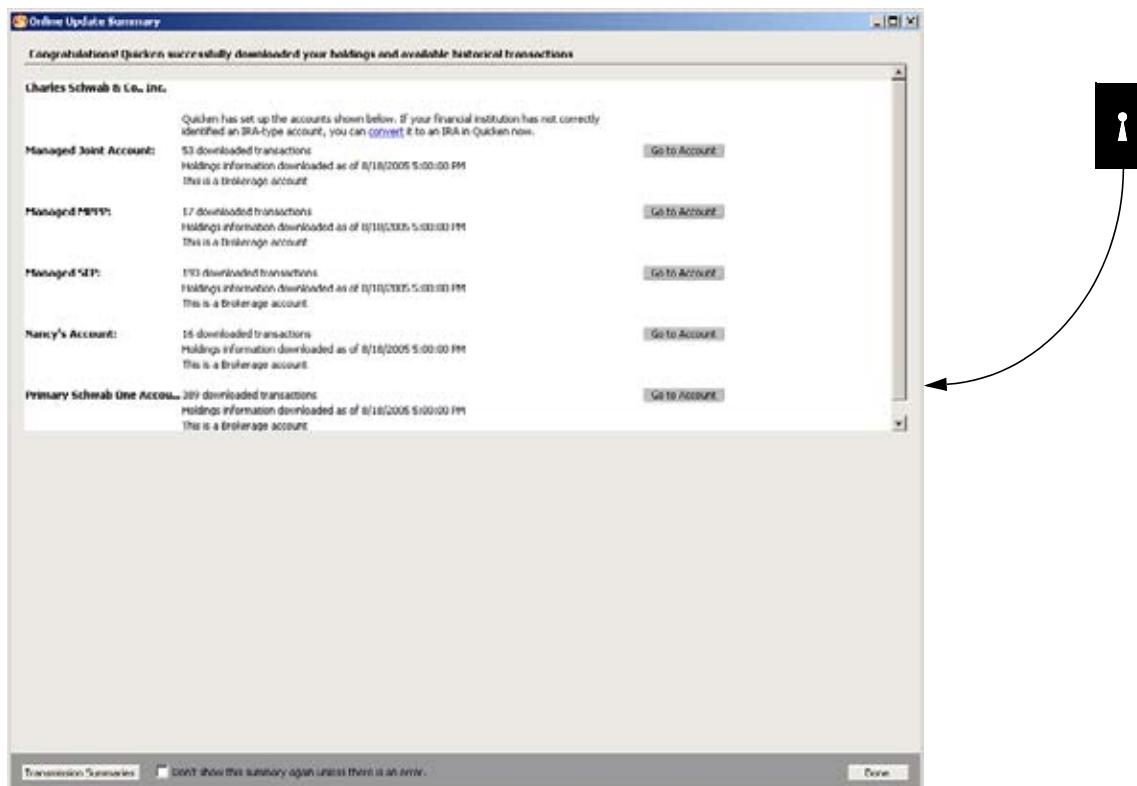
- Add this to each call to window.open: **resizable=yes**

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 17

The Fixed Size Window Keyhole

Intuit's Quicken Premier 2006 offers a curious variation:

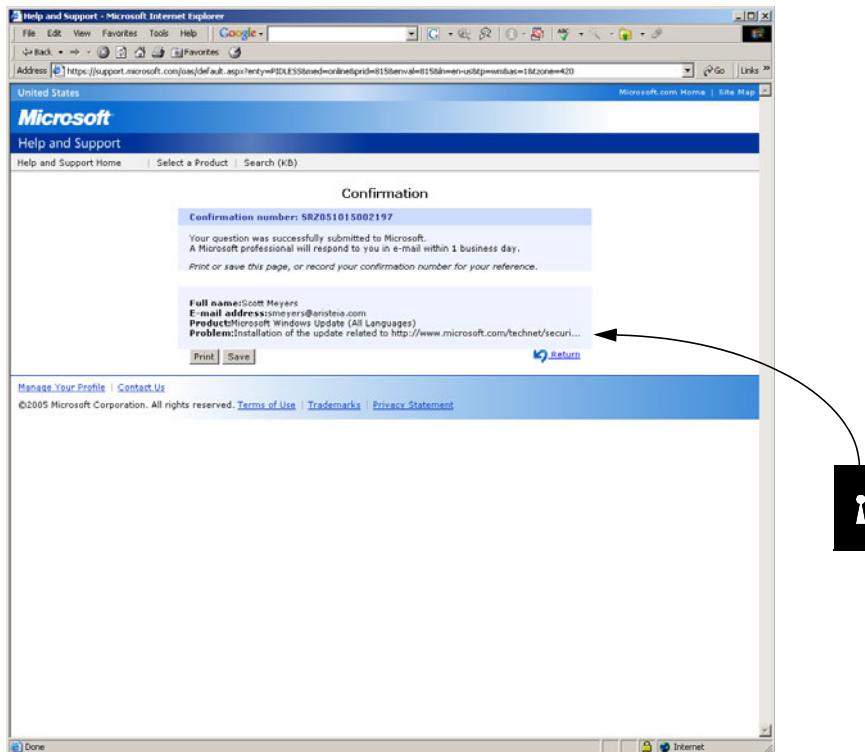


Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 18

The Fixed Size Window Keyhole

Microsoft's support web site has an equally curious variation on the variation:

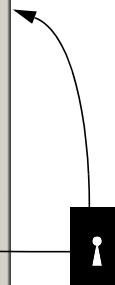


Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 19

The Fixed Size Dialog Keyhole

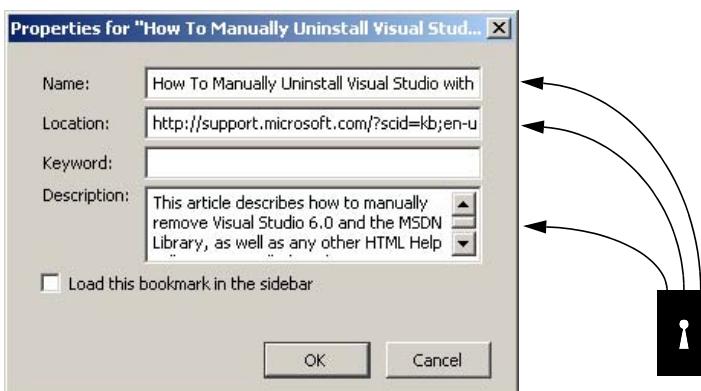
When long paths collide with narrow dialogs. From ACD Systems' ACDSee 7:



- Consider how useful this would be with network paths...

The Fixed Size Dialog Keyhole

Open source, closed source, it doesn't matter. From Mozilla's Firefox:



Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 21

The Fixed Size Dialog Keyhole

Microsoft's Windows 2000 Dial-Up Networking defines irony:

To enable operator-assisted calls or manual dialing

1. Open **Network and Dial-up Connections**.
2. On the **Advanced** menu, click **Operator-Assisted Dialing**.
3. Double-click the connection you want to dial.
4. Pick up the telephone handset, and then dial the number or ask the operator to dial it for you.

The number assigned to the entry is displayed in the dialog box for easy reference.

5. Immediately after you have finished dialing, click **Dial**.

6. Hang up the handset only when the connection is established, typically signaled with a change in signal strength.

It is always safe to replace the handset immediately after the connection begins verifying your user name and password. The system will remind you of this.

Notes

- To open Network and Dial-up Connections, click Start, Control Panel, and then double-click Network and Dial-up Connections.



Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 22

The Fixed Size Dialog Keyhole

When many tabs collide with narrow dialogs. From Microsoft's Visual Studio 6:



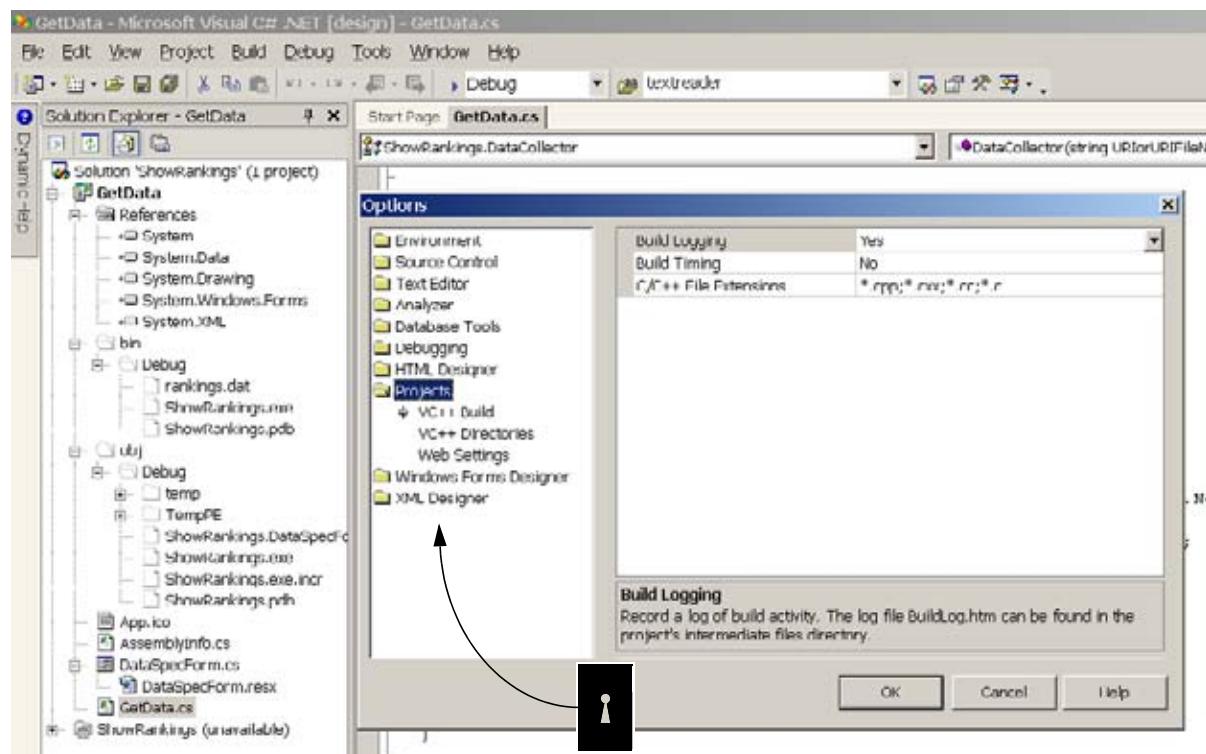
- This dialog is 417 pixels wide.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 23

The Limited Tree Expansion Keyhole

Microsoft's Visual Studio .NET 2003 replaces one keyhole with another:



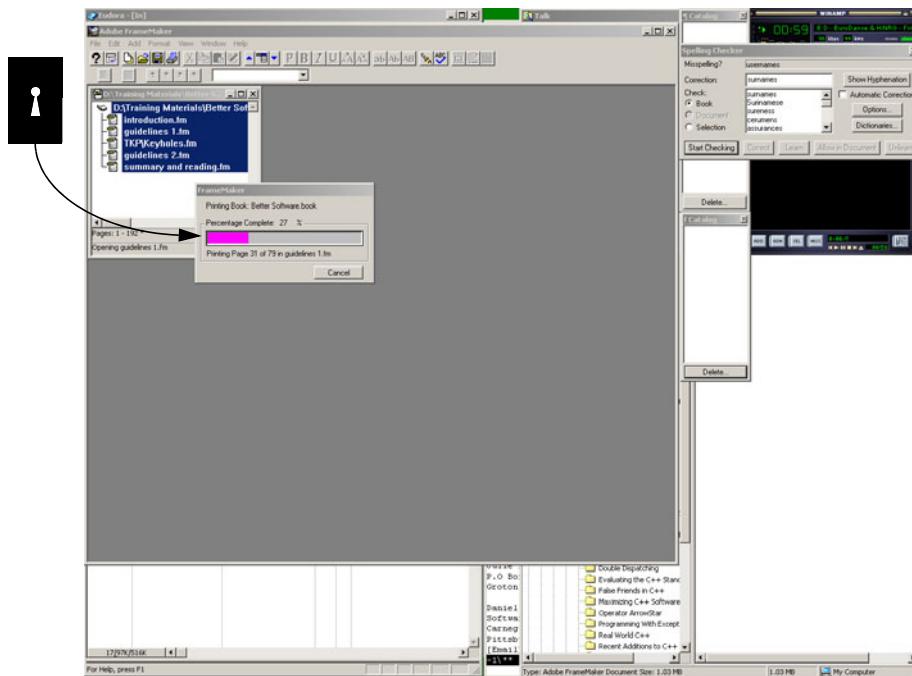
Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 24

The Modal Dialog Keyhole

Like too many dialogs, the VS Options dialog is modal.

- Why can't I scroll while the Print or About dialog is up?
- Adobe's FrameMaker 6 prevents closing the spell-checker when printing:

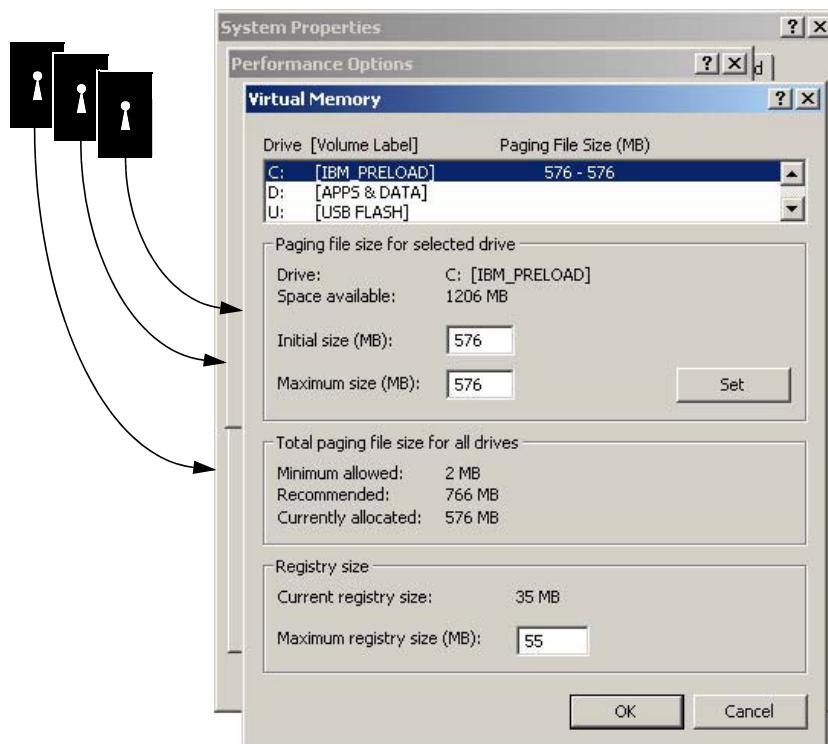


Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 25

The Modal Dialog Keyhole

Modal dialogs often spawn others, e.g., from Microsoft's Windows 2000:

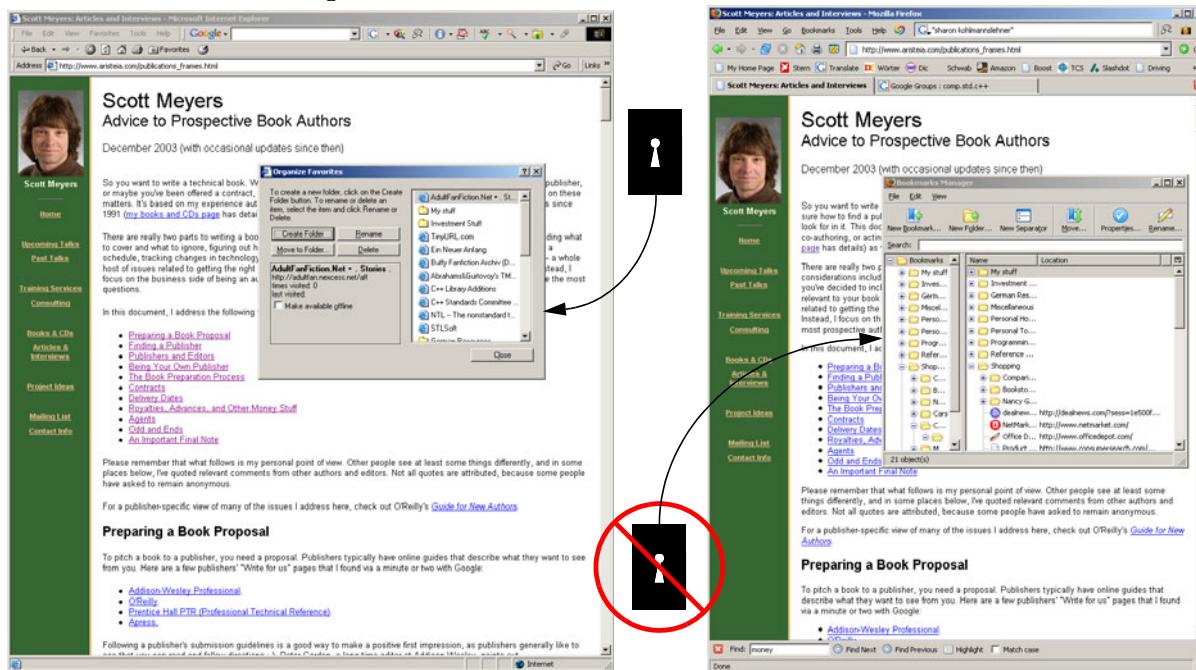


Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 26

The Modal Dialog Keyhole

Better design can help. Bookmark management under Mozilla's Firefox 1.07 and Microsoft's Internet Explorer 6:

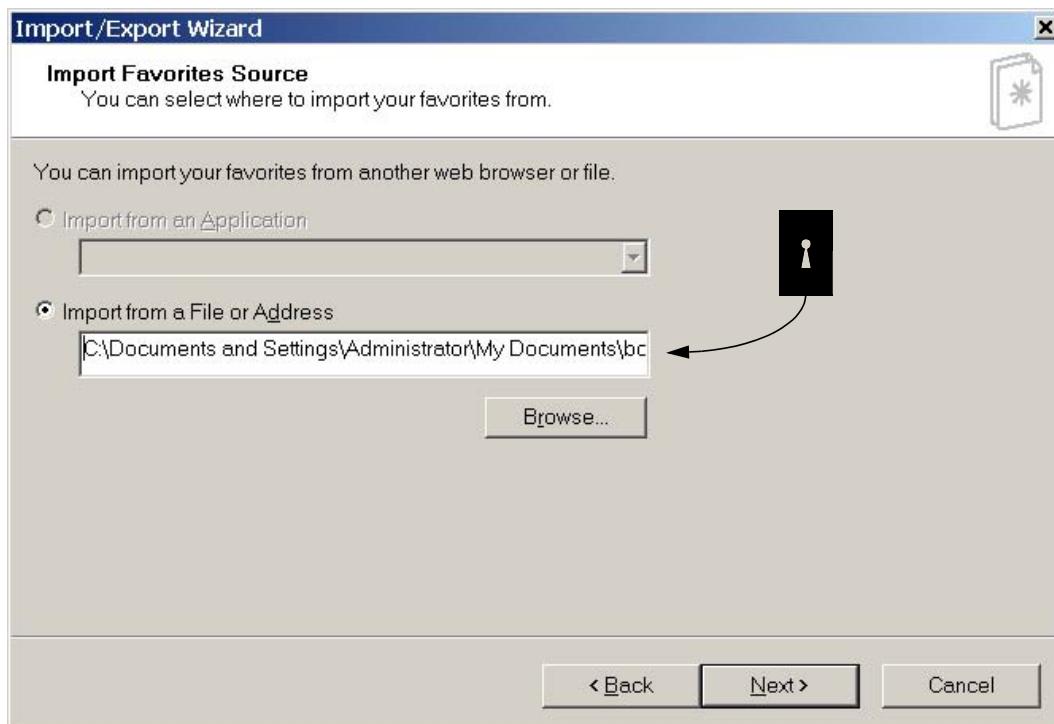


Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 27

The Fixed Width Edit Control Keyhole

From Microsoft's Internet Explorer 6:



Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 28

The Fixed Width Edit Control Keyhole

From OneSuite.com's web site (account setup):

A screenshot of a web form titled "STEP 2: Account Information". The form includes fields for User Name, Password, Re-Enter Password, Secret Question, Secret Answer, and E-Mail Address. A legend on the right provides instructions: "3 to 16 chars. Last character must be numeric." for the User Name field; "This question and answer are used if you lose your password." and "The answer to your secret question." for the Secret Question and Secret Answer fields; and "Account information is sent to this e-mail address." for the E-Mail Address field. A black keyhole icon is positioned to the left of the Secret Question field, with an arrow pointing from it to the field itself.

From OneSuite.com's web site (account maintenance):

A screenshot of a web form titled "Secret Question: What is your favorite color?". It includes an "Answer:" field, a "Save" button, and a "Reset" button. A black keyhole icon is positioned to the right of the "Answer:" field, with an arrow pointing from it to the field itself.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 29

The Fixed Width Edit Control Keyhole

From SourceForge's web site:

A screenshot of a Mozilla Firefox browser window showing a SourceForge.net project tracker page. The URL is http://sourceforge.net/tracker/?func=add&group_id=15&id=200001. The page displays a list of most active projects and a detailed view for a specific project. A black keyhole icon is positioned to the right of the "Detailed Description:" text area, with an arrow pointing from it to the area.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 30

The Fixed Width Edit Control Keyhole

For web pages, fixing the problem is trivial.

- Add this to each input and textarea tag: `style="width:100%"`

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 31

The Lying Fixed Width Edit Control Keyhole

From **Marriott**'s web site:

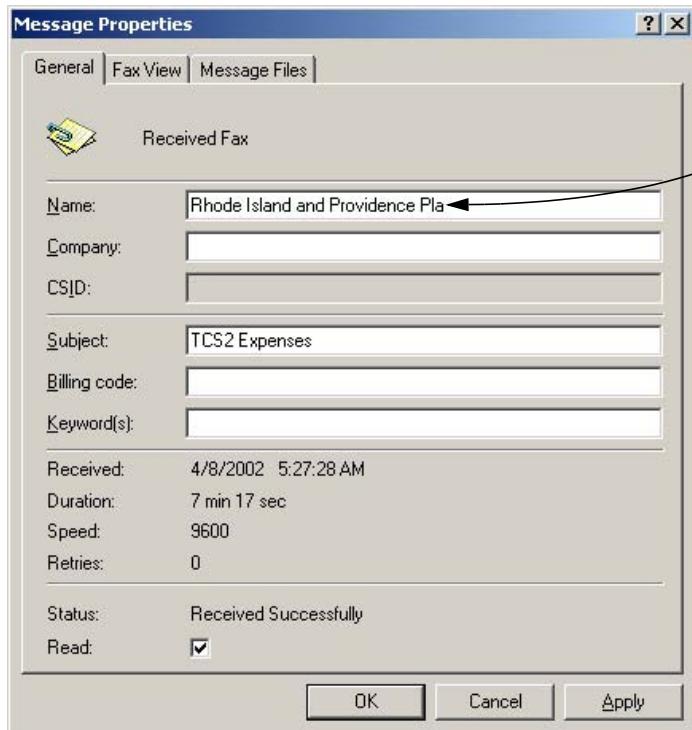


Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 32

The Lying Fixed Width Edit Control Keyhole

From Symantec's Winfax Pro 10:



Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 33

The Fixed-Width Web Page Keyhole



Examples on the bottom employ children's "reduced vegetable density" strategy.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 34

The Fixed-Width Web Page Keyhole

Inconsistency manifestation I: from **Ebay's** web site:

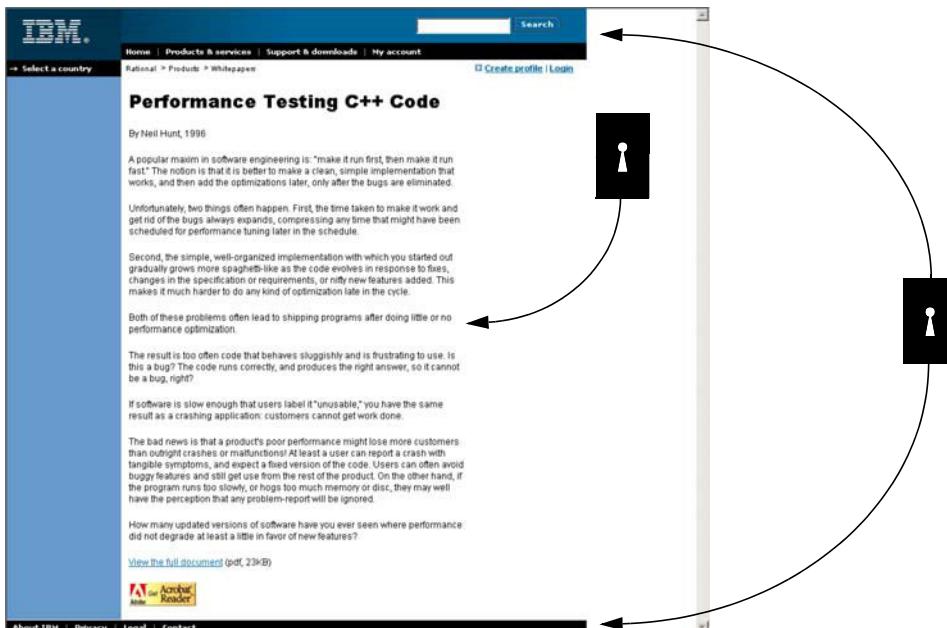


Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 35

The Fixed-Width Web Page Keyhole

Inconsistency manifestation II: from **IBM's** web site:



Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 36

The Fixed-Width Web Page Keyhole

The Massachusetts Convention Center Authority's web site confuses "looks good at" with "looks good *only* at":

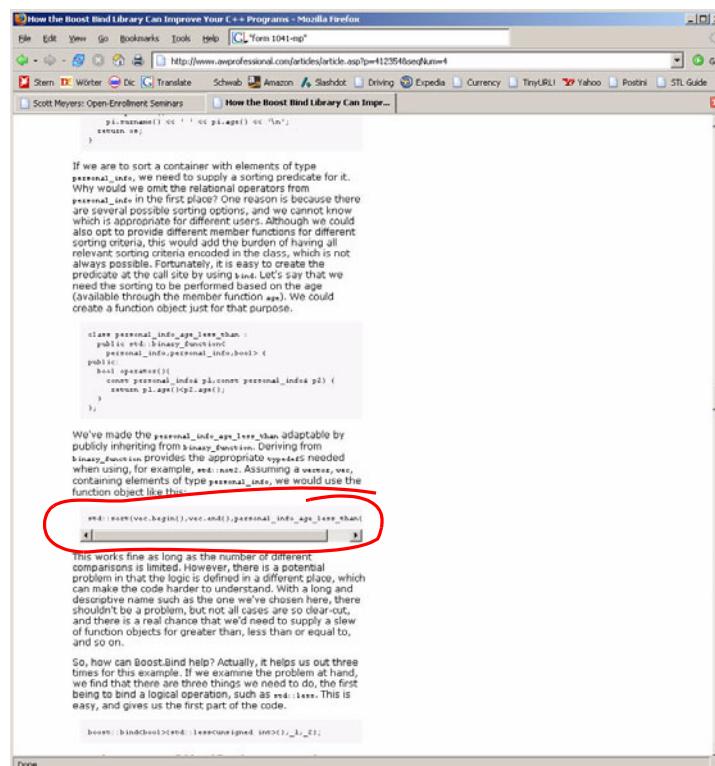


Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 37

The Fixed-Width Web Page Keyhole

Addison-Wesley fixes content width, then adds embedded scroll bars for too-long lines!



Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 38

The Fixed-Width Web Page Keyhole

Artima is one of many sites that let browsers do what they are designed to do — determine line lengths dynamically:

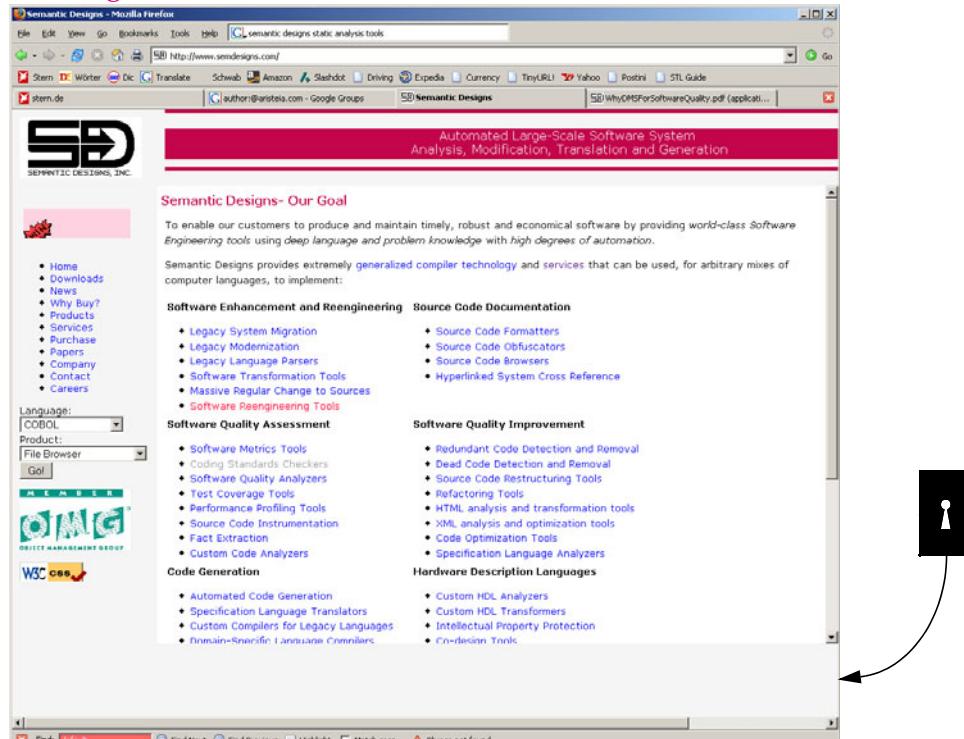


Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 39

The Fixed-Height Web Page Keyhole

From Semantic Designs' web site:



- The keyhole is present under Firefox 1.07, absent under Internet Explorer 6.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 40

Invisible Keyholes (Non-GUI Components)

Important: **Invisible ≠ Undetectable!**

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

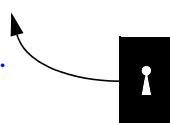
Copyrighted material, all rights reserved.
Page 41

The Too Few Bits Keyhole

From Roxio's web site (but the software is due to **Adaptec**):

If you have Easy CD Creator 4.00 and have more than 8Gb of free space, please create TEMP directories and fill up space in these until less than 8Gb of free space remains.

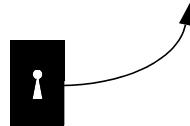
8GB is 33 bits...



The Unnecessarily Signed Int Keyhole

Naviscope's Naviscope status report:

Naviscope: Active, 10 Prefetched, -4116 Ads Blocked

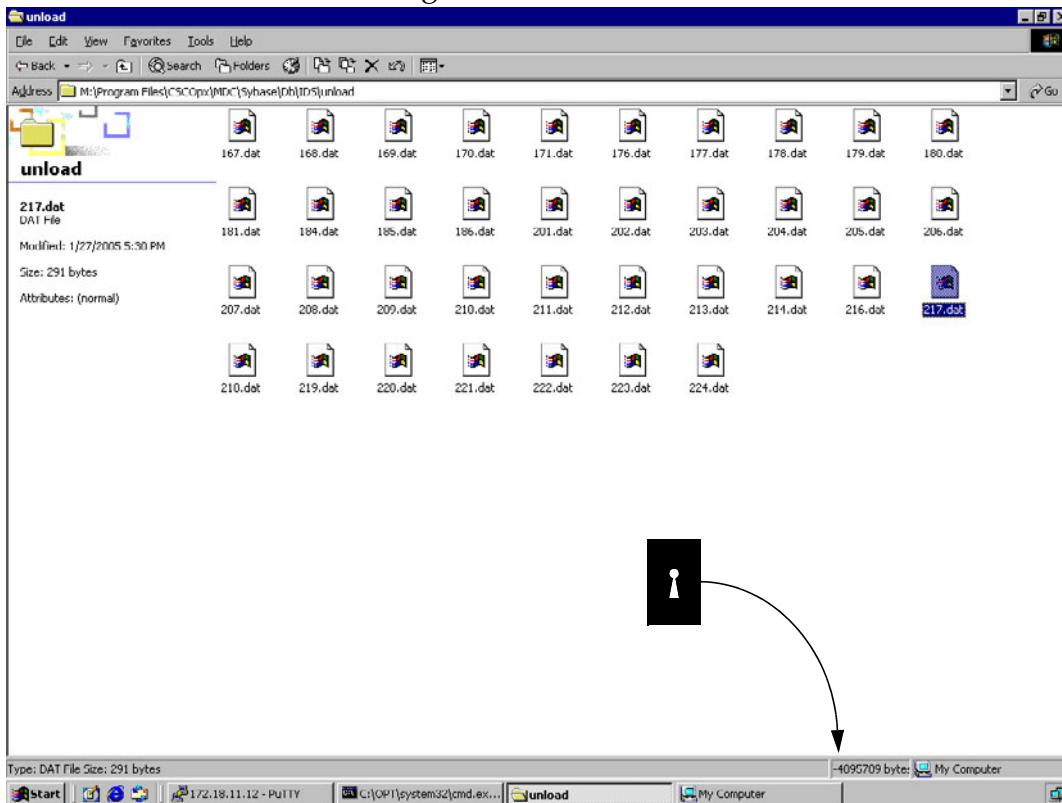


Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 43

The Unnecessarily Signed Int Keyhole

Microsoft, too, has trouble counting. From Windows 2000:



Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 44

The Restricted Field Size Keyhole

A phrase to index using Adobe's FrameMaker 6:

Prefer iterator to const_iterator, reverse_iterator, and const_reverse_iterator.

I want several different index entries:

iterator: vs. other iterator types
const_iterator: vs. other iterator types
reverse_iterator: vs. other iterator types
const_reverse_iterator: vs. other iterator types
iterators: choosing among types
containers: choosing among iterator types

With markup, here's the index entry I want to make:

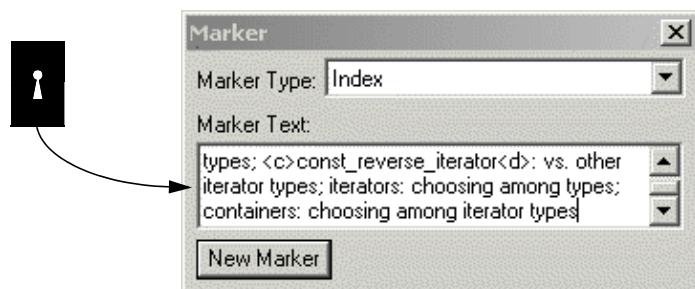
```
<$startrange><c>iterator<d>; vs. other iterator types;  
<c>const_iterator<d>; vs. other iterator types; <c>reverse_iterator<d>; vs.  
other iterator types; <c>const_reverse_iterator<d>; vs. other iterator  
types; iterators: choosing among types; containers: choosing among  
iterator types
```

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 45

The Restricted Field Size Keyhole

FrameMaker makes me type it into this lovely keyhole,

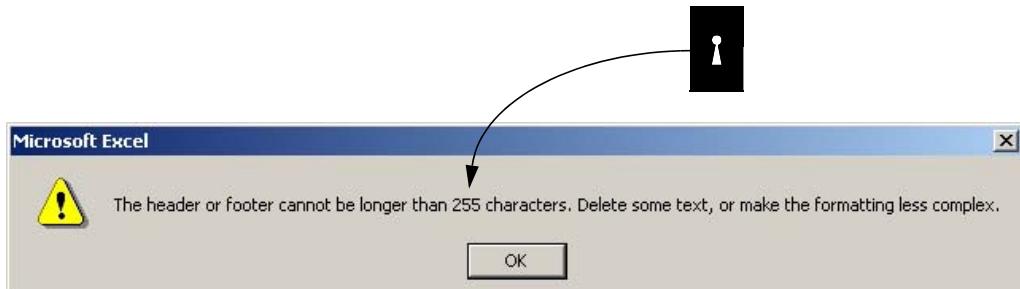


but when I try to add it to the document...



The Restricted Field Size Keyhole

Microsoft's Excel 2002 has a similar limitation:



PowerPoint 2003 limits the size of an outline it will import (to 249 slides):

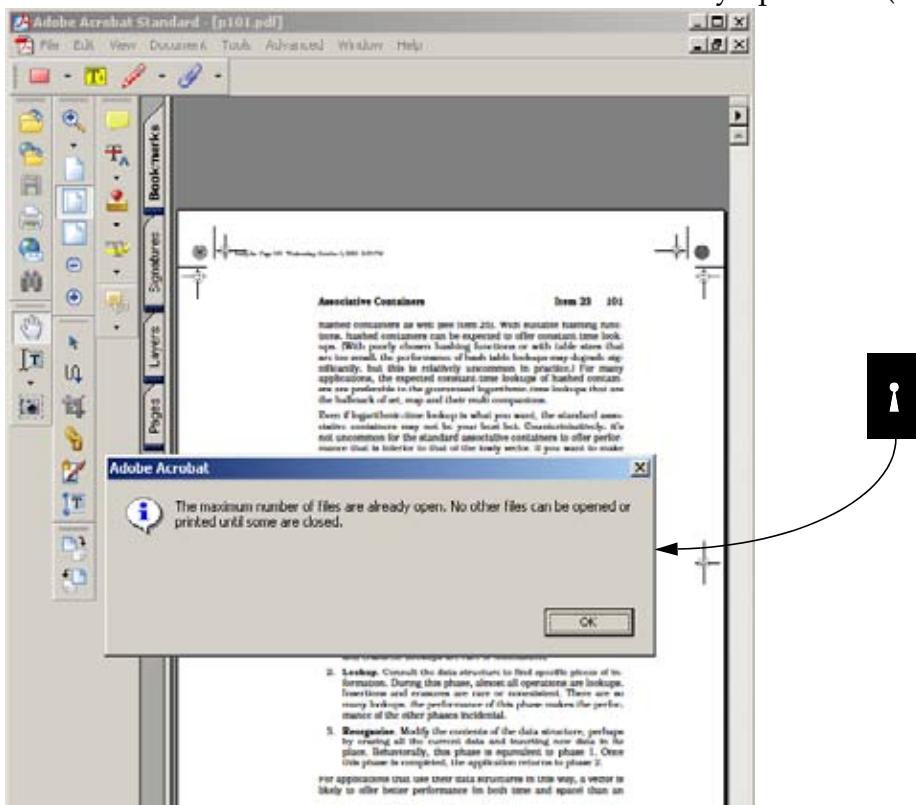


Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 47

The Restricted Field Size Keyhole

Adobe's Acrobat 6 limits the number of simultaneously open files (to 20):



Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 48

The Restricted Field Size Keyhole

From Microsoft's Visual C++ 6:

Compiler Warning C4786

The debugger cannot debug code with symbols longer than 255 characters.

From Microsoft's Visual Studio .NET Readme:

Setup fails if any path and file name combination exceeds 260 characters.
The maximum length of a path in Visual Studio is 221 characters;
accordingly, you should copy files to a path with less than 70 characters.

If you create a network share for a network image, the UNC path
to the root install location should contain fewer than 39 characters.

Annotations in Adobe Acrobat 6 are limited to 5000 characters.

Nostalgia:

- Local phone numbers will never have more than 7 digits.
- Two digits will always suffice to identify the year.

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 49

The Restricted Domain Keyhole

Unbelievably common for user IDs and passwords, often in conjunction with field size keyholes. From Ameritrade:

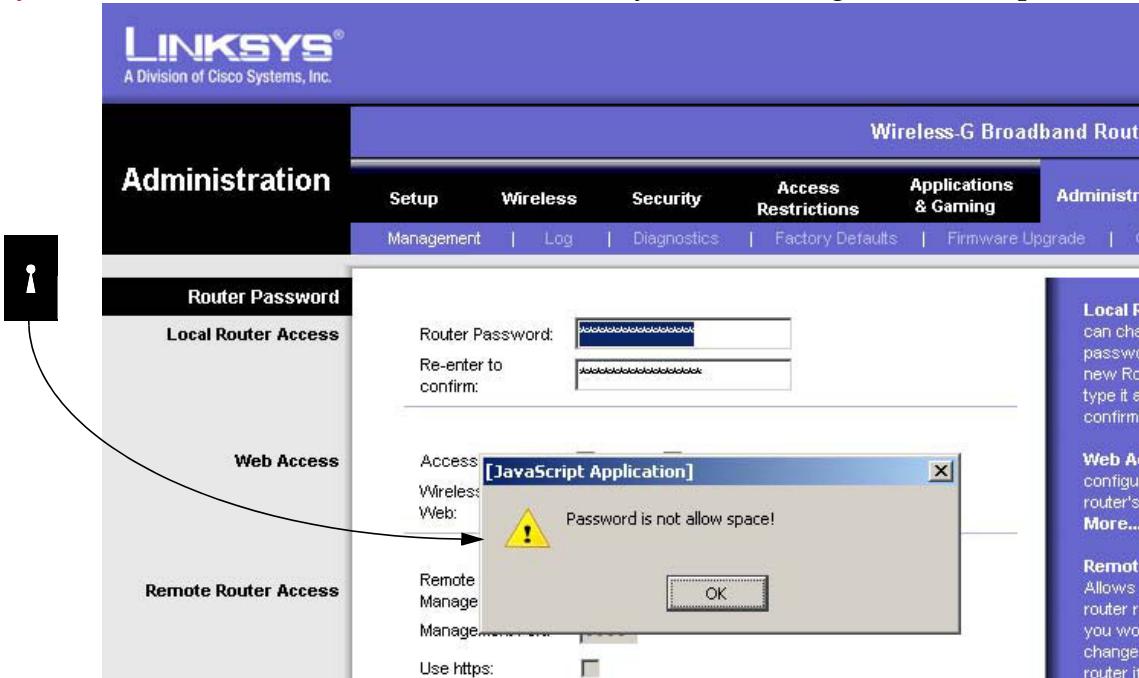
Your PIN...
1. Must be a 4-digit number
2. Cannot begin with a zero
3. Cannot include letters or symbols

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 50

The Restricted Domain Keyhole

Linksys's WRT54G router software combines a keyhole with linguistic incompetence:



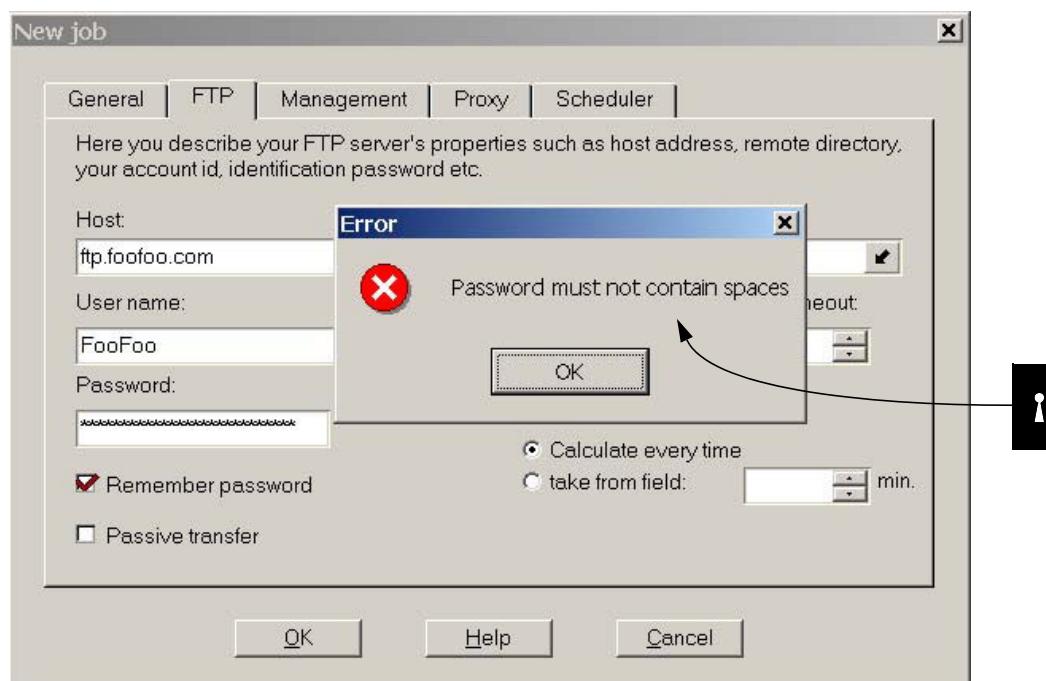
- But my former Linksys router *had* a space in the password!

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 51

The Restricted Domain Keyhole

Rifco's DC SmartFTP's keyhole actually keeps the program from working!



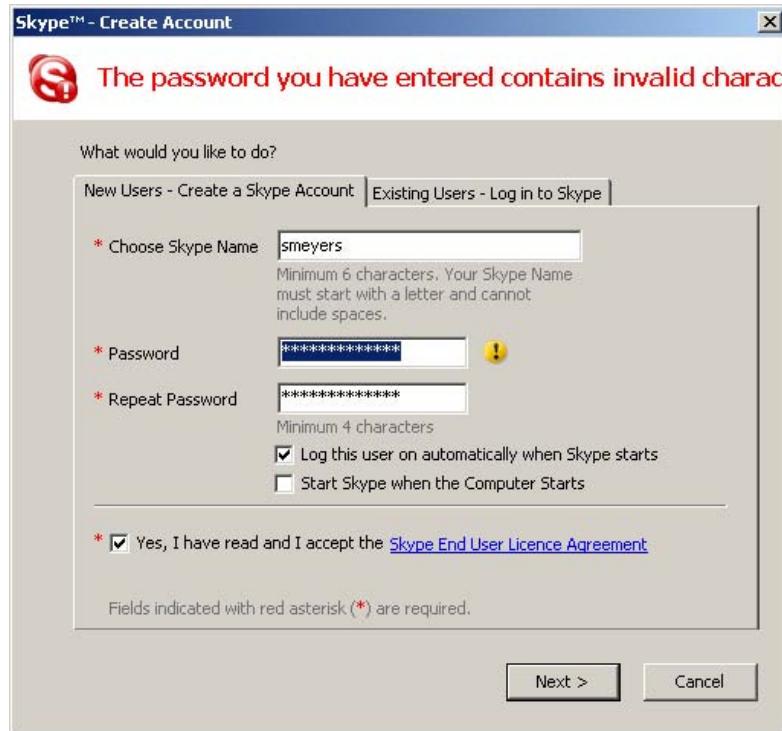
- Most sites I FTP to have passwords with spaces in them!

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 52

The Restricted Domain Keyhole

When Skype's error message about a restricted domain meets a fixed-size dialog:



Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 53

The Restricted Domain Keyhole

From 1800flowers.com's web site:

A screenshot of a web form titled 'Enter a Card Message'. It has several input fields and associated notes:

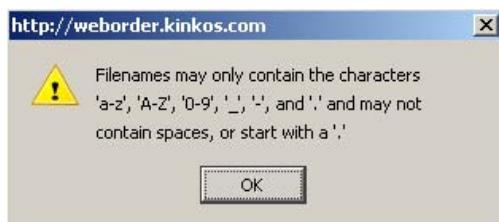
- greeting type:** A dropdown menu labeled 'Click to select...' with an arrow pointing to it.
- greeting:** An input field containing 'To all our good buddies out at'.
- message:** A large text area with the note 'There is a 150 character limit for your message due to the size of the card. Note: please do not use any special symbols like "&".' An arrow points from this note to the message area.
- signature:** An input field containing 'Fred, Wilma, Barney, Betty, an'.
- At a loss for words?** A note in a callout box: 'We can help you express your sentiments. [Click here to view](#)'.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 54

The Restricted Domain Keyhole

Look what FedExKinko's has to say *four screens after the data is entered*:



- No spaces allowed in a Windows filename?

Keyholes by Induction

If 1 is good, n must be better.

