Using the Electronic Proceedings

Once again, PNSQC is proud to announce our electronic Proceedings on CD. On the CD, you will find most of the papers from the printed Proceedings, plus the slides from someof the presentations. We hope that you will enjoy this addition to the Conference. If you have any suggestions for improving the electronic Proceedings, please visit http://www.pnsqc.org/hot/cdrom.htm to give us feedback, or send email to cdrom@pnsqc.org.

Adobe Acrobat Reader

The electronic Proceedings are in Adobe Acrobat format. If you do not currently have Acrobat Reader 4 installed, you can download it from Adobe's web site: http://www.adobe.com/acrobat.

Copyright

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.
1986, 1987, 1989, 1992, 1995, 1999, 2000, 2001

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_____

TWENTY-FIRST ANNUAL PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

OCTOBER 14 - 15, 2003

Oregon Convention Center
Portland, Oregon

TABLE OF CONTENTS

Management Track – October 14

Testing Track – October 15

Process Track – October 15

Management Track – October 15

Welcome to the 2003 Pacific Northwest Software Quality Conference.

With the help of our many dedicated members, we have been promoting good software practices for over 20 years. Every year we gain new members and new ideas, which are vital to keep us going for another 20 years.

I cannot begin to tell you how important our cause is in the current global environment. Software touches or controls almost everything. Software development is being globally distributed. The systems are getting both bigger and more complicated. The only way to control this evolution is with good software quality standards and processes.

It is you, the people who Design, Code, Test, QA and Manage software that have the understanding and knowledge that we all need. I have the utmost respect and admiration for those of you who are willing to present at our conference and share what you know. I learn more at conferences than I do on the job or in school. I get to learn from your experience. I can think of many instances where I used an idea that I got at PNSQC and saved myself a lot of work. I would like to encourage anyone who has not presented, to give it a try. We would love to hear what you have to share at next year's conference. I would also like to thank all of you who have taken the time to share with us this year. I am sure that I will learn a lot.

As you experience this year's conference, please take a few minutes to recognize how much work goes into planning and executing PNSQC. This is a non-profit organization that relies primarily on volunteers. We typically meet at some very odd hours to accommodate the fact that the volunteers are doing this in addition to their work and family lives. Thank you Board of Directors and thank you Committee Chairs and members. And very importantly, thanks to Terri Moore of Pacific Agenda who keeps us all in line. This (very honestly) could not have happened without all of you.

Enjoy the conference! Welcome new members and hello to existing members. Thank you to presenters and volunteers. If you have comments, questions or comments, please contact us through www.pnsqc.org.

Laura Anneker
PNSQC President and Conference Chair

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Laura Anneker – PNSQC President and Chair, Workshop Co-Chair
Symantec

Paul Dittman – PNSQC Vice President, Program Co-Chair

Richard Vireday – PNSQC Secretary
Intel Corporation

Debra Schratz – PNSQC Secretary
Intel Corporation

Cindy Oubre – PNSQC Treasurer, Exhibits Co-Chair, Volunteer Coordinator

Sue Bartlett - 2003 Keynote Chair, Program Co-Chair
System Harmony, Inc.

David Butt – Program Co-Chair

Rick Clements – Publicity Co-Chair
Cypress Semiconductor

Jim Fordenwalt, Workshop Co-Chair
Hewlett-Packard Company

Shauna Gonzales - Birds of a Feather
Inspiration Software

Bhushan Gupta - Strategic Planning
Hewlett-Packard Company

Randy King – Communications Chair

Ganesh Prabhala - PNSQC Software Excellence Award
Intel Corporation

Doug Reynolds – Exhibits Co-Chair
Tektronix, Inc.

Patt Thomasson – Publicity Co-Chair
SAS Institute

CONFERENCE PLANNING COMMITTEE

Hilly Alexander
Comsys

Pieter Bothman
True North Systems Consulting

Kit Bradley
PSC Inc.

David Butt
Oregon Health Science University

Manny Gatlin
Timberline Software

Cynthia Gens

Regina Grazulis
ADP, Inc.