Keyholes Everywhere

Here's my environment (PATH is highlighted):

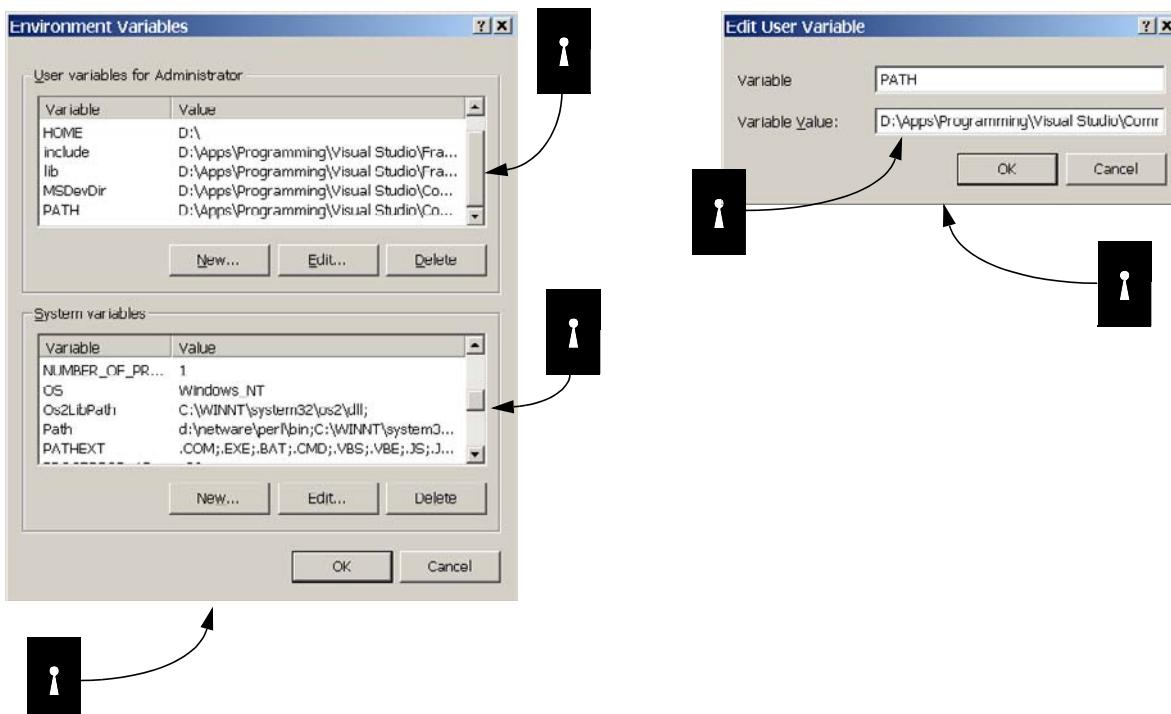
```
Sc Select Command Prompt
ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\Administrator\Application Data
CommonProgramFiles=.:Program Files\Common Files
COMO_MS_INCLUDE=D:\Apps\Programming\Visual Studio\VC98\Include
COMPUTERNAME=ATALANTA
ComSpec=C:\WINNT\system32\cmd.exe
C:\Windows\TEMP=d:\apps\programming\MWCW 6
HOME=D:\\
HOME DRIVE=C:\\
HOME PATH=\\
INCLUDE=D:\Apps\Programming\Visual Studio\FrameworkSDK\include;D:\Apps\Programming\Visual Studio\VC98\atl\include;D:\Apps\Programming\Visual Studio\VC98\mfc\include;D:\Apps\Programming\Visual Studio\VC98\include;D:\C++\boost\boost_1_27_0
LIB=D:\Apps\Programming\Visual Studio\FrameworkSDK\Lib;D:\Apps\Programming\Visual Studio\VC98\mfc\lib;D:\Apps\Programming\Visual Studio\VC98\lib
LM_LICENSE_FILE=d:\apps\programming\MWCW 6\license.dat
LOGONSERVER=\\ATALANTA
MSDevDir=D:\Apps\Programming\Visual Studio\Common\MSDev98
MWCIncludes=d:\apps\programming\MWCW 6\MSL\MSL_Common\Include;D:\apps\programming\MWCW 6\MSL\MSL_Common\Include;D:\apps\programming\MWCW 6\MSL\MSL_Win32\Include;D:\apps\programming\MWCW 6\Win32-x86\Support\Headers\Win32
SDK=d:\apps\programming\MWCW 6\MSL\MSL_C\MSL_X86;D:\C++\boost\boost_1_27_0
MWC Libraries+=D:\apps\programming\MWCW 6\MSL;D:\apps\programming\MWCW 6\Win32-x86\Support\Libraries
MWC library files=sdI32.lib;user32.lib;kernel32.lib;ansicx86.lib;ansicppx86.lib;MWCRTI.lib
NUMBER_OF_PROCESSORS=1
OS=Windows_NT
Os2LibPath=D:\WINNT\system32\os2.dll;
Path=D:\netware\per\bin;C:\WINNT\System32;C:\WINNT\System32\WHEM;C:\PROGRAM FILES\THINKPAD\UTILITIES;D:\Netware\Borland 5.5\bin;C:\Program Files\Microsoft SQL Server\80\Tools\BINN;D:\apps\programming\MWCW 6\Bin;D:\apps\programming\MWCW 6\Other Metrowerks Tools\Command Line Tools;D:\Apps\Programming\Visual Studio\Common\Tools\WinNT;D:\Apps\Programming\Visual Studio\Common\MSDev98\Bin;D:\Apps\Programming\Visual Studio\Common\Tools;D:\Apps\Programming\Visual Studio\VC98\bin;D:\Apps\Programming\Comeau_C++\bin;D:\Netware\g++\Bin;D:\Netware\Emacs\GnuServ;D:\Netware\Emacs\emacs-20.7\bin;D:\Bin
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
PROCESSOR_ARCHITECTURE=x86
PROCESSOR_IDENTIFIER=x86 Family 6 Model 8 Stepping 10, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=080a
ProgramFiles=C:\Program Files
PROMPT=$PSG
SystemDrive=C:
SystemRoot=C:\WINNT
TEMP=C:\WINNT\TEMP
TMP=C:\WINNT\TEMP
USERDOMAIN=ATALANTA
USERNAME=Administrator
USERPROFILE=C:\Documents and Settings\Administrator
VSCOMMONTOOLS=D:\Apps\Programming\Visual Studio\Common\7\Tools\
windir=C:\WINNT
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 57

Keyholes Everywhere

Here is Microsoft's Windows 2000's interface for editing it:



Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Copyrighted material, all rights reserved.
Page 58

Keyholes Everywhere

From Crucial Technology's web site:

The screenshot shows a Microsoft Internet Explorer window displaying the Crucial Technology website. The main page header features the company logo and the slogan "The Memory Experts". Below the header, there are links for "store/home", "company", "library", "help", and "log in". A navigation bar at the bottom includes "International Customers | Order Status | FAQ | Shopping Cart | Contacts | Site Map".

A large black keyhole icon is overlaid on the screen, with arrows pointing from it to various parts of the interface: the "Expert Online" form, the "Motherboard Specifications" table, the "Shipping Rates" box, the "Need Help?" box, and a warning dialog box.

Expert Online - Microsoft Internet Explorer

Type your question below:

I could add a 512MB module to a system currently holding two 128MB modules. Is that a correct assumption?

Submit Exit

Motherboard Specifications: SUPER P6SBA

• Chipset: Intel 440BX	• Accepts Parity: N
• Accepts PC133 SDRAM: Y	• DIMM Socket Count: 3
• Accepts SDRAM 100MHz: Y	• Sockets per DIMM Bank: 1
• Accepts SDRAM 66MHz: N/A	• 72-pin SDRAM Socket Count: N/A
• Accepts Registered SDRAM: Y	• Sockets per 72-pin SDRAM Bank: N/A
• Accepts EDO: Y	• 30-pin SDRAM Socket Count: N/A
• Accepts Fast Page Mode: N	• Sockets per 30-pin SDRAM Bank: N/A
• Max EDO/FPM: N/A	• Buffering: Unbuffered
• Accepts DDR: N/A	• Max 1 Unbuffered SDRAM: 284 MR
• Accepts ECC: Y	
• Accepts RDRAM (Rambus)	

Shipping Rates

FedEx 2nd Day	FREE
Oversize (arrives 4:30 pm) ...	\$11.50
All domestic rates	click here
All international rates	click here

Need Help?

Currency Converter
Explanation of Memory Terms

Microsoft Internet Explorer

Question should not exceed 255 characters.

168-pin DIMM (picture)

Module Size	Price (ea.)	V	Buy	Part Details
256MB	\$62.99	\$66.89	<input type="checkbox"/>	CT132M72S4R8E SDRAM, PC100 • CL=2 • Registered • ECC • 8ns • 3.3V • 32Meg x 72
120MB	\$25.99	\$23.39	<input type="checkbox"/>	CT16M64S4D75 SDRAM, PC133 • CL=3 • Unbuffered • Non-parity • 7.5ns • 3.3V • 16Meg x 64
128MB	\$25.99	\$23.39	<input type="checkbox"/>	CT16M64S4D8E SDRAM, PC100 • CL=2 • Unbuffered • Non-parity • 0ns • 3.3V • 16Meg x 64
128MB	\$29.99	\$26.99	<input type="checkbox"/>	CT16M72S4D75 SDRAM, PC133 • CL=3 • Unbuffered • ECC • 7.5ns • 3.3V • 16Meg x 72
128MB	\$29.99	\$26.99	<input type="checkbox"/>	CT16M72S4D8E SDRAM, PC100 • CL=2 • Unbuffered • ECC • 8ns • 3.3V • 16Meg x 72
120MB	\$30.99	\$27.09	<input type="checkbox"/>	CT16M64S4D7E SDRAM, PC133 • CL=2 • Unbuffered • Non-parity • 7.5ns • 3.3V • 16Meg x 64

Copyrighted material, all rights reserved.
Page 59

Beyond Inconvenience

What happens when the log exceeds 65.535" in width?



"It was a very serious bug. It really could have killed someone."

Beyond Inconvenience

Fixed-size buffers + programmer incompetence or oversight = Hackers' Delight.

- Code Red and Code Red II worms

A quick Google search yields reported buffer overrun vulnerabilities all over the place:

- **gzip** under **Linux**
- **xdat** under **AIX**
- **count.cgi** everywhere
- MIT's **Kerberos**
- SGI **df**, **pset**, and **eject** under **IRIX**
- HP's **Openview Network Node Manager Alarm Service**
- Microsoft **Clip Art Gallery**
- **Microsoft Index Server 2.0** under **Windows NT**.
- **rpc.espd** daemon under **IRIX**
- **xterm** under **OpenBSD**
- **statd** under **Unix**
- **talkd** under **HPUX**
- **NTP** under **NetBSD**
- **AOL Instant Messenger** under **Windows**
- Microsoft **Windows Media player**
- Microsoft **HyperTerminal**
- Microsoft **SQL Server**
- Microsoft **Internet Explorer** under **MacOS**
- Microsoft **IIS**
- TalentSoft's **Web+**
- CDE's **dtspcd** daemon under **Unix**
- **login** under System-V-derived **Unix**
- Microsoft's **C Runtime Library**
- Apple's **sudo** under **MacOS**

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 61

Beyond Inconvenience

San Francisco, November 5, 2004:



Unexpectedly high voter turnout caused vote-counting software to crash.

- Cause: the amount of data exceeded a pre-set limit.
- The vendor later determined that the limit was not necessary.

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 62

Beyond Inconvenience

The stakes can be high...

- March 1979:
TMI-2 comes
within 30
minutes of
total meltdown
- Contributing
factor:
A steam
temperature
readout is
programmed
never to
display values
over 280°F
 - ⇒ This is a
“Restricted
Range”
Keyhole.



Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 63

Topics Revisited

What are “keyholes?”

- Primarily *gratuitous* restrictions on what a user can see or express.
 - ⇒ Not all restrictions are gratuitous.
 - ⇒ Length restrictions on usernames are typically justifiable, on passwords typically not.
- Often:
 - ⇒ **Use of a constant where a variable would be better.**
 - ⇒ **Imposition of a constraint where none is warranted.**
- Missing features are typically *not* due to keyholes:
 - ⇒ Keyholes generally keep you from seeing everything at once.
 - ⇒ Missing features are not visible at all.
- Not all usability problems are due to keyholes.
- Not all keyholes lead to what are usually considered “usability issues.”

Topics Revisited

Why are keyholes important?

- They typically degrade a system's usability.
 - Users can't see what they want to see.
 - Users can't express what they want to express.
 - Users must cope with inconsistent behavior.
 - Users are frustrated, unhappy, and "unloyal" to a product or system.
- They make systems brittle in the face of change:
 - "Obviously reasonable" constants become unreasonable (e.g., PDA meets SXGA)
 - "Obviously sufficient" limits become insufficient (e.g., Warning C4786)
- They can lead to serious security and safety vulnerabilities.

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 65

Topics Revisited

How can we eliminate or avoid keyholes?

- **Give them "a seat at the table"**
 - Along with functionality, schedule, performance, etc.
- During design and development, avoid imposing gratuitous restrictions.
 - Determine things dynamically instead of statically.
 - If constraints must be imposed, make them as lax as possible.
 - Reject components that impose keyholes.
- During web site design and implementation:
 - Follow the advice above.
 - When in doubt, copy Amazon.
 - When not in doubt, check Amazon anyway.
- Before putting something somewhere, make sure it will fit!

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 66

But Everybody Uses Keyholes!

Listen to your mother:

- “If everybody jumped off a cliff, would you jump, too?”

“Different” can be a synonym for “competitive advantage.”

Scott Meyers, Software Development Consultant
<http://www.aristeia.com/>

Copyrighted material, all rights reserved.
Page 67

More on Keyholes

- [The Keyhole Problem Web Site](http://www.aristeia.com/TKP/), <http://www.aristeia.com/TKP/>.
 - ⇒ Draft chapters of a book I’m supposed to be working on.
 - ⇒ Information on a mailing list on keyholes.
- [“Bugopedia,”](#) Niall Murphy, *Embedded Systems Design*, December 2005.
 - ⇒ His “Jurassic Park bug” is my Restricted Range keyhole (e.g., TMI).

Please Note

Scott Meyers offers consulting services in all aspects of the design and implementation of software systems. For details, visit his web site:

<http://www.aristeia.com/>

Scott also offers a mailing list to keep you up to date on his professional publications and activities. Read about the mailing list at:

<http://www.aristeia.com/MailingList/>

Leading Change: Collaboration and Collaborative Leadership

Pollyanna Pixton Evolutionary Systems

It is no longer enough to respond to change—organizations must lead change or be left behind. Innovation and creativity is essential, as are collaborative leaders to create environments that stimulate powerful ideas and deliver quality products and services.

Pollyanna will discuss how to lead and create collaborative environments where adaptive practices emerge and thrive. Such practices embrace change and deliver the right products and services at the right time. She will address how to integrate business value and ensure quality and marketable results.

Presenting models from her think tank on innovation and leadership, Pollyanna will discuss how collaboration and collaborative leadership delivered the Swiss Electronic Stock Exchange, power grid control systems at Asea Brown Boveri, and rolled out the Homeless Information Management System in Utah.

Pollyanna Pixton is widely recognized for her ability to lead the collaborative efforts of talented people who want to expand what they already do very well to being even better. She works with leaders and teams inside corporations and organizations to improve their productivity and effectiveness to achieve lasting results using collaboration. She founded Evolutionary Systems in 1996, and brings over 35 years of executive and managerial experience from a variety of successful business and information technology ventures to her work as a consultant. As the result of a think tank she formed two years ago to address how to improve innovation in today's organization, she co-founded and directs the Institute of Collaborative Leadership.

She was primarily responsible for leading the development of the Swiss Electronic Stock Exchange, developing sophisticated control systems for electrical power plants throughout the world, and merging the complex technologies and data systems of large financial institutions. Her background includes leading the development of complex e-commerce projects, real-time applications, positioning systems, and researching computational methods in theoretical physics.

Pixton co-founded the Agile Project Leadership Network (APLN) and serves on that Executive Board. She is chairing the ADC 2006 Leadership Summit, and at Agile 2005, she presented a tutorial and workshop on collaborative leadership. In 2004, she chaired the ADC Executive Summit and led the Getting Leaders Onboard workshop at XP Universe with Mary Poppendieck.



L e a d i n g C h a n g e: Collaboration and Collaborative Leadership



P o l l y a n n a P i x t o n

President, Evolutionary Systems
Director, Institute for Collaborative Leadership
ppixton@evolutionarysystems.net



Why Collaborate?

- Unleash Innovation
- Increase Business Value and Quality
- Answers are in Your Organization
- Lead Change



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Implement and Review

Unleashing Innovation



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Unleash Innovation

"The way you will thrive in this environment is by innovating – innovating in technologies, innovating strategies, innovating business models."

- IBM CEO Samuel J. Palmisano

[BusinessWeek, April 24, 2006]

Institute for Collaborative Leadership

Evolutionary Systems Consulting



Unleash Innovation

How?



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Unleashing Innovation

- Collaboration Model
- Collaboration Process
- Collaborative Leadership



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Unleashing Innovation

Collaboration Model



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration Model

Create an Open Environment



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration Model

**Convene the Right People
From the *Entire Enterprise!***

Customers
Marketing
Sales
Finance
Technology
Manufacturing
Stakeholders



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration Model

**Foster
Creativity &
Innovation
via
Collaboration Process**



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration Model

**Stand back,
Let
Them
Work.**



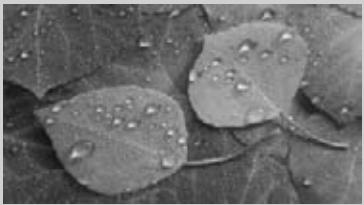
Institute for Collaborative Leadership

Evolutionary Systems Consulting



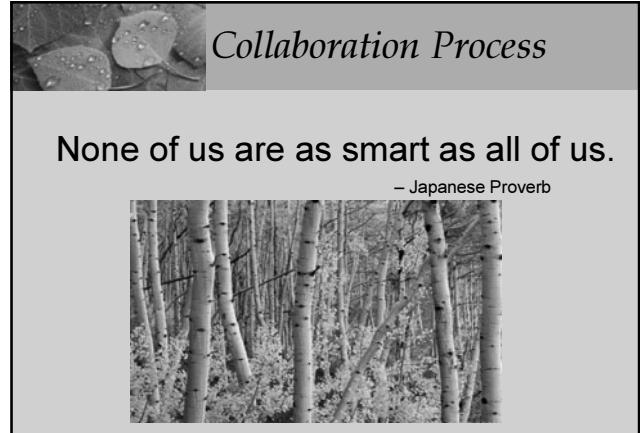
Unleashing Innovation

Collaboration Process



Institute for Collaborative Leadership

Evolutionary Systems Consulting



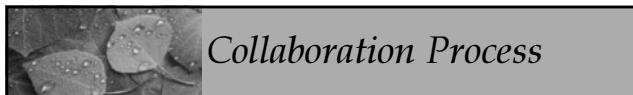
Collaboration Process

None of us are as smart as all of us.
— Japanese Proverb



Institute for Collaborative Leadership

Evolutionary Systems Consulting



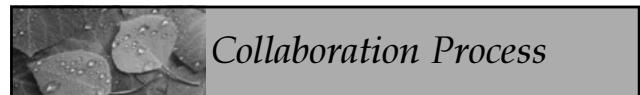
Collaboration Process

1. Agree to:
 - Goals
 - Objectives
 - Purpose



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration Process

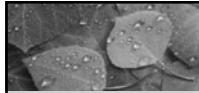
2. Brainstorm
3. Group
4. Prioritize

Based on Business Value



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration Process

5. Individuals
Volunteer
For What
And
By When



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration Exercise

■ How Do We Integrate

- Quality,
- Business Value, and
- Innovation?



➡ How do we do this?

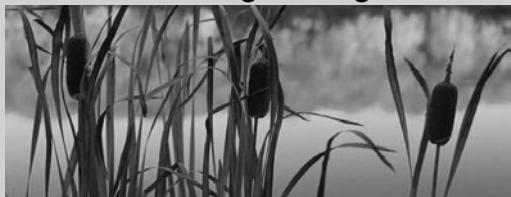
Institute for Collaborative Leadership

Evolutionary Systems Consulting



Leadership Model

Leading Change



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Leadership Model

"It's no longer enough to respond to change; today organizations must lead change or be left behind."

- Pollyanna Pixton

Institute for Collaborative Leadership

Evolutionary Systems Consulting



Unleashing Innovation

Collaborative Leadership



Institute for Collaborative Leadership

Evolutionary Systems Consulting

Collaborative Leadership

The Right People



Institute for Collaborative Leadership

Evolutionary Systems Consulting



The Right People

Hire and promote:

- First on the basis of integrity
- Second, motivation
- Third, capacity
- Fourth, understanding
- Fifth, knowledge
- Last and least, experience



- Dee Hock, CEO Emeritus VISA International

Institute for Collaborative Leadership

Evolutionary Systems Consulting



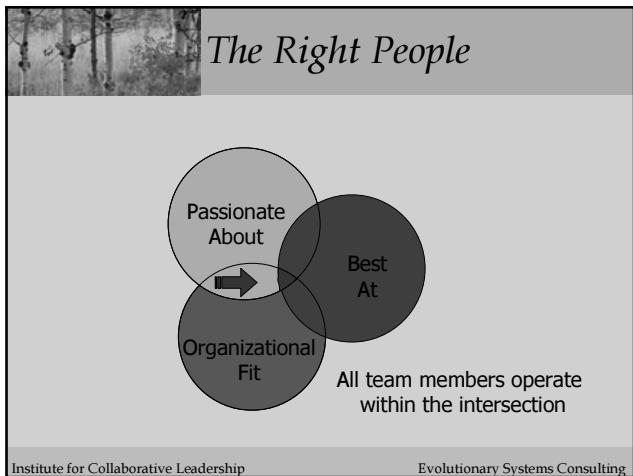
The Right People

- Authenticity
- Attitude
- Intelligence
- Talent



Institute for Collaborative Leadership

Evolutionary Systems Consulting



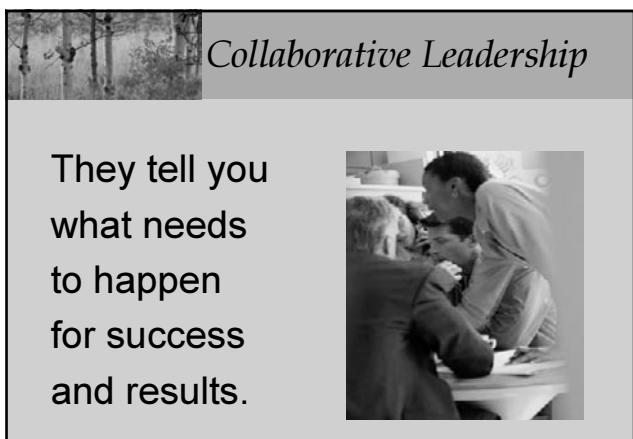
Institute for Collaborative Leadership

Evolutionary Systems Consulting



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaborative Leadership

Leading Collaboration



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Leading Collaboration

Ricardo Semler, CEO of Semco, believes that all people desire to achieve excellence and that autocracy dampens people's creativity and motivation.

- *The Seven-Day Weekend*

Institute for Collaborative Leadership

Evolutionary Systems Consulting



Leading Collaboration

Influence Not Authority



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Leading Collaboration

**Keep
the purpose
and
the vision
alive**



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Leading Collaboration

**Everyone
has full and
entire view:
“Total
Transparency”**



Institute for Collaborative Leadership

Evolutionary Systems Consulting

Leading Collaboration

**Take the ‘fun’
out of
being
dysfunctional**



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Leading Collaboration

**No Such Thing
As
‘Constructive
Criticism’**



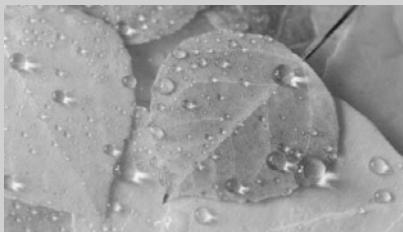
Institute for Collaborative Leadership

Evolutionary Systems Consulting



Leading Collaboration

**Free team to question, analyze
and investigate.**



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Leading Collaboration

The opposite of control is discovery



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Leadership Focus

Does everyone have what they
need to succeed?

Institute for Collaborative Leadership

Evolutionary Systems Consulting



Leadership Focus

How do I create a place where
people want to be?

Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration

Collaboration and Quality



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration For Quality

**1. Define Quality
Goals
and
Business
Goals**



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration For Quality

**2. Teams
Develop
Strategies**



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration For Quality

**3. Review
Strategies
With Goals.
Inline?**



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration For Quality

**4. Teams
Generate
Action Plans**



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration For Quality

Iterate!
Every
Two Weeks or
Every Month



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration For Quality

- Who are the Right People?
- How do we prioritize by Business Value and Quality?
- How do we recognize and find the balance?



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration For Quality

What is the **Business Value** of **Quality**?



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Summary

To Unleash
The Innovation to
Apply to Quality
And Lead Change
➡ Collaborate



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration Model

- Open Environment
- Right People
- Foster Innovation:
Collaboration Process
- Step Aside



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Collaboration Process

- Agree to Goal
- Brainstorm
- Group
- Prioritize
- Individuals volunteer



Institute for Collaborative Leadership

Evolutionary Systems Consulting



Leadership Model

- The Right Talent
- Trust First!
- Let them tell you what they need to do to be successful
- Stand back!



Institute for Collaborative Leadership

Evolutionary Systems Consulting



References

- *Good to Great*, Jim Collins
- *The Seven-Day Weekend*, Ricardo Semler
- *Leadership Is An Art*, Max DePree
- *Organizing Genius, The Secrets of Creative Collaboration*, Warren Bennis
- www.collaborative-leadership.com

Institute for Collaborative Leadership

Evolutionary Systems Consulting



Contact

Pollyanna Pixton:

- www.evolutionarysystems.net
- www.collaborative-leadership.com
- www.apln.org
- 801 . 209 . 0195
- P2@ppixton.com

Institute for Collaborative Leadership

Evolutionary Systems Consulting

Software Testing In An Agile World

Paul Hemson

QA Director

McAfee

20460 NW Von Neumann Drive

Beaverton, Oregon 97006

Abstract

The introduction of any new development methodology within a project brings risk and potentially turmoil. A great deal has been written about agile software development, but mostly from the perspective of a development team planning on adopting this methodology. There is very little information available to guide a testing organization. Most of the testing related material is focused either on unit testing, or on “context based testing”. Little of this information will help in planning an effective testing strategy for an agile project, in fact much of the agile literature leads the reader towards either a risky, or a very ineffective testing strategy.

This paper presents what can be either nine easy steps, or nine hard lessons to consider when planning the system testing on an agile project. Awareness of these lessons will help ensure a much smoother adoption of agile techniques, both during planning and execution. These lessons were learnt over the course of several projects, often painfully, as McAfee experienced some of the traps that in many cases agile techniques steer testing organizations towards.

The paper considers the complete agile development lifecycle, offering important lessons in areas such as:

- Creating the high level test plan, or testing strategy, that will be used to guide the testing activities on a project.
- Ensuring that the test team has appropriate expectations, assignments and deliverables during user story planning.
- Identifying and avoiding the gaps and risks which a purely user-story focused testing strategy tends to produce.
- Recognizing at an early stage if an agile project is headed for a death-march at later phases, identifying what can be done to both highlight and mitigate associated risks.
- Effectively planning testing time and resources for the final stages of a project.

A pragmatic application of the lessons presented here will help reduce the stress of interpreting how agile techniques should affect the testing organization, resulting in a more efficient and effective application of agile techniques within any organization.

Biography

Paul Hemson has spent the past 15 years in various QA leadership and management roles at Mentor Graphics, Intersolv, Cadence, and for the past 4 years as a QA Director at McAfee, Inc. in Beaverton. At McAfee he is responsible for a large, global QA organization. Prior to his QA roles he spent 6 years in software support and development positions at Mentor Graphics. He holds a BS in Electrical Engineering from Swansea University (UK).

Introduction

Much has been written about agile software development and how the application of its lightweight processes can produce improvements in some combination of delivery speed, schedule predictability, customer responsiveness, or quality. However, in spite of the volume of agile literature, there is very little published information relating to the system testing side of software testing in an agile environment. Instead, the available information on software testing focuses either on unit testing, or on “context-based” approaches. Both of these are relevant and valuable for certain aspects of the testing, but as any Test Manager or Test Lead responsible for planning the system testing on an Agile project will find, they offer little practical guidance on how to plan and run that system testing effort.

The situation is actually worse than this, since much of the agile documentation is likely to direct an otherwise sound system testing effort into potential problems. For example, most agile literature is very development centric and suggests, either explicitly or implicitly, that all testing should adopt the same user-story centered approach that the development organization will take. To some extent this may be true, but relying exclusively on this kind of an approach is likely to result in an ineffective or inefficient system testing effort.

At McAfee, agile techniques have been introduced into several project teams, and consequently we fell into many of the traps that any system testing organization is likely to encounter. Over time we have learned the lessons, and refined the details into the nine easy steps, or nine hard lessons, outlined in this paper.

The lessons are aimed primarily at the Test Manager or Test Lead, and they represent things which should be considered as the testing strategy for a project is being defined. Many of these items will become less of an issue as a team becomes more experienced with agile techniques, but on the first couple of agile projects they represent much larger risks, and are likely to be a lot more pronounced than on later ones.

As well as being directly applicable to the system testing group, these lessons are also relevant to other functional groups on the project, either as things to be aware of, as tradeoffs that should be happening, or as potential risks that need to be minimized.

Lesson 1: Make the Most of the Planning Meetings

All projects suffer to some extent from a lack of documentation, whether through missing or incomplete product requirements, specifications, or design related documents. Agile projects are no different in this respect. However they are different in that they provide a clear opportunity to gather and clarify this information, this opportunity is during the planning phase of each iteration.

Although the exact duration of an iteration, and of the phases within that iteration, will vary with individual implementations, agile teams always have a defined planning phase at the start of each iteration. Typically this will consist of 1-2 weeks of planning meetings to prepare for a 2-4 week coding phase. During these planning meetings the user stories are defined, refined, and updated.

With this pre-defined planning phase, agile teams typically spend more time planning than teams following most other development methodologies. The testing teams must get as much value from these meetings as possible, since this planning phase is the best opportunity to gather the information needed to define effective tests, both for the individual user stories and for the system as a whole.

The best way to ensure this information is gathered, and that testing related risks are identified, is to assign specific deliverables to the Testers attending the meetings. This means that attendance is an active not a passive activity. Attendance involves asking clarifying questions, making detailed notes, reviewing the notes for completeness and comprehension, and ensuring all information needed to design effective tests is collected. When the meetings are complete it will be necessary to design detailed tests, and these notes will need to be a reliable record of the rationale for a particular implementation, the risks associated with each user story, appropriate acceptance tests, and other complexities or assumptions outlined during the planning meetings.

Lesson 2: Work With the Developers to Write Good User Stories

Lesson two is closely related to the first lesson and the active role required from QA in the planning meetings. On the first couple of agile projects a project team undertakes, it is likely that nobody on the team will be an expert at writing user stories, so don't assume some team members are. The testing role begins during these meetings by validating that what is being defined is a complete, logical, and usable story. To perform this role the Testers will need some education on how to identify a good user story, and that involves some pre-work. An excellent and very readable text is "User Stories Applied" by Mike Cohn [1].

Without this kind of a focus it is easy for what is listed as the user stories to be little more than the development teams "to-do" list. Remember that this is a critical stage of the project, and although poorly defined user stories will impact the development team through various forms of re-work, they will have a much greater impact on the test team.

Lesson 3: Beware of the "Tyranny of the Urgent"

Since a fundamental part of the agile process is the user story, this methodology encourages the testing organization to focus almost exclusively on user story testing. Also, most development teams will further encourage this focus through pressure to:

- a. Test user stories as soon as they are coded
- b. Test iterations as soon as they are complete
- c. Provide feedback on updated user stories immediately
- d. Quickly verify bug fixes.

All of these represent valid requests for testing time, but it is the responsibility of the test manager to manage these requests in the context of the overall testing strategy.

This testing strategy must consider a lot more than validation of individual user stories. It might include performance, security, stress, load, database, beta, or a variety of other system testing requirements. These system testing requirements need to be identified and planned for in advance. They all require time from the test team, and that time needs to be identified, appropriately sized, and assignments scheduled during the planning phase of the project. These testing activities will then be built into the testing schedule for the project.

The best way to plan for this is by using the following sequence to identify the assignments for the testing group:

1. First identify how much testing effort will be available during the project. This is the number of available man-days of tester time.
2. Identify and size all of the required system testing activities. This includes test planning, test definition, test environment set up, execution of the tests, analysis of the results, and any other tasks required to complete these activities.
3. Calculate remaining effort available after the requirements of “2” are met. If the effort is clearly insufficient then iterate now on the process of refining the requirements from “2”, or lobbying for changes to the available time, feature set, or people assigned to the project.
4. Once any iterating within item “3” is complete, create a high level Gantt chart for the system testing activities. Assign start and end dates to each task, assign owners to each, remember to include re-execution of tests where it is likely to be required. Clearly identify the entry criteria for each testing assignment in preparation for adjusting the schedule if required deliverables are delayed (e.g. if scheduled builds do not arrive on time).
5. Based on the schedule this process created, subtract the required system testing effort from the total available effort to identify the effort available for the validation of user stories and individual iterations.
6. At this point the testing plans need to be reviewed with other functional groups (development, program management, etc) to ensure everyone agrees with the assumptions, trade-offs, and goals they contain. Iterate on this review until appropriate adjustments are made so that everyone involved considers the plan workable.

This plan will become a guide to help ensure the system testing activities are not deferred in favor of “urgent” testing assignments that arise once code delivery begins. It will still be necessary to make adjustments on-the-fly, but keep this plan nearby and stick to it as closely as possible. Don’t make the mistake of filing it away and never using it again until the project post-mortem.

Lesson 4: Nothing is Fully Baked at the End of Iteration

Unless it is explicitly stated, not everyone will have the same expectations for specific deliverables to be provided, or for the actions to be performed within the iterations. It must be defined in advance who will do what. Some of the tasks to be planned for are:

1. Having the developers who code them demonstrate any new stories as a part of the hand-off to the test group. This ensures that basic functionality works, is clearly understood, and provides an opportunity for any initial feedback.
2. Having the test team run an acceptance or smoke test on the delivered package before validating individual user stories. This ensures that the code has not been destabilized by the addition of new functionality.
3. The testing team performing detailed testing of newly delivered functionality within the iteration where it is first provided.
4. The testing team also needs to perform whatever level of additional testing has been defined as appropriate within each iteration. This could be some of the system testing, it could be re-validation of functionality delivered in earlier iterations, or some other testing assignment.

Once the developer has demonstrated the functionality, and the build passes the acceptance tests, then it is appropriate to start entering bugs against the build. If the functionality is not demo-able, or if the build fails its acceptance test, then it probably makes sense to reject the build. This course of action must also be clearly stated during project planning.

There are a variety of iterative development methodologies in use, and different philosophies on how stable the code should be at the end of each iteration. It would be convenient if everything were complete at the end of each iteration; all functionality working as planned, all entered bugs fixed before moving on. Even if a project defines this as a goal, there is a good chance this goal will change as the project progresses. The most likely scenario is that not all bugs will be fixed at the end of each iteration. Instead, only the truly “blocking” bugs will be fixes, and other bugs or incomplete user stories will either be deferred, or will appear as new user stories. These new user stories will then be targeted for a future iteration or release.

The testing team must be kept focused on increasing levels of stability and functionality at the end of an iteration rather than on perfection. The alternative risks pitting the test and development organizations against one another and will only create frustration.

Since any new code will not be frozen at the end of an iteration, it will be necessary to re-run most, if not all of an iteration’s tests later in the project. This could be a partial run at the end of a future iteration, or a full pass of all tests after the feature complete milestone. Again, this needs to be considered during project planning, rather than treated as a surprise or as a change to the plans when it occurs.

Lesson 5: Agile Does Not Mean We Are Cowboys

Agile does not mean “fast and loose”, nor is it an excuse to cut corners. There is a certain amount of pragmatism which everyone on the project needs to employ, after all this is not a cure for unrealistic schedules or insufficient resources, but the team are still expected to behave as professionals.

This starts with the need for the overall scope of the project to be clear early during the planning stages. As the team gets into the detailed planning, detailed design (and reviews of those designs) will still be needed for key or complex components. Similarly the system level attributes, such as performance targets, security requirements, and system compatibility needs, will need to be identified and clarified in advance to avoid problems during the implementation phase.

Since much of the discussion is centered on user-stories, i.e. a specific perspective, the testing group should include alternative perspectives in their thinking. Modeling techniques can be a great help in clarifying and refining the behavior of the system. These techniques could be relatively informal, such as diagramming the high-level architecture to show the major components, communication between those components, external interfaces, etc. If the diagrams do not already exist then the testers could create them as they define their tests, then iterate on reviewing and updating those diagrams with developers. Pretty soon the testing team will create some solid, usable, reliable documentation on the system.

Lesson 6: Integrate External Partners and Customers Into the Testing

A strong focus on working with customer is one of the main attributes that sets agile apart from other iterative methodologies. This typically begins early in the lifecycle, with requirements definition and validation, and then progresses through iterative deliveries of functionality, and on to the delivery and feedback on a feature complete beta build.

The test manager must use these customer relationships to validate the overall testing strategy and the specifics of individual tests. This means reviewing the testing strategy with the cross-functional team, and with the external customers involved in the project. They must be asked how they will implement new or changed functionality in their environment, and for their risks or concerns. Their responses must be understood and incorporated into the testing strategy. Whenever possible this needs to be done in-person, preferably by visiting the customers, since this provides better visibility into their environment and workflow. There is no effective substitute for “face-time” in these situations.

Agile projects typically have regular, possibly weekly, conference calls with groups of customers. The Testing Team must be positioned to gather additional testing clues from these weekly calls. This means assigning someone listen to the feedback, interpret it, and ask the questions necessary to understand the details behind bugs found, the reasons why questions were asked, and why the customers are struggling with certain features or why they find them confusing.

As the project progresses, the same level of review must be applied to beta feedback. The Testing Team must spend the time necessary to truly understand the meaning behind all beta feedback. Similarly if there is very little feedback, the reasons for this and what the options are to improve things must be identified.

The only real risk in this step is having a small number of very vocal customers mixed in with a larger group. In this case care is needed to avoid having feedback from the vocal few mask any overall trends or issues.

Lesson 7: Don't Forget the Basics

Regardless of any differences imposed on a project team as a result of adopting agile techniques, the test manager is still responsible for the overall testing of a software product. As a result, many of the fundamental testing techniques applicable within other methodologies also apply on agile projects.

For example traceability matrices are important both to map tests to user stories, and also to map tests back to the complete feature set of the application. It is possible that everyone else on the project will be focused solely on the user story perspective. In this case it is critical that as well as incorporating this perspective, the Testing Team also adopt different perspectives. Two areas that can easily be overlooked are:

1. Mapping the tests back to the complete feature set. A very important step since user stories rarely represent the complete feature set of an application.
2. Performing negative testing along with the positive testing. An over-emphasis on user stories can lead to tests that are focused purely on ensuring the correct operation of a user story (i.e. positive testing). Clearly this could result in a lot of missed bugs.

Similarly agile is not a cure for complexity. Developers on any project will struggle in certain areas, so the testing team must understand which code is new, innovative, confusing, or otherwise difficult to write. An appropriate amount of testing effort must be directed to these areas since they are indicators of potential bugs. Developers will typically provide a lot of this information, so they must be asked what they are struggling with, and what they are worried about, both in their code and the project as a whole, then testing effort should be directed accordingly.

Finally, bearing in mind the earlier warnings related to the “tyranny of the urgent”, the testing team must keep up with bug fix verifications. These are also areas that can produce a lot of additional bugs, so the appropriate amount of time and effort must be directed here.

Lesson 8: Plan for an “Iteration X”

A common way that agile projects can appear to stay close to “on schedule” is by deferring incomplete work to later iterations. Even on well planned projects some of the user stories won’t be fully implemented, additional required functionality will be identified, and ship-critical bug

fixes will be deferred. Although it may be possible to drop some of the planned functionality from the release, much of this work will be required for the current project. This means either an extra, unplanned iteration, or a greatly extended iteration will be required in the latter stages of the project.

Clues that this is coming appear during early iterations, and the testing strategy must be flexible enough to accommodate it. If this deferred work results in an additional iteration being added to the project then, of course, it is very visible and it is likely that the project schedule will be adjusted accordingly

Less visible delays are those which occur at the individual user story level. Typically, delivery of individual user stories will be scheduled evenly throughout each iteration. In practice it is likely that many user stories will be delivered close to, or at the end of the development phase of an iteration. This means that delivery of the functionality from an iteration may be on-schedule, even though completed individual user stories may have been delivered late to the testing team.

The testing strategy must allow for this, if the user stories are delivered on time testers must ready to test them, but if they are late then those testers need other valid assignments to work on so that the overall testing schedule is kept on track.

This becomes less likely to occur as a team becomes more familiar with agile techniques, but it remains a significant risk even with more experienced teams.

Lesson 9: The “End Game” is still the “End Game”

With a similar message to “Lesson 7”, the important point here is that agile projects are still software development projects, and the differences in this development methodology should not be expected to radically change the system testing requirements, especially on the first couple of agile projects a team undertakes.

For example, the incremental delivery of functionality will result in a much more stable “feature complete” build than is typically achieved with a “big bang” methodology, but all of the functional tests will still need to be re-run on the feature complete build. Similarly although bugs will be found in the feature complete build, and testing will iterate on rebuilds, it is typical that fewer bugs, and fewer rebuilds, will be required than with “big bang” methodologies. However the bugs will exist and retesting will need to happen. This must be considered and planned for within the testing strategy.

The final point is that testing schedules will still be squeezed, especially as the final release date nears. This could be a result of schedule dates included in the original plans, or a result of (formal or informal) re-planning that occurred during the project. Thus the testing team will be short of testing time at the end of the project, so they must be prepared for the same end-game challenges as would be encountered with any other development methodology.

Conclusion

The emphasis of the lessons learned is of course on working around or avoiding potential problems that may occur when adopting agile methodologies. However this is not meant to imply that agile techniques are any more problematic than other development methodologies.

Applied intelligently, and not as the next “silver bullet”, agile techniques can be very effective. They have allowed McAfee to deliver innovative functionality that could not have been delivered using most other methodologies. The close working relationship with customers is invaluable, especially on innovative projects, and the iterative process can result in a more reliable end product even with highly complex technologies.

Bibliography

- [1] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley, 2004

Defining Test Data and Data-Centric Application Testing

Chris Hetzler
Microsoft Corporation
chris.hetzler@microsoft.com

Chris Hetzler B.S. computer science, North Dakota State University, Fargo, ND 2003. M.S. software engineering, North Dakota State University, Fargo, ND 2005. He has been employed with Microsoft in the Business Solutions division since 2003 and has spent most of his time there working on next-generation financial applications. His master's thesis was published in 2005 and was written on the testing process that was created for the testing of Microsoft Dynamics Business Portal. He has been invited to speak at several conferences in 2006 on the topics of test data and testing within agile development methodologies.

Abstract—As applications grow larger and more complex with time, and as automated testing of these applications is increasingly adopted, the data that is used during the execution of the automated tests needs to be clearly defined and identified early in the development lifecycle. Currently, the Software Engineering Body of Knowledge does not contain a definition for “Test Data”, nor do any of the top books on the subject of software testing. This paper and presentation will propose a definition for the term “Test Data” and outline what testers can do to ensure that teams are considering it in their planning phases and provide useful ideas on how to isolate what those data needs might be. The paper will also give accounts of software development projects that have defined their test data early and of those that have not in an effort to provide the reader with some context and examples.

I. INTRODUCTION

Data-centric applications are nothing new in the software industry. Accounting packages were among some of the first widely adopted software applications for personal computers. The complexity of these applications has increased greatly over the decades since their introduction. Increasingly, testers are faced with very complex and tightly integrated test scenarios for the ever increasing feature sets in this class of applications. When automating the testing of a large, enterprise-class application that is data-centric, the test data that is used in the automated test suites is critical to the testing effort. As Linda Hayes points out in [1], “It’s basically impossible to develop repeatable automated tests without a known, stable data state”.

This paper and presentation are based on my testing experiences with the Microsoft Dynamics GP (formerly known as Microsoft Business Solutions Great Plains), Business Portal, and Dynamics GP Web Services in addition to my education in the Computer Science and Software Engineering disciplines. The intent of the paper is to provide others involved in testing these types of applications with common definitions, properties, practices, and suggestions to help them in their efforts. I also hope to gain acknowledgment from the industry at-large that these types of applications are in a class of their own and warrant special consideration during the various phases of the development cycle, regardless of the development process being followed.

Since the Software Engineering Body of Knowledge does not include definitions for these terms (Test Data and Data-Centric Application) in its software testing section [2], it is first necessary to provide the definitions that will be used throughout this paper.

II. DEFINITIONS

A. Test Data

The alpha state of an object is defined by Binder in [3] as: “The α state is a null state representing the declaration of an object before its construction. This is different from an initial state, which is the first state an object takes after being constructed or initialized”. Similarly, the omega-state is the state of an object takes after its deconstruction.

Some might see this definition as simply stating that the alpha-state is the declaration of an object and it’s type, for our discussion this analogy will be useful. Using the analogy, what happens when the concept is abstracted to the system level, and in particular the enterprise system level? At the object level we might declare an object of type account to be used when creating sales orders. What happens when the test is complete? The entire sales order object graph moves into the omega state and the account that was created is removed from memory. If another test wanted to use this account it would have to create it, and if your tests require hundreds of different accounts more time could be spent in establishing the tests alpha-states then in the actual testing. So at the enterprise system level I see this alpha-state as a point in time before test case execution in which all of the data required for the test case to exercise its features has been loaded into the systems backing-store. To summarize, test data is any piece of information required by the system under test (SUT) to place that system in the desired alpha state prior to test execution. This information would include application setup data, system setup data, application initialization data, data store initialization information, and any data that the feature being tested would need in order to achieve the desired alpha state. A great example of this type of data is given in [1]: “The test was to issue a loan against a 401(k) plan. First she had to find a plan that permitted loans, as well as a participant within that plan who had a sufficient cash balance for the loan, had not taken out a loan within the past year, and did not have an outstanding loan from a previous year.” In the setting of an ERP application, the alpha state of the system would require company and system databases to be created. This would also require that the databases be populated with sufficient data to allow the features under test to be exercised fully. In the previous example, the data requirements that are listed, in addition to all of the setup data for the system, would produce the alpha state for the test. This statement also applies to any negative, or exception testing that is to be completed on the system. In other words, the test case’s alpha-state is the state that the test case assumes prior to its execution, regardless of the type of test that is being executed. This is similar to the classical definition of a test case’s pre-condition, however there is a subtle difference. I would hold that defining the test data (and thus the alpha-state) for a test case is *part* of the definition of the test case’s pre-condition. I would hold that the pre-condition also includes such things as the test environment, other software dependencies, and so forth. If we carry this abstraction a bit further, we can see that when we are running a suite of

test cases in a regression system, the omega state of one test case is actually the alpha state of the next test case to execute. If we reduce the number of states, it can be inferred that the alpha state for a test suite is the state that every test case in the suite assumes prior to its execution. This allows us to define a test suite as a set of test cases that all assume the same alpha state and omega state in their life cycle. And, in fact, a set of test cases whose alpha-states and omega-states are interchangeable, as long as sequential dependencies have been eliminated or clearly defined.

One last note on this definition, if you are working with a test case that requires you to reload test data or otherwise change your dataset to a large degree, say load two more companies that each have three different currencies defined, this should really be a candidate for a new test case or even a new test suite. Since there certainly could be other tests that will need that dataset loaded for their alpha-state requirements to be met, this new dataset should be saved in the normal manner to allow its re-use by other test suites.

B. Data-Centric Application

A class of computer software that is characterized by heavy, if not total, reliance on the structured data being accessed for its *functionality* and *observed behavior*. These applications typically interface with a database system for the storage and retrieval for the data that they require. An application's reliance on a database backing store is not required for an application to be considered data-centric, however. As more and more applications use and depend on structured data, such as XML, these documents can be seen as the backing store and the test data that is required for the alpha state to be achieved. Increasingly, applications are relying on XML configuration files and if the data in these files is static, I would not consider them to be data-centric in our definition. If the data is dynamic in nature and causes the application accessing it to behave in different ways, the application would be data-centric in the proposed definition. By dynamic I mean that the data contained in the configuration file can change or can be changed in a number of ways at runtime and that these changes would directly affect the functionality of the application. These changes could be made by the application itself or by third-party applications as well.

One major obstacle to classifying these applications as data-centric is that sometimes if the data in the application's configuration file changes, the application has to be re-initialized for the new settings to take effect. The applications that this definition is dealing do not require a re-start for the new data to change the behavior of the system. To use the example of a multi-currency accounting system, normally the system will need to have a default currency setup and the functionality of the currency sub-system will depend on this setting. This leads to tests that can potentially return differing results based on this default currency setting, just as the system normally would. Another example would be an ERP system that could post transactions across multiple companies, but only if multiple companies are setup in the system and those companies have the cross-company posting feature enabled. Testing this feature set would require several sets of data that each represent that various combinations of these situations.

C. Alternative Definitions

Some may claim that all applications are data-centric, since data is what software deals with by its very nature. Certainly a program that takes no input and produces no output could be a bit boring for a tester to test! This belief does not acknowledge that there are different classes of applications. For example, this definition would state that a large enterprise class ERP application is in the same class as a word processor or game. This is just not true, since a word processor *processes* data, while an ERP application *consumes* data. For example, a word processor opens a file and allows a user to modify the contents of that file and then saves the file. The application enables the user to process the data within a file. An ERP application allows a user to view data in some format and then use that data to produce new data, transform the existing data, or modify the existing data. This list is only a few of the operations that are possible. One scenario could flow as follows: a user selects general ledger transactions for posting that are not assigned to a batch, they create a new batch, assign the selected transaction to that new batch and then post the batch. Thus the data that was consumed in the application (general ledger transactions) has been transformed into a new type of data (batch-posted general ledger transactions). Another difference arises when we look at the two applications in the context of the definition presented earlier in this paper. The data contained in the word processor's input file does not change the functionality of the application, once the file is opened it can be worked with and saved again. When an ERP application consumes its data, that data can change the application's feature set. An example of this would be if one user's role within a role-based system allowed them to use multiple currencies

for the transactions that they process while another user couldn't. This means that the data which is stored by the application can dynamically change the application's feature set, based on the user that is currently logged into the system; this only one example there are certainly others.

Experience has shown me that there is a very real difference in testing applications that deal with unstructured data and those that produce and consume structured data. The biggest difference that I have seen is what this paper is all about: test data and its definition. Applications that are not data-centric by our definition and that do not consume structured data generally do not need any starting data for automated test execution. They might use starting data to decrease the total test run time, but the tests could also create the data that they need during their execution. Automated test for data-centric applications cannot be run without the test data that they require being present. In other words, the alpha states of the two applications are fundamentally different in their requirements on the test environment. Using Microsoft Word as a sample non data-centric application, it can be shown that Word does not require any starting data (data that determines its alpha state) by our definition, since you can launch the application without any files that word consumes on the host system. An ERP application, however, will require a data-store of some sort to be present first as well as requiring that the data-store contains system and company data.

III. BACKGROUND

This section will cover some of the existing documentation on the subject at hand and what, if any, the deficiencies are in the existing sources that are listed.

ANSI / IEEE standard 1008-1987 states what the industry standards should be for software unit testing. In this standard, section 2 defines a data characteristic as: "An inherent, possibly accidental, trait, quality, or property of data (for example, arrival rates, formats, value ranges, or relationships between field values) [4]." This is the closest to a definition of test data in this document that functions as a standard for software unit testing. So are we as software engineers to believe that test data is a potentially accidental trait of our system? Certainly not, on the contrary, I would contend that test data is an intrinsic aspect of the system under test. In [3] we are given no definition of test data, nor is there one given by Pressman in his classic software engineering text [5].

When discussing this subject with other testers, one is frequently greeted with a quizzical look and a simple question. That question is "How is test data any different than the data that *any* application creates or consumes?" Indeed, unless a tester has worked on an application that fits this definition, they will not be aware of just what they are in for when they begin to automate their test cases, and in fact they may not even be able to create test designs without a satisfactory set of test data.

On the other hand, when discussing this subject with testers who are familiar with data-centric application testing, as I did in preparing this paper, you will receive as many definitions for these concepts as there are testers in your research group.

It has been my experience, across both legacy and next-generation applications that any effective test strategy begins with defining what the testing data needs are and what the required test infrastructure will be.

IV. AUTOMATED TESTING

This section is intended to give the reader some insight into how the test organization that I am a part of performs automated testing. It is not intended to be a full discussion of test automation theory, nor is it a description of how Microsoft as a whole performs its testing for its wide array of products.

Our test organization in Fargo is a highly automated one. In fact, over ninety percent of the testing that we do is fully automated. That means tens of thousands of tests cases for the various products that we are responsible for. What this means to us is a great investment in automation technologies and in resources to produce that automation. Our goal is to automate as close to 100% as possible of all testing that we can. This includes performance, integration, API, user interface, security, and business logic testing. The Fargo campus has a long history of automated testing due in large part to the testing that went into Great Plains accounting.

Our test creation process varies a bit from product to product, however it does follow a fairly consistent model. Our testers are involved in the development process for a feature as soon as possible. Normally this means a primary and possibly secondary tester is present at all of the requirement review sessions, a test manager is present for all program level meetings, and a test representative is sent to all development meetings. Once the requirements are

signed off for the feature, the tester begins the process of creating the test design. After the design is signed off, another tester creates the test automation driver for the feature. In our managed code applications, this means a C# executable that uses our own testing API, which is similar in nature to NUnit. This driver is then put into our regression system and run as part of our daily or weekly regression runs. When a tester is creating the test design for a given feature, they are required to define what test data the driver writer will need to fully test the feature. This is why we involve our testers as early as possible in the process –so they can begin the task of defining test data as early as possible. Quite often this means looking for a data set within our existing test data that we use for testing Dynamics GP. We do have the benefit of leveraging a vast amount of test data that has already been created for previous testing efforts. We also have a fantastic test data team in Fargo whose sole task is to maintain our test data from release to release.

During the recent development of our first version of the Dynamics GP Web Services, I was present at all of the early planning and vision creation sessions. This was in an effort to provide our test organization with an early voice in the development of these services. The development cycle for this program was one of the shortest we have ever achieved, and yet quality was always at the forefront of our development efforts. As this product is new, I am unable to report any bug report incidences at the time of this writing. It fell to our test organization to develop the test strategy as early as possible, to ensure that the highest quality was delivered in as short as time as possible.

When I first started working in our test organization there were several different meanings for the term “test data”. Developers defined test data as the data that their unit tests required to run. Feature level testers defined test data as the data that their assigned features’ tests needed to run successfully. Performance testing teams saw test data as their very large, very precisely crafted database backup sets. The list goes on and on which ultimately results in a simple fact that we faced and that many development projects can face. Simply put, every test group had different definitions of the term and, consequently, different data sets. This adds more fuel to the fire that this class of application is different from others, since different groups within the project can affect the performance of other groups, simply by changing the data that they use. For example, on one project the developers created their own sets of test data and their own processes to load it. When testers, or other developers, created their tests they found that it was almost impossible to duplicate the functionality they found in a given unit test driver. This was because there was no central process for using, creating, or consuming test data. We found that the build verification tests were hard to maintain since each developer was maintaining their own data sets and the tools to load that data. This problem was compounded by the fact that our development group was spread across three different geographical locations and this added yet another set of definitions to the mix. When testers began the process of creating their test designs, they first looked to the developer’s unit tests to determine what the data needs of the feature were. This also led to problems, since most testers were working with functionality that spanned several components that were developed by different developers and therefore used different test data sets. Often times the tester found that the data sets were incompatible with each other. An example of this would be if a developer had used a data set that did not have multiple currencies defined in it for unit testing general ledger accounts and another had used a multicurrency data set for the unit testing of, say sales orders. This leads to tests that cannot run on the same data set as the sales orders will try to use currencies that are not defined in the general ledger data set. Certainly the various data sets could be loaded in between test runs, but this takes time and we strive to have our regressions run as quickly as possible in an effort to reduce the turnaround time on builds and bug fixes. So the tester in this situation faced a difficult decision, namely which data set to use in the creation of their tests which could span these tests and several others, as an integration or installation test would. Also, this made it difficult for testers to determine what data sets to use in their own API level test drivers.

There had always been a desire in the test organization to eliminate this bottle neck in the testing process. Developers wanted testers to use their data and testers wanted developers to use one single data set. This is not to say that the test team wanted the developers to limit the scope of their testing, but rather to all use the same dataset where possible in an effort to make their tests more understandable and reusable. It must be stressed here that the test team did not want to create one monolithic dataset for every test case created for a product to use. On the contrary, the test organization is very quick to recognize that this is just not possible for our products. What we did want to achieve was joint access to a limited number of datasets for both of our organizations to use in the same manner. When it came time for a new release of this product, the various test organizations sat down to determine how they could increase their efficiency and decrease the amount of time their testing was taking. One of the major areas identified in these meetings was test data.

V. IMPROVING THE PROCESS

A. Identifying existing data sets

We are very fortunate to have a great number of existing test data sets for our use in our testing. These data sets have been in use for testing the Great Plains application for years and now we are able to leverage them in our next generation products that are based on the Dynamics GP database data model. As mentioned earlier, we adapted our test process to require the identification of the test data set for a given test design as early as possible in the planning and requirements phase for a feature, once we had sign-off from management on a tester's early involvement in the development process. So we were able to move the effort that was normally required at the end of the test creation cycle, to the beginning and allowed the tester to find existing test data when they had the time in their schedule to do so. The types of products that fall into the data-centric category are very different in their functionality and feature sets as well as in the way they store their data, so it is virtually impossible to recommend a strategy for identifying these data sets for your particular product. I can summarize what we have done, in an effort to provide some guidance to the reader, however. Our Dynamics GP product has several modules that each require their own subsets of test data. These modules have numbers that the core products uses to identify them and we number our test data sets based on the same numbering scheme that the Dynamics GP core product uses, which is published in a document on our internal testing SharePoint site. This way if we know that we need multi-company, multi-currency, sales-person data, we can identify the existing data sets that we have with this data in them by the number that identifies the dataset. This allows us to reuse the datasets as they are created, and as the creation of these datasets is very time-intensive we are able to speed our development process as more complete datasets are available for given product.

B. Identifying test data gaps early

One of the most difficult issues to overcome when defining test data is ensuring that the data contains all the values that a consuming test would need to achieve the desired alpha state prior to test execution. This is not to say that there is one all encompassing data set. On the contrary we have a great number of data sets, but they all serve clearly defined purposes that relate directly to the requirements for the feature or features being tested. For example, we need test data that represents an enterprise with multiple companies and companies within that enterprise that have a variety of currencies defined, for testing inter-company transactions and exchange rates. But we also need data for a single company without a default currency defined. So the functionality of our product determines what the test data for a given scenario will be. The criteria for determining what the gaps are in the existing data sets vary from product to product, but almost always deal with new functionality. If we see that a new feature or piece of functionality is coming into the product then there is a good chance that we will not have existing test data in our sets to sufficiently test it. One of the key areas that we found for improvement was the amount of time spent creating test data sets. When reviewing our previous testing process, we found that when a feature's testing required data that we did not have in our currently available data sets, several days were lost in the creation of these data sets. This is because at the end of a development cycle resources are scarce and the persons who are versed in test data creation might not be available. To reduce the risk of this issue we decided to task a tester with determining their data needs as early as possible in the test design process. This way, when gaps in the existing test data were identified, the tester would be able to coordinate with a data specialist for the area that they were testing. This aided us in our effort to get our test data defined as soon as possible in the development cycle.

When a product is as mature as Dynamics GP, there tends to be a great number of test data sets, so generally speaking we almost always modify existing data sets when a new feature or piece of functionality comes into the product. On some of the other Dynamics product we are at an earlier stage in the product lifecycle and in those testing efforts we are creating new datasets as required. One of the ways that we determine if new data is required is to look at the overall product requirements, such as multi-currency versus non multi-currency. Generally there are conflicting requirements in a product around these two features, so two different datasets would also be required. Another example would be data sets for different cultures and geographies, since many countries have different business rules and tax structures.

C. Involving testers as early as possible in the development lifecycle

As you may have noticed, there is a theme to this paper and it is the involvement of tester as early as possible in the development of an application or feature. I have been involved in several projects that brought testers into the development cycle at the traditional times –at or near the end of the project. And as one might expect these projects have experienced the traditional delays and troubles that applications developed in a waterfall-like manner have. I have also had the privilege to be a part of a development team that included testers as early as possible, and not just from a management standpoint but also from an individual contributor one as well. As I stated earlier I was involved in the earliest planning sessions for our web services release as a voice for the technical aspects of our testing. I was tasked with developing the test strategy (which included test data definition) at the same time the program managers were beginning to create the requirements for the various web operations. The strategy was then able to be created with every stakeholder in the testing process accounted for. Before the strategy was adopted, we required agreement from all of the various testing groups for the project as well as management and development. This also empowered the testers for a given feature to drive the processes that dealt with testing the code and requirements at all levels.

D. Testers and developers can use the same data in their testing efforts