Warren Harrison
Portland State University

Kathy Iberle
Hewlett-Packard Company

Mark Johnson
North West Evaluation Association

Patricia Medvick

Howard Mercier
Epicor Software Corporation

Jonathan Morris
Bellwether, Inc.

Elizabeth Ness
Hewlett-Packard Company

PS Rao
Intel Corporation

Jean Richardson
BJR Communication, Inc.

Ian Savage

Eric Schnellman
JanEric Systems

Erik Simmons
Intel Corporation

Wolfgang Strigel
Software Productivity Centre

Ruku Tekchandani
Intel Corporation

Scott Whitmire
Odyssey Software & Consulting, Inc.

Mark Wiley

How Many Light Bulbs Does It Take to Change a Tester?

Cem Kaner

Twenty-plus years ago, we developed a model for the software testing effort. It involved several "best practices," such as these:

- the purpose of testing is to find bugs;
- the test group works independently of the programming group;
- tests are designed without knowledge of the underlying code;
- automated tests are developed at the user interface level, by non-programmers;
- tests are designed early in development;
- tests are designed to be reused time and time again, as regression tests;
- testers should design the build verification tests, even the ones to be run by programmers;
- testers should assume that the programmers did a light job of testing and so should extensively cover the basics (such as boundary cases for every field);
- the pool of tests should cover every line and branch in the program, or perhaps every basis path;
- manual tests are documented in great procedural detail so that they can be handed down to less experienced or less skilled testers;
- there should be at least one thoroughly documented test for every requirement item or specification item;
- test cases should be based on documented characteristics of the program, for example on the requirements documents or the specifications;
- test cases should be documented independently, ideally stored in a test case management system that describes the pre-conditions, procedural details, post-conditions, and basis (such as trace to requirements) of each individual test case;
- failures should be reported into a bug tracking system;
- the test group can block release if product quality is too low;
- a count of the number of defects missed, or a ratio of defects missed to defects found, is a good measure of the effectiveness of the test group.

Each of these is sensible under some circumstances, but there is nothing magical about any of them. As our field matures, we should be challenging the applicability of all of them. Many of these practices are more likely to block quality-improving practices than to move them forward.

Cem Kaner, J.D., Ph.D., is Professor of Software Engineering at the Florida Institute of Technology. His primary test-related interests are in developing curricular materials for software testing, integrating good software testing practices with agile development, and forming a professional society for software testing. Before joining Florida Tech, Dr. Kaner worked in Silicon Valley for 17 years, doing and managing programming, user interface design, testing, and user documentation.

Dr. Kaner is the senior author of LESSONS LEARNED IN SOFTWARE TESTING (with James Bach and Bret Pettichord) and TESTING COMPUTER SOFTWARE (2nd Edition) (with Jack Falk and Hung Quoc Nguyen) and of BAD SOFTWARE: WHAT TO DO WHEN SOFTWARE FAILS (with David Pels).

Dr. Kaner is also an attorney whose practice is focused on the law of software quality. Dr. Kaner holds a B.A. in Arts & Sciences (Math, Philosophy), a Ph.D. in Experimental Psychology (Human Perception & Performance: Psychophysics), and a J.D. (law degree).
www.kaner.com, www.badsoftware.com

Agile Testing: A Year in Review

Brian Marick

Agile software development has become quite trendy since the publication of the Manifesto for Agile Software Development (www.agilemanifesto.org) in early 2001. Like most trends, there's been both hype and substance. What there wasn't, at least at the beginning, was much about testing, at least the kind of testing commonly practiced by independent product test groups.

That began to change around the middle of 2002, and the change has been accelerating since. This talk will describe the state of the practice of agile testing as of October 2003. What will those who now call themselves testers do on agile projects? What are their roles? What are their tasks? What are their skills? What are their tools?

Change is in the offing. It is the aim of this talk to help you understand how to use that change to make your job better: more productive and more enjoyable.

Brian Marick has been a tester and programmer since 1981 and a testing consultant since 1992. He specializes in programmer testing, test design techniques, and context-driven test strategies. He is the author of The Craft of Software Testing (Prentice-Hall, 1995), a coauthor of the Manifesto for Agile Software Development, and a technical editor for Software Testing and Quality Engineering Magazine. Many of his writings can be found at www.testing.com. He can be reached as marick@testing.com

Agile Testing:
A Year in Review

Brian Marick
marick@testing.com

What Are Agile Projects?

- An attitude toward change
- An attitude toward software
- Some attitudes toward people

The Manifesto for Agile Software Development, www.agilemanifesto.org

Changing Requirements Are Swell

Oh! Let's add this!

Now that I see my feature, I don't like it.

Sure!

How should we change it?

Software Can Be Soft

- **Programs can become better, cleaner, and more capable**
- **They become changeable by being successfully changed**
  - *not* mainly by planning for change
- **Frequent new requirements "train" the code and the coders**

"Agile methods, the Emersonian worldview, and the dance of agency"
www.visibleworkings.com/papers/agile-methods-and-emerson.html

People

- **Written documentation is a poor substitute for continuous conversation**
- **Generalists trump specialists**
- **Teams can self-organize**
- **Trust**
  - "… if you ask for help, someone has to help you" - Lisa Crispin