We were able to include system integration testers, performance testers, API testers, and developers in the process of defining what our testing for web services would look like. Starting this process so early and including so many various disciplines in these discussions enabled us to determine what the best course of action would be for several common stumbling blocks. Again, test data came to the forefront and we were able to get development, API testing, and integration testing all on the same data sets, using the same tools and process! It was truly amazing to see, in fact the developers agreed to have their unit tests reviewed by the tester assigned to the feature. They also agreed to consolidate their unit tests into as few executables as possible, and the grouping of these tests was based upon the data that the test needed to consume. It was determined, for example, that the general ledger account object and the customer object could be tested with the same data set, so these tests became suites within one test driver or executable. Our testing framework also allowed for the prioritization of test cases and we used this to our advantage. The priority is zero-based, so we labeled the build verification tests as priority 0, units test as priority 1, and API tests as priority 3. We included our user interface tests in the priority 3 tests as well, since the user interface for this project was extremely minimal. This in turn allowed us to run *all* of the tests for the application in the same way with the same regression harness. We simply passed the priority of the tests that were to be run as a command line parameter to the executable. This told our test services tools to run all of the tests in the assembly with the priority that was passed in along with all of the tests that are of a lower priority. It could be argued that using different data sets for the different tests would provide more coverage, but recalling the definition of data-centric application, it can be shown that the system would not even be able to be started if the proper data set was not being used.

VI. CONCLUSION

As mentioned earlier, I was assigned the task of creating the test strategy for the testing of our Web Services for Dynamics GP. Having just received my Master's degree I was eager to put the lessons I learned in software engineering in school to use, along with all of the learning that had come out of re-vamping our test process. Since we had identified the areas listed above, I was able to create a strategy that incorporated all aspects of the development lifecycle into an overall test strategy. For example, since we had been involved in the early planning stages of the program, we knew that when the user interfaces came along at the end of the release cycle, we would need help in testing them. So, from the very beginning of the project the test organization was asking for additional resources from other areas, such as program management, to aid in testing. We were also able to start working with the test data team extremely early in an effort to ensure that when we needed our test data it would be there for us and that they would have resources available to help us if needed. Unfortunately, we do not keep metrics that would allow me to state in man-hours or man-months how much efficiency we gained in this project over Business Portal, but we did achieve over ninety percent code coverage on a several hundred thousand lines of code application. And we did this with less than ten full time testers and in less than eight months. I am totally convinced that this success was due in large part to our emphasis on the test data that we used.

By emphasizing the test data and the test data process from the very start of the project we were able to accomplish several goals. We were able to get testers and developers to use the same test data sets, for a given

feature. Next, our test data sets were filled out more during the process as we identified gaps in the existing data. Third, our testers were involved in the project as early as possible and their knowledge of test data and processes is what got them involved that early. Lastly, we identified several datasets that could be used for multiple tests and decreased our test run time by several factors of magnitude.

ACKNOWLEDGMENT

I would like to thank MBS for allowing me the time to pursue this paper. I would also like to thank my manager Shawn Hanson, for finding opportunities that challenge me and all of the other testers that I work with. I would also like to thank my wife Janelle, for understanding my need to spend time writing, working, and researching this paper.

REFERENCES

- [1] Linda Hayes, (2004) "Automation or Not, it's All About the Data," Available: http://www.stickyminds.com/sitewide.asp?Function=WEEKLYCOLUMN&ObjectId=9033&ObjectType=ART_COL&btntopic=artcol&tt=WEEKLYCOL_9033_title&tth=H.
- [2] *Guide to the Software Engineering Body of Knowledge*. Los Alamitos, CA: The Institute of Electrical and Electronics Engineers, Inc. 2004, pp. 5-1 – 5-16.
- [3] Robert V. Binder, *Testing Object-Oriented Systems*. Boston, MA: Addison-Wesley, 1999, pp. 1074.
- [4] *IEEE Standard for Software Unit Testing* (Standards style), ANSI / IEEE Standard 31008-1987, 1986.
- [5] Pressman, Roger S, *Software Engineering A Practitioner's Approach*. New York, NY: McGraw Hill, 2005.

**Pacific Northwest Software Quality Conference
PNSQC 2006**

**Test Case Maps
In Support of Exploratory Testing**

Claudia Dencker
Software SETT Corporation
P.O. Box 718
Los Gatos, CA 95031
408-395-9376
cdencker@softsett.com

Abstract

With software becoming more complex than ever before and time (or the lack thereof) the ever-steady constraint, testers look to other techniques to help identify defects more quickly and still get the coverage that is essential. In the 1980's testing methods were structured, followed the waterfall method and the IEEE 829-1983 standard on test documentation. In the 1990's iterative and agile programming forced testers to pick up the pace. In the commercial sector detailed test procedures were a thing of the past. Test matrices became more popular and sharp testers could effectively test off these. Additionally, test tools came available for testers to automate the more tedious aspects of testing. In the 2000's ad hoc testing came of age and was beautifully articulated and defined as exploratory testing. But test procedures and testing tables were too heavy for this light, highly agile and wonderfully effective form of testing. This paper shows that test maps can be used as a complement to exploratory testing, solving the problems mentioned above.

With test case maps testers are able to provide management the objectivity they need to track the testing process and still achieve high productivity and efficiency as realized with exploratory testing. This paper defines test maps, what role they play with respect to other forms of test documentation, and their real importance as design/thinking documents. We all know that we can't test everything nor is a careful analytical evaluation the only true way to get the essential coverage. Instead test maps leverage off our common sense and the expected usage of the functionality for which we are responsible.

Bio

Ms. Dencker is President of Software SETT Corporation (www.softsett.com), established in 1987, a company specializing in hosted test case management and tracking for global and virtual software teams. She has been active in software testing and QA for over 25 years and has taught classes worldwide through the IEEE, University of California-Santa Cruz Extension and to major Silicon Valley companies. She is also Program Coordinator for a new program in Software Quality Engineering and Management at UCSC-Extension (www.ucsc-extension.com/softwarequality). Ms. Dencker received a Bachelors degree and a teaching credential from San Jose State University and is a certified software quality engineer (CSQE).

We gratefully acknowledge the test maps from two interns at the Portnov Computer School in Mountain View, CA, Sobha Rani Putta and Juby Paulose, who have allowed us to use their test maps as examples in this paper.

Introduction

Throughout my career as a software test consultant, I have been involved in many different testing assignments using a wide range of methodologies, some of which were overly ponderous and others delightfully light and quick. At the beginning of most projects and where I had the latitude, I invariably included exploratory testing as a fundamental aspect of my test strategy.

Even though I consider myself a senior test professional who can out test most others, a dilemma I have found with exploratory testing is maintaining focus and not loosing track of where I've been and where I want to go. Sometimes the functionality under test is complex enough or can be used in many different conditions that I'm not always sure I've covered everything. To ensure that my coverage is adequate and that I'm not overlooking conditions worth testing, I create test maps. These are simple tables that show how the functionality can be tested against itself, other functionality, or against boundaries and default settings. These tables or maps are created quickly and show the tester intersections of interest worth exploring further or those intersections that should be ignored completely. Furthermore, these maps can be used to monitor testing progress which will satisfy management's need to control product schedule.

As we race to the product finish line, our goal as professional testers is to find as many defects as possible in as little time. For young companies this race is the difference of being in business and crossing the finish line or getting winded or tangling with others and falling onto the side lines. Even large, established companies are aware of this race as they have to continue to find ways to compete and stay in the running. Overly heavy methodologies that require extensive maintenance slow the racer down. But, as we all know, management needs to be able to keep its finger on the pulse, and bug metrics, while important, are not the only way. In fact, most testers who consider themselves professionals will not be fully satisfied with bug metrics alone. They'll want the professional assurance that they've covered everything they need to. Test maps will allow them to do this and give management the confidence they need to objectively monitor the exploratory tester.

What is Exploratory Testing?

So what is exploratory testing and why is it so good? For a great discussion on this important form of testing and how it contrasts to scripted testing, refer to the article by James Bach, *Exploratory Testing Explained*, April 16, 2003, on www.satisfice.com. In a nutshell exploratory testing is "... simultaneous learning, test design and test execution. In other words, exploratory testing is any testing to the extent that the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests."

Bach cites several examples of exploratory test teams who outperform their scripted test colleagues. In fact, in a few cases, these teams are still a part of their companies while the scripted test organizations have been closed down.

Years ago I conducted a QA assessment whereby the customer had devoted their entire test strategy to automated scripted testing. They were frustrated that too many bugs continued to slip through and couldn't understand why the scripting team couldn't catch them. The answers were obvious at the time, and further validated by the test team who found themselves in a maintenance cycle of no end. Software continued to change rapidly but the scripted tests were like concrete blocks in the flowing river of change. The automation testers couldn't keep up with the change and ended up maintaining their tests at the expense of writing new tests. Needless to say, I recommended that the company lighten up on their automated scripting and focus on a more fluid process of testing that didn't involve scripting.

Not only does scripting place a heavy burden on the testing process, but so does a heavy test methodology. On another project we found ourselves confronting the needs of regulatory agencies that required every aspect of the test to be spelled out to the 'n' degree. Due to the needs for manual scripted testing, we asked our customer whether they were interested in proper documentation or good testing (that yields as many defects given the time available). Not surprisingly the customer requested proper documentation for the upcoming audit since their rationale was first and foremost to get the formal approval. If they didn't have this, then quality software didn't really matter. They would worry about quality after the approval which wasn't a reassuring position for us testers.

With all the advantages evident in exploratory testing, there is the exposure of omitted paths, conditions, and combinations as the exploratory tester "explores every test idea that emerges in the

moment of test execution.” This doesn’t mean to say that I advocate a more rigorous approach to exploratory testing, but rather how can we identify the “bread crumbs” necessary for returning back to “home base.” What are the signs ahead that will ensure that we take the correct or necessary paths and omit the unnecessary paths?

How Test Maps Extend Exploratory Testing and Minimize a Key Risk

To ensure the kind of coverage that a professional tester expects and to leverage off the strengths of exploratory testing, I create and use test maps. So what is a test map? Isn’t it just another test document that requires maintenance?

The answer is “Yes” and “No.” A test map is the end result of a thinking process whereby the software under test is mapped against obvious and not so obvious boundaries, against itself and other functions in order to identify conditions and combinations worth testing. The map is laid out on an x- and y-axis using the table feature in a word processor or using a spreadsheet tool. My maps are typically two-dimensional in order to keep the information simple and useful. Three or more dimensions can be identified but then maintainability and complexity increase that will diminish usefulness and usability.

The end result is a map from which a tester can run tests or build more formal test cases. A map is not a set of test cases, but rather reflects the thought process that helps to narrow down the test cases worth implementing and/or automating to ensure coverage (though coverage may not be absolute). As in exploratory testing, the three activities of learning, test design and execution are all involved in generating the map. Lastly, test maps are generated quickly and with as little constraint as possible (to enhance the creative process). Maps help the test professional to collect his/her thoughts and use his/her experience prior to much testing being done or where additional clarity is needed.

In fact, maps don’t need to be created for all areas of functionality, just those areas where a need arises or where complexity assures that the tester could forget certain combinations or conditions,

A test map can become part of the project artifacts, but need not be. Because a map is a thinking tester’s aid, it can remain as a personal checklist or it can be formalized as a pre-document for the test cases themselves. If it remains informal (that is, it is subject to change and not distributed), the primary downside is that the test lead or project management will be unable to assess where the exploratory tester is in their testing assignment. Certainly high bug yields will impress test and project management, but this should not satisfy the professional tester who will want to ensure good coverage as well.

In fact, I like to formalize my maps so management can partner with me in my testing. With assurance management can report objective progress and won’t be caught off-guard at product delivery. Think: the infamous 20% project remainder that takes 80% of the time or that the testers remain at 80% complete for weeks on end. While maps won’t guarantee perfect scheduling, they do provide management vital input as to the kinds of tests that you are planning or have run. This information along with bug metrics should satisfy the immediate needs of most projects in which the exploratory tester participates.

How do I create my test maps?

This question is best answered by a recent project in which I worked with a group of interns. For most beginning testers, working together as a tight group is optimum, but on this project, we were unable to do this. In fact, we met once a week as a group, reviewed progress and answered questions. The bulk of the work was completed at another location or at home. In this situation we used test maps as a learning tool and for tracking purposes.

The software under test was a new release of our test management tool, kSETT. We had implemented the query/filter capability across all objects and this new functionality required testing. This functionality allows the user to query the data for a specific subset of data, view it online or use it for reporting. We partnered with a local school in the San Francisco Bay Area, Portnov Computer School (www.portnov.com), and were assigned four testers. The students were degreeed from universities in other countries, had completed coursework at the Portnov Computer School and now sought experience on a real project.

All interns were eager but did not have a lot of experience creating test cases. Because the team did not work side-by-side, it was imperative that they understood how to develop test conditions that had merit and be able to identify conditions that were redundant or low risk. Some interns started writing extensive test procedures but due to time pressures we resynced the team along the lines of test maps.

Having done testing as long as I have, I preferred that the interns move as quickly as possible to test execution but not at the loss of good test design or some pre-thought of where they needed to go. Also, I was simply not interested (and am rarely interested) in detailed test procedures with extensive action steps because functionality needed to be checked out quickly. Furthermore, the interns would be testing their own area; there would be no hand-off of testing from one person to another with the resulting loss of product and test knowledge.

Below is an example of a sample test map we used to start for test case filter testing.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	SAMPLE TEST MAP												
2													
3			One (min)	Two-same			Two-different			Many (max)			
4				AND	OR	NOT	AND	OR	NOT	AND	OR	NOT	MIX
5	1	Assigned to					1, 8	1, 7	1, 6	1, many others	1, many others	1, many others	1, many others
6	2	Component											
7	3	Created After											
8	4	Created Before											
9	5	Created by											
10	6	Created on											
11	7	Executed											
12	8	Failed											
13	9	Has History											
14		...											
15													
16	LEGEND												
17	C5	Assigned to <tester>											
18													
19	D5	Assigned to <tester1> AND Assigned to <tester2>											
20	E5	Assigned to <tester1> OR Assigned to <tester2>											
21	F5	Assigned to <tester1> AND NOT Assigned to <tester2>											
22													
23	G5	Assigned to <tester1> AND Failed											
24	H5	Assigned to <tester1> OR Executed											
25	I5	Assigned to <tester1> AND NOT Created on <date>											
26													
27	J5	Assigned to <tester1> AND Component <component> AND Created After <date> AND ...											
28	K5	Assigned to <tester1> OR Component <component> OR Created After <date> OR ...											
29	L5	Assigned to <tester1> NOT Component <component> NOT Created After <date>											
30	M5	Assigned to <tester1> AND NOT Component <component> OR NOT Created After <date>											

Figure 1: Sample Test Map for Filter Functionality

The possible filter values appear in column B and include Assigned to, Component, Created After, etc. This column represents the functionality we want to test. Because filters could be very simple or highly complex, we sought to show this range on the map. The simplest filter or our minimum is represented in column C where only one value is queried. A user attempting to display only the tests assigned to a particular individual in the project would create the test case filter, Assigned to = <tester 1>. This query is identified in cell C5 above.

It is also possible to create a filter whereby tests can be displayed for more than one tester on the project but not all project members. This is represented in cells D5, E5, F5 and specified in rows 19-21. Two different filters can be combined to create yet another unique view of the data. This is represented in cells G5, H5, I5 and specified in rows 23-25. Finally many different filters can be combined for more sophisticated querying. This is represented in cells J5 – M5 and specified in rows 27-31.

By building simple to complex queries in our map, we're incorporating boundary tests though our boundaries are tempered by business needs, not technical needs. Others could argue that our

boundaries are not true minimum/maximuns but given the project objectives and timeline, we were satisfied with the best, reasonable attempt at defining these ranges.

Highlights in the figure above show what the tester wanted to test for the first test case filter, Assigned to. Initially we were interested in having the interns complete the entire test map but realized very quickly that the interns would have to sample. This sampling is conveyed by the highlighted cells in Figure 1. In our template we suggested to the testers that the following conditions seemed reasonable to explore when testing the Test Case filter functionality. We left the actual combinations of criteria up to the interns to decide.

- Using ONE value from ONE criterion, filter = Assigned to “Joshua Tester” to determine the test load for one individual on the test team
- Using TWO values from ONE criterion, filter = Assigned to “Joshua Tester” and “Barbara Breaker” to determine the test load for two members on the test team
- Using TWO criteria in the same filter (AND), filter = Assigned to “Joshua Tester” and execution status of “failed” to identify the number of failed tests during Joshua’s testing
- Using TWO criteria in the same filter (OR), filter = Assigned to “Joshua Tester” or execution status of “failed” to determine the test load for one individual on the test team OR the number of tests that have failed for the entire team
- Using THREE criteria in the same filter (NOT), filter = Assigned to “Joshua Tester” and not including component “wireless” or not created after October 1, 2006 to determine the test load for one individual on the test team but excluding the out of scope tests, wireless, or those tests created after October 1st.

Shown below is a completed test map for the test case filtering capability and priorities for test execution.

	A	B	F	G	H	I	O	P	Q	R	S	T	U	V	W	X	Y	AA	AB	AC	AD	AE	AF	AG	AH	AI
1	Test Case Filter Map																									
3	No	Test Case Criteria		One (min)		Two- Same				Two- Different				Many (max)												
4		A	B	C	D	E	F	G	H	I	J	K	L													
5		P/F	Not P/F	AND P/F	OR P/F	NOT P/F	AND P/F	OR P/F	NOT P/F	AND P/F	OR P/F	NOT P/F	AND P/F	OR P/F	NOT P/F	AND P/F	OR P/F	NOT P/F	MIX P/F	OR P/F	NOT P/F	MIX P/F	OR P/F	NOT P/F	MIX P/F	
6	1	Assigned to	A1	P*	B1	P*	C1	P*	D1	P*	E1	P*	F1	P*	G1	P*	H1	P*	I1	P*	J1	P*	K1	P*	L1	P*
7	2	Component	A2	P	B2	P	C2	D2	P	E2	P	F2	P	G2	P	H2	P	I2	P*	J2	P*	K2	P	L2	P	
8	3	Created after	A3	P	B3	P	C3	D3	P	E3	P	F3	F	G3	P	H3	P	I3	P	J3	P	K3	P	L3	P	
9	4	Created before	A4	P	B4	P	C4	D4	P	E4	P	F4	P	G4	P	H4	P	I4	P	J4	P	K4	P	L4	P*	
10	5	Created by	A5	P	B5	P	C5	D5	P	E5	P	F5	P	G5	P*	H5	P	I5	P	J5	P	K5	P	L5	P	
11	6	Created on	A6	P	B6	P	C6	D6	P	E6	P	F6	P	G6	P	H6	P	I6	P	J6	P	K6	P	L6	P*	
12	7	Executed	A7	P	B7	P	C7	D7		E7		F7	P	G7	P	H7		I7		J7		K7	P	L7	P	
13	8	Failed	A8	P	B8	P	C8	D8		E8		F8	P	G8	P	H8	P	I8	P	J8	P	K8	P	L8	P*	
14	9	Has History	A9	P	B9	P	C9	D9		E9		F9	P	G9	P	H9		I9		J9	P	K9	P	L9	P	
15	10	Has Links	A10	P	B10	P	C10	D10		E10		F10	P	G10	P	H10		I10		J10		K10	P	L10	P	
16	11	Has Notes	A11	P	B11	P	C11	D11		E11		F11	P	G11	P	H11		I11		J11		K11		L11	P	
17	12	In Folder	A12	P	B12	P	C12	D12		E12		F12	P	G12	P	H12		I12		J12		K12		L12	P	
18	13	Last Modified After	A13	P	B13	P	C13	D13	P	E13	P	F13	P	G13	P	H13	P	I13	P	J13	P	K13	P	L13	P	
19	14	Last ModifiedBefore	A14	P	B14	P	C14	D14	P	E14	P	F14	P	G14	P	H14	P	I14	P	J14	P	K14	P	L14	P	
20	15	Last Modified on	A15	P	B15	P	C15	D15	P	E15	P	F15	P	G15	P	H15	P*	I15	P	J15	P	K15	P	L15	P	

Figure 2: Executed Test Map for Test Case Filter Functionality

As you can see in Figure 2 the intern decided to apply the highest priority for testing to the conditions representing the minimum (columns F and H), maximum (column AH) and middle (columns U and W). Medium and low priority conditions appear in the middle of the spreadsheet in pink (columns Q, AB, AF) and blue (columns O, S, Y, AD).

Each cell in the map is coded to a Legend that appears in Figure 3 with test condition priorities carried forward onto the Legend.

	A	B	C	D	E	F
1	<i>Explanation of each cell of Test Case Filter map</i>					
2						
3	1			2		
4	Assigned to	A1 <Tester 1>		Component	A2 <Component1>	
5		B1 Not assigned to <tester1>			B2 Not <Component1>	
6		C1 <Tester 1> and Assigned to <Tester 2>			C2 <component1> And Component <component2>	
7		D1 <Tester 1> or Assigned to <Tester 2>			D2 <component3> Or Component <component2>	
8		E1 <Tester 1> and not Assigned to <Tester 2>			E2 <component1>And Not<component3>	
9		F1 <Tester 1> and Failed			F2 <component3> And New	
10		G1 <Tester 1> or Created by <Tester 2>			G2 <component2> Or Type<type2>	
11		H1 <Tester 1> and not Executed			H2 <component1> And Not Priority<priority>	
12		I1 <Tester 1> and Modified by<tester2> and Failed			I2 <component2>And Assigned to <tester1> AndType <type>	
13		J1 <Tester 1> or Last run on<date> or Modified			J2 <component 3> Or type <type1>Or assigned to <component3>And Not priority<priority1>And Not last run by<tester2>	
14		K1 <Tester 1> and not created before <date> and Priority <priority>			K2 <component3>Or created after<date>Or number before <date>And Not last modified after <Date>	
15		L1 <Tester 1> and Component<component> and not Executed or Failed			L2	

Figure 3: Legend to Test Map for Filter Functionality

Typically test maps act as design documents and not necessarily execution documents, but in this case, both objectives were met. The benefit for this technique was to ensure that beginning testers would not lose the big picture of what they were trying to test, i.e., that they understand the many conditions available to them and that they made good choices in the conditions to exercise.

An added benefit to these maps was for the senior staff to know where the testers were in their testing and where they needed to go. To ensure they were on track and testing as well as they could be, we found the test maps to be a useful tool. We required no further documentation other than these maps and bug reports.

In this first example the test condition selection process was easy. The intern picked the minimum, maximum and middle range for all filter options. But, sometimes the test conditions worth testing are spread throughout the map as shown in Figure 4.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	File Filters												
2	Filter Criteria	One	Two-Same			Two-different			Many (max)				
3			AND	OR	NOT	AND	OR	NOT	AND	OR	NOT		MIX
4	2 Created After	C3				G3	H3	I3	J3	K3	L3	M3	
5	3 Created Before	C4				G4	H4	I4	J4	K4	L4	M3	
6	4 Created by	C5	E5			G5	H5	I5	J5	K5	L5	M5	
7	5 Created on	C6	E6			G6	H6	I6	J6	K6	L6	M6	
8	6 Has History	C7				G7	H7	I7	J7	K7		M7	
9	7 Has Links	C8				G7	H8	I7	J6	K7		M8	
10	8 Has Notes	C9				G9	H7	I9	J6	K3		M9	
11	9 In Folder	C10	E10			G10	H10		J7	K3	L10	M8	
12	10 Last Modified After	C11				G11	H11	I11	J7	K11	L11	M10	
13	11 Last Modified Before	C12				G11	H12	I12	J12	K12	L12	M11	
14	12 Last Modified on	C13	E13	F13	G13	H11	I11	J13	K3	L13	M12		
15	13 Modified by	C14	F14	F14	G10	H15	I14	J4	K4	L11	M5		
16	14 New	C15			G5	H3	I15	J3	K14	L15	M5		
17	15 Please Review	C16			G16	H6	I16	J4	K15	L16	M3		

Figure 4: Test Map for File Filter Functionality

Here, the intern decided that conditions under “Two-same”, two filters with the same criteria (columns D, E, F) and a few other conditions in the Test Map were not worth exploring. These conditions are highlighted in orange (cells D4-17, E4-5, E8-10, E12-13, and L8-10) and essentially represent uninteresting or redundant conditions. Also, in reviewing the various conditions, the intern noticed that there was overlap; she highlighted these conditions in yellow (cells G9, G13, C15-16, H14-17, K9-11, and M15-17). All other conditions have an ID and are defined in the Test Map Legend. This test map demonstrates how upon review, some conditions just fall away or become combined with others. The tester’s evaluation of the conditions to exercise and which ones to ignore or combine is based on personal experience, judgment and business expectations, important aspects of exploratory testing.

Test Maps from Two Other Projects

On two other projects we decided to map our test conditions somewhat differently. On the most recent project, we were responsible for testing a corporate search function on our client’s public website that would allow users to find information more easily as well as allow our client to display focused ads. Display ads were live links that allowed the user to learn more about a particular product, solution, or service. And, users came in several flavors: guest, logged-in users and corporate partners.

The overall functionality was fairly straightforward and test maps were not required. But, one area of functionality proved to be much more complex than we had originally thought. Our approach to testing was based on a review of the specification, but we quickly found ourselves down a rat hole of complexity. Because we couldn’t get our minds wrapped around the software, we returned to the test map technique.

	A	B	I	J	K	L	M
1							
2			MINIMUM Site Sections Cases for PCA Link Pages				
3			Case 6	Case 7	Case 8	Case 9	Case 10
4			Min - PL	Min - PL	Min - PL	Min - PL	Min - PL
5	Online Status		On	On	On	On	On
6	Ignore Keyword match		Off	Off	Off	Off	Off
7	Site Sections						
8		All Pages	On	Off	Off	Off	Off
9		Products	Off	Off	On	Off	Off
10		Services & Industry Solutions	Off	Off	Off	On	Off
11		Support	Off	Off	Off	Off	On
12		Training	Off	Off	Off	Off	Off
13		For Partners	Off	Off	Off	Off	Off
14		Corporate	Off	Off	Off	Off	Off
15	Locale (Country)						
16		Country	USA	USA	USA	USA	USA
17		Language	English	English	English	English	English
18	Start date		7/1/2006	9/1/2006	7/1/2006	7/1/2006	7/1/2006
19	End date		10/1/2006	10/1/2006	10/1/2006	10/1/2006	10/1/2006
20	Keywords		servers	servers, tape systems	tape systems, workstations	servers, tape systems, workstations	servers, tape systems, workstations, VOIP
21	Action Link						
22		News bulletins					
23		Product Literature	Valid URL	Valid URL	Valid URL	Valid URL	Valid URL
24		Static Link					
25	Action Link Title						
26		Download the case study		Selected			
27		Download the white paper			Selected		
28		Learn more				Selected	
29		Listen to the podcast	Selected				Selected
30	Partner Type						
31		Level 1		Selected			
32		Level 2	Selected				Selected
33		None			Selected	Selected	
34	Other fields						
35	Headline		Valid value	Valid value	Valid value	Valid value	Valid value

Figure 5: Partial Test Map for Focused Ad Functionality

Our source of confusion was in the many different ways a focused ad could be tagged. Ads could be:

- online or offline
- displayed for any or designated keywords
- displayed for certain sections of the website, some or all sections
- displayed in a certain date range
- displayed for some, all or no corporate partners

The list seemed endless and we quickly got lost. We didn't know where we were, where we needed to go and what we had already done. We also realized that it was impossible to test every possible combination of criteria against each criterion. So we opted to sample our conditions. Figure 5 shows how we laid out the various ad options (column A and B) and how we built our tests (columns I-M) . In fact in this layout we were able to see patterns in such a way that it was very easy to see coverage and the needed variability (see sections Action Link, Action Link Titles and Partner Types in Figure 5).

While the application of the test map to this project is by no means a perfect solution, it did allow us to focus on the combinations of options quickly, lay-out the tests and get started with the job of testing. It also provided us a vehicle by which we could get client feedback on our approach without them having to read a mountain of test cases and test procedures. A quick snapshot of our approach as in this map allows reviewers to see what we are doing, how we are approaching the test and gives them the needed overview to provide meaningful feedback. A final benefit is that the map lends itself to the fluid nature of exploratory testing because maintenance of the map is quick, focused and easily adapted to new input. In fact, when we implemented the tests in our test management system, we created simple test titles and test intents and then referenced the map. This way test results could be attached to a test "shell" with the important detail remaining in the map.

On an earlier project we used the same style of test map to direct a team of engineers located in India. Again, I wanted not to focus on heavy test documentation, but to keep the documentation light and focused in support of solid testing. I was the test lead and responsible for identifying the tests that the India team would automate. While I wasn't thrilled with the automation solution, this strategy was in place prior to my having become the test lead and so I had to make the best/most of it. The software under test was an upgrade to an internal enterprise system in support of customer information. Various organizations in the corporation would request regular feeds from the customer data base for their mass mailings, targeted ad campaigns and other customer sales and marketing efforts.

As in the earlier examples, testing was complex. The many variables and combinations needed to be laid out in a simple, yet complete manner. A test map seemed the most appropriate. Figure 6 displays the resulting test map with functionality across the y-axis and potential test cases identified down the x-axis, column A. There were six variables:

- Timing of the data strip
- Volume data to be published (in each file)
- Format (of outgoing file)
- Delete (option of log files)
- Archive (option)
- Data base table (that was updated in the data base and used for verification purposes)
- File destination

Within each option, additional criteria were identified and embraced boundaries where it made sense. The focus was to group as many of the valid criteria together in as few tests as possible to ensure that we started with tests that were less obvious and larger than the typical unit tests. The test map was reworked a lot as I learned more about how certain options worked together and to spread criterion out as equally as possible and as made sense. In fact, during the process of completing the map, it was discovered that the test conditions identified in row 17 were irrelevant and not worth pursuing. Hence, it was grayed. I also accepted that it was impossible to test every option in combination with other options and so again, my approach was to sample and to allow the software and its bugs to "direct" the testing effort. My knowledge would improve the more I worked with the software and hence, my testing should sharpen and the map would be adapted.

	A	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1		Subscriber Tests - Valid Conditions														
2		Timing (STATUS=...)				Volume to be Published			Format			Delete?	Archive? (true/false)	Subscription Table		
3	Test Case ID	Subscriber	DEFAULT	RECORD	TIME INTERVAL	CLOCK	1rec/file	>1rec/file	xml rec	delimited txt file (pipe)	delimited txt file (comma)	true/false	Publisher	Event Type	Source	Dest
4																
5	1	TST1				noon		(Have a large number of records in the file.)				T	T/high	All	All	Gateway
6																
7	2	TST2		max				(Have the number of records in the file match the RECORD count.)				T	T/high	All	PMO	Gateway
8																
9	3	TST3				11:59:59						T	T/high	Create	All	Gateway
10																
11	4	TST6		Make all values bogus.								F	T/high	All	A,B,C	Corp website
12																
13	5	TST7		min+1	min	0:00:01						T	T/high	Create	B	Corp website
14																
15	6	TST8				23:59:59						F	T/high	Update Privacy	J	Corp website
16																
17				avg	avg	avg		avg				Y		Create	G,H,I	Gateway
18																
19	7	TST10		max	max	midnight		max				F	T/high	Update Privacy	All	Gateway
20	8	TST11			max			max				F	T/high (4)	All	F,G,H,I,J	Gateway

Figure 6: Test Map for Subscriber Updates to Customer Data

Conclusion

In summary, a test map is a handy test design tool that can augment exploratory testing. It has a number of benefits:

- Organizes your thoughts and aids the design process that allows you to get your ideas down on paper
- Can be the only test documentation on a project (aside from bug reports)
- Is one objective measure for determining or gauging testing progress and provides project mgmt oversight over an inherently fluid process
- Allows reviewers and/or management to understand the team's test approach and gives the reviewers and management the needed overview to provide meaningful feedback
- Is adaptable and lends itself to the fluid nature of exploratory testing because maintenance of the map is quick and focused
- Serves small, start-up companies as well as large, stable, highly regulated companies

From these maps further test case matrices and/or procedures can be written, but are not required. Exploratory testing, as recently as April 2006 in Bach's article, was defined as "an ongoing interaction of three activities: learning, test design, and test execution." Let test maps help with the "design" in exploratory testing and allow you to get your head wrapped around the testing challenge.

Adopting and Adapting Agile Development Practices in the Real World

Don Hanson

Abstract

Transitioning to an agile development model is a complex endeavor that impacts many individuals outside the immediate development team. Such an endeavor cannot be taken lightly, and a team must carefully assess their product and environment before beginning.

The most important of these questions are:

- Will this work for software products that go to market, as opposed to IT projects?
- How can I implement agile development in the real-world, where other stakeholders with additional constraints must be accommodated?
- With so many changes to make at once, what is the minimum set of changes I can start off with?

In this paper we'll answer these three questions and provide descriptions of what worked, and in some cases didn't work, at McAfee.

Biography

Don Hanson founded his first company and began writing commercial software in the early 90's, developing an evolving line of animation plug-in products. He has since developed commercial software products for the enterprise market and lead user-interface development for a mobile wireless navigation startup. Currently, Don is the development manager for McAfee's flagship enterprise security management product, ePolicy Orchestrator, used to manage security products with \$500 million in annual sales at McAfee, Inc.

Introduction

The benefits of agile development practices are well established. There are a myriad of books which describe agile development. However, these texts do not provide guidance to transition from traditional models to an agile one in real-world environments. In this paper we'll provide this guidance with descriptions of what worked.

Long before the first line of code is written, people in other parts of the company are making decisions based on the committed release date; marketing campaigns are

The individual aspects of your development situation are as varied as the number of companies developing software. Part of adopting a new development practice is adapting it to the realities of your situation. Such adaptations can be made to the process, its ownership, or its speed of adoption.

Over the past four years we have adopted and adapted agile development practices to fit the large corporate environment for software products.

The key practices of agile development reinforce one another, which is great once they're in place. The problem is getting them in place. While the number of practices is relatively small, each may be quite different from how your teams do things today. Anyone who's been around the block once or twice understands that making permanent changes in how people work together is hard to do.

With that in mind, here are the practices that worked for us, grouped by order of implementation. Each practice contains solutions that worked for us in overcoming the major hurdle to adoption.

Preparing for Agile Development

These practices lay the foundation for agile development. They have a low political risk because they work with almost any style of development.

High-bandwidth communication

"The most efficient and effective method of conveying information to and within a team is face-to-face conversation."ⁱ

I believe this practice is the foundation upon which the others are built. All of the other practices can be adopted and adapted more quickly and effectively with high-bandwidth communication in place. Management support of this practice is critical for agile development to work.

We like to co-locate in groups of four by replacing cubicles with "pods", e.g. removing the inner cubical walls to create a group space. We've learned that everyone must agree to the shared space. It will not succeed if you force anyone.

At McAfee we encourage teams to personalize their pod, to make it more than just extra floor space. A conference table for team meetings is a popular starting point. Server racks, projectors, lazy-boys and mini-refrigerators all have their place. Successful pods dispel the sterile cookie cutter sameness favored by corporate space planners and encourage team building.

For distributed teams instant messaging is a worthwhile advancement over email only interactions. This is one area where corporate standards can be a blessing. Avoid the tower of babble problems and select a single IM tool to be used by everyone on the project.

Regardless of your team's geographical distribution an easy to use collaboration tool is invaluable. We use a wiki; basically a website that you can edit from a web

browser. The wiki's ease of operation and organization fit well with our teams. Other sites have adopted Sharepoint, which provides a more structured experience by giving up some of that ease of operation.

Your team's communication needs will change over the course of a project. Early on there may be more discussions than on later iterations. Corporate environments can be a constant source of productivity killing interruptions. Near the end of a project, or when the interruption level seems too high, we specify a "quiet time" during a certain part of the day where team members can focus on their tasks free from interruptions.

Automated unit tests and nightly builds

Automated unit tests make your software development more predictable and repeatable. They help find problems in the code earlier in the development cycle. They also enable refactoring and egoless coding by validating that everything still works after changes are made.

While most people agree automated unit tests provide value, getting started using them can be difficult. Tight deadlines seem to preclude the time required to setup a unit test framework and adopt the pattern of writing unit tests.

Our answer has been to adopt automated unit tests between projects.

The automated unit tests are also integrated into the nightly builds. A build fails if any of the unit tests fail. It may be hard for some to imagine now, but this was not the case a few years ago.

In my experience most developers who've successfully used automated unit tests on a project adopt the habit for life. Projects I've been on have used the JUnit and CppUnit test frameworks.

Implementing Agile Development

In my experience these practices are the minimum set required to gain the benefits of agile development in a corporate environment.

Joint Development Program

Getting working code with valuable new features in front of real customers; that's what it's all about. One of the main problems of waterfall style development is the lack of feedback during software development. After the requirements are gathered and the project given a green light, customers are typically not involved again until the beta, which is far too late to have any meaningful impact on the product. The primary reason for a Joint Development Partnership (JDP) program is to get early and repeated feedback from actual customers on the delivered functionality. It involves working closely with a small number of customers during the development of the software. At the end of each iteration JDP members install, use and provide feedback on the software.

At McAfee JDP members are asked to:

- Install and use iteration code drops, and provide feedback.
- Help prioritize and provide feedback on user stories.
- Participate in usability testing.
- Participate in the weekly conference call.

In return, JDP members receive the following:

- Access to the user stories for all iterations.
- Access to relevant design and quality assurance documents (usually a small set of items).
- Early access to the product.
- Opportunity to influence the development and feature set of the product. As a result of the feedback user stories *may* be reprioritized.
- Interaction with the development, agile QA, and Design teams on an intimate level that is rarely experienced by 99.9% of customers.
- Opportunity to influence the look and feel and the workflow of the product.

We've run a number of JDPs at McAfee, some more successful than others. Over time we've learned a few best practices for running a successful JDP:

- Keep the number of JDP members small, e.g. 6 ~ 10. Strive for fewer, motivated participants.
- Use multiple means of communicating: weekly conference calls, regular emails to a mailing list that includes JDP partners and the team members, web presentations, surveys and when possible onsite customer visits.
- Incorporate JDP feedback when prioritizing your user story development. As early as possible have them rank your user stories by importance. One of my favorite techniques is to give JDP members a point budget to spend on appropriately priced user stories and features. Try to keep the number of options to less than 20. Be sure to get their feedback multiple times over the course of the project. This provides much better results than giving them your entire list of engineering stories and asking them to pick their top 500.
- Don't squander the opportunity by asking JDP members if they would like ice cream and cake (assuming they like both). More interesting is asking them;
 - Which they would like first?
 - What is their optimal ratio of ice cream to cake?
 - Would they be satisfied with just the one (for now)?
- Watch out for JDP members who don't participate or deliver feedback. It's natural to lose a couple over the course of a project. Set expectations up front and include them in your JDP agreement.
- Engagement doesn't have to be all or nothing. Often the reasons a JDP member no longer fully participates are beyond their control. A good relationship can often be maintained by asking them to participate in usability studies, broader email-only discussions, etc.
- Remember to make the JDP worth your customers' time. Ensure they feel appreciated, and that they can justify the cost of participation with their organization.

We went through a lot of pain learning what level of quality should be targeted for the JDP releases. In a nutshell, don't require beta or release quality for each iteration's code drop. With the limited amount of time between code complete and the code drop, it is not practical to try to achieve this quality.

Iteration code drops are tested to JDP quality. JDP quality enables testing of new

functionality in a meaningful way. Remember, the goal of an iteration code drop is to get customer feedback on the new, useful functionality.

It's worth mentioning that the overwhelming majority of JDP members I've worked with have reported the quality of the iteration drops being superior to earlier beta versions delivered using waterfall style development. When a team knows real customers will use and discuss with them in a few short weeks the software they're building today, they naturally tend to aim for a higher standard.

User and engineering stories

User stories are features described from the perspective of the customer's need or use of the feature.

Examples of user stories include:

- Users can configure their own dashboards.
- Administrators can publish dashboards that are visible to all users.
- Users can print dashboards.

Aligning expectations of what will be built and delivered is a persistent problem throughout a project. For example, it's hard to be sure everyone is talking about the same thing during project planning negotiations between stakeholders over specific features.

User stories help align expectations by removing ambiguity. On agile projects at McAfee the initial set of user stories is an integral part of the project planning process. They enable everyone to better understand how development plans to provide the requested functionality.

As projects get larger it's hard to balance the granularity of the user stories to keep the number small enough to be useful for planning but detailed enough to give confidence your estimates are accurate. We solve this problem by breaking user stories down into engineering stories. Engineering stories represent the engineering work required to satisfy the user story.

This two tiered breakdown helps with organization. Discussions with JDP members can focus on the smaller set of user stories and it is easy to keep track of what is required to support a given user story during iteration planning.

Characteristics of engineering stories include:

- Represent five days or less of effort.
- Has an associated QA acceptance test, e.g. a brief functional acceptance test
- Has an associated UI acceptance test for any UI components, if you have a separate design team

Using the example user story of "Users can configure their own dashboards", associated engineering stories might include:

- Create visual and interaction design (Design Team)
- Create db schema for storing dashboard & dashboard element configurations
- Show list of current dashboards
- Add dashboard (create empty, then edit)
- Edit existing dashboard
- Delete dashboard

QA acceptance tests

When we dropped the “big design up front” as part of our transition to agile development we threw a wrench into our QA planning process. We were left without a process for sharing detailed knowledge of what will be built.

To solve this knowledge transfer problem we introduced QA acceptance tests; a brief test which validates that the engineering story is complete by specifying the expected functionality to be delivered.

At the beginning of each iteration we use the following process:

- Agile QA talks with developers about the selected user and engineering stories
- Agile QA creates the acceptance tests
- Developers reviews acceptance tests and either sign off on them or propose changes and explain the reason for the changes

Acceptance tests are initially run by agile QA when a developer marks story completed and a second time at the end of the 6 week development cycle. The acceptance tests are also reviewed by whitebox QA when planning which areas of testing should be automated next.

Egoless coding

Egoless coding is a mindset briefly described as you are not your code. Developers have been likened to artists in that they often personalize criticism of their code. This unwillingness to discuss the inner workings of their code can eventually lead to a silo'd development team, where each person only works on code in “their” own area.

Egoless coding ensures many people review the code, by providing an environment that encourages team members to suggest improvements without fear that the originator of the code will be offended. It facilitates effective code reviews, pair programming and a shared understanding of the code base.

We found egoless coding fairly easy to adopt by focusing on the benefits to the developers. “Code Review Wednesday” has become a popular attraction. The code is projected on the wall; the chips and salsa come out while one person presents a particular system to the rest of the team.

With egoless coding everyone gains. Team members enjoy gaining an understanding of parts of the code with which they were less familiar, as well as exposure to new patterns or uses of patterns they hadn’t considered before.

In addition, a deeper knowledge of the entire system is spread across the team. This benefits development management by increasing the team’s collective flexibility and in the long run the company through the delivery of a more cohesive product.

Refactoring

"Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior." – Martin Fowler, "What is Refactoring"

Adopting refactoring as a development practice requires automated unit tests and nightly builds. This gives developers the confidence to change the code and know it still works. We adopted automated unit tests and nightly builds right away to enable refactoring.

We also encouraged egoless coding because it facilitates the adoption of refactoring in two ways. Developers are far more likely to refactor if potentially changing other people's code (for the better) is accepted and encouraged by the team. If the team's reaction to your brilliant refactor is to yell at you for touching their code, it's unlikely you'll be doing any more refactoring.

Egoless coding also spreads a deeper knowledge of the system across the team. More eyes on a given section of code and better understanding of how that code is used throughout the product translate into fewer missed opportunities for refactoring.

Iterative development

"Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."ⁱⁱ

Integrating iterative development into the corporate project planning process for the first time can be a harrowing experience. Big boss "Can you deliver these features by that date?" Expected answer; "Yes!" Wrong answer; "Maybe, if those are the most important features."

Correct agile answer; "Based on our detailed planning we should be able to deliver on that date and our development model guarantees you'll see working code throughout the project with the most important features being delivered first."

Within the corporate world you'll end up planning everything twice. Once to enable the project stakeholders to make the business case to proceed with the project and once more over the course of the project as you plan each iteration.

The good news is iterative development provides a more accurate assessment of project status throughout the development of the software. This is primarily due to the working code drops delivered at the end of each iteration. There is no integration Hell to go through. Everyone can see what is working and what is not.

Using this system it makes sense to prioritize delivering complete new workflows over partial ones. For this reason we create a theme for each iteration describing the new functionality that will be delivered. This theme plays an important role in the iteration planning meeting, which we'll cover in a following section.

Iterative development also provides an excellent opportunity to improve your developers' estimation accuracy because working code is delivered to customers

every few weeks, instead of once or twice a year. A short one on one conversation reviewing estimate accuracy and general impression of quality is often sufficient to increase estimate accuracy on succeeding iterations.

It is important the real focus of developers be on improving their accuracy and not matching the speed of other developers to complete a given story. Otherwise this will drive undesired behavior, such as sacrificing quality or unit tests to meet estimates. We use egoless coding, code reviews, code coverage and the Whitebox team to help reinforce the message that doing most of your stories well is preferred over doing all of your stories poorly,

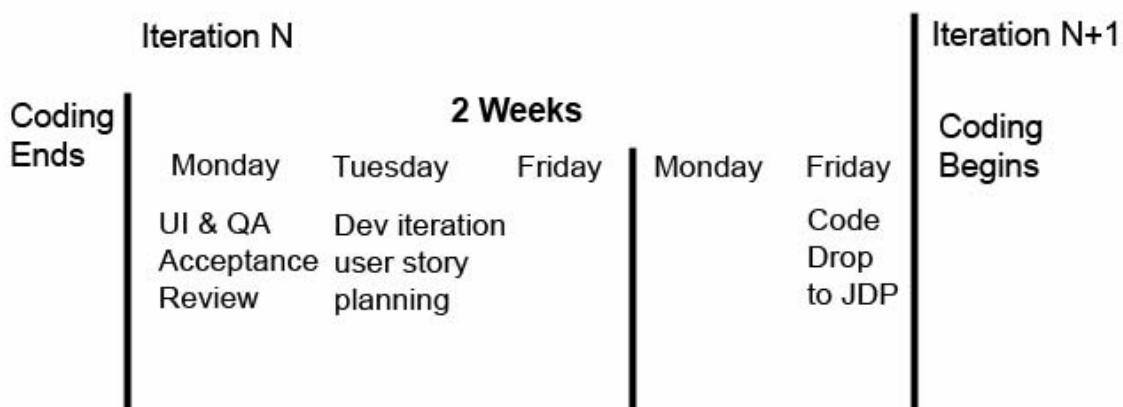
We'll cover the determining the length of your iterations and the iteration planning process below.

Determining the length of the iteration

So, how long do you let the team develop new user stories before calling the iteration complete? Much of the current literature advises "as short as possible," which is a lot like saying you should buy low and sell high in the stock market.

When considering iteration length, remember the goal is to optimize the number opportunities for meaningful feedback on useful, new features. If the installation is easy, customer representatives attentive and features fine grained then a shorter iteration length is possible. If the opposite is true, then lengthening the iteration is desirable.

After a fair amount of trial and error we settled on 8-week iterations; six weeks of development followed by two weeks of testing and planning. This budgeted enough time to create new features and get useful feedback from JDP members running the software.



Our eight week cycles are on the long end of what is recommended by some of the currently popular agile development methodologies for a number of reasons. When

we tried shorter cycles, we discovered that our JDP members and QA required more time with the software. We found that:

- JDP members returned some of their best feedback after they had time to become familiar with the new features and consider how they could be used in their daily routine.
- JDP members had a limited amount of time each week available to test the software due to their day-to-day duties (also known as their “real” job), firefighting and other demands on their time.
- Test labs, no matter how automated, take time to reserve, setup and prepare for each new code drop.
- QA required substantial time (longer than a week) to approve the software as JDP quality. This was due in part to the longer iteration length which in turn meant more new functionality was delivered each code drop, and in part to growing pains as our QA migrated from a traditional waterfall process.

When we tried longer cycles, we discovered that:

- The schedule was difficult to meet because it was hard to engage the development team right away due to a lack of a sense of urgency.
- There was not enough time to discuss at the weekly JDP meetings.
- There were not enough opportunities to get feedback on working code.

Initially there was a lot of resistance when people heard about the two weeks of test and planning. Reactions included comments that we’re wasting 25% of the team’s time over the course of the project. Nothing could be further from the truth.

Developers are busy fixing defects and incorporating feedback from the UI acceptance reviews. During the coding period there is no time to follow through on creative ideas for needed refactors or vexing problems. Any researching or prototyping is targeted for this time. If there are no open questions and bug counts are low, typically earlier in the project, developers can get a jump on the next iteration’s engineering stories.

Iteration planning meetings

Selecting the engineering stories for the next iteration from a list of 1200 plus items can be daunting. Not only are there too many items to think about at once, verifying completeness of the selected stories for an iteration is almost impossible. We’ve adopted the following process to make iteration planning easier and faster.

One or more themes for the iteration are selected before the meeting. This is usually done by the manager in conjunction with the architect or team lead.

Prior to the meeting the iteration themes are written on the whiteboard. Selection of the user stories for the iteration follows these steps:

1. Determine the team’s capacity for the iteration by accounting for holidays and each person’s planned vacations.
2. Team members select stories that support the themes. We project the user story spreadsheet on the wall and perform a group review of the appropriate categories of user stories. Using themes enables us to do this fairly quickly; 2~4 hours for up to 1200 engineering stories. We try to select stories that represent the mix of team members’ skills, but don’t try to be exact at this

- stage. The goal is to select all of the stories that support the themes and keep the selection equal to the team's capacity. If you consistently end up with a story selection more than double the team's capacity, try reducing the number of themes.
3. Team members prioritize the selected stories. The list of selected stories is quickly reviewed and each engineering story is assigned a priority.
 4. Team members choose stories to implement from the prioritized list. Starting with the highest priority stories, team members decide who owns which stories until their capacity is filled. The goal is to have people speak up for the stories they would like to do, but for new members there may be some assigning of stories. At some point the team's capacity is met though there may be stories left unselected. This is normal. During the meeting the team lead, architect, or both are responsible for ensuring the selected stories do not have any dependencies on un-selected stories. For example, the stories for creating the UI were selected but not the stories representing the required database work.

Newly discovered user and engineering stories are synched up with the master user story spreadsheet every week. They are placed in the bucket of unfinished stories for possible selection in future iterations.

Additional Best Practices

When to introduce these practices depends upon your situation. You may be doing some already, while some may not be appropriate for your situation.

Development and QA pairings

This one is as simple as it sounds. To facilitate high-bandwidth communication we try to have one or more members of the QA team work with the same developer over the course of a given iteration.

We did this one right off the bat. This simple practice really got our dev and QA teams talking on a level that had not been previously present. Many teams may already be doing some form of this. If so, don't stop! Other teams may need a little more time to adjust to the increased levels of communication.

Integrated QA and development team

To improve code quality as early as possible we split our traditionally waterfall oriented QA into two teams; agile and iterative.

The smaller agile team is comprised of co-located QA, including our white box team. During the six-week development phase they focus on risk-based testing. During the two-week testing and planning phase they re-run the QA acceptance tests and participate in planning the next iteration. To facilitate information sharing they are also responsible for the weekly code drop to the iterative team.

The larger iterative team contains local QA and all of our distributed QA members. They are responsible for shipping a quality product and the innumerable tasks that requires.

The challenges faced by this team when migrating from waterfall to iterative development models should not be discounted. Many basic processes change; for example, test cases are now defined and built up over the life of a project. In addition, a more active and participatory role is now required; an understanding of the current and planned functionality is critical to writing useful defects.

If you decide using an integrated-QA approach is not for you, then I recommend occasionally revisiting the question of using one. You might be in for a pleasant surprise down the road.

Continuous integration

Integration is difficult. We've all been through "integration Hell" in one form or another. Why would anyone want to do it more often?

The main reason is to reduce the complexity of each integration; fewer changes since the last integration mean fewer things that can break and fewer places to look when things do break.

Continuous integration is the practice of integrating on every code check-in via an automated build and test framework. This sounds hard but builds quite nicely upon automated unit tests and automated nightly builds with pass/fail reporting.

We've had good success with CruiseControlⁱⁱⁱ, an open source framework for a continuous build and integration process, for our Java and C++ based projects. Upon every code check-in, CruiseControl:

- Performs a full build.
- Runs all unit tests.
- Initializes test database.
- Starts server product.
- Run all system tests.
- Shuts down server product.
- Sends e-mail notification of the results.

We know whether it passed or failed within 10 minutes after check-in!

There are projects where continuous integration won't work, such as builds that require 24 hours, but once you've been on a project with continuous integration it's hard to imagine life without it.

Code coverage

Most developers have extremely negative reactions when the subject of code coverage is brought up. Upon further inquiry, most respond favorably to what code coverage tools provide. Their dissatisfaction usually stems from situations in which code coverage was misused as a management tool.

Developers and whitebox testers are beginning to use code coverage tools to improve their unit- and system-level automated tests on new code, and direct their efforts to add tests for existing code.

We selected Clover as our code coverage tool for Java code due to its tight integration with JUnit, our automated unit test framework, and CruiseControl, our continuous integration server. Emma, a popular open source code coverage tool was a close second.

Conclusion

The benefits of agile development over traditional water practices are well established. How to make the transition from traditional models to an agile one in real-world environments is less well defined.

The individual aspects of your development situation are as varied as the number of companies developing software. The transition to agile development will involve customizing agile best practices to your environment.

We discussed the set of agile practices that enabled us to integrate agile development into the enterprise software development.

We saw how user stories and iterative development with working code drops can help integrate agile development in situations with other stakeholders that must be accommodated.

We also called out the minimum set of changes required to get started. Using these practices to build upon a foundation of high-bandwidth communication it is possible to make the transition to agile development in the real world with a high probability of success.

References

"Refactoring" by Martin Fowler

"Continuous Integration" by Martin Fowler -
<http://www.martinfowler.com/articles/continuousIntegration.html>

ⁱ Principles of Agile Software, <http://agilemanifesto.org/principles.html>

ⁱⁱ Principles of Agile Software, <http://agilemanifesto.org/principles.html>

ⁱⁱⁱ CruiseControl homepage: <http://cruisecontrol.sourceforge.net/>

Improving Test Code Quality

Brian Rogers <brian.rogers@microsoft.com>
John Lambert <jlamb@microsoft.com>

Errors in test code are frequently overlooked. Hunting for test errors is often deemed a laborious and time-consuming process that is not worth the effort. Some feel that any time a tester spends away from writing new tests is time lost. However, with proper tooling and a formalized, low-impact process, large gains in test quality can be achieved with a small investment.

There are hidden costs involved when teams neglect to account for test code quality. Time spent analyzing test issues translates to loss of productivity for testers and less time overall that could be spent finding bugs in production code. Tests with significant issues may not run reliably and may obscure product bugs.

In general, test issues can be divided into categories of severity (errors, warnings, etc.) and general issue type, ranging from environmental sensitivity (e.g. reliance on non-invariants such as operating system versions, languages, and locales) to correlation problems (e.g. incorrect mapping between test definition data and test implementation).

The test quality process, to be most effective, should rely heavily on automation and be handled by a small, knowledgeable group of individuals. When addressing test errors, it is important to set realistic and properly paced goals and to obtain the cooperation of management and individual testers. Education of other testers, including explanations of the underlying reasons for why the errors must be fixed, is key to garnering support from management and individuals.

The tools involved in finding test issues are essential to keeping the process efficient. Tools need not be complex; simple static analysis using a lightweight rule-based engine is enough to find the most common test errors. Over time, maintenance of the tools and process will ensure long-term success of the test quality process.

Brian Rogers is passionate about software engineering. In his current role as a software design engineer in test at Microsoft, he leads a small working group of testers who oversee test quality and enforce best practices for a set of feature test teams. He received a BS in Computer Engineering from the University of Washington in 2003. When not in front of a keyboard and monitor, Brian likes to cook, hike, and write and play music.

John Lambert is a test technical lead at Microsoft. He works on test automation, test methodologies, and penetration testing for the Windows Communication Foundation (“Indigo”). He is an inventor or co-inventor on six patents, four of which are related to testing. John graduated with Bachelor of Science and Master of Science degrees in computer science from Case Western Reserve University in Cleveland, OH. During college, he spent a summer as a Program Manager Intern at Microsoft and a summer as a Research Intern at Digital. John has presented at the Pacific Northwest Software Quality Conference (PNSQC), Seattle Area Software Quality Assurance Group (SASQAG), and STAR.

1. Introduction

The primary duty of a test team is to provide an objective measurement of the quality of a given product before it is released to customers. This is largely achieved by writing and running tests, reporting defects based on the results, and pushing for fixes where appropriate. Of course, this process is not without its problems.

Many testers are well aware that testers themselves are not infallible. Test code can contain issues which cause tests to run improperly and report incorrect results. This is a simple fact of the test development process and can occur no matter how careful or experienced the tester.

The situation is not hopeless, however. With appropriate tools and a lightweight process, test code quality can be measured and enforced in a largely automated way. It is our hope that the following ideas will lead to improvements in test code quality and fuel the creation of test quality initiatives across many test teams.

2. Motivation

Before diving into a test quality initiative, it is instructive to explore the reasons why we must care about test code quality. First and foremost, the time we have to test a product before it is released is limited. To make sure we as testers are using our time in the most efficient way, we should be sure that the majority of our energy is spent finding product defects and exercising new scenarios. It follows that the more time we spend fixing existing tests, the less time we will be able devote to new and untested features. If we can quickly find common test issues that cause such failures, we can apply fixes and then move on to more important work.

Another important consideration and motivator for test code quality is the fact that test code, as with any code, often outlives the author. Tests ownership can change between members of a feature team. In larger companies, tests can be handed off to a completely different organization after a product is completed. If these tests do not adhere to a minimum level of quality, they are more likely to be fragile, prone to error, and hard to understand and debug. If the original author is no longer available to deal with these issues, the new owners will often waste much more time finding and fixing the issues.

Finally, there are some important psychological aspects of testing to consider which are impacted by test code quality. Testers that produce test code with few issues are more likely to be confident in their work. On the contrary, test teams with test code that often fails, regardless of the state of the product, will lose confidence. Loss of confidence leads to loss of trust in test results, as each new failure will naturally be blamed on issues in the test automation itself. This leads to the perception that the test team cannot be counted on to give an accurate measure of product quality; thus, test will have failed in one of its fundamental responsibilities.

3. Categorization of Issues

To be able to measure test code quality, we must first be able to categorize test issues. We have found that most issues fall into five general categories: environmental sensitivity, correlation problems, standards violations, design issues, and statistical anomalies.

3.1 Environmental Sensitivity

Environmental sensitivity refers to the reliance of tests on non-invariant environmental factors. This issue can reveal itself in a variety of forms which are best represented by example.

Consider a test of a user control called a FileSaveWidget which accepts an input file path and saves data to the specified file. A simple test of this component is shown in Figure 1.

Figure 1: Sample code for FileSaveWidget test

```
1 void TestFileSaveWidget()
2 {
3     // this file contains the expected output of the file save widget
4     string baselineFilePath = @"\\test-server\ TestData\baseline.txt";
5
6     // the local path for the save file
7     string outputPath = @"D:\datafile.txt";
8
9     FileSaveWidget widget = new FileSaveWidget();
10    widget.SetDataFile(outputPath);
11    widget.Save();
12    Try
13    {
14        VerifyDataFile(baselineFilePath, outputPath);
15        WriteTestResult("PASS");
16    }
17    catch (Exception e)
18    {
19        String errorMessage = e.Message;
20        if (errorMessage.Contains("File not found"))
21        {
22            WriteTestResult("FAIL");
23        }
24    }
25 }
```

The code looks straightforward enough but there are at least three issues in this example. On line 4, we notice the use of a remote file share on a machine called `test-server`. There is no guarantee that this server will be visible if the test is run on a different network than the tester originally intended.

Line 7 contains a hard coded reference to drive D. Perhaps all machines that this tester has encountered so far have writeable partitions on D, but this assumption may lead to failures later on (D could be a CD-ROM drive, or simply nonexistent).

Line 19 hides a more subtle but important issue. The expected error message in the case that the save operation fails may indeed be “File not found” – if the test is running on an English operating system or English build of the product. However, a Spanish localized build may return “Archivo no encontrado” in this case, which would not be caught by this test.

3.2 Correlation Problems

A test case rarely consists of only a single source file. Testers rely on test case databases to mark the status of a test and to store links to test case files. Makefiles are used to tell the build system what code should be compiled and where to put the resulting test binaries. External data files may be required for some tests to run (see Figure 2). If even one of these pieces is not

synchronized appropriately, runtime errors can occur. These errors are especially common because it is easy to forget to update the test case database when a filename changes, or to miss a trailing backslash in a makefile.

Test metadata should also match up with reality. Tests often go through changes in ownership. If someone leaves the team or the company, the tests written by that person will not leave with him. As such, it is necessary to periodically check that all user names associated with a test are valid and current.

Metadata also includes test requirements, the prerequisites that must be fulfilled before a test can run. A well-established product that has gone through several releases will often have several versions that must be concurrently supported. In cases like this, test teams will often maintain multiple versions of the tests, some of which are valid on only certain product versions. Lines 13 and 14 below show a possible way to express such requirements. Suppose, however, that the tester made a typo and the test cannot run on version 3.0, or even worse, 3.0 does not exist. In the best case, the tester has to waste time analyzing a test that should not have run. In the worst case, a test that should have run will be skipped over and the feature will not be exercised until the problem is noticed and resolved.

Figure 2: XML test definition file

```
1  <Test>
2    <Owner>JLamb</Owner>
3    <Name>TestFileSaveWidget</Name>
4    <Description>
5      Tests basic features of the
6      FileSaveWidget
7    </Description>
8    <Variations>
9      <!-- use external data file -->
10     <Expand File="TestFileSaveWidgetVariations.xml" />
11   </Variations>
12   <Requirements>
13     <OSVersion>5</OSVersion>
14     <ProductVersion>3.0</ProductVersion>
15   </Requirements>
16 </Test>
```

3.3 Standards Violations

Most test teams have a set of accepted practices that testers are trusted not to violate. For example, it is now increasingly common for customers to be granted access to source code; understandably, offensive terms are frowned upon and should be avoided in product and test code to avoid negative publicity. Similarly, it is unwise from a security standpoint to put passwords and other sensitive information directly in source code files.

Occasionally a test will take longer to implement than expected. This remaining work is sometimes encoded in “TODO” comments interspersed throughout the code. It is certainly acceptable to do this while a product is still in beta. However, it is not advisable to release a product when tests are only partially implemented. Such “TODO” comments should at least be periodically audited and surfaced as issues to avoid losing track of them.

3.4 Design Issues

A design issue is a fundamental flaw in how a test is implemented. Exactly what constitutes a design issue will vary depending on the specific test organization, but many practices are not recommended in general. The sample code in Figure 3 illustrates some of these problems.

Figure 3: Sample code for FileSaveWidget test

```
1 void TestFileSaveWidgetWithLargeFile()
2 {
3     FileSaveWidget widget = new FileSaveWidget();
4     string veryLargeFilePath = @"VeryLargeFile.txt";
5     widget.SetDataFile(veryLargeFilePath);
6     string displayedPath = widget.GetDisplayedFilePath();
7     WriteTestLog("Checking if display path is correct.");
8     if (displayedPath != veryLargeFilePath)
9     {
10         WriteTestResult("FAIL");
11     }
12
13     // this will take a while, but the I/O runs on a different thread
14     widget.BeginSaveInBackground();
15
16     Thread.Sleep(10000);
17     WriteTestLog("Checking if file exists.");
18     if (!File.Exists(veryLargeFilePath))
19     {
20         WriteToConsole("File didn't exist.");
21     }
22     else
23     {
24         WriteTestResult("PASS");
25     }
26 }
```

The code in Figure 3 is another test of the FileSaveWidget mentioned earlier. This particular test ensures that the FileSaveWidget displays the correct file name and that it can handle large files appropriately.

If the widget displays the wrong path, we respond to this on line 10 by writing a “FAIL” result. However, what might be more useful than that is printing out why the test failed, e.g. “Fail: Expected path X but got path Y.” Descriptive failures are very helpful, especially when the test author and the test runner/analyizer are different employees.

Since the test is attempting to save a very large file, it is likely that the operation will take a while to complete (the tester even makes a note of this fact on line 13). The FileSaveWidget does input/output (I/O) operations on a different thread so the call to `widget.BeginSaveInBackground()` returns immediately. To compensate for this, the tester calls `Thread.Sleep` on line 16. However, this is nondeterministic and prone to failure on any machine which happens to have a slower disk drive. It is far better to wait on an event that fires explicitly when an operation is complete. (If the FileSaveWidget does not provide an API like this, it is probably a product design issue.)

Assuming the I/O operation is complete, the test then checks if the file exists on disk. If it does not, the tester makes a note of this on line 20 by writing a message to the console. However, this is at odds with what was done on lines 7 and 17, where messages are instead written to the test log. Perhaps only the test log is saved and console output is simply discarded, in which case the message on line 20 will never be seen by anyone.

3.5 Statistical Anomalies

Depending on how test results are stored, it may be prudent to track potentially problematic trends in test automation. For instance, it is interesting to call out tests that have never passed. A test that was written one week ago that has never passed may not be an issue; if six months later the test has still not passed, it is definitely worth figuring out why. Also, we should look at tests that have never run in the lab and tests that have always passed. Tests that have always passed are deceptively problematic because few people will think to look for problems in a test that appears successful. However, if a test passes even when the product being tested is not installed, this surely indicates a lack of verification steps in the test.

Test durations can also be a useful metric to track efficiency in the test lab. While runtime performance is rarely a primary concern of a test developer, it may be a worthy consideration if it is found that, for example, 10% of feature tests take 90% of the machine time in the lab.

4. Prioritization of Issues

Not all test issues are created equal. Some issues are inherently more important than others, and prioritizing them is essential to guiding test quality work to the most urgent areas. Issues can be logically grouped into three severity categories: errors, warnings, and reporting.

4.1 Severity: Error

Errors are the most severe test issues and are generally regarded as “must fix.” If an issue prevents a test from running or prevents a test from executing correctly, it should be considered an error. Issue types that might be considered errors are the use of hard coded strings instead of properly localized resource strings (especially if globalization testing is a priority), or missing data files required at runtime.

4.2 Severity: Warning

Warnings, just like compiler warnings, are less severe than errors and may even be acceptable in limited circumstances. Generally, warnings alert testers to potential problems and are treated as “should fix.” Examples of issues that might be considered warnings are uses of hard coded paths and server names or TODO comments.

4.3 Severity: Reporting

Reporting level issues are mostly informational and may require no specific action. Any of the statistical anomalies listed previously could be seen as reporting level issues. Items in this category can be used to guide investigations that lead to overall test improvements. For example, if a set of tests is identified as taking an unusually long time to run in the test lab, it

may be appropriate to investigate the source of the performance issues and resolve them to decrease the total duration of a test pass.

5. Implementation of the Process

A successful test quality initiative should follow a lightweight well-defined process. In general, there are four parts to this process: defining the scope, addressing the backlog, prevention of regressions, and continuous improvement.

5.1 Defining the Scope

To define the scope of the test quality initiative, we must decide who owns the initiative. This role is ideally suited for a test architect, but almost any senior tester can do this job. A small group of additional participants should be assembled to take care of general work items of the initiative, including tool development and support tasks. The group should be large enough to be able to meet the demands of the test quality initiative but small enough to maintain agility. We have found that four people can effectively oversee test quality for a 100 person test organization.

The owner must define the goals of the initiative and is responsible for obtaining management approval of these goals. Test quality goals should be clearly defined and explicitly state a timeline when results should be achieved. For example, “All error-level issues will be eliminated before the start of the Milestone 5 full test pass.”

5.2 Addressing the Backlog

At the beginning of the process, there will undoubtedly be a large backlog of test issues that were previously unknown. It is helpful at this point to look for outliers in the result set and identify areas where a small amount of effort can eliminate the largest number of issues. For example, testers may use auto-generated code and data files for their tests. If thousands of test files are generated by a single tool which creates bad code, thousands of errors can be eliminated by simply fixing the tool. When large scale test fixes are made by a few individuals on behalf of the test team, this should be clearly communicated to the rest of the team; this will promote understanding of the specific test issues and thus prevent the problems from reappearing.

Once the result set has stabilized and the core test quality issues have been identified, the test quality group can drive individual testers to fix their errors. This process works best when the management team is directly involved in tracking progress for their direct reports; this is especially true if the members of the test quality initiative are not managers themselves. The message should be “Tests with quality issues are not finished.” However, it is not sufficient to simply state the problems; to ensure that test issues are addressed, testers should be presented with ways to address the problems and given information about why it is a problem in the first place.

Goals should be aggressive but also mindful of other scheduled activities. Testers are already pulled in many different directions due to the daily demands of the job so appropriate tradeoffs should be made if time constraints begin to obviate the originally stated quality goals. In the event where adjustments are necessary, quality issues should be addressed in severity order. For example, the initial goal may have been to eliminate all errors and warnings before the next full test pass; in reality, teams may not have time to address every TODO comment in

their source code, but should definitely fix their hard coded resource strings before running tests on a localized build.

5.3 Prevention of Test Quality Regressions

As testers are well aware, product development teams struggle with regressions; a developer may fix one bug but neglect to fix the root cause, leaving the product vulnerable to future issues of the same type. A similar problem exists with respect to test code quality. We must avoid becoming trapped in a cycle of fixing problems, only to see quality issues being reintroduced upon the next check-in.

Maintaining a consistent level of test quality is key. The owner of the initiative should send periodic reports to the team as a whole that show how many errors, warnings, and so on are present for each feature team. Transparency and visibility are essential; no one wants to be on the feature team that checks in 100 quality issues, especially if everyone is watching the issue counts.

Fixing current issues is good, but preventing introduction of issues is far better. We recommend empowering individual testers to audit and ensure their own test quality by giving every tester access to the scanning tools and providing instructions on how to run them. This allows everyone a fair chance to catch bad code before it is checked in.

5.4 Continuous Improvement

After a few iterations of the process, some patterns may develop. If testers continually make the same mistakes, this could indicate one of several problems. One possibility is that test development guidance is lacking or incorrect; to address this, test quality education meetings can be held. Another possibility is that the test automation area is simply problematic and more difficult than it should be; this can be addressed by appropriate changes in test libraries or improvements in automation tools. For instance, if testers are consistently forgetting to sync up their checked in test cases with the test case database, perhaps this process should be fully automated to remove all possibility of error.

Feedback from testers is an important part of the improvement process. In our experience, close to 50% of the issue types that we have identified have been brought to our attention by testers that have actually made the mistakes. On a related note, it is imperative to give testers a forum of some sort to ask questions and make suggestions to further the test quality initiative. An e-mail list works well for this purpose and provides a written record of communication as opposed to less formal “hallway chats.”

As more quality issues are addressed, the error threshold can steadily be lowered. When there are thousands of legacy issues, it makes sense that only critical errors will be addressed; however, as soon as the backlog is cleared, it may make sense to promote some warning level issues to errors.

6. Implementation of the Tools

The test quality initiative can only thrive with appropriate scanning tools to automate the laborious task of finding test issues. While code reviews can certainly uncover some of the issues, static analysis tools are much faster and more efficient. Given that even a medium sized test organization may have hundreds of thousands of lines of test code, it quickly becomes infeasible to rely on humans to do this work.

The most generic test quality scanning tool should adhere to a few basic principles. First of all, it should be able to check any and all test artifacts. This will include code files, but should also be applicable to data files, test case databases, and more. The scanning tool should be able to use rules to scan these artifacts; rules should be simple to add and remove as needed.

Since many teams already have a set of static analysis tools, the quality scanning tool should integrate with them; it is not the goal to replace existing processes, but to make the results of such processes more visible and useful. Following from this point, it should be simple to collect scan results and compile them into reports suitable for consumption by upper management all the way down to individuals responsible for fixing issues. Finally, the tool should be easily extensible and provide a simple object model for adding new rules and artifacts.

6.1 Object Model

A reference implementation of a quality scanning tool should define six components: Teams, Targets, Rules, Issues, Exclusions, and Reports.

Teams are essentially groups of people and provide a way to enforce ownership of quality issues. A Team in the object model in most situations will reflect the makeup of an actual feature team; this makes reporting results per feature team easier. However, there is no requirement that a Team follow a particular organizational chart. Some test organizations may have a group of informally connected individuals (often called a “virtual team”) that own common code libraries; this set of individuals could still be represented by a Team in the object model.

Targets are generic test artifacts, as described above. The Targets can be thought of as the data on which the scanning tool operates. In general, Targets should not be tied to any particular type of object and should be flexible enough to include flat text files as well as database rows, test case definitions, and anything else that a test organization produces.

Rules are the operators of the scanning tool. Each Rule knows how to check a specific type of Target and will generate zero or more Issues as a result of these operations. For example, a test team might want to write a rule that flags hard coded passwords in test files. In the object model, this might result in a PasswordRule which operates on TextFileTarget to raise a HardCodedPasswordIssue.

Issues represent test quality problems. As stated earlier, each Issue has an associated severity level and should ideally be assigned to a Team (or member of a Team). Issues may contain information about how to resolve problems, or contain links to additional documentation to help guide testers toward an appropriate solution.

Exclusions are the mechanism by which specific Targets or Issues can be removed during the scanning process. No tool is perfect and hence false positives are bound to show up when using static analysis tools. It is important to allow for limited Exclusions to make the result set as accurate as possible. This is comparable to the use of “pragma” directives to disable compiler warnings. In simple terms, we need to handle occasions when the tester knows better than the tool. Exclusions should be rare and as such, over-excluding may indicate that legitimate Issues are being hidden or that the tools need more work to improve scanning accuracy.

6.2 Reporting

Reports are the output of the scanning process and at a minimum should contain a summary of Issues found. We have had success using an Extensible Markup Language (XML) format for Reports, as they can be easily parsed and reformatted in a variety of ways using XPath

and Extensible Stylesheet Language Transforms (XSLT). Two reports should be prepared on a regular basis, a summary report and a detailed report.

The summary report contains the minimum information necessary to communicate the bottom line, appropriate for managers or others who only want at-a-glance statistics. If there are multiple smaller sub-teams in the test organization, the results should be rolled up by sub-team. See Figure 4 for an example. In this particular report, each team can get more details by following a link from their team name. It also includes contact information for members of the test quality initiative who are presumably the best contacts to assist testers in fixing issues. We recommend widely distributing the summary report in nightly e-mails to keep everyone informed of progress being made.

Figure 4: The summary report

Test Quality Report: October 12, 2006

Team	Errors	Warnings
<u>Widgets</u>	12	17
<u>UI</u>	9	8
<u>Servicing</u>	4	11
<u>Performance</u>	3	1
<u>Stress</u>	2	9

Click on the team name to view the detailed report. For more information, please contact [Brian Rogers](#) or [John Lambert](#).

The detailed report contains the full list of issues found, where they were found, and optionally, how to fix them. This report is typically more fine grained than the summary report and may be generated separately for each sub-team or even on a per tester basis. Figure 5 shows a portion of the detailed report for the Widgets sub-team. Where possible, each issue is given an owner who will be expected to follow up and fix the problem, using the guidance given in the Details section.

Figure 5: The detailed report

Test Quality Issues for Widgets: October 12, 2006

Issue	Location	Owner	Details
Hard-coded string	Tests\Widgets\TestFileSaveWidget.cs:20	BRoger	Hard-coded exception string “File not found”; please use ResourceLibrary to comply with localization testing.
Incorrect product requirements	Tests\Widgets\TestFileSaveWidget.testdef:14	JLamb	Specified non-existent version “3.0”; please correct typo.
Invalid user name	WidgetTestDatabase: Test case ID 31337	(N/A)	Test case assigned to invalid user “Nobody”; please assign to a current team member.

7. Conclusion

It is imperative for test teams to address and eliminate test quality issues. Quality issues vary in type and severity but will invariably lead to annoyances in the near term and wasted time in the future. Improving test code quality involves looking at not only test code files but the subtle interactions between test definitions, test data files, and other such artifacts.

A successful test quality initiative is best handled by a small well-organized team employing a well-defined but lightweight process. The quality process should make heavy use of automation to efficiently find issues and generate regular reports for consumption by management and individual testers.

Long term success of the test quality initiative requires involvement from all members of the team. Feedback and suggestions from testers should be taken seriously. Continuous improvement of the process and tools as well as increasingly aggressive quality goals can lead to fast reductions in the backlog of issues. The more quickly that test quality reaches an acceptable level, the sooner the test team will be able to move past existing tests and write new tests to exercise more product functionality and eventually find more product defects.

CyberHunters
Diving into the Inner World of Test Engineers
John Copp

Biography

John Copp studied Cree Indian subsistence hunters on Canada's James Bay under a fellowship from the Guggenheim Foundation. As a consultant to the United States Fish and Wildlife Service, he designed and managed the annual migratory waterfowl subsistence harvest survey conducted among Yup'ik Eskimos on Alaska's Yukon-Kuskokwim Delta National Wildlife Refuge. The survey provides a model for similar surveys elsewhere on Federal lands in Alaska. Extensive use of computers in research provided a segue to a career in industry, where he has served as a developer, a database administrator, and a test engineer. He holds an M.S. in Computer Science, a Ph.D. in Psychology, and has been a commercial fisherman on Alaska's Bristol Bay. He currently works for McAfee, Inc. in Beaverton, Oregon.

Abstract

Reality is often unexpectedly complex, both in Nature and in engineered systems. Software in particular often favors us with ugly surprises. Software testing has evolved as a means of mitigating against such surprises. However, the growing complexity of software systems poses a cognitive challenge to testers. Uncovering defects hiding deep beneath the surface is not always easy. To hunt them down, a tester must often think and act in unconventional and creative ways. Some testers are exceptionally successful in doing so. But what personal traits contribute to such success? Software testing resembles other human activities, such as hunting and commercial fishing, intended to discover and capture elusive prey. Examples are drawn from these seemingly disparate domains to illustrate common traits such as persistence, inventiveness, and sense of humor. Hunting for tiny computer bugs may, in fact, simulate the hunting for large, dangerous game undertaken so successfully by our Paleolithic ancestors. A better understanding of the mind of the software tester could yield more effective recruitment, motivation, and management of talented test engineers. The beneficiaries would not be just the testers themselves, but also companies and the industry as a whole. Opening our own minds to new ideas about human psychology could indeed help us forge a culture of excellence.

“Belief and seeing – they’re often wrong.”
Robert McNamara, *The Fog of War*

Mind

Mind, that cloud of neural dust swirling through the brain’s vast inner universe, is the key difference between our species, *Homo sapiens*, and other life forms. Some scientists and popular writers have gone a long way to downplay the difference between ourselves and other creatures. Chimpanzees, for instance, have been shown to grasp the rudiments of language, make primitive tools, and murder each other. In that regard, they are just like us. But no chimpanzee has yet to write a symphony, solve a calculus problem, or test computer software. Although given the quality of some of the software I have seen, a chimpanzee would do just fine.

“Sapiens” is derived from the Latin *sapere*, to taste, be wise. Let’s savor those two words for a moment. Together they provide a foundation for our most cherished freedom, the freedom to choose, and our most compelling responsibility, the need to make good decisions. Taste means to sample, to consider, to savor. To taste properly requires adequate time, appropriate experience, sufficient curiosity, and proper self-restraint. A wine connoisseur doesn’t swill down a glassful, although a drunkard will. The connoisseur makes an informed decision.

Wisdom: The ability to discern or judge what is true, right, or lasting.
- Answers.com

Wisdom, then, is about making the right decision. But the right decision is not necessarily the obvious decision. The truth may lie hidden from view.

“Rational decisions are often unwise, and wise decision often appear to be irrational”
- Scott A. Whitmire

Decision-making in turn is driven by uncertainty. Creatures with a simpler neural cortex never agonize over decision-making. Trout sees fly, trout eats fly, only later, when he’s sizzling in a pan, to realize that the fly was a phony. In contrast, uncertainty dogs the human condition: where to eat, who to marry, what to believe in, who to vote for, and when to release software.

Testing and Tasting

Reality is often unexpectedly complex, both in Nature and in engineered systems. Outcomes are determined by multiple events, often interacting with one another in non-linear, unexpected ways. Things are almost never as they seem. Ask Robert McNamara, architect of the Vietnam War. Software systems, too, favor us with ugly surprises. Truth can twist and turn right before our very eyes. First glance is often off the mark. “*Belief and seeing – they’re often wrong.*” To successfully navigate in this space, Mind must remain agile, open, and inventive.

Making wise decisions is a cultivated skill, part method and part art. One key ingredient is testing, or symbolically tasting, various alternatives. Testing – testing of anything – has certain logical prerequisites: a problem, a method of discovery, and a solution. There is a nebulous cognitive space between recognizing a problem and discovering a solution, a space that can be called the “Maze of Uncertainty”. There are many paths inside the Maze, but only a few, and sometimes only one, leading out. Take too many wrong turns, loose your way, take too long, and a monster devours you. Not every person has the nerve to enter the Maze of Uncertainty, and not every one who enters is clever enough to escape. Yet some enter it repeatedly, willingly, even

gleefully. Software testers spend their days wandering through the Maze – agonizing over issues, bumping their heads against the wall, and watching over their shoulder while the clock ticks.

Method and Purpose

I must confess to a secret agenda. I want to lure you out of your cube and into the fresh air. I want you to rub shoulders with commercial fishermen and Native subsistence hunters. Cube life is a little too cozy. It's comforting to sit there in a temperature controlled environment, safe from physical danger, and think that you understand reality.

The purpose of this exercise is not just to tell hunting stories but to identify human characteristics that contribute importantly to success in software testing. But what's novel about that?

Bookstores are stuffed with metric tons of How To computer books. None of them, however, deal with the human side of software testing that way I think it deserves to be dealt with. Let me describe experiences that shaped my views.

My second career is software engineering. My first career was as a researcher, using a method called participant-observation. That means that I participate in the activity that I am studying. This kind of work isn't done in a laboratory or a library. Participant-observation is action-oriented: you learn by both observing and by doing. I've had plenty of opportunity to watch other people working at different tasks, during which time I struggled to do the very same tasks, shoulder to shoulder with the others.

The result has been intensive contact not just with computer professionals, but with commercial fishermen and Native subsistence hunters in Canada and Alaska. I've worked next to them, slept next to them in the same tent and on the same boat, killed animals (by the thousands) with them, and starved when they starved, either economically or literally. One of the things that I've learned is how difficult it is to succeed in all these arenas. When the stakes are high and the consequences of failure can be severe, success factors become more compelling, more urgent. Luck helps but is never enough.

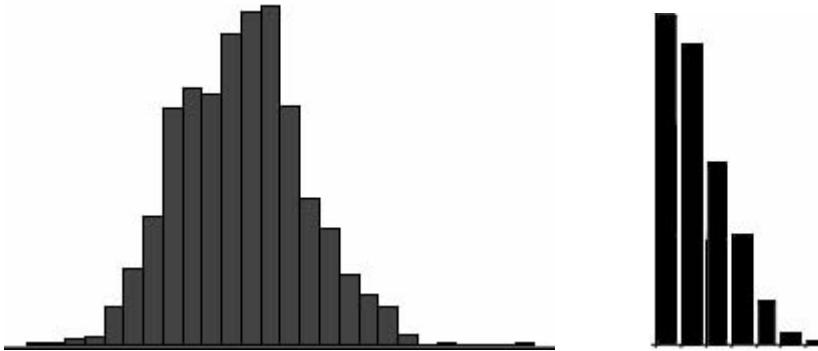
So the first purpose is to identify some of the factors contributing to success – some, but certainly not all. The method is to compare observational data across different high-risk human endeavors, extracting common factors. True, these endeavors differ greatly in outward form. But the thread that connects them is the need to make quick decisions under difficult, at times urgent, conditions. And why shouldn't there be similarities? Dealing with uncertainty and risk has shaped human destiny for millions of years. It's perhaps the essence of who we are as a sentient species.

Characteristics of successful hunters and fishermen may offer insight for software testers. I hope to illustrate how a social science approach can generate insights useful to us in the industry. I also want to offer some forward looking ideas about designing workplace culture.

In this paper, I present ideas based on my own experience rather than upon laboratory research, literature review, or statistical analysis. The goal is not to be definitive but to be provocative. If the observations are interesting, if the ideas are stimulating, further investigation may follow. The first step is to get the ideas on the table.

Far from the Golden Mean

Collections of numbers distribute themselves in interesting ways. Here, for instance, are two common distributions, Gaussian or “normal” (left) and Poisson (right):



There is a widespread and enduring infatuation with the Golden Mean, the center of a normal statistical distribution. Marketing studies and survey research of all kinds target the average this and the average that. The tacit assumption seems to be that anything of importance is distributed normally. Interestingly, the historical roots of normal distribution theory lie with the study of observational error. Error is, in fact, reliably normal. Skewed distributions, such as the Poisson, are far more interesting for our purposes here and, to be truthful, often far more important, both in Nature and in human behavior. Hunters or commercial fishermen provide classic examples. If you plot the distribution of success, the distribution is typically skewed, sometimes sharply so. A minority of participants account for the majority of game taken, fish caught, money made.

The focus of this discussion isn't about statistical averages, but about excellence. The spotlight, therefore, is on the elite players, the kind of players who not just win the game, but change the pace, even the rules, with which it is played.

Why Study the Best?

Among California waterfowl hunters, there is a widespread myth regarding the “Northern Birds”. The idea is that somewhere far to the north, there is a swarm of duck and geese collectively waiting for the “Go!” signal to head to the hunting grounds in the south. When hunting success is low, many hunters attribute their own miserable performance to the missing Northern Birds. A hunter would tell me, in very authoritative terms, that the birds were staged “by the millions” in Klamath or Tule Lakes or Canada or Alaska - somewhere in the distant and invisible North. Yet, in that at the very moment, hundreds of thousands of waterfowl are clearly visible four hundred yards overhead. The astonishing thing is that the hunter doesn't see them. The Northern Bird myth renders a sky full of waterfowl completely invisible. By blaming factors seemingly beyond his control, the hunter side-steps accountability and soothes disappointment. But not all hunters behave like that. Some know how to consistently beat the odds.

One opening day of California waterfowl season dawned hot and clear – supposedly the worst conditions for duck hunting. I had been studying hunters for a couple of years and decided, quite late in the afternoon, to apply what I had learned. I drove to a public hunting area in the Sacramento Valley and registered at the check station only an hour before closing time. The downcast hunters moping around the check station told me that the day's average kill was less a quarter a bird per hunter. One informed me that I was an “idiot” for going out at all. Wearing only a t-shirt and carrying only a shotgun and three duck decoys, I sauntered out to nearby flooded millet field a mere hundred yards from the check station. I returned to the check station exactly forty-five minutes later, carrying a full limit of seven prime drake pintails, one of the most coveted ducks. Nearby hunters were incredulous. One accused me of collecting ducks that were already dead. I had him examine the ducks more closely. Pellet wounds in their chests oozed fresh blood, and their bodies were as warm as the barrel of my shotgun.

I wasn't a lifelong, hardcore hunter. I was a graduate student in psychology. My success arose directly from observation, both of birds and of hunters. I had discovered a cadre of hunters, a quite small percent, that put more faith in their own powers of observation than they did in myths and excuse-making. They never followed the crowd, and their path was often contrarian. When other hunters were running about in a frenzy, trying to get in position for the hunt, I would see these individuals standing aside, waiting, observing quietly, before they made a move. Either that, or I wouldn't see them at all. Like Northern Birds, they attained invisibility by being somewhere unexpected. And they were immensely productive, far beyond the average. I learned how to succeed in hunting by watching and interviewing these individuals. To a person, they excelled in naturalistic observation, and their assessment of causality was spot-on. They had discovered how to systematically navigate the Maze of Uncertainty. So if you want to understand success, study people who succeed.

Success Defined

Success has an endless number of definitions: money, fame, political power, popularity. The definition used here reflects a hunting perspective: hitting the target. A 'target' consists of harvestable prey, both physical and symbolic: fish, geese, caribou, software bugs. Factors like career advancement, salary, kind of car owned, marital bliss, and conventional happiness are deemed irrelevant. Hitting the target may, or may not, increase the likelihood of making other gains in life. But that doesn't matter, because for many serious players in these kinds of competitive games, hitting the target is what counts. Hitting trophy-grade targets is even more important. A trophy bear is worth a thousand rabbits. A fat showstopper bug is worth a thousand cosmetic defects.

Keys to Success

Here are some of the characteristics of successful hunters and commercial fishermen. Let's see how well they apply to software testers.

Persistence

Persistence is an essential characteristic: dogged, relentless, at times almost fanatic. Neither bad weather, bad advice, nor bad luck proves a deterrent. Once plugged into the chase, they don't let go until they either succeed or run out of food, fuel, or ammunition. One time, I was stranded on a tiny island in the middle of James Bay with Cree Indian hunters after all our boat engines failed. A fierce storm was approaching, and the ice was forming contiguous round pans all over the Bay. A few more hours, and we would be stuck there for weeks until the ice was solid enough to walk on. The hunters had taken all three boat engines apart, and were switching parts from two to make one working engine. Suddenly a flock of sage grouse broke from the tree line, circled over us, and flew back into the bush. To a person, the men dropped what they were doing, grabbed guns, and took off on a dead run, leaving me to hold the boat against crashing waves. A few gunshots later, the beaming hunters returned, each carrying one or more dead grouse. They then quickly, yet calmly, went back to engine repair. They succeeded, despite cold-numbed fingers, and we were saved, grouse and all.

Persistence is the life blood of great software testers. Here's proof. Late some evening, stroll past local government offices, insurance companies, banks. Closed. Now wander over to your own computer company, enter the building, walk into the test lab. There are a dozen people in the lab, more maybe, eating pizza, drinking pop, shooting nerf balls at posters of unpopular political figures, banging away on the keyboard, cursing, then suddenly shouting "Yes! Yes!

Killed it!” She’s just found the bug of the century. Now she can’t wait to find another one. Don’t these people ever go home?

Born of a Seal: Systematic Observation and Pattern Recognition

In testing software, persistence alone is not enough to assure success. Systematic observation and pattern recognition are essential. One follows the other: you have to observe in a systematic way in order to detect patterns. Discovering software timing errors provides a good example. I was running automated interoperability tests against a database product. The tests were bedeviled by intermittent data access errors which I couldn’t pin down. It reminded me of record locking issues I had created for myself in writing database code. I tested each of several co-existing applications separately, then again in pairs. Turned out that one particular application was locking the whole table instead of rows like it was supposed to, but only if it was the first one to request the lock. It was first sometimes, other times not, hence intermittent. If I hadn’t made systematic observations, I wouldn’t have seen the pattern.

Pattern recognition clearly plays a role in hunting and fishing as well. Alaskan Native fishermen recognized the effects of global warming well before it was a talk show topic. Yup’ik Eskimo villages along the Yukon River depend on annual salmon runs, both for food and for cash. Traditionally, they laid gillnets right against the river bank, anchoring them against the current. But in the 1980s, they changed. Instead of fishing close to the bank, they drifted their nets from skiffs, toward the middle. The Yukon is a big river, so this is a major shift. Local fishermen know that salmon are very sensitive to water temperature. Big kings tend to run deep, following the thermocline. As it became increasingly difficult to catch kings in the shallows, Native fishermen switched to deeper water, where they again found kings. A Yup’ik elder explained to me that the temperature change along the banks was due to progressive decreases in snow melt and icy run-off in recent years, resulting in much warmer water along the banks. This drove the kings away. “Something is changing in the weather,” he asserted.

The Yup’ik Eskimo village of Emmonak lies near the Bering Sea, at the mouth of the Yukon River. It is a stark, wind-whipped, brutally cold place, where cash is in short supply. Seals and salmon are the centerpiece of local diet and economy. Success in fishing and hunting, especially seal hunting, conveys high social status. One winter day, I had the opportunity to meet the greatest seal hunter of all. Late one afternoon, I heard a knock at the door. An older man, a village elder, introduced himself, and asked if he could visit. “I hear you’re interested in hunting,” he said, then for two hours poured out a stream of meticulously accurate details about biology, botany, meteorology, and animal behavior. At the end of our conversation, he drew me over to the window and pointed to a small barrel a good hundred meters away. “See that? That’s a seal,” he said, “when he comes up to breathe, I shoot him, shoot him right in the eye, every single time. You believe me?” I’ve been in small boats, hunting seals, in nasty chop, with howling winds. For me at least, it would be an impossible shot. Catching the dubious look on my face, he laughed and said “every time”.

That evening, I was eating dinner with a Yup’ik family. I casually mentioned my visitor. Conversation stopped, and sideways glances were exchanged. “What did he say?” I was asked. I shared the stories, including the “every single time” assertion. No one laughed or jeered. Instead, they nodded their heads, affirming it to be factual. Then they explained why he was a gifted hunter. “He is not a man,” I was told, “he only looks like a man.” It was stated, as a fact, that he had been born of a seal, that his mother had been a seal, so that he actually thought like a seal and knew his prey as kindred. True, in the Yup’ik cosmology, man and animals are intimately connected, and souls recycle through physical form in an infinite loop. Yet here, 2,800 miles away from California duck hunters, myth still plays a role.

Risk-taking

Software testing is not a job for the risk-averse. To be successful, a software test engineer has to be willing to take risks – not once, but over and over, day in and day out. These risks are both intellectual and professional. They are intellectual in the sense that the engineer has to be willing to spin new theories and new explanations, none of which are guaranteed to be true. They are professional in the sense that a big mistake or (ironically) even a big discovery can permanently damage a career. On one occasion, I stayed up all night testing a theory that I had regarding database performance. Given what I knew of the database architecture, I believed that a relatively small number of simultaneous service requests would dramatically degrade performance. The experiments worked beautifully: I was right. I couldn't wait to show the printed charts to my manager. Next morning I stood in the doorway to her office, excitedly describing my findings. She cut me off in mid-sentence. "Can't you see that I'm planning the office party?" she said, slamming the door in my face. I stood there, literally with my nose against the door, one hand raised, still holding a chart. She never liked me after that. She had seen me stop shipment before and to her, as a fanatic company loyalist, that was heresy. A few months later, after I left the company, I saw identical charts in a computer magazine, along with a brutal product downgrade based upon poor performance.

The great hunters, the great fishermen, are all risk takers. John, a Yup'ik Eskimo, worked for me in the coastal village of Scammon Bay. He often went seal hunting miles out on the Bering Sea in his homemade plywood boat (with a gigantic engine). One day his boat's hull fasteners suddenly started giving way. While racing full speed back to the village before the boat sunk, John spotted a seal. He hesitated briefly, worried about possibly sinking the boat. But he shot the seal anyway. In order to retrieve the seal, he had to stop the boat, thereby accelerating the flow of water through the holes in the hull. By dint of great skill and steady nerves, John made it back to the village dock just as the boat sank.

This particular incident illustrates a common characteristic among the best hunters. Not only do they put themselves into risky positions, they are skillful at surviving. It's worth noting that these people do not engage in risky behavior just for the thrills, like bungee-jumpers. Instead, this kind of risk-taking is purposeful and part of a larger effort to secure a tangible and valuable end. So the lesson in all this for software testers? Keep taking those risks. You couldn't live without the excitement. But keep your resume polished.

Inventiveness

A good software tester has to be inventive. Each day offers a challenge to inventiveness, and each new product requires new ideas. Here's an example.

It was my first QA job. I had been a registered developer for the company and knew the product internals quite well. In particular, I knew that the architecture connecting the GUI to database functionality was convoluted and highly suspect. I bet myself that "out in left field" events could detonate the system. I built test automation to exercise unlikely combinations of key strokes. I crashed the GUI, froze the computer, and corrupted the entire database with something like Alt+Shift+space+F7. The bug stopped shipment. As an aside, it is this very kind of bug, the often-dismissed "never could happen" bug, that can reveal seismic fractures in an entire system. Atomic fission is an unlikely event, too, but when it occurs, it takes out a whole city.

Although more at an abstract, symbolic level, the inventiveness required by testers is not unlike that required of subsistence hunters. Witness the instance where, despite freezing weather, the Cree hunters disassembled and re-assembled three boat engines. Once, when I had taken my new

bride to live in a tent on the shores of James Bay, we were stranded by weather and ran short of food. We were on a little raised spit of gravel, all alone, surrounded by open water. There were a few Black ducks around, but they proved exceptionally wary, and I had no decoys. I lashed small sticks into tripods, set them in the water, put large mud balls on them, topped by a stick for a neck and toilette paper wrapped around smaller mud balls for a head. My invention worked great.

Curiosity

A person without curiosity makes a terrible tester. Little kids are endlessly curious. They turn over rocks, watch the bugs wiggle out, then gleefully step on them, dismember them with tweezers, or blow them to smithereens with firecrackers. They'll do it all day long or until they run out of firecrackers. By some kind of magic, hardcore testers have remained little kids. That's why many of us make worse managers than the ones we complain about. We don't want to be grownups. We want to keep on hunting bugs.

Like elite testers, elite hunters are driven by insatiable curiosity. In both cases, their curiosity is not confined to an eight hour work day. Looking for the next challenge, navigating previously unknown terrain, learning about new techniques and new equipment is a ceaseless pre-occupation. These individuals never assume that what works today is good enough. They know that it might not work tomorrow. To them, knowledge is golden.

On one occasion, I was stuck on a tiny rock pile island in James Bay with a couple of Cree hunters. We were there for days, with nothing to do but talk, drink tea, and wait for something to fly overhead so that we could shoot it for food. I had two bird books with me, Peterson's and the Golden Guide. The three of us spent hours each day going through the books, page by page, while the two Cree asked endless questions. They wanted to know about the birds they were unfamiliar with – how they behaved, what kind of habitat they lived in, and, above all, whether or not they were good to eat. Their interest never slackened, although, two days and hundreds of questions later, mine definitely did.

Playfulness and Sense of Humor

Despite the best of intentions, the tide often runs against you. Anyone who has been in the computer business for more than a day fully understands. Once again, childlike characteristics pay off – not just wonder and curiosity, but a willingness to laugh in the face of adversity. Once we were crossing James Bay, far from land, in a twenty-four foot open skiff in nasty weather. The boat was stuffed to the gunwales with Cree Indians, dogs, and gear. Water appeared in the bottom of the boat, and it kept steadily rising. Everyone started getting tense and nervous. Gerry, the steersman, said something in Cree, and everyone cracked up laughing. I asked him what he had said. He paused for drama then simply said "Sinking". From then on, everyone joked, laughed and teased, seemingly indifferent to the water sloshing around inside the boat.

Recently, I circulated the trait list amongst my some of testing colleagues. I was hoping to get real-world examples. The response was minuscule (we're dealing with engineers here), but 100% of the respondents selected Playfulness and Sense of Humor. Here are a couple of the responses.

One of the products I've QA'd was starting to become a catchall for disparate common functionality that several other products used. As a result it was acquiring a lot of functional areas that didn't fit together, logically or practically. When coming up with a code name for the next version I suggested "Frankenstein." While not everyone thought this was funny, it got the ball rolling talking about the design problems.

An elite QA engineer takes great joy in being mischievous. Learning new tools, skills and techniques is an effort towards the playful discovery of new ways to cause havoc with an AUT. The elite QA engineer takes pride in doing things to software that no one else thought of and often find themselves in a fight to apply proper weighting to an issue they find. This struggle with dev and project management is part of their fun whether they admit it or not. You'll often hear their mischievous laughter as they recount how they found that one nasty showstopper that halted the release at the last possible moment.

Playfulness and humor reflect the ability to see the other side of things, to build a distance between inner self and events. It's a manifestation of creative thinking in general. If nothing else, humor triggers fresh ideas, lowers fear and anxiety, and puts soothing ointment on the sting of defeat. Whenever I hire a test engineer, the first trait that I look for is a sense of humor. The testing game is rife with frustration, disappointment, and meandering paths leading to nowhere. It is all too easy to get lost in the Maze of Uncertainty – not once or twice, but many times daily. There are moments where sanity itself pivots on having a sense of humor

Fault Tolerance

Taking risks and being inventive can lead to wonderful adventures. At the same time, risk-taking and inventiveness opens the door to failure. Many people are risk averse because they have little stomach for the taste of defeat. Defeat can be embarrassing, humiliating, profoundly unsettling. In other words, failure isn't fun. For some people, however, it's worth taking the chance.

Elite testers and elite hunters alike seem less disposed to blame others for reversal of fortune. Instead, they lean toward rational analysis, while accepting their own responsibility. They thereby place the *locus of control* within themselves rather than in some external agent over which they have no control. High performance players in risky games must necessarily have high fault tolerance for the failures that their own risk-taking causes. Accordingly, they are quite good at quickly moving to alternative strategies. Remember that a leading trait among such individuals is unusual persistence. If the one approach doesn't work, the next one will, or the next or the next. In their view, false starts, dead-ends, and failed experiments simply represent useful learning opportunities.

Cognitive and Social Independence

Elite players can function within a group, yet willing to think and act on their own. When push comes to shove, they trust their own judgment and know that the group can be wrong. In one instance, I was testing an Internet shopping mall that had a flashy UI. Management thought it was perfect for luring customers. They would broach no complaint. I decided to investigate the database architecture. It was a train ready to run off the tracks. I immediately pointed out the dangers and stated that it would take only a couple of weekends to fix the design. A manager took me aside and told me to shut up. Making the release date was all that mattered. We made the date, but soon afterwards, customers were furious. Database misbehavior had become intolerable. Management hired consultants to fix the database, which now was filled with financially valuable information. It took them weeks to make the repairs and cost \$125,000.

What would a commercial fisherman do with a \$125,000? Buy a new boat. On Alaska's Bristol Bay in the 1980's, prices were good and salmon were abundant. Boats became progressively bigger, fancier, and costlier. But not everyone bought a fancy boat. A Sicilian man in his seventies continued to fish in a tiny, inexpensive, old-style wooden cannery boat. Younger fishermen teased him about fishing in "an old piece of junk". I asked him if the teasing bothered him. "To hell with those bastards," he said with a grin. They didn't know his secret: he out-fished almost everybody. The old wooden boat was so light that it could float through shallows

where newer, heavier boats dared not go. The man fished on mud flats as the tide was going out, drifting down tiny channels that were invisible from only a few yards away. He had started fishing the Bay as a teenager, when they still used sailboats to fish. His job had been to pull the boat through the mud as the tide went out. So he knew the mudflats like the back of his hand. But there was another angle. He didn't always play by the rules, anybody's rules. "If the Game cops seize the boat, so what?" he said, "she's an old boat. I go buy another one for nothing."

Tool Savvy

Nature and computers are alike in that they often refuse to cooperate at the worst possible moments. Knowing how to make and use tools is one of the best defenses against cruel fate. In the midst of a commercial fishing season, we suddenly faced a serious mechanical problem, without a tool to fix it. We were stuck on the beach. One of my guys, a high-school graduate but brilliantly inventive, went rummaging through a tumbled down cannery that had been abandoned a century ago. He found a wrench that was thickly encrusted with rust and barnacles. He then crafted a small forge out of an abandoned oil drum, built a roaring fire inside it, then repeatedly heated the wrench red hot, then dipped it first into water then engine oil. Soon it looked almost new. We finished the mechanical repair with a wrench that hadn't been used in a hundred years.

Elite software testers often relish the opportunity to build new tools. I was testing an object-oriented database that had a rich front-end and a very sophisticated logic layer that instantiated objects both in code and in the database. But timing errors were destroying database integrity. It was simply impossible to properly test the system manually. So we built rather elegant test automation that fully understood the data factory rules and could spelunk inside the database surgically. It took several weeks to complete, but was a thing of beauty. And it worked.

Some tools software testers build are merely simple scripts, analogous to crude stone tools used by our most distant ancestors in the early Pleistocene. Other tools that testers build become elegant works of art, reminiscent of the beautiful flaked spear heads crafted by the Cro-Magnon people. And they are not just tools. These are weapons to spear nasty bugs and drag them, screaming and pleading, to their just rewards.

Are Software Testers Hunters in Disguise?

Our earliest human ancestors presumably started as prey species themselves, subsisting by collecting plants, small animals, and birds' eggs. Over time, our role changed from hunted to hunter. The size of the human brain, as well as of the animals being hunted spectacularly increased over the course of millions of years. By the late Pleistocene, we had become supreme Big Game hunters. The transition from hunting societies to agricultural societies happened only quite recently, perhaps ten thousand years ago. That's just a blip in geological time, so brief that our genes are not likely to differ significantly from those of our Big Game hunting ancestors. Hunting for tiny computer bugs may indeed be simulated hunting for wooly mammoths and cave bears. The excitement of the chase remains alive, without the bloodshed. A stop-ship bug isn't just another routine, boring, bureaucratic event. It's an arrow straight into the lion's eye.

Mind in Action: For Managers and Testers Alike

One more gift from our ancestors: we are creatures of culture. That thick, rich layer of neurons called the cerebral cortex stores entire libraries of ideas, names, relationships, tactics, and values. And it is this which accords us the flexibility to invent, imagine, remember, dream, and do. Every human environment – school, military, family, workplace, society at large - creates its own distinctive cultural milieu. That opens the door to this possibility: engineering an effective culture, shaping its values and directives, rewarding the most useful personal traits.

In general, engineers, and engineering managers in particular, have little awareness of, or interest in, consciously designing an effective workplace culture. The tacit assumption is that the standard set of rewards (giving money) and punishment (taking money) is good enough. It may be good enough, but it is as good as it could be? Does it promote the quest for excellence? Ask those same questions of work environments you have experienced. You may conclude that what passes for management in this industry is often at best benign neglect. Take my last employer, for instance. In six years I had twenty six managers, the classic bungee cord exercise.

I recall discussing incentives with an old-school manager. As he stubbed out his cigar in an ashtray shaped like a naked ballerina, he said, with solid conviction, “the military has it right: the best reward is simply the absence of punishment.” Is this the best we can do? Tactics I’ve sometimes seen being used to manage software professionals seem akin to tuning a dual cam Jaguar engine with a sledge hammer. Sledge hammers are designed for wood, steel, and concrete, but not for ideas – and not for people who create ideas. Exploring the depths of Mind may not just be an interesting exercise, but may have a practical pay-off as well.

Judged from a social science perspective, the computer industry is an historical anachronism. It lives in pre-Freudian times. Decades of research in psychology, anthropology, sociology, and psychiatry are simply ignored. We consign an entire body of knowledge into the “soft stuff” bit bucket. Never mind that social science ideas drive contemporary public policy, politics, economics, media, military planning, and that centerpiece of capitalism, marketing. My real purpose here is to encourage my colleagues to consider a topic that, in our rarefied cerebral business, is more taboo than sex or violence – namely, our own inner world.

Let me repeat: the goal of this paper is not to be definitive but to be provocative. My discussion barely scratches the surface. It spotlights only a handful of traits, and it completely bypasses the complex details of cognition and perception. There are doubtless dozens of additional traits, waiting for discovery. There are textbooks filled with psychological constructs, from cognitive dissonance to neural processing, that are the deep stuff of understanding Mind. There are proven techniques for orchestrating a research effort that would parcel out relevant functionality in a statistically reliable way. But without the will to investigate, all that remains essentially invisible, just like a sky full of waterfowl above a hunter waiting for the mythical Northern Birds.

Opening our minds to social science ideas could help lead us toward a true culture of excellence. We could, as a profession, take an important step on the journey towards wisdom, a journey begun by our sapient ancestors a hundred thousand years ago: discovering who we are. I want to encourage us all, managers and engineers alike, to take that step. Let us envision a new and better workplace, a workplace that understands and welcomes the fullness of human character and capability. We should think of those who will follow in our footsteps and remember the proverb: plant a tree today so that your grandchildren can enjoy the shade tomorrow.

Requirements Testing

IBM

Kelly Whitmill

Tel: 303-924-9145

E-mail: whitmill@us.ibm.com

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

Abstract:

As a software industry we supposedly understand the importance of requirements and the financial advantages of removing problems as early in the cycle as possible. However, our practice is often far short of our understanding. To close this gap, start testing requirements rather than just reviewing them. When requirements testing becomes as much a part of the process as code testing, we will see significant improvements. To make that happen, we must understand when and how to test requirements. This paper presents six techniques for testing requirements.

1. 3 Q's
2. Use Cases
3. Check Lists
4. High Level Test Cases
5. Ambiguity Review
6. Visual Requirements

It is important that this testing happen early in the cycle, before coding starts. There are many opportunities to utilize requirements throughout the development process, but the focus of this paper is on testing the requirements before the code exists. It is not imperative to have a formal requirements process to start testing requirements.

Requirements Testing

If the best opportunity to improve software is with requirements then why do so few do anything meaningful about it? Are we intentionally ignoring the obvious at our own peril? Do we just not know how to go about it? This paper is based on the assumption that we would like to improve requirements, but we just have not found a practical way to do it. There are many areas we could focus on to improve requirements including:

- gathering requirements
- writing requirements
- testing requirements

This paper addresses only one of those ways – testing requirements.

The term requirements testing rather than requirements review may only be semantic, but it is a very important semantic. To review generally implies to read, consider, and comment. To test implies to a more rigorous approach. When we start focusing on testing requirements as much as we do our function test and system test we will start seeing significant improvements.

In most cases, it is not reasonable to assume that your organization will adopt a new set of processes to accommodate testing the requirements. My experience has been that test must be the one to allocate time and effort to develop a requirements testing plan and strategy and to perform the test cases.

Each organization has their own methods for creating test plans, strategies and test cases. Organizations differ in format, formality, and process. This paper will not attempt to detail how your organization should accomplish these tasks. Rather than bogging down on those details, the focus of this paper is on presenting testing techniques and practical tips for testing requirements.

Why do requirements testing? Why do it now?

Many factors are driving a need to improve requirements including development and maintenance costs, outsourcing, new methodologies and legal liabilities.

Development costs mandate better requirements

It is commonly understood that the later in the cycle that an error is discovered the more it costs to fix it. The relative cost of fixing a bug can increase by an order of magnitude or more as we progress through each phase of development and testing. If that is true, how can an organization justify investing so much in testing at the end of the cycle and so little in investing early in the cycle? How many organizations and projects have formal function tests and system tests but have no requirements tests? Very few organizations test requirements early despite the data that shows the high cost of finding defects later. Therefore, one can assume that as an industry we do one or more of the following:

- Ignore the obvious to our own detriment
- Don't believe the studies
- Don't know how to test early

Outsourcing mandates better requirements

Outsourcing work can provide a competitive advantage but it also creates communication challenges. Prior to outsourcing, daily office communications and interactions could compensate for requirements deficiencies. With outsourcing, the requirements play a more important role in defining legal obligations. As co-workers get more and more separated by space and time we have fewer daily interactions and become more dependent on clear and accurate communication through requirements. Improving requirements through testing may be one of the great differentiators of success and failure in the current environment.

Competition mandates better requirements

We can get by with less effective practices as long as the competition isn't doing any better. When the competition improves then so must we or we fail to compete. Agile development has introduced the practice of test first design. In many cases, the tests are written before the code is designed and the tests become the requirements. As agile development companies push the testing to earlier less expensive phases, costly investing in late test phases will not be competitive.

Legal liabilities mandate better requirements

The software industry is a prime target for law suits. Software bugs cost business money and businesses want to recover that money. Better requirements testing not only reduces the liability by reducing the number of errors but it also demonstrates to the courts that due diligence has been exercised to prevent harm.

Just because we have succeeded thus far, even without great requirements, is no guarantee that we will be able to continue to do so. We must improve requirements and requirements testing is key to making that happen.

The data to support earlier testing is too overwhelming to reject outright. Business is too cost competitive to ignore the obvious. Therefore, it is essential that we start understanding and implementing a practical process for improving requirements. This paper presents a very practical approach to improving requirements through testing; an approach that can be applied successfully regardless of where you are at right now with requirements. This is an approach to improve your process not replace it.

How do I get started?

Even if your organization has very terse or no requirements at all, you can still test them and add significant value. Whether you are starting from scratch or have a well defined

requirements process, there is a basic set of practical techniques that you can apply that will improve your requirements.

Start with what you have, whether it is a formal written requirement, an idea in someone's head, or anything in between. The following tests/techniques can be adapted to fit your level of requirements formality.

How do I test requirements?

The primary focus of this paper is to present techniques that are practical to apply. It is important not to get too hung up on the definition of a requirement. Consider a requirement to be anything that provides input on what is to be developed. Ideally, there is a requirements document. However, it could take the form of a design document, a specification, e-mail, notes on a napkin, or even a verbal conversation. Consider requirements testing to be the process and techniques you apply to discover the completeness and/or problems and challenges associated with the requirement. The learning curve needed to apply most of these techniques is remarkably simple.

It is not intended that any project will apply all of these techniques. You should be able to find one or more of the following techniques that can be applied effectively and practically to your project.

1. Ask the 3 Q's?

I have often found myself in the predicament where there is too much project left to complete and too little time to complete it. At that point we start doing some serious risk-based searching on what is really needed and who needs it. Eventually, I have come to realize that these questions are better asked at the beginning rather than at the end of a project.

Ask the questions:

- Who wants this?
- Why do they want it?
- How are they going to use it?

These three questions are as powerful as they are simple. Don't allow the simplicity to lure you into ignoring these questions. First, the answers are not always obvious. Second, you will be surprised at how illuminating the answers can be. If the answers are clearly understood then communicating them during the requirements process will promote a more focused, unified, efficient and effective development effort. If the answers are not clearly understood then you have identified an area with high risk of potential problems.

Unclear answers to these questions can lead to the following problems:

- It is difficult to understand whether the business case for the requirement is justified or not.
- It is difficult to develop a solution that will satisfy the customer if we don't know who the customer is or what they want to accomplish.
- It is difficult to test the code for customer suitability.
- It is likely that waiting to discover the answers throughout the process will result in different stakeholders working toward different goals causing significant rework and costly inefficiencies.

It is not unusual to discover that the requirement is based on the needs of some stakeholder other than the customer. This often times has a significant impact on the business case as well as the development and test strategy.

Forcing this thought process early on is essential to the success of the project. Just getting the stakeholders to communicate on these essential items early in the process will have a significant positive impact on quality.

2. Create Use Cases

Use cases can be instrumental in discovering what is missing from the requirements. Prior to implementing use cases my typical experience in requirements reviews and design reviews consisted of a general reading of the document and a general agreement that things looked okay. When I started using use cases, I found that it exposed so many deficiencies that I had to prioritize my discoveries so as not to overwhelm the requirements providers.

Even if you don't have a lot of experience here, with very little guidance you can create use cases. This approach will keep it simple and practical. Creating use cases will accomplish the following:

- Provide context for the requirement. This will answer the 3 Q's mentioned earlier.
- Expose error cases and alternative cases not covered by the requirement. Most requirements reviews are good at reviewing what is there but are poor at detecting what is missing. Use cases are effective at detecting what is missing.

Ideally, you wouldn't have to create use cases because they are provided as part of the requirement. However, in many if not most cases, as the requirements tester you will have to provide the use cases. There are many different versions of use cases. Most of them will probably work. The four simple steps identified here will take you through the process. This is largely based on Alistair Cockburn's *Structuring Use Cases with Goals* at <http://alistair.cockburn.us/crystal/articles/sucwg/structuringucswithgoals.htm>.

Four Steps for creating use cases include:

Goals – identify the goals of the requirement.

Interactions – List 3-9 interactions/steps needed to accomplish the goal

Variations – list the variations for each step.

Exceptions – list the exceptions for each step.

Goals:

Identify all of the goals or tasks that should be accomplished with the requirement. Goals should be stated from the perspective of a user of the system/requirement. Sometimes the user of a system/requirement is another system or application. As you review this list with the stakeholders it will generate a lot of understanding. It will go a long ways towards accurately defining the scope of the requirement. Without this process, I often find myself in the middle of a system test debating what the software should or should not do. System test is much too late for this debate.

It is not critical that the list you generate is completely accurate. Once the list exists, other stakeholders will help refine it and the scope can be better defined earlier in the process. It may be okay to have some tasks or goals where it is not known without further investigation whether they can be included in the scope of the requirement or not. At least by creating the list all stakeholders have visibility to the scope and the open issues related to the scope of the requirement.

Even for complex requirements there should be a pretty limited list of goals. If you start getting a large list it may be worthwhile to review Alistair Cockburn's paper or other reputable references on use cases. To create the list it may be helpful to list all of the users of this requirement and then list what they would expect to be able to accomplish with the code developed as a result of this requirement.

Example Requirement: Provide accounting and restart capabilities for print jobs sent to PCL printers.

Goal: Provide accurate accounting information for billing customers for print jobs

Goal: After a printer intervention, restart the print job from the last printed page

Interactions

List 3-9 interactions or steps needed to accomplish the goal. Specify the simplest successful set of interactions that will accomplish the goal.

Example Steps:

Goal: Provide accurate accounting information for billing customers for print jobs

1. Set up printer definitions to turn on job accounting
2. Submit customer jobs
3. Gather billing data for each customer

Variations

For each interaction, list the possible variations. Now, you will start to discover things that were not considered in the requirement. As we look at the example interactions above we may come up with more questions than answers.

Example Variations:

1. Set up printer definitions to turn on job accounting
 - Are there various ways to turn on accounting? What are the possible interfaces for turning this on?
 - Can I turn it on for all printers at once or do I have to do it for each individual printer?
 - Can I turn it on for some jobs and off for others?
 - Can the printer be printing when I turn it on? What happens to the current job when I turn it on?
2. Submit customer jobs
 - Submit jobs from local system, remote system
 - Submit jobs in the following formats: PCL, Postscript, PDF, AFP, . . .
3. Gather billing data for each customer
 - while jobs are printing, while printers are idle
 - at the end of the hour, day, month

We did not take these variations to completion, but you can see that it starts exposing a lot of questions that developers and testers are going to have to answer to be able to do their jobs. This is a great way to expose any ambiguities before we invest time and money into the development process.

Exceptions

For each interaction list the possible exceptions. In other words, for each interaction ask yourself, “What could go wrong here?”

Example Exceptions:

1. Set up printer definitions to turn on job accounting
 - What if a non-authorized user tries to turn on/off accounting?
 - What happens if all the correct data is not provided?
2. Submit customer jobs
 - What if a job fails in the middle? Does the customer get charged?
 - What happens if we run out of a resource? For example, disk space where accounting information is stored
 - What if the job is sent to a non-PCL printer?
 - What if there are lots of 1 page jobs with header and trailer pages? Does the customer get charged for header and trailer pages?
3. Gather billing data for each customer
 - What happens if the system goes down before data is gathered? Power outage, normal shutdown, etc?
 - What happens if all the correct data (customer name, etc.) is not available?

It is not uncommon that requirements fail to adequately consider error cases. Handling the error cases can sometimes significantly change the scope of the project. Exposing these items before we enter the development process should only increase our ability to more effectively define our scope and plan our development.

Note that no formal UML diagrams were used for these use cases. There is nothing wrong with UML diagrams but you can also accomplish this technique with normal intuitive steps. Keep it short and simple. This process will expose many questions about the requirement.

3. Checklists

You won't realize how enlightening this simple exercise is until you try it. Every time I present this to people the reaction is an unenthusiastic "Yes, that is a nice idea and it makes sense" sort of response. Every time I get someone to actually do it, it is an eye-opening experience that has impact.

It is hard to discern if requirements provide the needed information and it is even harder to discover what is missing. In some cases requirements can be large enough that you suffer from information overload. Almost always, it is really difficult to know what should have been included but isn't. One way to efficiently discover essential information and problem areas in requirements is to specify categories of information then read the requirement and fill in the information. This process includes.

1. Make a list of items that you need to learn about the requirement (See sample lists below.)
2. Keep a separate piece of paper, or note card, etc. for each category on your list
3. Fill in the category with brief notes gathered from the requirement. If the requirement is not in written form then extract the information from whatever knowledge is available about the requirement.
4. Review the category.
 - a. If there is little or no information this raises a red flag that the requirements is missing information you need.
 - b. Where information is provided, you will get a different view as you have all the related information from the requirement isolated into one place. Your ability to discern the adequacy of information for this category will greatly increase by this exercise.

The contents of your list will be dependent on your context and your goals. Here are a couple of lists that have shown to be useful.

C	Capability	What does this do? What functions are provided?
U	Usability	How easy is this to use?
P	Performance	What are the performance requirements?
R	Reliability	What are the reliability requirements? What are the requirements for running without crashes, deadlocks, hangs, corruptions, etc?

I	Installability	What are the requirements for installing?
M	Maintainability	What are the requirements for being able to apply fixes?
D	Documentation	What are the requirements for documentation? Online Doc?
S	Serviceability	What are the requirements for being able to discover, understand, and diagnose problems?
S	Security	What are the security requirements?
T	Testability	What are the testability requirements?

The most obvious problems are categories that end up with no information. For example, if the requirements said nothing about performance or serviceability you would need to go back and ask for additional information. This simple yet organized manner of reading the requirement is very effective at discovering missing material. If the requirement is not written down, you can do this same exercise to gather information from a discussion or whatever form the requirement takes. This checklist tends to be useful to look at the requirement from a system point of view.

Another useful checklist is the following:

I	Inputs	What are all the inputs?
O	Outputs	What are all the outputs?
P	Processing	What does it do?
I	Interfaces	What are all the interfaces?

This checklist tends to be useful to understand the requirement from a functional point of view.

You can construct a checklist that fits the needs for your situation.

4. High level test cases

Create high level test cases to test the requirement. The purpose of the test cases is to challenge the adequacy of the requirements. This can be an effective means of exposing omissions, errors, and ambiguities in requirements.

For example if the requirement is to allow the selection of messages in a log by hours not just days then some possible test cases might be:

- TC 1: Select log messages for the last 3 hours
- TC 2: Select log messages for the last 1 day 7 hours.
- TC 3: Select log messages for 3 days and 27 hours.
- TC 4: Select log messages for the most recent 2 days.
- TC 5: Select log messages for 6 am to 8 am

If you cannot determine the expected results from the requirement then more information needs to be provided with the requirement. In the above case, it is not clear what input is acceptable. Can the user specify to get messages for a specified range of time? Can it only be the most recent days + hours? Does the input have to be in days or hours but not

a combination? A few test cases can often shed a lot of light on what is ambiguous in the requirement.

Normally, high level test cases would be redundant if you have developed use cases. But, in those instances where you have not developed use cases creating some high level test cases can be an effective means of discovering problems in the requirements.

These test cases are only high level descriptions of test cases. There are not any expected results or information on how to run the test. In most cases there is no intention of running the test cases. The primary goal is to discover ambiguities and flaws in the requirement. To be effective, you need to have a reasonable idea of what the customer wants or needs then try to think of test cases that the user might exercise to accomplish their tasks.

Generating these high level test cases should go a long ways towards ensuring the product meets the needs of the customer rather than just meeting the requirements specification.

Like the preceding techniques, this too can be applied to requirements in whatever form they exist.

5. Ambiguity Review

Do an ambiguity review. Clearing up ambiguities in the requirements, regardless of formality level, can avoid a lot of future problems. When two people can carefully read a requirement and come up with different interpretations, there is a high likelihood of introducing errors into the product.

For example, consider the following requirement:

Software must prevent inadvertent delivery of a secure print job to a printer that is not secure.

Possible Ambiguities

- What software is responsible for this prevention?
- What is meant by inadvertent delivery? Could it mean accidentally sending it to the wrong printer? Is it okay to intentionally send it to the wrong printer? Does it have to prevent intercepting a transmission to a printer?
- What is a secure job? A job that is encrypted? A job classified as secure? A job submitted to a secure printer?
- What is a secure printer? A printer behind a badge lock door? A printer designated as secure? A printer that decrypts jobs? A printer that requires a password to print a job?

Natural language is very ambiguous. If we don't put some effort into clearing up ambiguities in requirements then we should expect to deal with the costs, inefficiencies, conflicts and errors resulting from differing interpretations.

Richard Bender teaches a course on ambiguity reviews which teaches how to identify ambiguities in a requirement. It teaches to identify dangling else statements such as “The number must be an integer.” The question is “Else what?” What if it isn’t? Is it an error condition? It also covers ambiguous statements, ambiguities of reference, ambiguities of scope, ambiguous operators and more. It is beyond the scope of this paper to teach ambiguity reviews but see <http://bdn.borland.com/article/borcon/0,1919,31588,00.html> for more detailed information on doing an ambiguity review.

6. Visual Requirements

Sometimes pictures illuminate things that are obscured by words. There are several types of diagrams that you can draw to help you understand the requirement. This technique requires a little more time and practice, but in some instances it may be what you need. Even with no tools and very little experience, the process of trying to sketch out one or more of these diagrams can expose areas where more information is needed.

When drawing these diagrams you are not aiming for perfection. The goal is usefulness. If you know the formal UML notation, then use it. If you don’t know UML then just start sketching a diagram. See *STQE May/June 2003, Visual Requirements* for more details but just to give a flavor of what this entails here is a brief description of the process.

Data Flow Diagram

1. Identify the functions or processes in the requirement and represent them in circles on your diagram.
2. Identify data/information stored or maintained – you can show this in boxes
3. Identify the information flow between functions/processes and between the functions/processes and data stores. Show the information flow with directional arcs/arrows.

The data flow diagram will expose functional dependencies as well as help you to understand the interfaces and system interactions. The data flow diagram helps you to understand the function of the requirement.

Class Diagrams

1. Identify the objects in your system/requirement
2. Identify the basic data associated with each object
3. Identify the behaviors/functions associated with each object
4. Identify the relationships or associations between objects.
Note: put each object in a box with its data and behaviors. This becomes a class.
5. Identify the relationship rules that govern how the classes interact.

For example, an airline reservation system might have the following objects/classes: Flight, airplane, passenger, reservation, and airport.

In steps 2 and 3 you would identify the data and behaviors/functions for each class/object. In step 4 you might draw a line between flight and passenger and note that a flight has space for passengers. You might draw a line between passenger and reservation and note that a passenger can have 0 or more reservations on a flight. Draw a line between a flight and an airplane and note that a flight is assigned an airplane and so on.

The class diagram helps you to understand all the components of the requirement and their relationship with each other.

State Transition Diagram

1. Represent states with boxes or circles
2. Use directional arrows to show how to get from one state to another.
Label the arrow with the event or condition that triggers the path to be taken.

You may also consider any other diagram that is useful to you, such as context diagrams, interaction diagrams, etc.

If you don't have a background with UML diagrams it is not likely that you will try to draw diagrams to understand the requirement. If you do have some UML in your background you may find these visual representations of the requirement to be very helpful. As you start to draw a diagram you will realize that you don't know how the pieces fit together. You may not even be able to identify the pieces. You will have to ask questions to understand how to build the diagram. The rest of the team will learn with you as you discover the requirements didn't provide enough information to draw a picture. As you show and discuss your diagram with others it will become a tool to build a common understanding amongst all stakeholders.

Non-technical issues with requirements

Requirements are not just a technical issue. The people issues involved with requirements are just as important as the technical issues.

You cannot specify what you do not know.

Requirements testing will expose many weaknesses in a requirement. Sometimes merely exposing them allows us to fix them. Sometimes we are aware of the problem but still don't know the answer. It is sometimes necessary to learn the answer over time. As testers, we must learn to discern when it is appropriate to wait for an answer until later in the development cycle. Process has the ability to greatly enhance the productivity and efficiency of an organization. Process also has the ability to suck the life out of an organization. Don't be a slave to the process. Use requirements testing as a tool to help you in your work not as a

master to dictate your actions. When it is necessary to learn as you go then be willing to do that.

Where do we draw the line between requirements and design?

Inevitably, the question will arise as to whether the information you are seeking really belongs in a requirement or should it be in the design. Remember, a primary motivation for doing this testing is to detect and fix problems earlier. Rather than getting hung up on semantics it may be useful to focus on “What is the earliest reasonable point where this information can be provided?” However, specifying requirements is hard and we always want to think it is someone else’s job. The following question is sometimes useful in understanding if the issue is a requirements issue or a design issue. “Does it have to work a certain way for the customer or is it developer’s choice how it works?” If it can be left to the developer’s choice then it is a design issue. If not, it is a requirements issue. In all cases, you are going to have to make decisions based on your context. Be firm, but be reasonable.

In many cases, it is the developer that fully develops the requirement in the process of understanding what needs to be developed. It doesn’t really matter who did the work. It is important that you test the requirement before too much time and effort is invested in design and coding. It is more important to focus on getting access to the information prior to coding and design than to worry about which document it ends up in.

How do we push for requirements improvements without alienating the requirements providers?

It is certainly possible that the requirements testing will expose an overwhelming number of problems. You don’t always have to tell everything you know. Rather than overwhelming the providers with issues and problems, select the most important issues and work on getting those resolved. Or, decide with the providers which issues should be focused on. Don’t expect to go from zero to perfection in a single release. You may have to be satisfied with incremental improvement from release to release.

Approach requirements testing with a helpful attitude rather than with a requirements police attitude. Developing requirements is a team responsibility. The person or group who provides the requirement is only part of that team. Each stakeholder must contribute to the quality of the requirement. View defect discovery via requirements testing as an expected contribution rather than a measure of the value of upstream work. Testing is just as much a part of building in quality as is writing the requirement.

It is always easier to work with someone than against them. Be sure to show appreciation for any progress that is made. We are all the same. We are willing to do difficult tasks if we are convinced that it is worthwhile and it is appreciated.

Time spent planning on getting acceptance from the requirements providers and addressing the people issues may be even more important to your success than the test techniques themselves. Be sure to:

- Genuinely listen to and understand any objections or issues
- Treat objections and issues as valid concerns that need valid answers.

In some cases, you may have to begin by implementing some of these techniques on your own. You don't have to have a formal approved process to apply these techniques. You don't have to have permission to ask questions and raise issues. You can begin by using these techniques on your current process to discover problems and expose the problems through your current process. As others begin to recognize your contributions you can begin to introduce the requirements testing techniques.

How can this apply to my work if I don't use the waterfall model of development?

These techniques are not tied to a development model. They can be adapted to your development model whether that model be waterfall, agile, or something else. If you are using an agile model then just test those requirements that are part of the current iteration.

What if key players think this whole notion is of little or no value?

Some key players may be convinced that traditional reviews are sufficient. Some may think that this focus on requirements in general is not needed. Some may be convinced that whatever value it has, it is not worth making the necessary changes. In most of these cases, it is not likely that you will make much headway by trying to convince them of the goodness of requirements testing. Amongst the alternatives available the course I would recommend is:

- Begin to implement the techniques on your own. If the techniques prove useful and you are able to identify important problems earlier you will begin to develop listening ears among the stakeholders.
- Find new avenues to present the information. Most people are going to have to be taught a new concept at least 6 or 7 times before it starts to sink in.
- Focus on finding an advocate amongst key stakeholders that can influence other stakeholders.

Conclusion

Requirements-testing is not an all-or-nothing proposition. Applying any of these techniques, will likely result in your team having a better understanding of the real

requirements. Six techniques have been presented that can be applied in a very practical way without imposing burdensome process on your organization.

1. Ask the 3 Q's
2. Create use cases
3. Use checklists
4. Create high level test cases
5. Do an ambiguity review
6. Draw diagrams

You already have most of the skills you need to apply these techniques. The learning curve is small. You don't have to implement formal process changes to begin using these techniques. Think of it as techniques to teach you how to discover the important questions that need to be asked. However, to reap the benefits to a much fuller extent you probably will need to start having a test that focuses on requirements just as you have function tests and system tests. This sort of testing should provide you with the following benefits.

- Reduce the cost of improving quality
- Prevent many development and test issues
- Get test involved early. This is one place where test can actually test quality in to the product in a meaningful way before the code is implemented.
- Get the stakeholders effectively communicating early in the process.
- Make test more efficient by doing testing that not only detects errors but also prevents errors.
- Provide test with much of the essential information needed for later testing.

BIBLIOGRAPHY

Cockburn, Alistair, "Structuring Use Cases with Goals", *Journal of Object-Oriented Programming*, Sep-Oct, 1997 and Nov-Dec, 1997. Also available on
<http://alistair.cockburn.us/crystal/articles/sucwg/structuringucswithgoals.htm>

Becky Winant, "Visual Requirements", STQE , May/June 2003

Bender, Richard, RBT Incorporated, "The Ambiguity Review Process ",
<http://www.benderrbt.com/postionpap.htm>

Rick Craig, "The Value of Requirements Based Testing",
<http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=ART&ObjectID=6455>

Pairwise Testing in Real World

Practical Extensions to Test Case Generators

Jacek Czerwonka

Microsoft Corp.

One Microsoft Way

Redmond, WA 98052

jacekcz@microsoft.com

ABSTRACT

Pairwise testing has become an indispensable tool in a software tester's toolbox. The technique has been known for almost twenty years [22] but it is the last five years that we have seen a tremendous increase in its popularity.

Information on at least 20 tools that can generate pairwise test cases, have so far been published [1]. Most tools, however, lack practical features necessary for them to be used in industry.

This paper pays special attention to usability of the pairwise testing technique. In particular, it does not describe any radically new method of efficient generation of pairwise test suites, a topic that has already been researched extensively, neither does it refer to any specific case studies or results obtained through this method of test case generation. It does focus on ways in which pure pairwise testing approach needs to be modified to become practically applicable and on features tools need to offer to support a tester trying to use pairwise in practice.

The paper makes frequent references to PICT, an existing and publicly available tool built on top of a flexible combinatorial test case generation engine, which implements several of the concepts described herein.

Categories and Subject Descriptors

D.2 [Software]: Software Engineering; D.2.5 [Software Engineering]: Testing and Debugging—*testing tools*

General Terms

Verification, Design

Keywords

Pairwise testing, combinatorial testing, test case generation, test case design

1. BACKGROUND

A set of possible inputs for any nontrivial piece of software is too large to be tested exhaustively. Techniques like *equivalence partitioning* and *boundary-value analysis* [17] help convert even a large number of test levels into a much smaller set with comparable defect-detection power. Still, if software under test (SUT) can be influenced by a number of such factors, exhaustive testing again becomes impractical.

Over the years, a number of combinatorial strategies have been devised to help testers choose subsets of input combinations that would maximize the probability of detecting defects: *random testing* [16], *each-choice* and *base-choice* [2], *anti-random* [15] and finally *t*-wise testing strategies with pairwise testing being the most prominent among these (see Figure 1).

Pairwise testing strategy is defined as follows:

Given a set of N independent test factors: f_1, f_2, \dots, f_N , with each factor f_i having L_i possible levels: $f_i = \{l_{i,1}, \dots, l_{i,L_i}\}$, a set of tests R is produced. Each test in R contains N test levels, one for each test factor f_i , and collectively all tests in R cover all possible pairs of test factor levels (belonging to different parameters) i.e. for each pair of factor levels $l_{i,p}$ and $l_{j,q}$, where $1 \leq p \leq L_i$, $1 \leq q \leq L_j$, and $i \neq j$ there exists at least one test in R that contains both $l_{i,p}$ and $l_{j,q}$.

This concept can easily be extended from covering all pairs to covering any *t*-wise combinations where $1 \leq t \leq N$. When $t = 1$, the strategy is equivalent to *each-choice*; if $t = N$, the resulting test suite is exhaustive.

Covering all pairs of tested factor levels has been extensively studied. Mandl described using orthogonal arrays in testing of a compiler [16]. Tatsumi, in his paper on Test Case Design Support System used in Fujitsu Ltd [22], talks about two standards for creating test arrays: (1) with all combinations covered exactly the same number of times (orthogonal arrays) and (2) with all combinations covered at least once. When making that crucial distinction, he references an earlier paper by Shimokawa and Satoh [19].

Over the years, pairwise testing was shown to be an efficient and effective strategy of choosing tests [4, 5, 6, 10, 13, 23]. However, as shown by Smith et al. [20] and later by Bach and Shroeder [3] pairwise, like any technique, needs to be used appropriately and with caution.

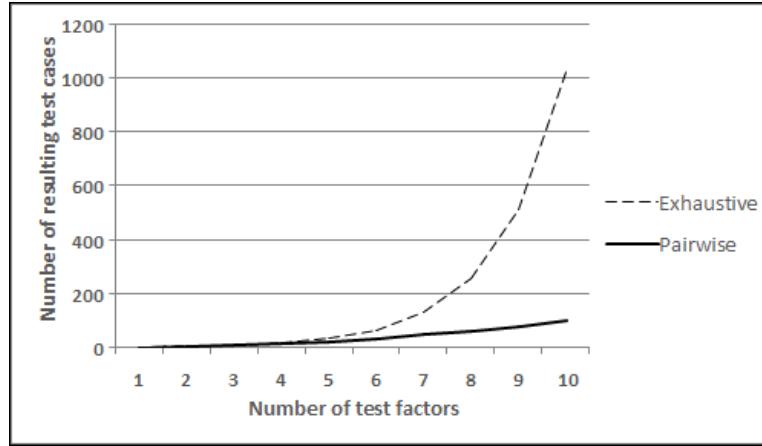


Figure 1: Increase in number of exhaustive and pairwise tests with number of test levels

Since the problem of finding a minimal array covering all pairwise combinations of a given set of test factors is NP-complete [14], understandably a considerable amount of research has gone into efficient creation of such arrays. Several strategies were proposed in an attempt to minimize number of tests produced [11].

Authors of these combinatorial test case generation strategies often describe additional considerations that must be taken into account before their solutions become practical. In many cases, they propose methods of handling these in context of their generation strategies. Tatsumi [22] mentions *constraints* as a way of specifying unwanted combinations (or more generally, dependencies among test factors). Sherwood [18] explores adapting conventional t -wise strategy to *invalid testing* and the problem of preventing input masking. Cohen et al. [6] describe *seeds* which allow specifying combinations that need to appear in the output and covering combinations with *mixed-strength arrays* as a way of putting more emphasis on interactions of certain test factors.

This paper describes PICT, a test case generation tool that has been in use at Microsoft since 2000, which implements the t -wise testing strategy and features making the strategy feasible in practice of software testing.

2. INTRODUCTION

2.1 Models

PICT was designed with three principles in mind: (1) speed of test generation, (2) ease of use, and (3) extensibility of the core engine. Even though the ability to create the smallest possible covering array was given less emphasis, efficiency of PICT's core algorithm is comparable with other known test generation strategies (see Figure 2).

Input to PICT is a plain-text file (model) in which a tester specifies test factors (referred to as test *parameters* later in this paper) and test factor levels (referred to as *values* of a parameter). Figure 3 shows an example of a simple model used to produce test cases for volume creation and formatting.

By default, the tool produces a pairwise test array ($t = 2$).

It is possible, however, to specify a different order of combinations. In fact, any t is allowed if only $1 \leq t \leq N$.

2.2 Test Case Generation Engine

The test generation process in PICT is comprised of two main phases: (1) preparation and (2) generation.

In the preparation phase, PICT computes all information necessary for the generation phase. This includes the set P of all parameter interactions to be covered. Each combination of values to be covered is reflected in a parameter interaction structure.

For example, given three parameters A , B (two values each), and C (three values) and pair-wise generation, three parameter interaction structures are set up: AB , AC , and BC . Each of these has a number of slots corresponding to possible value combinations participating in a particular parameter interaction (4 slots for AB , 6 slots for AC and BC). See Figure 4.

Each slot can be marked *uncovered*, *covered*, or *excluded*. All the *uncovered* slots in all parameter interactions constitute the set of combinations to be covered. If any constraints were defined in a model, they are converted into a set of *exclusions*—value combinations that must not appear in the final output. Corresponding slots are then marked *excluded* in parameter interaction structures and therefore removed from combinations to be covered. The slot becomes *covered* when the algorithm produces a test case satisfying that particular combination. The algorithm terminates when there are no *uncovered* slots.

The core generation algorithm is a greedy heuristic (see Figure 5). It builds one test case at a time, locally optimizing the solution. It is similar to the algorithm used in AETG¹ [6] with key differences being that PICT algorithm is deterministic² and it does not produce candidate tests.

¹AETG is a trademark of Telecordia Technologies.

²PICT does make pseudo-random choices but unless a user specifies otherwise, the pseudo-random generator is always initialized with the same seed value. Therefore two executions on the same input produce the same output.

Task	AETG [14]	PairTest [21]	TConfig [24]	CTS [12]	Jenny [1]	DDA [8]	AllPairs [1]	PICT
3^4	9	9	9	9	11	?	9	9
3^{13}	15	17	15	15	18	18	17	18
$4^{15}3^{17}2^{29}$	41	34	40	39	38	35	34	37
$4^13^{39}2^{35}$	28	26	30	29	28	27	26	27
2^{100}	10	15	14	10	16	15	14	15
10^{20}	180	212	231	210	193	201	197	210

Figure 2: Comparison of PICT’s generation efficiency with other known tools

Type:	Single, Spanned, Striped, Mirror, RAID-5
Size:	10, 100, 1000, 10000, 40000
Format method:	Quick, Slow
File system:	FAT, FAT32, NTFS
Cluster size:	512, 1024, 2048, 4096, 8192, 16384
Compression:	On, Off

Figure 3: Parameters for volume creation and formatting

	AB	AC	BC
	00	00	00
A: 0, 1	01	01	01
B: 0, 1	10	02	02
C: 0, 1, 2	11	10	10
		11	11
		12	12

Figure 4: Parameter interaction structures

```

# Assume test cases  $r_1, \dots, r_{i-1}$  are already produced
# Slots in  $P$  reflecting combinations selected by  $r_1, \dots, r_{i-1}$  are set to covered

If there are any unused seed combinations not violating any exclusions
    Add a seed combination to  $r_i$ 
    Mark all slots in  $P$  covered by the seed combination as covered

While there are parameters with no values in  $r_i$ 

    If  $r_i$  is empty
        Choose a parameter interaction  $p$  from  $P$  with most uncovered slots
        Pick the first uncovered combination from  $p$ 
    Else
        # Assume values  $l_1, \dots, l_{k-1}$  have already been chosen and added to  $r_i$ 

        Look at subset  $Q$  of  $P$  that covers at least one parameter with no
        representation in  $l_1, \dots, l_{k-1}$ 

        Look at slots in  $Q$  which values are consistent with already chosen values in  $l_1, \dots, l_{k-1}$ 

        If there exist uncovered combinations
            Pick a slot with values which when added to  $r_i$  would cover the most uncovered
            combinations with  $l_1, \dots, l_{k-1}$  and the resulting partial test case  $r_i$  would not
            contain an excluded combination
        Else
            Pick randomly a covered combination which when added to  $l_1, \dots, l_{k-1}$  would not
            contain an excluded combination

    Add values of this combination to  $r_i$ 
    Mark the chosen combination in  $P$  as covered

```

Figure 5: PICT heuristic algorithm

The generation algorithm does not assume anything about the combinations to be covered. It operates on a list of combinations that is produced in the preparation phase. This flexibility of the generation algorithm allows for adding interesting new features easily. The algorithm is also quite effective. It is able to compute test suites comparable in size to other tools existing in the field and it is fast enough for all practical purposes.³

3. ADVANCED FEATURES

3.1 Mixed-strength Generation

Most commonly, when t -wise testing is discussed it is assumed that all parameter interactions have a fixed order t . In other words, if $t = 3$ is requested, all triplets of parameter values will be covered. It is sometimes useful, however, to be able to define different orders of combinations for different subsets of parameters. For example (see Figure 6), interactions of parameters B , C , and D might require better coverage than interactions of A or E . We should be able to generate all possible triplets of B , C , and D and cover all pairs of all other parameter interactions. Importance of this feature stems from the fact that often certain parameter interactions seem to be more ‘sensitive’ than others. Possibly, experience had shown that interactions of these parameters are at the root of proportionally more defects than other interactions. Therefore they should be tested more thoroughly. On the other hand, setting a higher t on the entire set of test parameters could produce too many test cases. Using mixed-strength generation might be a way to achieve higher coverage where necessary without incurring the penalty of having too many test cases.

Cohen et al. describe the concept of *subrelations* as a way of getting an output with varying levels of interactions between parameters [6]. AETG actually uses seeding to achieve this. In PICT, since the generation phase operates solely on parameter interaction structures, they can be manipulated to reflect the need for higher-order interactions of certain parameters.

3.2 Creating a Parameter Hierarchy

To complement the mixed-strength generation, PICT allows a user to create a hierarchy of test parameters. This scheme allows for certain parameters to be t -wise combined first and that product is then used for creating combinations with parameters on upper levels of the hierarchy. This is a useful technique which can be used to (1) model test domains with a clear hierarchy of test parameters i.e. API functions taking structures as arguments and UI windows with additional dialogs or (2) to limit the combinatorial explosion of certain parameter interactions. (1) is intuitive, (2) requires explanation.

Tatsumi, when describing the process of analyzing test parameters [22], distinguishes between ‘input’ parameters which are direct inputs to the SUT and ‘environmental’ parameters which constitute the environment the SUT operates in. Typically, input parameters can be controlled and set much

³For instance, for 50 parameters with 20 values each (20^{50}), PICT generates a pairwise test suite in under 20 seconds on a Intel Pentium M 1.8GHz machine running Windows XP SP2.

easier than environmental ones (compare supplying an API function with different values for its arguments to calling the same function on different operating systems). Because of that, it is sometimes desirable to constrain the number of environments to the absolute minimum.

Consider the example shown in Figure 7 which contains the same test parameters as Figure 3 but with hardware specification added. To cover all pairwise combinations of all 9 parameters, PICT generated 31 test cases and they included 17 different combinations of the hardware-related parameters: *Platform*, *CPUs*, and *RAM*.

Instead, hardware parameters can be designated as a ‘sub-model’ and pairwise-combined first. The result of this generation is then used to create the final output in which 6 individual input parameters and 1 compound environment parameter take part. The result is a larger test suite (54 tests) but it contains only 9 unique combinations of the hardware parameters. Users of this feature have to be cautious, however, and understand that in this scheme not all t -wise combinations of all 9 parameters will be covered.

If the goal is to achieve low volatility of a certain subset of parameters, an even better solution can be implemented. Namely, generate all required t -wise combinations at the lower level (*Platform*, *CPUs*, and *RAM*) and use them for the higher-level combinations (all 9 parameters) with the requirement that in any test case a combination of *Platform*, *CPUs*, and *RAM* must come from the result of the lower-level generation. In this case, we would achieve low volatility and would not lose t -wise coverage. This feature has not been implemented in PICT yet.

3.3 Excluding Unwanted Combinations

The definition of pairwise testing given in section 1 talks about test factors (parameters) being independent. This is, however, rarely the case in practice. That is why constraints are an indispensable feature of a test case generator. They describe ‘limitations’ of the test domain i.e. combinations that are impossible to be successfully executed in the context of given SUT. Going back to the example in Figure 3, FAT file system cannot actually be applied onto volumes larger than 4 Gigabytes. Any test case that asks for *FAT* and *volume larger than 4 GB* will fail to execute properly. One might think that removing such test cases from the resulting test suite would solve the problem, however, such a test case might cover other, valid combinations e.g. [*FAT*, *RAID5*], not covered elsewhere in the test suite.

Researchers recognized this problem very early. Tatsumi describes the concept of constraints and proposes special handling of those by marking test cases with excluded combinations as ‘errors’ [22]. Later, several different ways in which constraints can be handled were proposed. The simplest methods involve asking a user to manipulate the definition of test parameters in such a way that unwanted combinations cannot possibly be chosen; either by splitting parameter definitions onto disjoint subsets [18] or by creating hybrid parameters [25]. Other methods could involve post-processing of resulting test suites and modifying test cases that violate one or more constraints in a way that the violation is avoided.

A: 0, 1	AB	AC	AD	AE	BC	BD	BE	CD	CE	DE
B: 0, 1	00	00	00	00	00	00	00	00	00	00
C: 0, 1	01	01	01	01	01	01	01	01	01	01
D: 0, 1	10	10	10	10	10	10	10	10	10	10
E: 0, 1	11	11	11	11	11	11	11	11	11	11
<hr/>										
	AB	AC	AD	AE	BCD	BE	CE	DE		
	00	00	00	00	000	00	00	00		
A:	0, 1	01	01	01	001	01	01	01		
B @ 3:	0, 1	10	10	10	010	10	10	10		
C @ 3:	0, 1	11	11	11	011	11	11	11		
D @ 3:	0, 1				100					
E:	0, 1				101					
					110					
					111					

Figure 6: Fixed- and mixed-strength generation

Test domain consisting of ‘input’ and ‘environment’ parameters:	
# Input parameters	
Type:	Single, Spanned, Striped, Mirror, RAID-5
Size:	10, 100, 1000, 10000, 40000
Format method:	Quick, Slow
File system:	FAT, FAT32, NTFS
Cluster size:	512, 1024, 2048, 4096, 8192, 16384
Compression:	On, Off
# Environment parameters	
Platform:	x86, x64, ia64
CPUs:	1, 2
RAM:	1GB, 4GB, 64GB
# Environment parameters will form a sub-model	
{ PLATFORM, CPUS, RAM } @ 2	
<hr/>	
Hierarchy of test parameters:	
- Type	
- Size	
- Format method	
- File system	
- Cluster size	
- Compression	
- <CompoundParameter> (t=2)	
- Platform	
- CPUs	
- RAM	

Figure 7: Two-level hierarchy of test parameters

Dalal et al. describe *AETGSpec*, a test domain modeling language which includes specifying constraints in the form of propositional formulas [9]. PICT uses a similar language of constraint rules. In Figure 8, three IF-THEN statements describe ‘limitations’ of a particular test domain.

PICT internally translates constraints into a set of combinations called *exclusions* and uses those to mark appropriate slots as *excluded* in parameter interaction structures. This method poses two practical problems:

1. How to ensure that *all* combinations that need to be excluded are in fact, marked *excluded*.
2. How to handle exclusions that are more granular than the corresponding parameter interaction structure; i.e. they refer to a larger number of parameters than there are in the parameter interaction structure.

The first problem can be resolved by calculating dependent exclusions. Consider the example depicted in Figure 9. Constraints on that model create a circular dependency loop between values $A:0 \Rightarrow B:0 \Rightarrow C:0 \Rightarrow A:1$ which results in a contradiction: if $A:0$ is chosen only $A:1$ can be chosen. In the end, if the generation is to proceed we have to ensure that we do not pick $A:0$ at all. Instead of the initial three, five combinations need to be excluded, among them all combinations of $A:0$.

The second, is a case where directly marking combinations as *excluded* in parameter interaction structures is impossible. Consider the example shown in Figure 10 in which 3-element exclusions are created but parameter interaction structures only refer to two parameters at a time. In other words, there is not a parameter interaction structure ABC which can be used to mark the excluded combination; AB , AC or BC are not granular enough. In such a case, one more parameter interaction structure ABC is set up. Appropriate combinations are marked *excluded* and the rest of them are marked *covered*. The generation algorithm will ensure that all possible combinations of AB , AC , and BC will be covered without actually picking $A:0$, $B:0$, $C:1$ combination.

Both steps, expanding the set of exclusions to cover all dependent exclusions and adding auxiliary parameter interaction structures, happen in the preparation phase. After that, produced test cases are guaranteed not to violate any exclusions. Since there is no need for any validity verification or post-processing of produced results, the generation phase can be very fast.

3.4 Seeding

Cohen et al. use the term ‘seeds’ to describe test cases that must appear in the generated test suite [6]. Seeding has two practical applications:

1. It allows explicit specification of ‘important’ combinations.
2. It can be used to minimize change in the output when the test domain description is modified and resulting test suite regenerated.

The first application is intuitive. If a tester is aware of combinations that are likely to be used in the field, the resulting test suite should contain them. All t -wise combinations covered by these seeds will be considered *covered* and only incremental test cases will be generated and added.

The need for the second application stems from the fact that even small modification of the test domain description, like adding parameters or parameter values, might cause big changes in the resulting test suite. Containing those changes can be an important cost-saving option especially when hardware parameters are part the test domain.

It happens often that the initial test domain specification is incomplete, however, the test suite it produces is a basis for the first set of configurations to run tests on. For example, when a test case specifies a machine with two AMD64 CPUs, a SCSI hard drive, exactly 1 GB of RAM, Windows XP with Service Pack 2, and a certain version of Internet Explorer, such a machine must be assembled and all necessary software installed. Later, when a modification to the model of SUT is required, it might perturb the resulting test cases enough to invalidate some if not all already prepared configurations. Seeding allows for re-use of old test cases in newly generated test suites.⁴

In PICT, these seeding combinations can be full combinations (with values for all test parameters specified) or partial combinations. Figure 11 shows two seeding combinations, one full and one partial. The former will become the first test case in the resulting suite. The latter will initialize the the second test case with values for *Type*, *File system*, and *Format type*. The actual values of *Size*, *Cluster size*, and *Compression* will be left for the tool to determine.

3.5 Testing with Negative Values

In addition to testing all valid combinations, it is often desirable to test using values outside the allowable range to make sure the SUT handles error conditions properly. This ‘negative testing’ should be conducted such that only one invalid value is present in any test case [7, 17, 18]. This is due to the way in which typical applications are written, namely, to take some failure action upon the first error detected. For this reason a problem known as *input masking*, in which one invalid input prevents another invalid input from being tested, can occur.

For instance, the routine in Figure 12 can be called with any a or b that is a valid *float*. However, it only makes sense to do the calculation when $a \geq 0$ and $b \geq 0$. For that reason, the routine verifies a and b before any calculation is carried out. Assume a test $[a = -1; b = -1]$ was used to test for values outside of the valid range. Here, a actually masks b and the verification of b being a non-negative *float* value would never get executed and if it is absent from the implementation this fact would go unnoticed.⁵

⁴Certain precautions must be taken in cases involving removal of parameter values, removal of entire parameters, or addition of new constraints.

⁵It might be desirable to test more than one invalid value in a test case but it should be done in addition to covering each invalid value separately. Both cases can easily be handled by PICT.

```

Type: Single, Spanned, Striped, Mirror, RAID-5
Size: 10, 100, 1000, 10000, 40000
Format method: Quick, Slow
File system: FAT, FAT32, NTFS
Cluster size: 512, 1024, 2048, 4096, 8192, 16384
Compression: On, Off

# There are limitations on volume size

IF [File system] = "FAT" THEN [Size] <= 4096;
IF [File system] = "FAT32" THEN [Size] <= 32000;

# And not all file systems support compression

IF [File system] <> "NTFS" or
([File system] = "NTFS" and [Cluster size] > 4096)
THEN [Compression] = "Off";

```

Figure 8: Parameters for volume creation and formatting augmented with constraints

Input:																															
A: 0, 1																															
B: 0, 1																															
C: 0, 1																															
IF [A] = 0 THEN [B] = 0; IF [B] = 0 THEN [C] = 0; IF [C] = 0 THEN [A] = 1;																															
<hr/>																															
Before and after calculating dependent exclusions:																															
A:0, B:1 B:0, C:1 ⇒ A:0 A:0, C:0																															
<hr/>																															
Before and after excluding dependent combinations:																															
<table border="1"> <thead> <tr> <th>AB</th><th>AC</th><th>BC</th></tr> </thead> <tbody> <tr> <td>00</td><td>00</td><td>00</td></tr> <tr> <td>01</td><td>01</td><td>01</td></tr> <tr> <td>10</td><td>10</td><td>10</td></tr> <tr> <td>11</td><td>11</td><td>11</td></tr> </tbody> </table>	AB	AC	BC	00	00	00	01	01	01	10	10	10	11	11	11	<table border="1"> <thead> <tr> <th>AB</th><th>AC</th><th>BC</th></tr> </thead> <tbody> <tr> <td>00</td><td>00</td><td>00</td></tr> <tr> <td>01</td><td>01</td><td>01</td></tr> <tr> <td>10</td><td>10</td><td>10</td></tr> <tr> <td>11</td><td>11</td><td>11</td></tr> </tbody> </table>	AB	AC	BC	00	00	00	01	01	01	10	10	10	11	11	11
AB	AC	BC																													
00	00	00																													
01	01	01																													
10	10	10																													
11	11	11																													
AB	AC	BC																													
00	00	00																													
01	01	01																													
10	10	10																													
11	11	11																													

Figure 9: Calculating dependent exclusions

Input:																																																																																																																														
A: 0, 1																																																																																																																														
B: 0, 1																																																																																																																														
C: 0, 1																																																																																																																														
D: 0, 1																																																																																																																														
IF [A] = 0 AND [B] = 0 THEN [C] = 0;																																																																																																																														
<hr/>																																																																																																																														
Exclusions:																																																																																																																														
A:0, B:0, C:1																																																																																																																														
<hr/>																																																																																																																														
Before and after adding new parameter interaction ABC:																																																																																																																														
<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>AB</th><th>AC</th><th>AD</th><th>BC</th><th>BD</th><th>CD</th><th>ABC</th></tr> </thead> <tbody> <tr> <td>00</td><td>00</td><td>00</td><td>00</td><td>00</td><td>00</td><td>000✓</td></tr> <tr> <td>01</td><td>01</td><td>01</td><td>01</td><td>01</td><td>01</td><td>001</td></tr> <tr> <td>10</td><td>10</td><td>10</td><td>10</td><td>10</td><td>10</td><td>010✓</td></tr> <tr> <td>11</td><td>11</td><td>11</td><td>11</td><td>11</td><td>11</td><td>011✓</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td>100✓</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td>101✓</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td>110✓</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td>111✓</td></tr> </tbody> </table> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>AB</th><th>AC</th><th>AD</th><th>BC</th><th>BD</th><th>CD</th><th>ABC</th></tr> </thead> <tbody> <tr> <td>00</td><td>00</td><td>00</td><td>00</td><td>00</td><td>00</td><td>000</td></tr> <tr> <td>01</td><td>01</td><td>01</td><td>01</td><td>01</td><td>01</td><td>001</td></tr> <tr> <td>10</td><td>10</td><td>10</td><td>10</td><td>10</td><td>10</td><td>010</td></tr> <tr> <td>11</td><td>11</td><td>11</td><td>11</td><td>11</td><td>11</td><td>011</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td>100</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td>101</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td>110</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td>111</td></tr> </tbody> </table>	AB	AC	AD	BC	BD	CD	ABC	00	00	00	00	00	00	000✓	01	01	01	01	01	01	001	10	10	10	10	10	10	010✓	11	11	11	11	11	11	011✓							100✓							101✓							110✓							111✓	AB	AC	AD	BC	BD	CD	ABC	00	00	00	00	00	00	000	01	01	01	01	01	01	001	10	10	10	10	10	10	010	11	11	11	11	11	11	011							100							101							110							111
AB	AC	AD	BC	BD	CD	ABC																																																																																																																								
00	00	00	00	00	00	000✓																																																																																																																								
01	01	01	01	01	01	001																																																																																																																								
10	10	10	10	10	10	010✓																																																																																																																								
11	11	11	11	11	11	011✓																																																																																																																								
						100✓																																																																																																																								
						101✓																																																																																																																								
						110✓																																																																																																																								
						111✓																																																																																																																								
AB	AC	AD	BC	BD	CD	ABC																																																																																																																								
00	00	00	00	00	00	000																																																																																																																								
01	01	01	01	01	01	001																																																																																																																								
10	10	10	10	10	10	010																																																																																																																								
11	11	11	11	11	11	011																																																																																																																								
						100																																																																																																																								
						101																																																																																																																								
						110																																																																																																																								
						111																																																																																																																								

Figure 10: Handling exclusions that are more granular than parameter interaction structures

Test parameters:																		
Type: Single, Spanned, Striped, Mirror, RAID-5																		
Size: 10, 100, 1000, 10000, 40000																		
Format method: Quick, Slow																		
File system: FAT, FAT32, NTFS																		
Cluster size: 512, 1024, 2048, 4096, 8192, 16384																		
Compression: On, Off																		
<hr/>																		
Seeding file:																		
<table border="1" style="width: 100%;"> <thead> <tr> <th>Type</th><th>Size</th><th>Format method</th><th>File system</th><th>Cluster size</th><th>Compression</th></tr> </thead> <tbody> <tr> <td>Single</td><td>100</td><td>Quick</td><td>NTFS</td><td>4096</td><td>Off</td></tr> <tr> <td>Mirror</td><td></td><td>Slow</td><td>FAT32</td><td></td><td></td></tr> </tbody> </table>	Type	Size	Format method	File system	Cluster size	Compression	Single	100	Quick	NTFS	4096	Off	Mirror		Slow	FAT32		
Type	Size	Format method	File system	Cluster size	Compression													
Single	100	Quick	NTFS	4096	Off													
Mirror		Slow	FAT32															

Figure 11: Seeding

Routine to be tested:																													
<pre>float SumSquareRoots(float a, float b) { if (a < 0) throw error; if (b < 0) throw error; return (sqrt(a) + sqrt(b)); };</pre>																													
<hr/>																													
Sample model with <i>invalid</i> values marked with ‘*’:																													
A: *LessThanZero, Zero, GreaterThanZero B: *LessThanZero, Zero, GreaterThanZero																													
<hr/>																													
Resulting test cases:																													
<table border="1"> <thead> <tr> <th>A</th><th>B</th><th></th></tr> </thead> <tbody> <tr> <td>Zero</td><td>GreaterThanZero</td><td>(positive)</td></tr> <tr> <td>Zero</td><td>Zero</td><td>(positive)</td></tr> <tr> <td>GreaterThanZero</td><td>Zero</td><td>(positive)</td></tr> <tr> <td>GreaterThanZero</td><td>GreaterThanZero</td><td>(positive)</td></tr> <tr> <td>*LessThanZero</td><td>GreaterThanZero</td><td>(negative)</td></tr> <tr> <td>*LessThanZero</td><td>Zero</td><td>(negative)</td></tr> <tr> <td>GreaterThanZero</td><td>*LessThanZero</td><td>(negative)</td></tr> <tr> <td>Zero</td><td>*LessThanZero</td><td>(negative)</td></tr> </tbody> </table>			A	B		Zero	GreaterThanZero	(positive)	Zero	Zero	(positive)	GreaterThanZero	Zero	(positive)	GreaterThanZero	GreaterThanZero	(positive)	*LessThanZero	GreaterThanZero	(negative)	*LessThanZero	Zero	(negative)	GreaterThanZero	*LessThanZero	(negative)	Zero	*LessThanZero	(negative)
A	B																												
Zero	GreaterThanZero	(positive)																											
Zero	Zero	(positive)																											
GreaterThanZero	Zero	(positive)																											
GreaterThanZero	GreaterThanZero	(positive)																											
*LessThanZero	GreaterThanZero	(negative)																											
*LessThanZero	Zero	(negative)																											
GreaterThanZero	*LessThanZero	(negative)																											
Zero	*LessThanZero	(negative)																											

Figure 12: Avoiding input masking

PICT allows marking values as *invalid* (Figure 12). The output of such model has the following properties:

1. All valid values are t -wise combined with all other valid values in positive test cases.
2. If a test case contains an invalid value, there is only one such value.
3. All invalid values are t -wise combined with all valid values in negative test cases.

The actual implementation of negative testing in PICT uses two generation runs: first, on test parameters with invalid values removed (positive test cases) and second, on all values augmented with extra exclusions to disallow two invalid values to coexist in one test case (negative test cases). Even without this feature implemented, a user could achieve the same results by modifying models and running the generator twice. In fact, this feature is not a part of the core generation engine and is implemented in a higher layer of PICT and is there for users’ convenience only.

In practice, this concept can be extended from disallowing two *values* to disallowing any two or more *combinations* to coexist in one test case. Since this situation also can be handled with appropriately crafted constraints and occurs less frequently than the need for handling individual invalid values, the author never felt compelled to implement it.

3.6 Specifying Expected Results

Having a simple way of defining expected results for test cases is another useful feature.

If there are only two possibilities: test cases with valid values must always succeed and test cases with invalid values should always fail, the task of specifying expected results is straightforward and does not require any support from the engine. Frequently, however, rules of deciding the outcome of a test are more complex than checking for existence of an invalid value in the input data.

Traditionally, for the kind of output PICT produces, either manual evaluation and assignment of expected test results or automated post-processing of test cases is used. Both are labor-intensive. The former is very hard to maintain when input model changes, the latter is typically implemented as a set of single-purpose scripts which have to be rewritten each time a new test domain is modeled. To simplify this task, PICT re-uses its existing artifacts, namely parameters and constraints, and allows for defining expected results within the test model itself.

Defining expected results requires (1) specifying possible result outcomes in form of result parameters and (2) defining rules of assigning result values to test inputs (see Figure 13).

Defining result parameters is as straightforward as defining input parameters. Result rules are syntactically the same as constraints which makes them easy to use. Semantically, however, rules must be both *complete* and *consistent* in the

Sample model for int Sum(int[] Array, int Start, int Count) with expected results specified:

```
# Input parameters:

Array: *Null, Empty, Valid
Start: *TooLow, InRange, *TooHigh
Count: *TooFew, Some, All, *TooMany

# Result parameters:

$Result: Pass, OutOfBounds, InvalidPointer

# Result rules:

IF [Array] IN {"Empty", "Valid"} AND
    [Start] IN {"InRange"}          AND
    [Count] IN {"Some", "All"}
THEN [$Result] = "Pass";

IF [Array] = "Null"
THEN [$Result] = "InvalidPointer";

IF [Start] IN {"TooLow", "TooHigh"} OR
    [Count] IN {"TooFew", "TooMany"}
THEN [$Result] = "OutOfBounds";
```

Test cases contain input data and expected results:

Array	Start	Count	\$Result
Empty	InRange	All	Pass
Valid	InRange	Some	Pass
Valid	InRange	All	Pass
Empty	InRange	Some	Pass
Empty	InRange	*TooMany	OutOfBounds
Empty	*TooHigh	All	OutOfBounds
Valid	InRange	*TooMany	OutOfBounds
Empty	InRange	*TooFew	OutOfBounds
*Null	InRange	All	InvalidPointer
Empty	*TooLow	Some	OutOfBounds
Valid	InRange	*TooFew	OutOfBounds
*Null	InRange	Some	InvalidPointer
Valid	*TooHigh	Some	OutOfBounds
Valid	*TooLow	All	OutOfBounds

Figure 13: Specifying expected results

context of values of a result parameter, which was not a requirement for constraints. Completeness and consistency of result rules are required to ensure that one and only one result value can be assigned to each possible combination of input parameters.

To deal with result parameters, PICT uses the same procedure that handles constraints. It employs its mixed-strength generation capability to combine the result parameters, which always have order of generation t set to 1, with the input parameters. It also adds pre- and post-processing steps to ensure consistency of expected results. At this time, there is no verification of completeness of result rules. Users must be careful to define result rules which assign at least one result value to each possible combination of input values. To allow the tool to distinguish between input and result parameters and apply additional processing steps to the latter group, names of result parameters in PICT are, by convention, prefixed with a '\$'.

3.7 Assigning Weights to Values

In practical applications of automated test generation, it frequently happens that certain parameter values are presumed more ‘important’ than others. For instance, a certain value among others could be a default choice in the SUT therefore the likelihood of it being chosen by a user is greater than her picking other values. Weighting feature in PICT allows putting more emphasis on certain parameter values. Figure 14 shows how to set weights on values.

An ideal weighting mechanism would allow the user to specify proportions of values and actually deliver a test suite that follows them exactly. However, this cannot be guaranteed for strategies whose primary purpose is to minimize the number of test cases covering all t -wise combinations. PICT uses value weights only if two value choices are identical with regards to covering still unsatisfied combinations. In fact, in the ideal test generation run when at each step there always exists a value that wins over others in terms of combination coverage, weights will not be honored at all.

In practice users may not want to define precise likelihoods of choosing values and frequently are satisfied with a mechanism that only allows them to pick certain values more often than others. PICT satisfies that requirement very well.

4. FUTURE WORK

Although PICT already has a reasonably rich set of features, further improvements are needed. Especially in the area of sub-modeling which at this time only allows for defining one level of sub-models. It is actually a limitation of the user interface; the underlying engine is able to handle any number of model levels and it should be considerably straightforward to enable it in the user interface as well. An entirely new and better sub-modeling schema, described in section 3.2, aimed at achieving low volatility of certain parameters could also be added.

Another refinement is required in the area of handling of result rules. Namely, automatic verification of result rules completeness is needed. It would remove the burden of manual work from users and fully ensure correctness of the result definitions.

5. SUMMARY

PICT has been in use at Microsoft since 2000. It was designed with usability, flexibility and speed in mind. It employs a simple yet effective core generation algorithm which has separate preparation and generation phases. This flexibility allowed for implementation of several features of practical importance. PICT gives testers a lot of control over the way in which tests are generated, it raises the level of modeling abstraction, and makes the pairwise generation convenient and usable.

6. ACKNOWLEDGMENTS

Special recognition to David Erb, who architected and implemented the original generation algorithm, and whose excellent design allowed for many of the advanced features to appear in later versions of PICT. Thanks to Keith Stobie, Noel Nyman and John Lambert of Microsoft Corp., Keizo Tatsumi of Fujitsu Ltd, and Richard Vireday or Intel Corp. for their insightful perusal of the first draft.

7. REFERENCES

- [1] <http://www.pairwise.org/tools.asp>.
- [2] P. E. Ammann and A. J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Ninth Annual Conference on Computer Assurance (COMPASS'94)*, Gaithersburg MD, pages 69–80, 1994.
- [3] J. Bach and P. Shroeder. Pairwise testing - a best practice that isn't. In *Proceedings of the 22nd Pacific Northwest Software Quality Conference*, pages 180–196, 2004.
- [4] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation, and test coverage. In *Proceedings of the International Conference on Software Testing, Analysis, and Review (STAR)*, San Diego CA, 1998.
- [5] K. Burroughs, A. Jain, and R. L. Erickson. Improved quality of protocol testing through techniques of experimental design. In *Proceedings of the IEEE International Conference on Communications (Supercomm/ICC'94)*, May 1-5, New Orleans, Louisiana, pages 745–752, 1994.
- [6] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions On Software Engineering*, 23(7), 1997.
- [7] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–87, 1996.
- [8] C. J. Colbourn, M. B. Cohen, and R. C. Turban. A deterministic density algorithm for pairwise interaction coverage. In *Proceedings of the IASTED International Conference on Software Engineering*, 2004.
- [9] S. R. Dalal, A. J. N. Karunanithi, J. M. L. Leaton, G. C. P. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the International*

Type:	Single (5), Spanned (2), Striped (2), Mirror (2), RAID-5 (1)
Size:	10, 100, 1000, 10000, 40000
Format method:	Quick (5), Slow (1)
File system:	FAT (1), FAT32 (1), NTFS (5)
Cluster size:	512, 1024, 2048, 4096, 8192, 16384
Compression:	On (1), Off (10)

Figure 14: Value weights

- Conference on Software Engineering (ICSE 99), New York*, pages 285–294, 1999.
- [10] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. In *Proceedings of the International Conference on Software Engineering (ICSE 97)*, New York, pages 205–215, 1997.
 - [11] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies - a survey. *GMU Technical Report*, 2004.
 - [12] A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *Discrete Mathematics*, 284(1-3):149–56, 2004.
 - [13] R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In *Proceedings of the 27th NASA/IEEE Software Engineering Workshop, NASA Goddard Space Flight Center*, 2002.
 - [14] Y. Lei and K. C. Tai. In-parameter-order: a test generation strategy for pairwise testing. In *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium*, pages 254–261, 1998.
 - [15] Y. K. Malaiya. Antirandom testing: getting the most out of black-box testing. In *Sixth International Symposium on Software Reliability Engineering, Oct. 24-27, 1995*, pages 86–95, 1996.
 - [16] R. Mandl. Orthogonal latin squares: an application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054–1058, 1985.
 - [17] G. J. Myers. *The Art of Software Testing*. John Wiley and Sons, 1978.
 - [18] G. Sherwood. Effective testing of factor combinations. In *Proceedings of the Third International Conference on Software Testing, Analysis and Review, Washington, DC*, pages 133–166, 1994.
 - [19] H. Shimokawa and S. Satoh. Method of setting software test cases using the experimental design approach. In *Proceedings of the Fourth Symposium on Quality Control in Software Production, Federation of Japanese Science and Technology*, pages 1–8, 1984.
 - [20] B. Smith, M. S. Feather, and N. Muscettola. Challenges and methods in testing the remote agent planner. In *Proceedings of AIPS*, 2000.
 - [21] K. C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Transactions of Software Engineering*, 28(1), 2002.
 - [22] K. Tatsumi. Test case design support system. In *Proceedings of the International Conference on Quality Control (ICQC), Tokyo*, 1987, pages 615–620, 1987.
 - [23] D. R. Wallace and D. R. Kuhn. Failure modes in medical device software: an analysis of 15 years of recall data. *International Journal of Reliability, Quality and Safety Engineering*, 8(4), 2001.
 - [24] A. W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the 13th International Conference on Testing Communicating Systems (Test-Com 2000)*, pages 59–74, 2000.
 - [25] A. W. Williams and R. L. Probert. A practical strategy for testing pair-wise coverage of network interfaces. In *Proceedings of the Seventh International Symposium on Software Reliability Engineering (ISSRE '96)*, page 246, 1996.

Accelerating Performance Testing – A Team Approach

Dawn Haynes

Independent Consultant
1600 Main St. SE
Albany, OR 97322
dawn@midnightdawn.com

Biography

Dawn is a consultant providing software quality, testing, and training services for companies like Security Innovation, Quality Tree Software, and SQE Inc. With over 21 years of experience in IT and high-tech, Dawn has spent the last 15 years focusing on software testing projects implementing process improvements, advanced testing techniques, and tools. Alongside these efforts, she has also delivered testing-related technical training, developed many training courses for various audiences, and managed training curriculums and departments. Dawn's career has included successful technical positions at companies like Xerox, Rational Software, SoftBridge Microsystems, and Ipswitch, Inc.

Dawn is a member of ASQ and ASTD, and has participated in several invitation-only industry initiatives like the Workshop on Performance and Reliability (WOPR). She is also a contributing author of the book "Quality Web Systems: Performance, Security & Usability." Dawn holds a BSBA in MIS from Northeastern University.

Abstract

While some organizations deserve kudos for even thinking about evaluating the performance of their software systems prior to deployment, it is often an afterthought resulting in a relatively unplanned, under-resourced, time-crunched, and unproductive effort. At odds with this common situation is that performance problems can be dramatic enough to cause the "failure" of the deployed system. And yet, even with such high stakes, I've seen this scenario repeat itself over and over again in both small and large companies, and on low-risk and very critical implementations.

This paper outlines a strategy for engaging key members of the development and deployment staffs to assist test teams in planning and executing performance tests - dramatically accelerating the process and minimizing down time during testing cycles.

Copyright: © 2006 Dawn Haynes / Midnight Dawn Consulting

Introduction

Performance testing is typically done, if at all, near the end of a development effort. This compresses a very difficult and expensive task into a small set of cycles compounded with high expectations of getting accurate and useful data prior to deployment.

However, test teams rarely have the support they need to be successful and often own the complete problem of planning, setting up, and executing performance test scenarios in isolation. Additionally, test teams are often expected to pinpoint errors so development teams can effectively diagnose problems and recommend solutions. Most average to experienced test teams will be gurus of testing practices and tools, but it is unlikely that they are cumulative experts in databases, middleware, networks, hardware, other technologies, and key components of their target systems under test. Any gaps in knowledge will invariably cause crucial details to be missed in the planning and test execution process. These gaps significantly bloat the entire performance testing and system tuning process downstream. Using the right performance testing team for planning, execution and debugging will reduce the time in getting accurate results dramatically.

Common pitfalls of performance testing

Over the past 10 years I have worked with, trained, and supported hundreds of individuals tasked with generating performance metrics for their projects prior to release or deployment. During that time I have observed a common set of issues that prohibit success on most projects.

- Not enough time given to perform the task correctly
- Not enough resources provided (budget, tools, staff)
- Lack of access to resources (staff or lab/deployment hardware)
- Incomplete or scaled-down test systems
- Insufficient training on target systems, underlying technologies, or test tools
- Throw it over the wall approach (test team works in isolation)
- Pinpointing, debugging and triage of issues is uncoordinated and poorly managed
- Testing is delayed while testers are waiting for resolutions from disparate teams (development, deployment, IT, etc.)
- Testing cycles become bloated when unnecessary/simple errors cause a test run to fail

These issues can severely interfere with the basic generation of performance metrics, but in some circumstances, success can be even more devastating. Inaccurate or misleading performance results can cause teams to make poor decisions or implement unnecessary or problematic changes. To add insult to injury, a lack of success on any performance testing project can often impede future efforts (specifically regarding support from management).

Given that there are many valid reasons for system performance testing to occur late in project life cycles, extra effort should be made to ensure the success of the testing and the accuracy of the results. Resources should be utilized as efficiently as possible with

the goals of reducing test cycle time, improving tuning and fix decisions, and having more confidence in the results. In most of the projects I've been affiliated with, I've found that even the best efforts still succumb to many of these common pitfalls.

An enlightening moment

While working at a large software company, I was brought in on an internal performance testing project which seemed to be suffering from many of the common pitfalls. Their testing efforts were stalled because they believed they were having a test tool problem (common fallacy) and I was tapped because I was an expert in that tool. They were hoping I could identify their blocking issues so they could perform their testing and get the metrics they were looking for before their go live date.

I happily joined the team and came up to speed on the situation as quickly as possible. Here are some of the important details:

1. It's Wednesday and their go-live target date is the following Monday.
2. There was one person responsible for the performance testing.
3. The test target was a newly architected web site that would replace an existing web site. The system utilized new technologies that were unfamiliar to the test lead, and still being learned by the development team.
4. The load balancing hardware that was to be used with the web servers had not arrived yet (it came on Saturday).
5. All the target servers were in another facility off-site. There was no way to observe them, gather data from them, or tweak/tune their settings directly.
6. The test "lab" was a 4-foot square piece of conference room table. The room was shared by at least six members of the development team.
7. The test "lab" consisted of one test driver machine connected to the shared network hub in the room.
8. The test lead analyzed the existing system's usage patterns and developed a well-designed workload for the performance tests.
9. The test lead was unable to generate more than a six-user load with the test tool.
The target load was 100 users.
10. The test lead determined the test tool was the problem.

With that as background, even though the project was being poorly managed, staffed, and funded, I had confidence in the test lead's approach. Because of this, I believed that the test lead had analyzed the problem thoroughly and I began to investigate common problems utilizing the test tool. This was very short-sighted on my part as I had seen dozens of situations where teams falsely accused the test tool of failing when their test environment or their system under test was flawed (and the tool was adeptly pointing that out). I spent the better part of a day determining the test design was solid and the test tool was performing accurately.

After a late evening on day one of my intervention, day two began early and was flooded with pressure and frustration. The project manager was inquiring about the

progress of the testing and none had been made. In the midst of the morning's frustration, after a couple hours of running several more unsuccessful tests, the test lead had a meltdown and said loudly, "why can't we get more than 6 users to run!!!" A head popped up on the other side of the tiny room. It was the database architect. He turned his head toward us and said, "six?" The test lead jumped out of his seat and said, "yeah, what do you know about six?" Some muttering occurred between them and the database guru suggested he could look into a few things. After much furious typing, exits and re-entries, a re-build and push to the server, the database architect suggested we try running the test again. Voila! Now we got up to ten users!

Now while ten users isn't an extraordinary success, it was the first roadblock that was overcome. Many more roadblocks were overcome in the next three days – all with the assistance of development staff and deployment engineers. There were problems with the network bandwidth to the room the test driver was in and it was subsequently upgraded. There was a problem with some database licensing limits and faulty license server configuration. The Web server's default timeout settings caused random failures generating inconsistent and hard to troubleshoot results. Implementing the load balancing hardware changed the test results so dramatically that all previous results had to be tossed out. And so on, and so on.

The amazing thing I extracted from this experience was witnessing the speed and accuracy of pinpointing problems when the right team of people was engaged in looking at the problem. In reality, I did almost nothing to assist the team on this project. I was merely an observer and they did all the work and delivered their own success. Funny that this success was not brought about by masterful management or clever planning, but instead by the pure accident of proximity as the bulk of the development team leads were all in the same room together while the testing was being performed.

Prior to that experience, I would have believed it very difficult to get truly useful performance testing results in five days (unless you were in the testing Garden of Eden). This team proved that it can be done when everyone is dedicated to the effort. As quick progress was shown, project management made anyone and everyone available to the test lead as a resource, and that was the key to success.

As a postscript, the new Web site went live on Monday and did not perform flawlessly. But the testing team was able to run tests effectively and engage the team to pinpoint problems so they could be resolved very quickly. This cycle went on for several weeks to work out the kinks, but no showstoppers were found. The project was deemed a success, and as with any successful project, I got a t-shirt.

I analyzed this whole experience to derive the critical elements that enabled the project to move forward in this accelerated fashion in spite of compressed time schedules, lack of sufficient resources, and incredible pressures to deliver. In a word, the critical element was T-E-A-M. The right team at the right time compressed a nightmarish testing effort into days and delivered unexpected success.

I have since developed a step-by-step process to integrate a team-based approach to accelerate performance testing projects of all sizes and regardless of constraints.

Common elements of a performance testing process

Most software testing processes contain a minimum set of basic tasks:

Plan > Design/Document > Execute > Evaluate/Report

It's easy to say that for every type of testing effort that these same tasks should be performed. But with performance testing, it's very common that no planning gets done (getting the effort off to a great start), and that very little design and documentation is performed. This is a great oversight and missed opportunity. A robust but short planning and design effort can dramatically improve results downstream since a poorly designed test will just be a poorly executed test producing inaccurate results. Planning and design are the key elements to a successful performance test effort, and can be greatly enhanced by applying a team approach.

Test execution is typically the domain of test engineers and tool experts, but with performance testing, it is rare that the testers possess key knowledge about all the domains involved (databases, networking, hardware, web technologies, etc.). These gaps in knowledge not only interfere with the successful execution of tests causing significant delays, but also can cause testers to misdiagnose the probable causes of performance issues. Engaging the right resources during test execution can significantly speed up the pace and improve the accuracy of results.

Testers can easily report the results of performance test runs, but pinpointing the source of performance problems is exceedingly difficult. If development teams adhere to an isolated process of defect triage, performance problems can end up in an endless cycle of defect reassignment (finger pointing) consuming valuable time without making progress. If the right team of people work together to perform triage, many potential solutions can be worked through in a fraction of the time typically spent to deliver one potential solution.

To break through the typical constraints of performance testing projects and processes, a different approach is necessary. This approach will require significant management buy-in and support on multiple levels. But if the performance testing effort is to succeed, commensurate resources should be applied.

Applying a team approach to accelerate performance testing projects

In order to accelerate performance testing efforts with this method, significant additional resources external to the test team will need to be applied at various stages in the process. For optimal results, the right personnel with the right skill set will need to be selected for involvement, regardless of their title or position, or departmental affiliation.

Key members of a performance testing "dream team" would include:

- User representative (actual user, business analyst, user acceptance tester, product manager, customer support representative)
- Test lead
- System architect / lead developer
- Component leads (Web developer, database architect, middleware expert, etc.)
- IT/deployment/production hardware experts
- Networking guru

Others could be involved if they have insight into how the system was built, how it will be deployed, or how it will be used. The test lead will act as a facilitator to get this team to provide support at crucial moments during the performance testing effort. In order to be successful, the test lead must have the latitude to access and engage these team members on an as-needed basis. However, it is imperative that the entire team be responsible for the stated goals and outcome of the effort.

Stage 1 – Planning & Design

The team works together to plan the effort. This may require one or more meetings, determined by the team, but regardless of the frequency, the key deliverables of this meeting should be to:

- Confirm that the appropriate team members are engaged
 - Recommend alternate or additional team members if necessary
- Evaluate the goals of the effort for feasibility, completeness, and attainability
 - Stop the process and report to management if they are not
- Confirm the top target transactions and measurements (load levels, response times, transactions rates, etc.)
 - Draw out the details needed for performing these transactions and gathering metrics (rough test design)
- Identify an appropriate data set for the test and determine how it will be generated
- Identify any potential blocks for running the tests, hitting the targets, or gathering the results
- Identify which team members need to present for test execution
 - This may vary according to the testing target
- Identify which team members will regroup for triage meetings

The most powerful parts of this team exercise lies in refining the testing targets and identifying the potential blocks to test execution or achieving the performance goals. An excellent method for roughing out the test design is to whiteboard or flip chart the transaction flow in a large conference room. Each part of the transaction needs to be captured combining the location (interface) it occurs at and any communication between system components or other interfaces. The resulting diagram shows the transaction as an overlay to key elements of the underlying deployment.

Example transaction – Process a query (see Fig. 1)

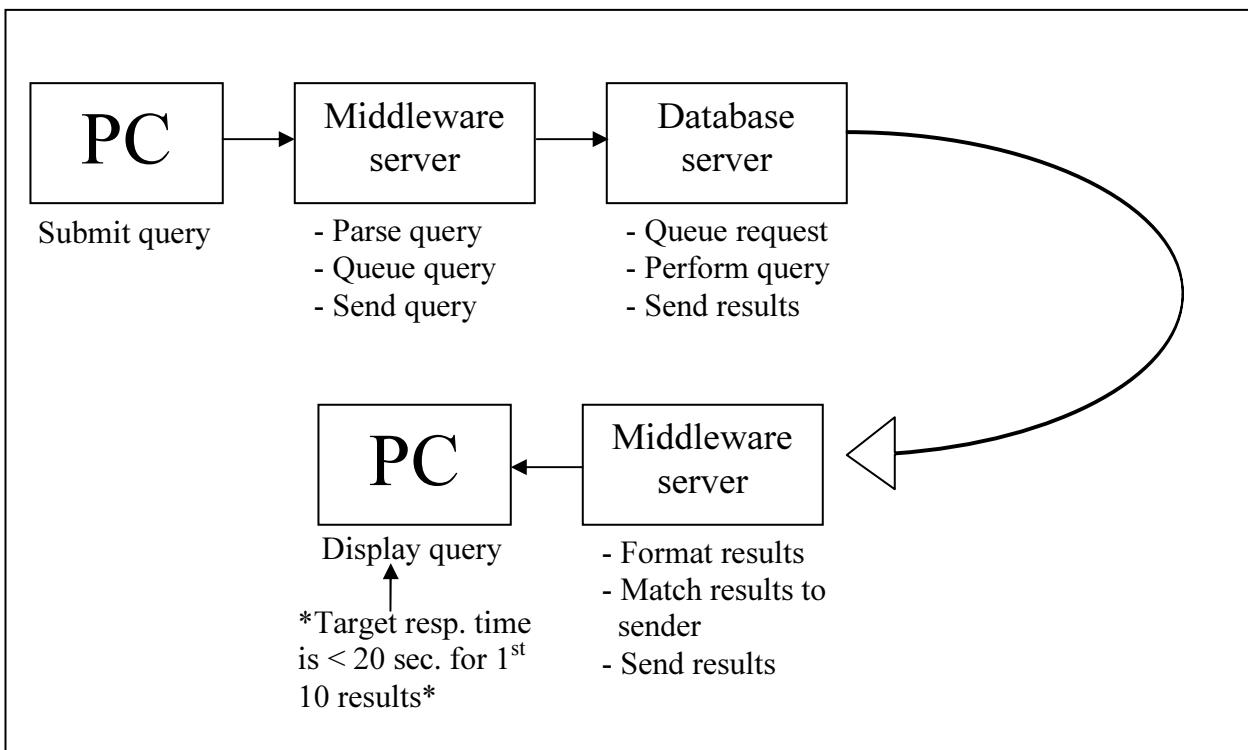


Fig. 1

Here is a step-by-step walkthrough of the first few elements of the diagram:

1. User fills out a query screen on a PC and clicks “Go”
 - a. Draw a box to represent the PC (write PC inside the box)
 - b. List SUBMIT QUERY under the box to indicate the key event/action
2. The query elements are interpreted by the client software, packaged and sent over the network to the middleware server
 - a. Draw a line from the box to indicate a network connection is present and show that the transaction must traverse this connection
3. The middleware server receives the query request, performs an integrity check for proper query format, and queues the request to be sent to the database server
 - a. Draw another box to indicate the middleware server (write the name of the server or “middleware” in the box)
 - b. List the discrete tasks performed by the server below the box
 - c. Draw a line from the box indicating that the query request is sent over the network
4. The database server receives the request and queues it along with other requests. The query is performed and the results set is returned to the middleware server.
5. Etc.

The team would continue adding elements and annotations until the complete transaction is represented and all the relevant system components are identified.

Once the transaction is sketched out, the measurement goals should be added to the diagram. In our example, the performance metric under scrutiny is response time of query results being displayed to the user. The target goal to be measured is that the first 10 results are displayed to the user within 20 seconds of submitting the request (90% of the time).

With all the team members having this common understanding of the transaction and the underlying system components, the test lead will facilitate cycling through the transaction with the goal of drawing out additional, crucial considerations for developing the tests, executing the test runs, and gathering the appropriate metrics. Any additional data will be added to the diagram. Some key questions to work through are:

- Are there any important processing elements missing from the diagram?
- Will data used in the transaction impact the performance results?
 - If so, discuss the impacts and note where they will occur on the diagram (if known, or make an informed guess using the group's wisdom)
- What are the important elements to monitor and how will they be monitored?
 - Note them on the diagram
 - Define what monitoring tool will be used (native, test tool, 3rd party utility, etc.)
 - Determine who will perform the monitoring (or interpret the results of the monitoring data)
 - Identify any elements that can't be monitored

Now the most valuable part of the process can occur. The test lead asks the team members if they can identify any potential bottlenecks or issues by walking through the transaction using the test targets as the variable. Here are some examples:

- Can anyone identify a problem with basic queries meeting the performance requirement? (This is a sanity test of the performance goal, metric or requirement.)
- Can anyone see a problem with simulating 10 users performing queries at the same time? How about 20 users? 50 users? 100 users?
 - Have the team members to identify where in the diagram the transaction is likely to bottleneck based on the respective load targets. Guessing is fine at this stage.
- If queries are fired off at a rate of 1 every 10 seconds, when would the system bog down (fail to meet the performance requirement)?
- If a query is submitted that will return a result set of over 1000 rows, what is the likely impact to the system?

With this information in hand, the meeting is adjourned. The test lead is responsible for documenting the results of the meeting and distributing the test plan for review by the team and other stakeholders.

Stage 2 – Test Execution

The test lead and other key testers are responsible for building the appropriate tests as defined by the planning meeting. The tests should be “tested” to confirm the transactions are being performed correctly and that any other design elements are being executed as expected.

Once the tests are deemed solid, the tests must be ramped up to generate the target scenarios, and monitoring must be turned on for all the key system components. Early ramp tests are plagued with a variety of “getting started” problems, and significant time can be wasted trying to run down all the potential root-causes. Since testers aren’t experts on all the system components and discrete elements of the transactions, it is most helpful to engage the larger team during this activity. With several eyes on the problem, troubleshooting the errors becomes a quick exercise. Getting the test team over this speed bump is a major key to accelerating the entire effort as down time here is completely unproductive as no useful performance measurements can be generated.

Once the target scenarios can be executed and the testers are comfortable with monitoring key components, the additional team members are no longer needed to observe and assist with execution. However, if there are inexplicable anomalies that interfere with test execution downstream, the team should be reconstituted to troubleshoot the new situation.

Stage 3 – Test Results & Triage

Testers typically report performance related issues and failures to meet requirements through a change management or defect tracking system. This is likely to be the same system used for other issue tracking and reporting, and therefore the supporting workflow and triage processes are likely to be the same. If this occurs, it is very likely that performance related issues will travel in hopeless circles without anyone ever owning the problem of determining the root cause.

Since the root cause of a performance problem can often be traced to a multi-tiered and complex set of issues, once again engaging the larger team to help to accelerate the generation of potential solutions. The test lead should organize a test results reporting meeting and facilitate the triage process. Because this particular team has already worked together to identify the transaction/deployment diagram during test planning, they are uniquely qualified to assess potential changes by working them through the system. As solutions are presented, the team can identify the feasibility of performance improvements as well as generate a list of potential impacts.

Another benefit to this triage process is that the testers and developers are not working in isolation, providing the testers with key insights about the system that can be extremely useful during test execution. This informal knowledge transfer is just another side effect that will dramatically accelerate the testing effort and reduce the impact to ancillary resources in the long run.

Conclusion

Through personal experience, I witnessed an accidental set of circumstances that significantly reduced the time expended in generating a reasonable set of performance testing metrics. This technique was repeated over and over compressing each test cycle and creating opportunities for generating quick results and follow-on performance improvements. This experience encouraged me to think of a way to integrate this technique into a more formal testing process that could improve the efforts regardless of the constraints.

As a result, I've identified how to harness the power of teams and apply it throughout the performance testing lifecycle to reap the biggest benefits – compressing overall testing time and generating more reliable performance results. This approach can be applied regardless of the constraints of time and resources. The only thing that is required for success is the availability of the team members.

Leveraging Model-driven Testing Practices to Improve Software Quality at Microsoft

Jean Hartmann, Test Architect
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
Email: jeanhar@microsoft.com

Biography

I currently hold the position of Test Architect in the Internet Explorer (IE) team. My main responsibility includes driving the concept of software quality throughout the IE development lifecycle. My technical interests, while diverse, have gravitated over the years towards three main areas of interest, namely testing, requirements engineering and architectural analysis. Previously, I was Manager for Software Quality at Siemens Corporate Research for twelve years and earned my Ph.D. in Computer Science in 1992 researching the topic of selective regression test strategies. My passion for model-driven development was kindled back in 1998 and I have researched and developed new approaches to model-driven test generation and consistency checking since then, which resulted in a number of patents.

Abstract

One area that the IE team believes holds promise for significantly improving our software process and in particular, software quality, is leveraging model-driven development practices. Our current improvement efforts are focused on three key areas: requirements engineering, architectural analysis and most importantly, testing. Some key factors driving this adoption are the strong emphasis on addressing customer scenarios, the need for improved product traceability, increased testability, security and reliability of our product architecture and more systematic and efficient process encompassing test design, generation and execution.

This paper chronicles our ongoing effort to introduce model-driven testing practices, which we are aiming to introduce to the team during the IE7.X and IE8 release timeframe. The goal of this effort is to reduce the overall test cycle through the automatic generation and execution of tests based on behavioral models of IE. Our approach, which actually builds on our recent improvement efforts in requirements engineering, leverages the UML (Unified Modeling Language) and its concept of use cases [1]. Use cases, together with behavioral diagrams such as activity, sequence or state chart diagrams, are used to specify our customer scenarios. These behavioral diagrams are then annotated by testers with test-specific directives or test requirements and processed by an automatic test generator developed by Microsoft Research, with the generated tests being executed using an IE-specific test execution environment.

This paper outlines our motivation for exploring the topic of model-driven development. Emphasis is placed on describing the modeling methodology and its impact on the existing requirements and testing processes rather than providing detailed descriptions of the underlying technology and tools. Examples are given to illustrate and emphasize key aspects of the methodology and use the *RSS (Really Simple Syndication)* feature of IE7.

1. Introduction

1.1 Motivation

Our motivation for exploring model-driven development practices to improve software quality and in particular, testing, needs to be considered in the context of three major factors:

- Product history
- Shifts in development culture
- Future market pressures

Product history plays an important role - the key milestone being the successful release of IE6 in 2001. While significant amounts of time were invested in creating regression test suites for that product, the rapid pace of development leading up to the IE6 release resulted in little time being spent on creating good product, design or test documentation. Compounding that, the original IE development team including testers was scaled back after release with key development staff dispersing throughout the company. Inevitably, they took with them much of their product knowledge and expertise.

With the renewed commitment of Microsoft to shipping IE7 for Windows Vista and XP, the newly reformed IE organization and especially the testing team, faces the challenge of validating a legacy product and at the same time testing new product features. We believe model-driven testing practices can help us address the above challenges:

- **Knowledge of product behavior and scope** – we need to quickly and efficiently gain (or regain) knowledge of not only our legacy product, but also absorb that knowledge in conjunction with new feature requirements. A model-driven approach is an effective and efficient way for the entire team and especially testers to explore and understand old and new behavior.
- **Cost of maintaining legacy test code** – we need to minimize the impact on cost for updating and executing large legacy regression test suites. A model-driven testing approach based on automated test generation technology, can be used to complement, and potentially replace over time, these legacy test suites with automatically generated test suites.

Apart from product history, a shift in development culture at Microsoft is encouraging us to apply model-driven testing practices. Company-wide, we are seeing the trends towards:

- **Emphasizing customer scenarios** – we want to elicit and capture key customer scenarios as models for use in product validation¹. Complementing our testing philosophy by embracing a scenario- or use case-driven testing approach represents a significant shift for us from our current, very feature- and API-centric testing activities.
- **Striving toward earliest possible bug detection** – we want to find more efficient and effective ways to promote software quality early in the product lifecycle. Model-based testing practices raise general team awareness by forcing us to define and review a set of model artifacts early on. It is during these model reviews, automated or manual, that give rise to product specification and design bugs. In effect, we are discovering bugs long before designing and executing any tests.
- **Better communications of intended product behavior** – we want to ensure that product intent can be clearly, concisely and unambiguously communicated across the IE organization and our Microsoft partner groups who rely on IE functionality. We believe that UML models are a very effective way to communicate such intent. This is true, both internally and externally, where our offshore test teams, partners and customers are involved. We believe that such improved communications can improve software quality as better product understanding ensures the development of better and more thorough test cases.

¹ As well as leveraging those models for architectural redesign and code re-factoring.

Finally, Microsoft just like any other IT company faces market pressures to deliver higher quality products in ever shorter development cycles. Two key issues that we are addressing with model-driven testing practices in this context are:

- **Encouraging iterative development processes** – we see that a model-driven process can support iterative development much more effectively than our existing waterfall-like processes, with particular benefits for the test team. In combination with automated test generation, we anticipate being able to derive new functional tests for updated scenarios and features very quickly.
- **Improved product traceability** – in order to better focus our development and testing efforts and avoid potentially costly, extraneous feature prototyping efforts, we need to ensure better traceability between requirements and tests. Model-driven testing practices and the supporting tool chain encourage this traceability by providing direct linkage between textual requirements, use case models and generated tests.

1.2 Vision

Figure 1 provides the vision for our approach. The process described in Section 2 envisages the evolution of a set of UML use case models, together with textual feature requirements, to represent our customer scenarios. In Section 3, we discuss how these models are then annotated by testers to transform them into test specifications and prepared for automatic test generation. For Section 4, we outline our automated test generation process, describing briefly the constraints testers can place on the generation step. We also discuss the impact of automated test generation on our existing test development and execution process. We discuss our plans to transform our existing test set into an automation library that can be leveraged by the test generator. In Section 5, we briefly outline our tool implementation strategy with Section 6 providing an overview of related work.

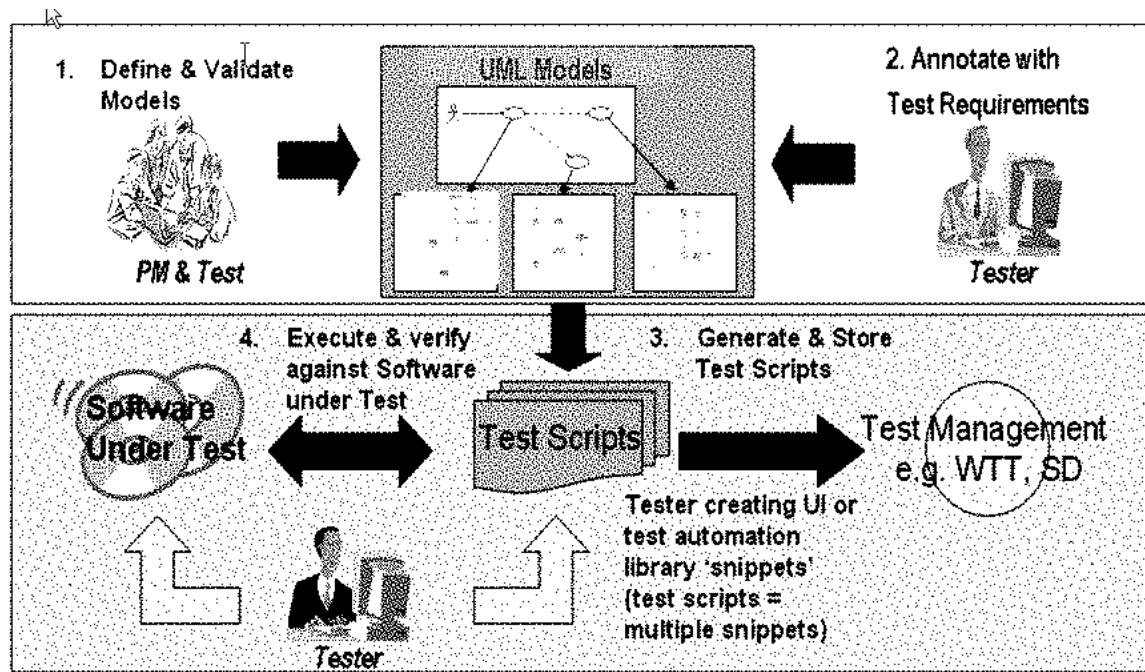


Figure 1: Vision for Model-driven Testing Practices

2. Modeling Customer Scenarios in UML

In this section, representing Step 1 of Figure 1, we describe how we intend to leverage UML use cases as well as other UML diagrams, such as activity diagrams, to capture our customer scenarios in preparation for automatically deriving test plans and scenario-based test cases from them. We then discuss how we are adapting our current development practices to maximize the benefit of model-driven practices. Key technical issues and decisions made during our strategy definition are also outlined.

Use case modeling represents a powerful methodology for depicting system behavior with customer scenarios in mind. The biggest benefit stems from the fact that all dependencies, whether between individual use cases or use cases and their actors/stakeholders, are explicitly described. We found that consumers of such use case documentation, especially testers, are able to digest this type of visual information and quickly gain a better understanding of intended product behavior and scope. Figure 2 shows an early use case diagram for an IE7 feature known as RSS.

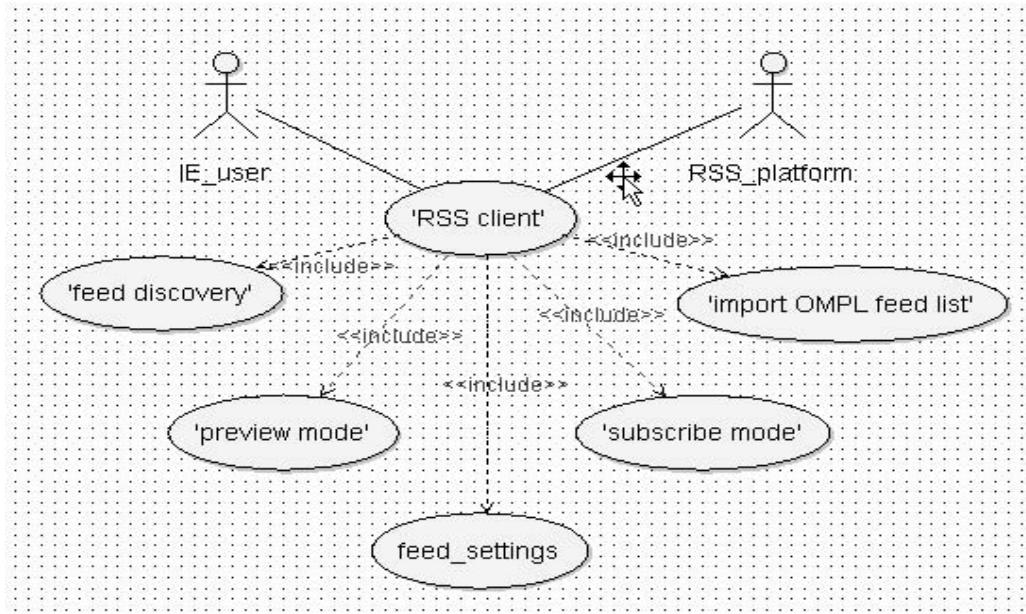


Figure 2: Early Use Case Diagram for RSS Client

Use case specifications come in two flavors: textual or graphical [2]. While several Microsoft groups have begun with the creation of *textual* use case documents, we believe that these documents very quickly become unmanageable when considering complex use cases and customer (test) scenarios, which include multiple, nested alternative flows. As scenario and product complexity rises, the IE team believes that testers may overlook important test scenarios when deriving their test procedures from such documents.

We believe that a much more concise and compelling way of eliciting, evolving and communicating such test scenarios is making use of visual modeling. A good example is an activity diagram, such as the one shown in Figure 3, which enables us to specify intended product behavior and leverage a nested hierarchy of such diagrams to hide scenario complexity or identify product line variations [3]. Apart from leveraging the inherent benefits of visualization, tools can be written to automatically review the resulting models for consistency, style and quality based on the underlying meta-model. This is not possible with textual use case specifications.

Activity diagrams, for example, enable Microsoft program managers (PMs)² and testers to visually capture and discuss the expected flow of control between the system and users. These diagrams are very well suited to reflect the user input and system response paradigm by being partitioning the diagram into so-called 'swim lanes'. The first lane depicts the user input, the second the system response, and together the pair

² Program Managers are Microsoft's equivalent of system analysts.

illustrate the main success or ‘happy’ path. Alternate and exception paths are fleshed out collaboratively by the team at a later date, notably after the main customer scenarios have been specified, and can be depicted in a third swim lane.

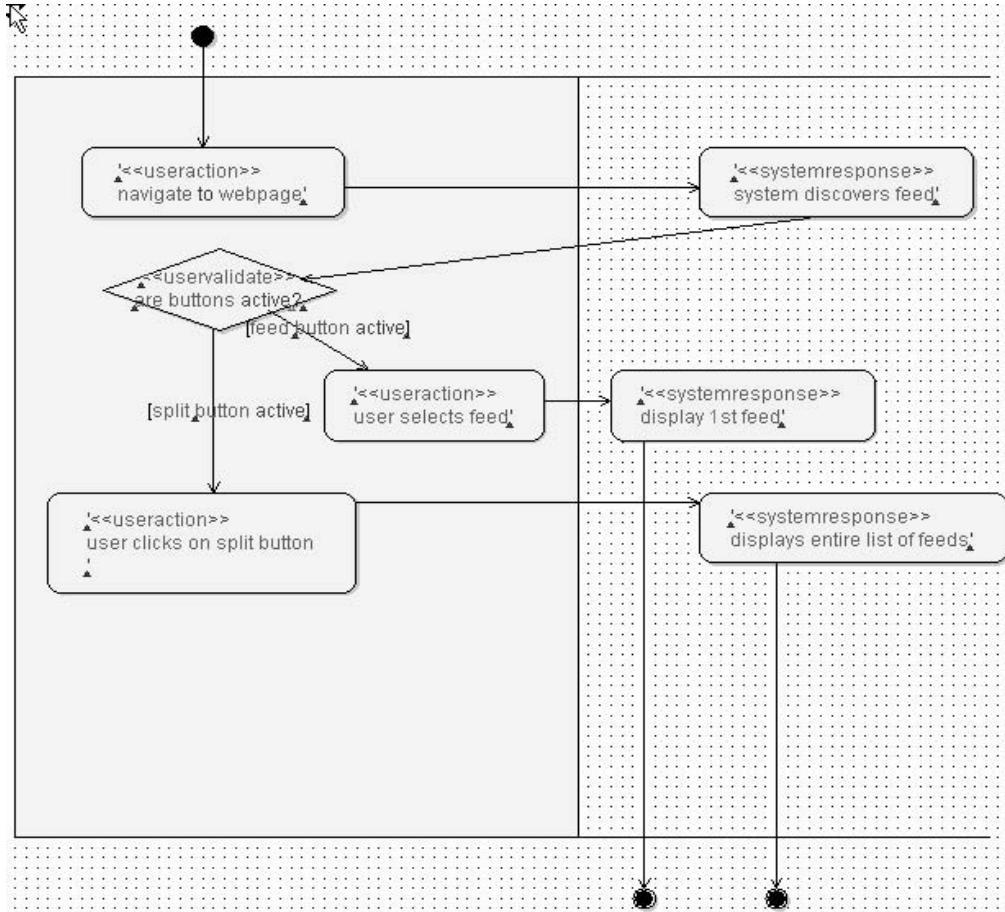


Figure 3: Activity Diagram for RSS Feed Discovery Showing Main Success Scenarios

Leveraging this visual UML use case modeling methodology, we are making the following changes to our existing requirements and testing practices and anticipate the following effects:

- Scenario- rather than feature-centric product specifications.** Rather than evolving the initial product specifications around a collection of Word-based documents, which are feature-centric in nature and are typically defined by PMs only, the new model-driven practices encourage PMs and testers to iteratively capture, structure and depict product behavior and scope around customer scenarios together using UML models as their main artifacts. Thus, specifications will no longer focus on a set of features and requirements, which have no clear and explicit dependencies or flow of control between them. With the new model-driven practices, features will be discussed in the context of customer scenarios instead. The resulting models will also be responsible for governing the outline of the supplementary textual product specifications where details that cannot be denoted graphically, such as user interface specifications, non-functional requirements and other quality attributes of the product, are described. As the scenarios are fleshed out, models and documents will be linked bi-directionally and kept in sync with the help of automated tools.
- Supporting change.** Currently, updates to the product specifications are limited to the initial product definition phase after which they are not consistently updated or communicated. By introducing model-driven testing practices, supported by good automated tools and an underlying process, specifications can be versioned, linked and traced to specific use cases and even tests.

Based on those practices, specification changes resulting from either bug triage or intended product enhancement will improve as they can be clearly scoped, communicated via emails and quantified in terms of the impacted scenarios to be retested. We anticipate that this approach will make a significant contribution to improving our current release criteria.

- c) **Earlier defect prevention.** The creation of a central set of model artifacts enables the IE team to leverage them in order to improve product quality. Foremost is the leveraging of these models for automatic test generation; the subject of this paper. However, other tasks that can help to improve product quality include consistency checking of models [4], impact and traceability analysis [5], and security modeling, such as Threat Model Analysis (TMA) [6]. All of these tasks can be automated to a large degree and run against the model base on a regular basis due to the underlying UML meta-model.

3. Adding Test Requirements to UML Models

In this section, representing Step 2 of Figure 1, we describe how once the use case models and associated activity, sequence or state chart diagrams have been defined and reviewed by the team, testers are responsible for annotating the models with so-called test requirements or test-specific directives. These directives are needed in preparation for test generation by the automatic test generator tool.

The goal of the previous phase (Step 1) was to model product *intent*. This is illustrated by the activity diagram in Figure 2, which describes the intent of user actions and system responses via descriptive text such as “*navigate to webpage*” or “*system discovers feed*”. In this phase, we focus on test case design – we start the iterative process of transforming and refining our intentions into the *mechanics* associated with the development of executable test cases (scripts). This process is started by testers through the specification of test requirements or test-specific directives as annotations in the behavioral diagrams. Testers are focusing on two key objectives:

- **Influencing the exercising of test paths** – by default, the coverage criterion for the control-flow in the models is transition coverage, which means that each arc in a given model is exercised at least once by a generated test case. However, the tester can influence the test generation process by specifying more stringent coverage requirements using the <<coverage>> stereotype. This enables testers to significantly influence the depth and breadth of their testing. For example, targeted application of these coverage requirements would enable tests to be generated that only exercise the main success scenarios or happy paths rather than also including the alternate or exception scenarios. At Microsoft, these tests are referred to as Build Verification Tests or BVTs.
- **Defining input data values** – To complement the generation of tests based on the elaboration of control-flow in the models, we leverage the concept of data variations. Data variations are a key part of the concept of category-partition testing [7]. Testers define so-called categories and partitions wherever they wish the test generator to vary input data values. This may be a field in the product’s user interface or typed input parameters of an API. In both cases, choices are defined, which represent the value ranges for those input data values. These choices are then explored during the test generation process to augment tests created using the model control-flows. To leverage the power of data variations, testers need to define in the models and mark them using the <<define>> stereotype. Further details concerning the use of such variables and their significance for the test generation process are described in [8].

We believe this annotation step brings with it the following changes to our existing testing practices and we anticipate the following effects:

- **Emphasize test design rather than generation-execution** – at present, test case design, whether automated or manual, appears to only validate a small portion of the potential input space for a given set of customer scenarios. By asking testers to think more deeply about test requirements in

- the context of modeling customer scenarios, we are advocating that testers return to what they were hired for – test design, exploring the depth and breadth of that potential product input space.
- **Detailed refinement of the product specifications with other team members** – In the process of explicitly annotating the models with test requirements, which will later be reviewed, we believe that testers will drive the rest of the IE organization to continuously discuss and refine the product specifications by considering details they typically would not be considering at this stage in the process. Explicit discussions concerning issues such as typical and boundary value conditions for the product are the result. Developers also profit from this close interaction with refinements being made to product architecture and design.

4. Test Generation and Execution

In this section, which represents Steps 3 and 4 of Figure 1, we describe the process by which testers can invoke the test generator from within the UML modeling environment and automatically generate tests - manual or automated. We also discuss how model-driven testing practices are acting as a catalyst for promoting better reuse of our legacy and future automated tests. The approach focuses on re-factoring the test code itself to ensure better quality, reliability and stability, but with an eye on utilizing that improved test code and structure by invoking it through the test generator tool.

Before proceeding with a description of the test generation and execution steps, we would like to emphasize the following:

- Our approach generates a set of black-box conformance tests. These test cases aim to ensure the compliance of the system specification with the resulting implementation.
- It is assumed that the implementation behaves in a deterministic and externally controllable way. Otherwise, the generated test cases may not be meaningful.

Our approach foresees testers interactively invoking the underlying test generation tool via the user interface of the UML modeling tool whenever the review of the model annotations is complete³. During this invocation, testers will be given control over the following aspects of the generation step:

- The ability for testers to select more stringent control-flow coverage criterion than the default of transition coverage. This reflects and aligns with the different types of testing currently being conducted at Microsoft for a given product. For example, testers may wish to only generate so-called BVT or Build Verification Tests corresponding to the main success scenarios or happy paths of the models and select the associated coverage criterion.
- The possibility for testers to scope their test generation effort. By interactively highlighting one or more use cases (or even the root use case) in the model diagrams with a mouse-click, testers can bound the generation effort to those scenarios for which they want to regenerate tests. For example, based on a bug triage, PMs and testers may want to invoke an automated impact analysis of the requirements to determine that only certain use cases are affected. Based on this affected set of use cases, testers will regenerate and execute only the selected tests.

With respect to test execution, the IE test team has typically created customized, automated test cases in JavaScript for validating IE functionality. As a result, we continue to accumulate a large number of regression tests.

However, a major benefit of model-driven testing practices is that it is leading us to rethink our entire test development and execution strategy. Going forward, we are strongly encouraging testers to invest additional effort in better structuring their new, and if time permits, legacy test code by developing modular test code - so-called test snippets - for later reuse. Our recommendation to testers has been to refine (generalize or parameterize) individual test steps, so that they are independently callable from an

³ The prerequisite is a tight integration of these two tools, which is described in Section 5.

automatically generated test script created by the test generator and driven using their arguments or the data structures that reference those arguments. We are also strongly encouraging testers to avoid dependencies between tests as the test generator creates tests in random order.

5. Tools

Model-driven development and, in particular, testing practices can only be applied effectively and efficiently when they are developed and deployed in conjunction with an integrated set of tools. For this ongoing effort within IE, we are leveraging two best-of-breed commercial tools from Telelogic [9], namely DOORS - the requirements management tool - and Tau - the UML modeling tool - and tightly integrating them with Microsoft tools for test generation and execution. A similar approach was prototyped by the author and his team at Siemens Corporate Research and was known as TDE/UML [8]. In this section, we describe our implementation strategy.

5.1 DOORS and Tau

The two commercial tools provide us with the flexibility of iteratively capturing, structuring and depicting product behavior and scope around customer scenarios. They support our notion of first developing UML use case models, which are reviewed, and then exporting these models into a set of DOORS requirements documents that can be expanded and filled with details including user interface specifications, non-functional requirements and other quality attributes of the product. The tools also support the team with performing various link management tasks, which are necessary to enable requirements traceability, impact analysis and thus regression testing during bug triage. While the DOORS requirements management tool provides automatic versioning of the textual requirements, the Tau UML modeling tool does not perform use case model versioning nor allow models to be modified by multiple users concurrently. Thus, we are actively working to integrate this tool with our in-house version management software to enable the entire team to work with this tool effectively.

5.2 Test Generation and Execution

The test generation tool being developed by Microsoft Research is the latest incarnation of the SpecExplorer tool [10] whose pedigree lies in model-checking. SpecExplorer2 follows that tradition by systematically exploring the models defined in the UML tool and stored in the meta-model to produce conformance tests that cover all explored model transitions. The output from the test generator is a set of XML-based files that can be transformed for presentation as a set of textual test procedures or executable test scripts based on the eXtensible Stylesheet Language (XSL) style sheet being used. SpecExplorer2 is being implemented in C# and integrated with the commercial Tau UML modeling tool via the latter's Microsoft COM API. In fact, our generator implements a COM server, that is, a COM component waiting for events, such as users invoking the test generator via the user interface of the Tau tool. Similar to other commercial UML modeling tools, Tau provides an extensibility interface for third-party tools that function as add-ins.

The test execution environment being deployed in conjunction with this tool solution is an in-house, JavaScript-based User Interface (UI) Tool. The intent is that conformance tests derived by the test generator from UML models are assembled into executable test scripts corresponding to customer (test) scenarios from a library of pre-packaged and parameterized test code components. These test scripts can then be stored and versioned in the in-house test management system, if necessary, for regression purposes.

6. Related Work

The use of UML for automatically generating test cases has been extensively studied in recent years [11-15]. Many of these papers discuss work related to the automated test generation and execution for individual or subsystems of components and is more applicable to unit and integration testing. Our ongoing efforts align more with those publications that discuss UML for the purposes of system testing [16-18] and those that focus on generating tests for user interface-based systems [19,20]. With respect to UML-based

modeling for system-testing purposes, Fröhlich and Link [18] discuss a method to automatically generate test cases from use cases. In their approach, a use-case textual description is transformed into a UML state chart and then test cases are generated from that model; however, the use of state charts in the context of representing system behavior is debatable - we believe that activity diagrams reflect the user-action–system-response paradigm in a more natural way. Briand and Labiche [17] propose the TOTEM system test methodology, which is based on the refinement of UML use cases into sequence (or collaboration) diagrams. It provides a good alternative approach to modeling with activity diagrams. Bertolino and Gnesi [16] present a methodology to manage the testing process of product lines. The methodology is based on annotation of textual-use cases with test requirements. While very similar in premise to our own strategy, we annotate the activity diagram instead of the textual-use cases.

7. Conclusions and Future Work

In this paper, we have described our ongoing effort to introduce and leverage model-driven testing practices to the team. The goal of this effort is to reduce the overall test cycle through the automatic generation and execution of tests based on behavioral models of IE. We have discussed the motivation and vision for our work as well as describing how we are deploying this methodology within the IE team. We believe that our solution will benefit from the use of COTS tools, such as those from Telelogic, and their integration with the in-house and state-of-the-art Microsoft tools. Together, they will provide a solid basis for helping us address some of the challenges that IE faces in the future.

8. Acknowledgements

I would like to thank Scott Stearns, Director of QA, and Tony Chor, Group Program Manager, for their support with the introduction of the model-driven development processes. I would also like to express my thanks to Wolfram Schulte and Wolfgang Grieskamp from Microsoft Research for their continued collaboration on the topic of automatic test generation.

9. References

1. D. Rosenberg and K. Scott, "Use Case-driven Object Modeling with UML: A Practical Approach", Addison-Wesley, 1999.
2. A. Cockburn, "Writing Effective Use Cases", Addison-Wesley, 2000.
3. J. Hartmann, M. Vieira and A. Ruder, "A UML-based Approach to Validating Product Lines", Proceedings of SPLiT 2004 Workshop, 2004.
4. B. Berenbach, "The Evaluation of Large, Complex UML Analysis and Design Models", International Conference on Software Engineering, 2004.
5. L. Briand, Y. Labiche and L. O'Sullivan, "Impact Analysis and Change Management in UML", International Conference on Software Engineering, 2003.
6. J. Jürgens, "Secure Systems Development in UML", Springer Verlag, 2005.
7. T. Ostrand and M. Balcer, "The category-partition method for specifying and generating functional tests", Communications of the ACM, Vol. 31, No. 6, 1988.
8. J. Hartmann, M. Vieira, H. Foster and A. Ruder, "A UML-based Approach to System Testing", NASA Journal of Innovations in Software Engineering, Vol. 1, No. 1, 2005.
9. [Telelogic @ www.telelogic.com](http://www.telelogic.com)
10. C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, M. Veanes, "Model-based Testing of Object-Oriented Reactive Systems with SpecExplorer", Microsoft Technical Report, 2005.
11. J. Hartmann, C. Imoberdorf, M. Meisinger, "UML-based integration testing", Proceedings of International Symposium of Software Testing and Analysis, 2000.
12. A. Abdurazik and J. Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation", Proceedings of 3rd International Conference on the UML, 2000.
13. A. Cavarra et al., "Using UML for automatic test generation", Proceedings of International Symposium of Software Testing and Analysis, 2002.
14. D. Lugato, C. Bigot and Y. Valot, "Validation and automatic test generation on UML models: the Agatha Approach", Electronic Notes in Theoretical Computer Science, Vol. 66, No. 2, 2002.

15. J. Offutt and A. Abdurazik, “ Generating Test Cases from UML specifications”, Proceedings of 2nd International Conference on UML, 1999.
16. A. Bertolino and S. Gnesi, “Use case-based testing of Product Lines”, Proceedings of the Foundations on Software Engineering, 2003.
17. L. Briand and Y. Labiche, “A UML-based Approach to System Testing”, Journal of Software and Systems Modeling, 2002.
18. P. Fröhlich and J. Link, “Automated Test Case Generation from Dynamic Models”, Proceedings of ECOOP 2000, 2000.
19. A. Beer, S. Mohacsi and C. Stary, “IDTAG: An Open Tool for Automated Testing of Interactive Software”, Proceedgins of COMPSAC’98, 1998.
20. A. Menon, “A Comprehensive Framework for Testing Graphical User Interfaces”, Ph.D. dissertation, Univ. of Pittsburgh, 2001.

Techniques That Inspired Workplace Improvement

George Yamamura

g.yamamura@gte.net

Abstract

Have you found thinking of improvement ideas was easy, but implementing the change was always the difficult part? In many cases, implementing can be difficult because the real, underlying problem was not addressed. Learn about a simple technique that looks beyond lateral thinking to lead you to the core problem and its solution. This paper presents one person's experience in successfully implementing changes and improvements, one after another, within a large organization without encountering the usual resistance.

Biography



George Yamamura has led several software engineering groups in major aerospace firms, including the Boeing Space Transportation System Software Development organization, which was awarded the Software Engineering Institute's Capability Maturity Model Level 5, a measure of excellence in processes and products.

Yamamura has been the subject of articles in the Department of Defense *CrossTalk Journal* and in the *Handbook on Software Quality Assurance*. He has been awarded the AIAA Technical Management Award, the Pacific NW Software Excellence Award and honored with the Lifetime Achievement Award by National Association of Asian American Professionals (NAAAP). He has recently published a book titled, *The 10th Inning – Winning Strategies in Baseball and Business*.

He has a B.S. and M.S. in Aeronautics and Astronautics from the University of Washington and a M.S. in Applied Mathematics from the University of Santa Clara in California.

1 Introduction

Implementing improvements in our work culture is always difficult because of the inherent resistance to change. Seemingly insurmountable problems invariably cause people to look for excuses or give up with helplessness. The problems we often encounter seem so difficult to resolve at first and we tend to chase the wrong problems, problems that are often unsolvable.

This paper is an attempt to elaborate to the conference the methods and process I discovered - a novel approach that overcame the typical resistance to change and chasing after the wrong problem by looking at the situations in a different way. Then the real, underlying causes were uncovered and the appropriate solutions easily became evident and were applied. I first learned of this approach while coaching a youth baseball team years ago and successfully applied it to many business situations resulting in dramatic changes and improvements.

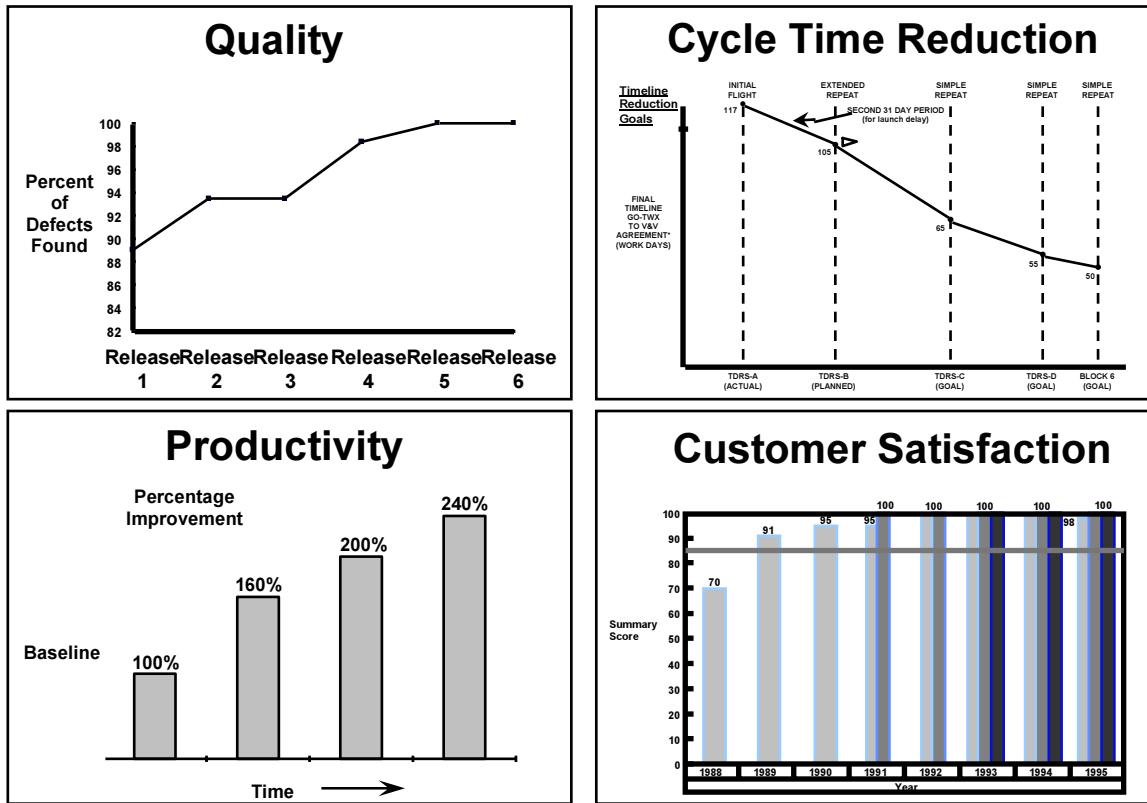
In our business environment, we focused on improving quality first and measured defects remaining after the design phase. As we implemented key design process improvements with each new product released, we found that fewer and fewer defects remained after design until we became nearly 100% effective in eliminating defects before starting the test phase. We then focused on cycle time reduction, cutting the cycle time in half. An 8% program cost saving resulted from the 140% software productivity improvement and we returned \$1.5 million to the customer resulting in their high satisfaction levels. A common term used at the time was “cheaper, faster, better”, but our experience showed that the proper order should be “better, faster, cheaper”.

We implemented process improvements, tracked results and measured employee satisfaction which shot up 40%. A team led by the Software Engineering Institute (SEI) later assessed us to be “operating” at CMM Level 5.

The initial rush of keynote presentations, seminars and workshops highlighted what we had accomplished and the results. The most popular questions were: “how long did it take”, “how many people were involved”, “how much did it cost” and “how did we get management sponsorship”. These were important questions, but this emphasis appeared to be more focused on replicating the successful assessment preparation. Questions about assessment preparation should be secondary to the importance of accomplishing organization improvements.

Now I do want to address the more crucial question of motivation and reveal how I inspired the workforce to accept change, over and over again, towards an improved, smooth-running organization. This is especially significant since I worked for a large company with the usual cultural issues that made change difficult.

Process Improvement Leads to Gains in Quality, Cycle Time, Productivity & Satisfaction



2 The Problem

I first realized how difficult it is to implement improvement changes in our company as a team member working on one of the change activities. We were bogged down trying to define the problem and every possible symptom. It was difficult to agree on all inputs and it was even more difficult to agree on potential solutions and implementation approaches. In the end, the team finally gave up, stopped meeting and no action was taken.

These change activities were called “initiatives” and people had negative impressions of them. They would comment: “oh no, not another initiative” or “we have too many initiatives already” or “this looks like another flavor-of-the-month initiative”. Most of the initiatives faded away with time with little positive results.

It is pertinent here to mention some of the reasons and excuses for failure of these initiatives. I recall remarks such as, “there was nothing I could do” or “I tried my best” or “it will never change in a 100 years”.

These rationales were mentioned enough times to make me believe them and I learned to use some of them myself. Then one day, during an interesting parallel situation, where I could have used the same excuses, I discovered an amazing technique that was so simple and effective, that it had the impact of changing my life.

Suddenly, I was able to implement one change after another before people realized what we were doing. We just implemented the improvements and benefited from the results. The changes were made with no big initiative or fanfare. I had simply asked myself, “What’s going on?”

3 “What’s Going on” Technique

The parallel situation I referred to earlier, where I first discovered this technique, was while coaching a youth baseball team. Coaching a baseball team was much like coaching a workplace team. They both require of the coach or leader - management skills and teamwork – especially decision-making, and they each have a goal.

The usual goal of coaching a baseball team was to teach the players the rules of the game, to have them play their best and to win the game.

I was coerced into my first experience of coaching a youth baseball team when I was also a new leader at work. I was asked to volunteer due to the lack of enough coaches; otherwise my son might not have been able to play on a team that year. As a leader at work, I thought I should be able to manage a youth baseball team. The kids were 5 and 6 years old and this was their first experience playing on a baseball team.

I taught the kids how to play the game the best I could, but we were soundly defeated in all the practice games. I later found out that the kids assigned to my team were all the leftovers, the kids the other coaches did not want.

The first impulse was to rationalize, like we often do at work, complaining that I got a raw deal and there was “nothing I could do” and that I will just “try my best”.

Of course, I was discouraged and shared the predicament with my wife one evening. I painted the picture of “what’s going on” out on the field. My players *could not make decisions* of where to throw the ball, and consequently, threw to the wrong bases allowing the other team players to run around the diamond and score.

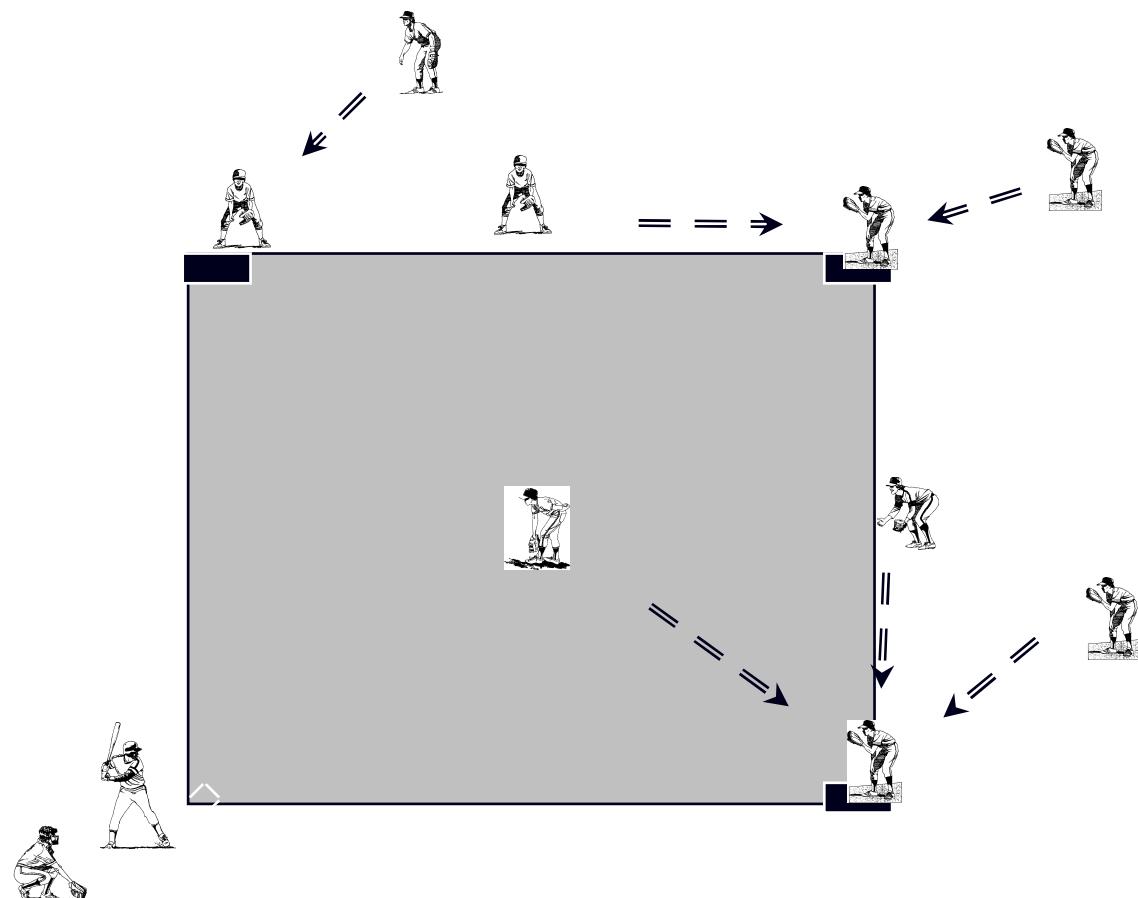
That’s when it hit me! I was describing a new – actual underlying - problem.

I thought the problem was to teach the players the rules of baseball. But by describing what was happening on the field as I saw it, I learned that the players were too young to make good decisions. That was the real problem!

Once I determined the real problem, I looked for a new solution – a new player position strategy. By placing the ballplayers in different positions, I diagramed a formation that minimized the decisions required of each player. The final formation actually required no decision by the players.

The usual position of basemen is between the bases and decisions are required to determine who covers the base and where to throw the ball. By making a small change of placing basemen on the bases, all decision-making was eliminated.

The fielders usually try to throw the ball towards the base near a base runner. But in a panic, they often threw to the wrong base. The confusion was even worse when there were multiple base runners. With my new strategy, I trained all fielders to throw only to the same, closest base every time they had a ball hit to them, eliminating all decisions for them, too.

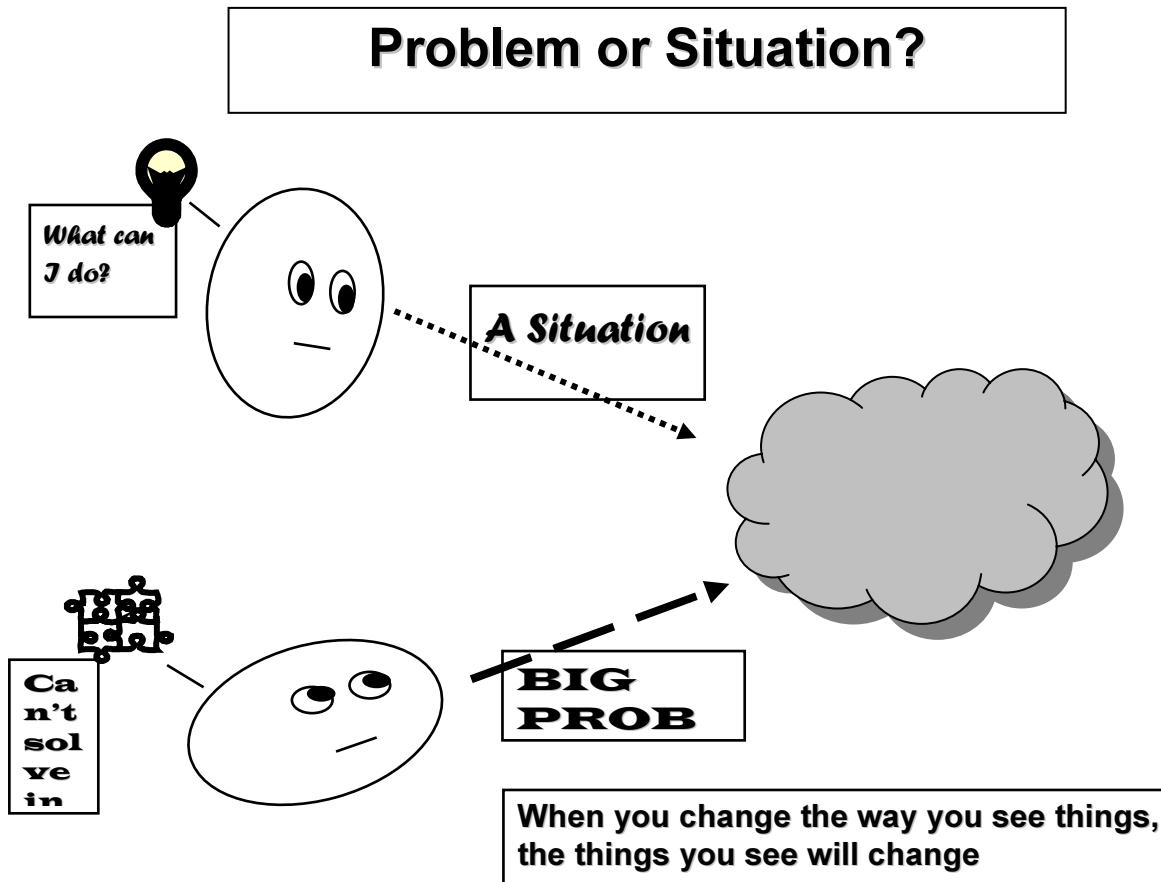


Minimize Decisions

The new strategy made an incredible difference. We not only won the next game, but every game that season and were the league champions, an amazing feat for that team.

I had discovered a very subtle technique to solving difficult problems. Instead of stewing about getting a raw deal or making excuses that nothing could be done, I looked at what was going on and described the situation as I saw it.

The original problem was to teach the kids the rules of baseball, but the real problem was that the kids could not make good decisions. By describing the situation, or asking myself what's happening, I came up with a different problem - one that was solvable and had a rich payback solution.



I learned that whenever I observed a tough – seemingly unsolvable problem, I should describe the situation to myself or narrate it to someone else. This description then uncovers the real underlying problem. And any situation I could describe could be changed, and therefore, always had a solution. Using this technique, we achieved changes and results before anyone realized what was going on, in other words, without a big initiative.

4 Describing the Situation

A very simple work example is the managing of my time. I never seemed to have enough time. I complained of being understaffed, of being given too much work, having to work so much overtime and I was extremely frustrated and unhappy. I put up with this for many years because I had not thought through my time management processes; I just did the best I could.

Then one day, I recalled my trick of asking, “What’s going on” out on the baseball field to determine the real problem. Could that apply here at work? I described the situation: I was so busy at work every day and *I didn’t know where the time went*. That was it; I had described a new problem. I didn’t really know where my time went, so the solution was to track my time for a week. I discovered I spent too much time reading cc’ed e-mail and going to non-productive meetings. I made small adjustments to correct that and recovered a significant amount of time.

Another example resulting in a major business impact for my company involved a multi-million dollar contract. We had a great opportunity for a new business with one of our satisfied customers. They were operating with an older version of software with limited capability and maintainability. We had ideas on increasing the capability, as well as, improving ease of use and maintenance. The customer had funds for an upgrade so the timing seemed right.

Our organization developed a convincing story and presented a glowing presentation to the customer. But to our surprise, the customer rejected the proposal.

We asked ourselves, “Where did we go wrong?”

By now, it had become a routine for me to automatically ask myself, “What’s going on?” Describing what happened, I discovered that we presented a wonderful story on why the customer should upgrade their software now, *but failed to identify that they were not comfortable with changing* (they were actually very impressed with the change we proposed).

That was it! The real problem was that the customer was not comfortable changing. We focused on addressing the customer’s comfort zone instead. We showed how process improvements had reduced defects, shared how the transition would be smooth and that there would be adequate training and support.

When we emphasized our understanding of issues usually encountered in changing to a new system, the customer was impressed with our focus on their fear of change. Subsequently, the customer agreed to the upgrade and approved the contract.

5 Looking at Other Difficult Problems

As we continued to encounter problems at work, we implemented solutions one after another, using my simple technique. Some other examples include engaging the employees and achieving strong participation and buy-in, replacing fire-fighting with process-focus and addressing unmotivated employees. We simply asked ourselves what was really happening, described the situation as we saw it and determined what the real problem was. I realized that each time we did this, the situation described was usually different than the original problem statement. I then concluded that the reason many problems are difficult to solve was because we often try to solve the wrong unsolvable problems.

This paper touches on only a few examples. Many other common situations experienced and additional details may be found in the references.

6 Conclusion

This is one employee's experience in successfully implementing change resulting in significant improvements. I did not try to change the world in big steps. I made small changes, one – after – another, by a process of narrating what I saw going on and changing only that. In the end, the small changes added up to major cultural changes. My point in sharing this is to consider that it may not be change that is difficult, but finding the right change to the right problem and only committing to small units of change at one time without any big initiative or aggrandizement.

The following references give additional pointers and elaboration of the “what’s going on” technique and also how to break down complex problems to simpler solvable narrative questions.

- *The 10th Inning – Winning Strategies in Baseball and Business*, George Yamamura, 2005
- *Handbook of Practical Leadership Techniques*, George Yamamura, 2004
- *Process Improvement Satisfies Employees*, George Yamamura, IEEE Software, September 1999
- *Handbook of Software Quality Assurance*, Gordon Schulmeyer, 1998, Chapter 13 written by G. Wigle and G Yamamura

Dynamics of Exploratory Testing

(In support of a presentation at PNSQC 2006 titled “Exploratory Testing as Competitive Sport.”)

© 2006, Jon and James Bach

Abstract:

Two decades after the term "exploratory testing" was invented, many people don't understand it. Some even call it "monkey testing" -- clicking around the screen with only a primal intelligence, hoping to stumble on to a bug. They decry its efficacy because they say the testing steps aren't precise or repeatable.

And no wonder ... to the untrained eye, exploratory testing may not look very understandable or reproducible, especially in relation to scripted tests like test cases. It may look like the sport of curling, for example, where a person at one end of an ice rink slides a rock to the other end, hoping to hit the opponent's rocks in the process. To the untrained eye, curling may look unskilled, and that's at the root of many misunderstandings of exploratory testing. But it's not a question of how it looks, but how skill is cultivated and used.

In this paper, I'll discuss an emerging set of tactics and skills that when understood, can be used to narrate a tester's exploration, just as sports commentators provide play-by-play and color commentary during curling matches. With this evolving language, testers can explain their own exploration so it can be seen not as "monkey testing", but as the thoughtful, "brain-engaged" style of testing that keeps it such a widely used approach, just as Olympic sports – yes, even curling – are a showcase for all kinds of athletic skill.

Bio:

Jon Bach is Manager for Corporate Intellect and Technical Solutions at Quardev Laboratories – an onshore, outsource test lab in Seattle, Washington.

In his ten-year testing career, Jon has worked from contractor to full-time test manager to consultant for many large companies such as Microsoft and Hewlett-Packard. He has written testing articles and presented testing talks, tutorials, and keynotes at over 40 different venues, both domestically and abroad.

In 2000, Jon and his brother James invented "Session-Based Test Management" – a technique for managing (and measuring) exploratory testing. He currently serves as the VP of Conferences for the Association for Software Testing, speaker chairman for Washington Software Alliance's Quality Assurance SIG, and as a regular guest speaker about agile test techniques for Seattle-area testing forums and university classes.

Exploratory Testing, defined

The difference between scripted testing and exploratory testing is as simple as describing the game “20 Questions”. In this game, one person thinks of an object (an animal, vegetable, or mineral) and another person is allowed 20 yes or no questions to discern what the object might be. If the guesser can’t determine what the object is by the 20th question, they lose the game.

The important aspect of the game is that the guesser asks one question at a time, and an answer is given before the guesser decides what their next question will be.

This is exploratory testing in action. The game would not work if a scripted testing paradigm was used, where the guesser had to submit their 20 questions in advance and could not adapt their questions to each answer.

Unlike scripted testing, exploratory testing *relies* on this adaptation. Like the “20 Questions” guesser, the tester is free to design and execute their next test idea based on the results of the previous test. To test wisely, the tester can draw from anything in their cognition – experience, heuristics, biases, observations, conjectures, notions, and hunches, and on and on -- but culling through all of that cognition, choosing that next test, playing that hunch and applying that experience -- takes skill.

But often, managers treat exploratory testing as if it requires none.

Over 20 years have passed since Cem Kaner (author of the best-selling book "Testing Computer Software") coined the term "exploratory testing".¹ In that time, only a handful of practitioners are considered experts in the approach, even though it's arguably the most widely used testing style in the software testing industry.

Could that be because so many people think they already understand exploratory testing?

Could it be that managers consider it something any untrained tester could do?

Perhaps it's because so many think exploratory testing can't be measured or managed like test cases can. They may associate exploration with randomness, or mindless pounding on the keys, or may have resigned to the fact that, yes, it takes cognition (because even experts have said that exploratory testing involves tester intuition) – but cognition is a mysterious, inexplicable notion that's impossible to teach, so what's the point of trying?

This paper was written to say that there IS a point in trying. It assumes you want to study what it is about exploratory testing that makes it so inexplicable. It assumes you have tried exploratory testing and found some success with its ability as an approach that helps you find bugs in a way that written test cases cannot. Maybe some of us have seen a tester find severe bugs without a test script and have stood back in awe of how they used their cognition. Maybe that tester was us, but we couldn't explain how or what we did.

But mainly, this paper is meant to help you set aside a few minutes to think about why exploratory testing works, why it's such a popular approach, and how you might find words to describe the phenomenon of unscripted, unrehearsed testing.

Exploratory testing is like driving a car -- you need to be able to see and react to changing situations all around your vehicle, not just one category of situation. For example, if you wore a neck brace and could only look to the right, it would impair your driving ability.

Analogies like this are useful to help dispel the notion that exploratory testing can be done by any unskilled tester. But to propel critical discussions of what it takes to be a *good* exploratory tester (just like there are good drivers and bad drivers), there have been focused efforts like James Bach's Heuristic and Exploratory Techniques workshops (WHET)², James Lyndsay's London Exploratory Workshops on Testing (LEWT)³, and most recently, the Exploratory Testing Research Summit (ExTRS)⁴ in February 2006.

In his recent keynote at the First Annual Conference for the Association for Software Testing (June 2006)⁵, Kaner stated a definition of software testing:

“...an empirical technical investigation conducted to provide stakeholders with information about the quality of the product or service under test.”⁶

As he focused on *exploratory testing*, he delivered a definition in one slide consisting of ellipses – perhaps a deliberate attempt to emphasize the key components of the richly cognitive approach:

“Exploratory testing is

*... a style of software testing
... that emphasizes personal freedom and responsibility
... of the tester
... to continually optimize the value of their work
... by treating test-related learning, test design, and test execution
... as mutually supportive activities
... that run in parallel throughout the project.”⁷*

I understood why Kaner would take such care with his definition. As a practitioner and trainer of exploratory testers since 2000⁸, I know the importance of slowing down and emphasizing exploratory testing dynamics. I have even struggled with how to describe it – as did my colleagues at the Exploratory Testing Research Summit.

So, what about a demonstration to help people understand?

For the Software Testing and Review (STAR) Conference in 2005⁹, my brother James (perhaps the most famous exploratory testing expert in our industry) and I were asked to present a demonstration of exploratory testing in action.

In preparing our presentation, we wanted to distill our demo into separate behaviors we could show in a two-hour timeframe. As we sat to prepare our strategy, we were creating the beginnings of a set of skills and tactics to help testers not just learn how to do exploratory testing, but do it well. We were creating a vocabulary – a language to help testers explain it their exploration to stakeholders, just as we had done on many projects.

Our aims were:

- to give testers a language to explain their work to project stakeholders
- to suggest a set of behaviors that can help them be more effective explorers
- to help testers understand how much cognition is involved in something that seems to be so taken for granted
- to dispel the notion that exploratory testing is mysterious and unstructured

That last point is a good place to start.

Is it mysterious?

No more mysterious than understanding the Olympic sport of curling. To understand the game (and ultimately to appreciate it), you need to know what to look for. It's more than just one person throwing a rock down a sheet of ice in an attempt to hit opponent rocks. It is a game of skill and strategy, with its own vocabulary. And hearing the commentators from this past Olympics, I found that I was impressed at how boring a sport could look but how interesting it was once hearing its language described in play-by-play and color commentary.

Consider the following a vocabulary that a play-by-play or color commentator would use if they were seeing you test and describing it to spectators or a list of drills to practice as if testers were in an exploratory testing pre-season training camp. I also hope it will take away the mystery of exploratory testing, and to give it a structure you can understand.

These skills comprise professional exploration of technology:¹⁰

Chartering: Making your own decisions about what you will work on and how you will work. Understanding your client's needs, the problems you must solve, and assuring all stakeholders that your work is on target.

Manipulating: Making and managing contact with the object of your study; configuring and interacting with it. Designing experiments and establishing lab procedures.

Observing: Gathering empirical data about the object of your study; collecting different kinds of data, or data about different aspects of the object. Designing experiments and establishing lab procedures.

Modeling: Composing, describing, and working with mental models of the things you are exploring; identifying dimensions, variables, and dynamics that are relevant.

Conjecturing: Considering possibilities and probabilities. Considering multiple, incompatible explanations that account for the same facts.

Questioning: Identifying missing information, conceiving of questions, and asking questions in a way that elicits the information that you seek.

Recording: Preserving information about your process, progress, and findings; taking notes, categorizing, sorting.

Reporting: Making a credible, professional report of your work to your clients in oral and written form.

Resourcing: Obtaining tools and information to support your effort; exploring sources of such tools and information; getting people to help you.

Refocusing: Managing the scope and depth and nature of your attention; looking at different things, looking for different things, in different ways.

Branching/Backtracking: Allowing yourself to be productively distracted from one course of action in order to explore a new idea that arises in the moment. Identifying opportunities and pursuing them without losing track of the process.

Generating/Elaborating: Working quickly in a manner good enough for the circumstances. Revisiting the solution later to extend, refine, refactor or correct it.

Roughing/Refining: Working quickly in a manner good enough for the circumstances. Revisiting the solution later to extend, elaborate, refactor or correct it.

Alternating: Switching among or contrasting different activities or perspectives so as to create or relieve productive tension and make faster progress.

Exploratory Testing Polarities

To develop ideas or search a complex space quickly yet thoroughly, not only must you look at the world from many points of view and perform many kinds of activities (which may be dualities), but your mind may get sharper from the very act of switching from one kind of activity to another.

Here is a partial list of polarities:

- Warming up vs. cruising (or cooling down)
- Doing vs. describing
- Careful vs. quick
- Data gathering vs. data analysis
- Working with the product vs. reading about the product
- Working with the product vs. working with the developer
- Product vs. project
- Solo work vs. team effort
- Your ideas vs. others' ideas
- Lab conditions vs. field conditions
- Current version vs. old versions
- Feature vs. feature
- Requirement vs. requirement
- Test design vs. execution
- Testing vs. touring
- Individual tests vs. general lab procedures and infrastructure
- Testing vs. resting

Evolving Work Products

Exploratory testing spirals upward toward a complete and professional set of test artifacts. Look for any of the following to be created or refined during an exploratory test session:

Test Ideas: Tests, test cases, test procedures, or fragments thereof.

Testability Ideas: How can the product be made easier to test?

Bugs: Anything about the product that threatens its value.

Risks: Any potential areas of bugginess or types of bug.

Issues: Any questions regarding the test project, or matters to be escalated.

Test Coverage Outline: Aspects of the product we might want to test.

Test Data: Any data developed for use in tests.

Test Tools: Any tools acquired or developed to aid testing.

Test Strategy: The set of ideas that guide our test design.

Test Infrastructure and Lab Procedures: General practices or systems that provide a basis for excellent testing.

Test Estimation: Ideas about what we need and how much time we need.

Test Process Assessment: Our own assessment of the quality of our test process.

Testing Narrative: The story of our testing so far.

Tester: Experience and skill set evolves over the course of the project.

Test Team: The test team gets better, too. Relationships evolve and mature.

Developer Relations: As you test, you also get to know the developer.

Testing Considerations

This is a compressed version of the Satisfice Heuristic Test Strategy model¹¹. It's a set of considerations designed to help you test robustly or evaluate someone else's testing.

Project Environment

Customers: Anyone who is a client of the test project.

Information: Information about the product or project that is needed for testing.

Developer Relations: How you get along with the programmers.

Test Team: Anyone who will perform or support testing.

Equipment & Tools: Hardware, software, or documents required to administer testing.

Schedules: The sequence, duration, and synchronization of project events.

Test Items: The product to be tested.

Deliverables: The observable products of the test project.

Product Elements

Structure: Everything that comprises the physical product.

Functions: Everything that the product does.

Data: Everything that the product processes.

Platform: Everything on which the product depends (and that is outside your project).

Operations: How the product will be used.

Quality Criteria Categories

Capability: Can it perform the required functions?

Reliability: Will it work well and resist failure in all required situations?

Usability: How easy is it for a real user to use the product?

Scalability: How well does the deployment of the product scale up or down?

Performance: How speedy and responsive is it?

Installability: How easily can it be installed onto its target platform?

Compatibility: How well does it work with external components & configurations?

Supportability: How economical will it be to provide support to users of the product?

Testability: How effectively can the product be tested?

Maintainability: How economical is it to build, fix or enhance the product?

Portability: How economical will it be to port or reuse the technology elsewhere?

Localizability: How economical will it be to publish the product in another language?

General Test Techniques

- Function Testing:** Test what it can do.
- Domain Testing:** Divide and conquer the data.
- Stress Testing:** Overwhelm the product.
- Flow Testing:** Do one thing after another.
- Scenario Testing:** Test to a compelling story.
- Claims Testing:** Verify every claim.
- User Testing:** Involve the users.
- Risk Testing:** Imagine a problem, then find it.
- Automatic Testing:** Write a program to generate and run a zillion tests.

Conclusion

If testers want to win respect and credibility as they perform exploratory testing, they have to be able to describe their work to a stakeholder’s satisfaction. They have to be able to use a language to define and report their “off-road” testing. But they also have to be committed to using this language as a foundation to cultivate their skill as effective explorers.

Perhaps if exploratory testing was viewed as a spectator sport that could be practiced like Spring Training to prepare for the Major League Baseball season or Pre-Season Camp to prepare for the National Football League season, it could be commented on (literally) as it unfolded and stakeholders would see exploration not as unskilled “monkey testing”, but “*... a style of software testing that emphasizes personal freedom and responsibility of the tester to continually optimize the value of their work by treating test-related learning, test design, and test execution as mutually supportive activities that run in parallel throughout the project.*”¹²

Bibliography

¹ Cem Kaner stated his definition of exploratory testing in a keynote speech at the first annual conference of the Association for Software Testing in June 2006. See <http://www.associationforsoftwaretesting.org>

² <http://serl.cs.colorado.edu/~serl/seworld/database/5899.html>

³ <http://www.workroom-productions.com/LEWT.html>

⁴ The Exploratory Testing Research Summit was a meeting of speakers and thinkers on exploratory testing in Melbourne, Florida on January 30, 2006.

⁵ See <http://www.associationforsoftwaretesting.org>

⁶ Kaner, keynote, CAST 2006

⁷ Kaner, keynote, CAST 2006

⁸ Presentation: “Measuring Ad Hoc Testing”, Software Testing Analysis and Review Conference (West) 2000

⁹ Presentation: “Testing Outside the Bachs”, Software Testing Analysis and Review Conference (West) 2005, <http://www.sqe.com/archive/sw2005/sessions2.html?from=glance&dg=date&dgd=thu&ac=t4#T4>

¹⁰ From a handout titled “Exploratory Testing Dynamics”, first published and presented at STARWest 2005 by James and Jonathan Bach

¹¹ See <http://www.satisfice.com/tools/satisfice-tsm-4p.pdf>

¹² Kaner, keynote, CAST 2006

Myths and Strategies of Defect Causal Analysis

*David N. Card
Q-Labs, Inc.
dca@q-labs.com*

Biography

David N. Card is a fellow of Q-Labs, a subsidiary of Det Norske Veritas. Previous employers include the Software Productivity Consortium, Computer Sciences Corporation, Lockheed Martin, and Litton Bionetics. He spent one year as a Resident Affiliate at the Software Engineering Institute and seven years as a member of the NASA Software Engineering Laboratory research team. Mr. Card is the author of *Measuring Software Design Quality* (Prentice Hall, 1990), co-author of *Practical Software Measurement* (Addison Wesley, 2002), and co-editor ISO/IEC Standard 15939: *Software Measurement Process* (International Organization for Standardization, 2002). Mr. Card also serves as Editor-in-Chief of the *Journal of Systems and Software*. He is a Senior Member of the American Society for Quality.

Abstract

The popular process improvement approaches (e.g., Six Sigma, CMMI, and Lean) all incorporate causal analysis activities. While the techniques used in causal analysis are well known, the concept of causality itself often is misunderstood and misapplied. The article explores the common misunderstandings and suggests some strategies for applying causal analysis more effectively. It does not provide a tutorial on any specific causal analysis technique.

Many different processes, tools, and techniques (e.g., Failure Mode Effects Analysis, Ishikawa diagrams, Pareto charts) have been developed for defect causal analysis. All of them have proven to be successful in some situations. Organizations often invest large amounts in the software and training needed to deploy them. While application of these techniques helps gain insight into the sources of problems, a checklist implementation of the techniques alone is not sufficient to ensure accurate identification and effective resolution of “deep” problems. Gaining a true understanding of concepts underlying causality and developing a strategy for applying causal analysis help to maximize the benefit of an organization’s investment in the tools and techniques of causal analysis.

Introduction

Causal analysis is a major component of modern process improvement approaches, e.g., the CMMI, Six Sigma, and Lean. The implementation of effective causal analysis methods has become increasingly important as more software organizations transition to higher levels of process maturity where causal analysis is a required as well as appropriate behavior. Petrosky [1] argues that failure, and learning from failure, is an essential part of engineering discipline. He describes how the investigation of some spectacular failures has led to improvements in scientific knowledge and engineering practice. However, software engineering organizations often are reluctant to acknowledge failure and often don’t act effectively to investigate it.

While many things can go wrong in the implementation of any new tool or technique, my experience suggests that four myths about the nature of causal analysis are common impediments to its success in software engineering:

- Causality is intuitive, so it needs no explanation
- Causal analysis can be done effectively by external tiger teams or graybeards
- Causal analysis is for “mature” organizations only
- Small improvements don’t matter

Resolving these misconceptions and developing a strategy to minimize their effects helps to maximize the benefits of this simple, but important learning activity.

While causal analysis of some form can be applied to almost any type of anomaly, defects are the most common subject of causal analysis. Thus, this article will focus on defect causal analysis, although these issues apply to investigations of other types of anomalies, as well. Before discussing the myths, let’s review some basic concepts.

Concept of Causal Analysis

Causal analysis focuses on understanding cause-effect relationships. A causal system is an interacting set of events and conditions that produces recognizable consequences. Causal analysis is the systematic investigation of a causal system in order to identify actions that influence a causal system, usually in order to minimize undesirable consequences. Causal analysis may sometimes be referred to as root cause analysis or defect prevention.

Many good examples of causal analysis efforts in software engineering have been published, e.g., [2], [3], [4], [5], and [6]. However, these efforts have adopted different terminology and approaches. (Card [11] proposes a unifying set of terms and concepts for describing causal analysis.) The differences between the analysis procedures obscure the commonality in the subject matter to which the procedures are applied. Further complicating the situation are apparent differences in the notion of causal analysis defined in the CMM [7] and CMMI [8], see Card [11]. This cloud of terminology creates an environment in which myths can thrive.

Myth 1: Causality is Intuitive

Causal analysis focuses on understanding cause-effect relationships. Searching for the cause of a problem is a common human endeavor that wouldn't seem to require much formalism. However, causal investigations often go wrong from the beginning because the investigators don't recognize what truly constitutes a "cause". Three conditions must be established in order to demonstrate a causal relationship:

- First, there must be a correlation or association between the hypothesized cause and effect
- Second, the cause must precede the effect in time
- Third, the mechanism linking the cause to the effect must be identified

The first condition implies that when the cause occurs, the effect is also likely to be observed. Often, this is demonstrated through statistical correlation and regression.

While the second condition seems obvious, a common mistake in the practice of causal analysis is to suppose that a cause-effect relationship exists between factors that occur simultaneously. This is an over-interpretation of correlational analysis. Consider the situation in which the time spent by reviewers preparing (reviewing) and the number of defects found while reviewing are recorded from peer reviews. (A separate meeting may be held to consolidate findings from multiple reviewers.) Figure 1 shows a scatter diagram of these two measures of peer review performance. These two variables frequently demonstrate significant correlations, thus satisfying Condition 1 (above). This diagram and a correlation coefficient computed from the data often are taken as evidence of a causal relationship between review and detection.

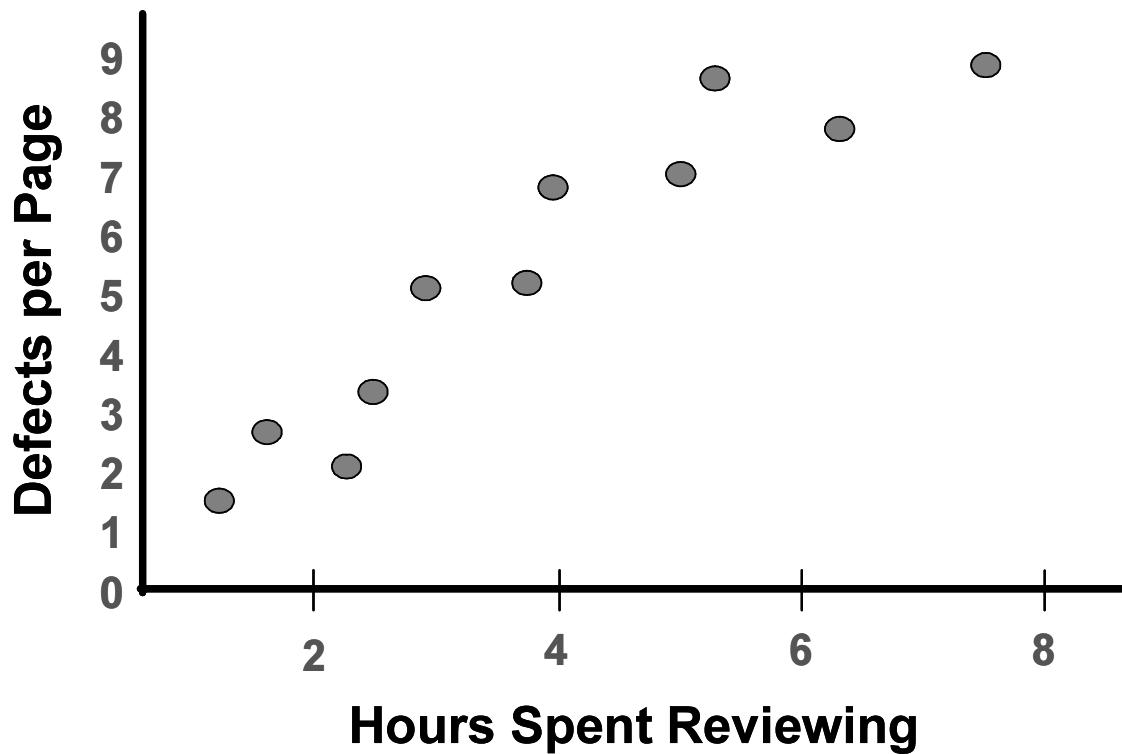


Figure 1. Example of Correlation Between Variables

However, most defects are discovered *during* the individual review time. A few more may be identified during a consolidation meeting. Both meters (review time and defects) are running simultaneously – one does not occur before the other, thus Condition 2 (above) is not satisfied. There is a necessary relationship between review time and defect detection, but not a causal one. The difference is important.

Defects cannot be found unless review time is spent – that's necessary. However, it is not necessarily true that simply directing reviewers to spend more time will cause more defects to be found. If the reviewer has already read the material once, reading it again (the same way) isn't likely to produce new findings. Some improvement to the reading technique is needed. If the reviewer didn't have time to finish reading the material, then a mandate to spend more time is likely to be ineffective, unless something is done to provide time in the reviewer's schedule. In each case a focus on increasing review time short-circuits the causal analysis and leads to an ineffective action. Instead, this correlation should suggest that some other factor that affects both review time and defect detection should be sought.

Issuing a mandate (as a corrective action) to spend more time reviewing may result in more time being charged to reviews (to show compliance), but it isn't likely to increase the defects detected. The underlying cause of low review time and defect detection may be a lack of understanding of how to prepare, schedule pressure, or other factors that affect both measures. That underlying cause must be addressed to increase both the review time and defect detection. Recognition of the correlational relationship helps to narrow the set of potential causes to things that affect both measures. Simply encouraging reviewers to spend more time reviewing is only likely to improve defect detection if the underlying problem is laziness or lack of concern for quality. (That is often management's perception.) However, if the reviewers are making a good faith effort to follow the defined process then this encouragement isn't helpful.

Some of the responsibility for this kind of misinterpretation of correlational relationships can be attributed to statisticians. The horizontal and vertical axes of Figure 1 are typically referred to as the "independent" and "dependent" variables

respectively. While these terms are simple labels, not intended to imply a causal relationship, they are often misunderstood. Applying the three criteria for causality helps to ensure that the causal analysis team probes beyond the obvious correlation.

Because causal analysis is perceived to be “intuitive”, many organizations do not provide meaningful training in it to their staff. Even Six Sigma training programs, which often devote a lot of time to causal analysis techniques, typically do not provide an explicit definition of causality such as the three conditions for causality discussed above. In the course of my work with clients I have reviewed the content of several Six Sigma Black Belt training programs. None of those I examined provided a definition of causality or of a causal system. Try to find one in your training!

One of the consequences of a poor understanding of the nature of causality is that causal analysis sessions become superficial exercises that don’t look deeply enough to find the important causes and potential actions that offer real leverage in changing performance. This reduces the cost-benefit of the investment in causal analysis expected of mature software organizations. Causal analysis training should address the underlying concepts, not just the forms and checklists that make up the techniques.

Myth 2: Causal Analysis by “Tiger Team”

Many organizations implement causal analysis by assigning “tiger teams” or “graybeards” drawn from outside the process or project to investigate the anomalies. The idea is that smart and experienced staff can solve problems created by less experienced and skilled staff. This reflects an “audit” mentality. The assumption is that problems occur because the staff fails to adopt correct behaviors, not because the staff is working in an ambiguous and changing environment.

It is informative to look at the context in which the Lean approach positions causal analysis in manufacturing situations. The Japanese term “gemba” may be translated descriptively as, “the scene of the crime”. The idea is that causal analysis should take place in the actual location (gemba) where the defect occurred. There may be factors in the environment, for example, that contribute to the problem that might not be apparent in a conference room far away from the factory floor. The most effective causal analysis will consider all influences on the problem.

The software development process is an intellectual process. It takes place in people’s heads. The people executing the software process need to be involved in the causal analysis process. The method recommended by Card [6] and others with practical experience put the responsibility for defect causal analysis on the software developers or maintainers who contribute to the mistakes. They are the best qualified to identify what went wrong and how to prevent it.

Some Six Sigma implementations exacerbate this problem by systematically assigning the responsibility for causal analysis to “black belts” who may have little experience with the specific process under study or the defects being investigated. No amount of training can substitute for the insight obtained by actually being involved in the situation.

Myth 3: Causal Analysis for High Maturity Only

The CMMI assigns Causal Analysis and Resolution to Maturity Level 5. Similarly, its predecessor, the CMM assigned Defect Prevention to Maturity Level 5. This gives the impression that only mature organizations can benefit from formal causal analysis. Six Sigma programs often embed causal analysis in a curriculum of much more challenging statistical techniques, discouraging the independent application of simple defect causal analysis techniques.

Causal analysis can be applied effectively to any well-defined process. This doesn’t require the entire process to be well-defined, only the parts to which causal analysis are applied. Moreover, the process definition needs to be complete, but doesn’t need to satisfy any specific CMM/CMMI maturity or capability level. Dangerfield, et al. [3] describe the successful application of causal analysis within a CMM Level 3 project. Card [6] describes the application of Defect Causal Analysis to an “independent” configuration management organization, with little software development activity.

Myth 4: Small Improvements Don’t Matter

Human nature tends to favor revolutionary and dramatic change over incremental and sustained improvement. Many of the recommendations resulting from causal analysis teams are small changes that improve communications and processes in small ways. However, these small changes can have dramatic impacts over the long-term.

Consider the real example (described in [3]) of a project that identified “inconsistent use of the computing environment” as a systematic problem and acted to eliminate it by standardizing on naming, security, and interface conventions. These actions “cleaned up” some aspects of the project’s process. Figure 2 shows the reduction in defect rate from Build 1 to Build 2 as a result of these actions. Clearly, these small changes had a big impact!

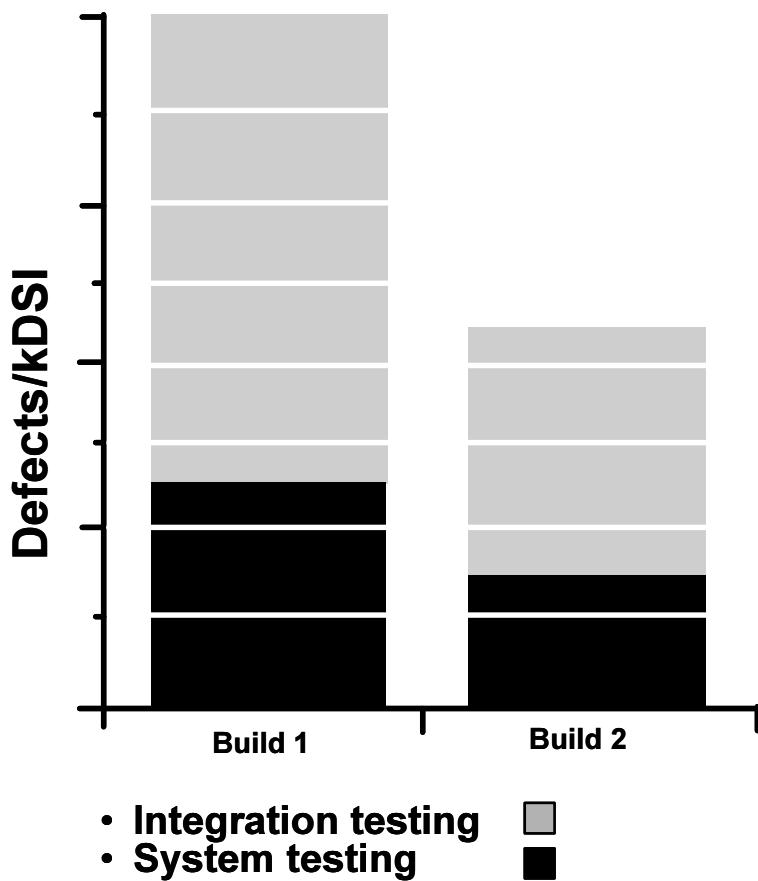


Figure 2. Example Effect of Incremental Improvement (from [3])

While many organizations might prefer to take more dramatic actions (e.g., new technology and processes) with a correspondingly larger investment, substantial improvements often can be obtained with incremental changes to current processes.

Strategies for Causal Analysis

Often, the causal analysis process and techniques are taught to staff, but little guidance is provided on when and where to apply them. Consequently, the application of causal analysis becomes ad-hoc. Moreover, management often assumes that once people are trained, they will somehow identify an important problem and work the causal analysis task into their schedule. A causal analysis strategy should define when and where causal analysis should be performed. It is based on an understanding of the organization’s process improvement objectives as well as its current performance levels. The causal analysis strategy helps to ensure that resources are applied systematically to important problems within the organization.

A causal analysis strategy should address the following elements of the causal analysis program:

- Training – explain the concepts, techniques, and context for causal analysis. Make the training widely available. Anyone in the organization could be a candidate for some causal team. Training should include the following:
 - Definition of causality and causal systems.

- Causal analysis techniques and tools. Limit training “for the masses” to the few most appropriate techniques
 - Description of the organizational context for planning and performing causal analysis, e.g., the strategy
- Focus – identification of key problem areas to be worked. Of course, some causal analysis may be triggered by “special causes”, but don’t just wait for a “black belt” to volunteer to work on an important and persistent problem. Establishing a focus requires the following:
 - Identify the production activities that are the greatest sources of defects
 - Identify the testing activities that are least effective
 - Establish causal analysis teams for the activities that represent the greatest improvement opportunity
 - Put effective data collection mechanisms in place for these activities
 - Schedule causal analysis of these activities at the appropriate point in the project life cycle
 - Allocate resources for performing causal analysis
- Action – implement the recommendations of the causal analysis teams. The benefits of causal analysis are lost without timely action. Ensuring that action is taken requires the following:
 - Establish an action team that includes management and technical experts
 - Schedule regular reviews of causal analysis activities and results
 - Allocate resources to implementing causal analysis team recommendations
- Communication – keep staff informed about the status of causal analysis activities and the lessons learned from the investigations.
 - If causal analysis team members don’t realize that their recommendations are being acted upon, then they lose interest in the process
 - If the larger staff isn’t informed of the outcome, they can’t learn from the lessons accumulated

Many of these elements are addressed in the Defection Prevention process area of the CMM, but have been dropped from Causal Analysis and Resolution in the CMMI (see Card [11]). Unless these four elements of strategy are addressed, the causal analysis program will remain ad-hoc and perform fitfully.

Summary

Defect causal analysis is becoming ever more common in the software industry as process maturity increases and new forces, such as Six Sigma and Lean, focus increasing attention on quality improvement. The four myths previously discussed often impede the effective implementation of a causal analysis process. While causal analysis seems “obvious”, meaningful training and thoughtful planning are needed to ensure its successful implementation. This article has attempted to identify some of the major problems and suggest solutions to them.

References

- [1] Petrosky, H., *To Engineer is Human: The Role of Failure in Successful Design*, St. Martin’s Press, 1985
- [2] Mays, R., et al., Experiences with Defect Prevention, *IBM Systems Journal*, January 1990
- [3] Dangerfield, O., et al., Defect Causal Analysis - A Report from the Field, *ASQC International Conference on Software Quality*, October 1992
- [4] Yu, W., A Software Fault Prevention Approach in Coding and Root Cause Analysis, *Bell Labs Technical Journal*, April 1998
- [5] Leszak, M., et al., Classification and Evaluation of Defects in a Project Perspective, *Journal of Systems and Software*, April 2002
- [6] Card, D., Learning from Our Mistakes with Defect Causal Analysis, *IEEE Software*, January 1998
- [7] Paulk, Mark, et al., *Capability Maturity Model*, Addison Wesley, 1994
- [8] CMMI Development Team, *Capability Maturity Model – Integrated, Version 1.1*, Software Engineering Institute, 2001
- [9] Ishikawa, K., *Guide to Quality Control*, Asian Productivity Organization Press, 1986
- [10] Harry, M. and R. Schroeder, *Six Sigma*, Doubleday, 2000
- [11] Card, D., Understanding Causal Systems, *Crosstalk*, October 2004

Proceedings Order Form

Pacific Northwest Software Quality Conference

Proceedings are available for the following years. Circle year for the Proceedings that you would like to order.

2006 2005 2004 2003 2002 2001 2000

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 _____

Mailing Address _____

City _____ State _____ Zip _____

Phone _____

Using the Electronic Proceedings

Once again, PNSQC is proud to produce the Proceedings in PDF format. Most of the papers from the printed Proceedings are included as well as some slide presentations. We hope that you will find this conference reference material useful.

Download PDF – 2005, 2004, 2003, 2002, 2001, 2000, 1999, 1998

The 2006 Proceedings will be posted on our website in November 2006.

Copyright

You can print articles or slides for your own private use, but remember they are copyright to the original author. You cannot redistribute the Acrobat files or their printed copies. If you need extra copies of the printed or electronic Proceedings, please contact Pacific Agenda at tmoore@europa.com.