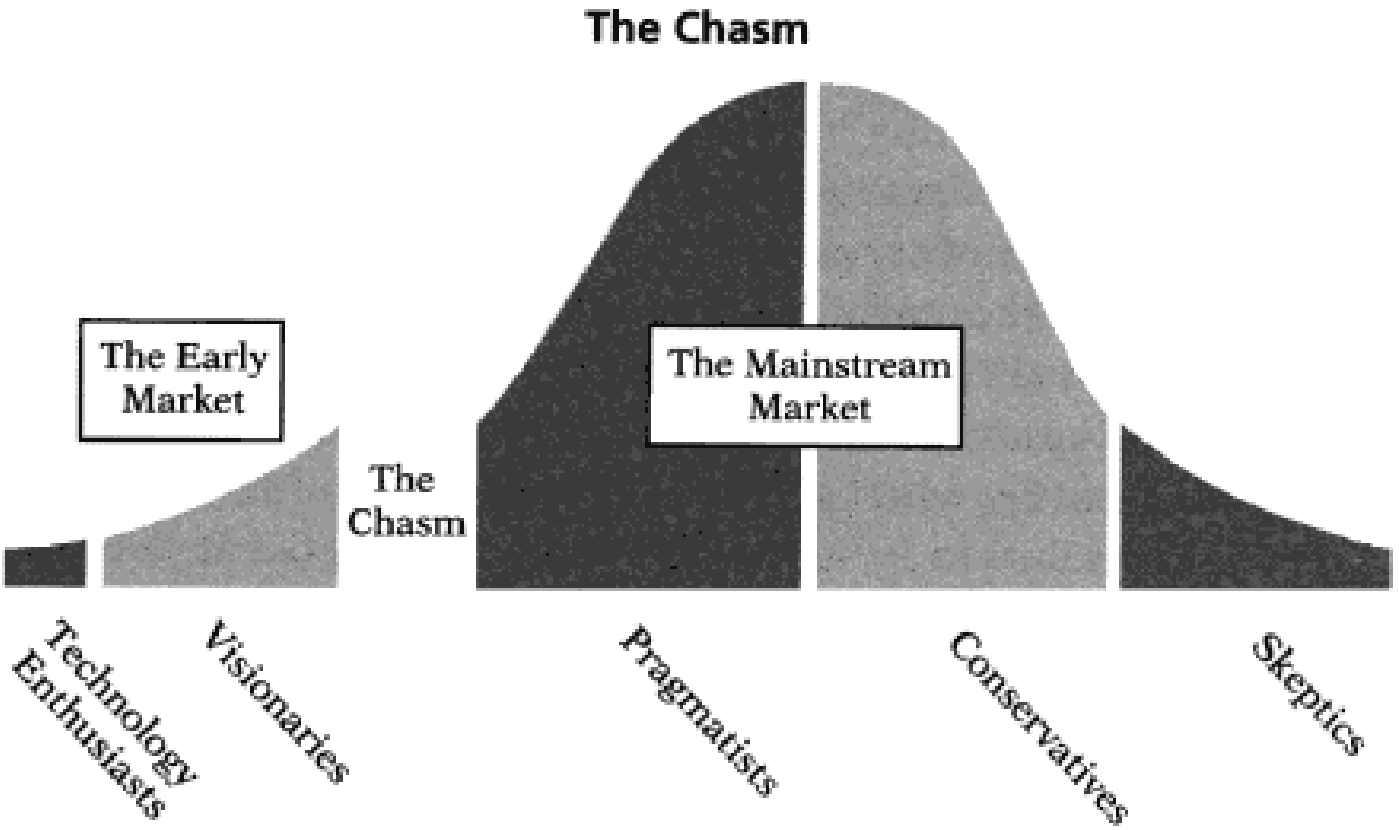
The Import

These attitudes toward change, software, and people are the context for testing in Agile projects

Kaner, Bach, and Pettichord, *Lessons Learned in Software Testing*
www.context-driven-testing.com/wiki/scribble.cgi

The Import

Four Types of Testing



Geoffrey Moore, *Crossing the Chasm*, p. 19

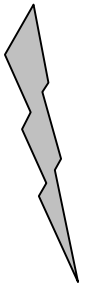
Programmer Testing

I need an object that records each time segment

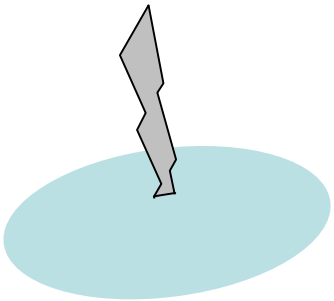
Test-driven design
Unit testing

Programmer Testing

The test comes first

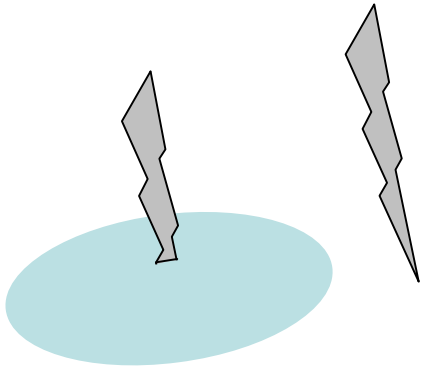


Programmer Testing



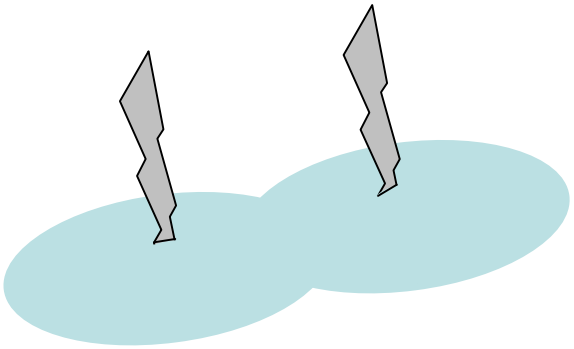
Code is written to pass the test

Programmer Testing



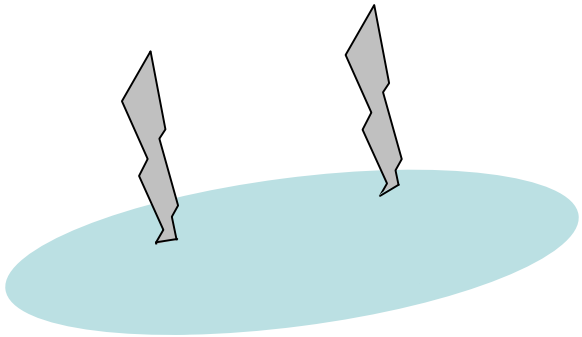
Not done?
Write another test…

Programmer Testing



And the code to pass it.
All earlier tests continue to pass

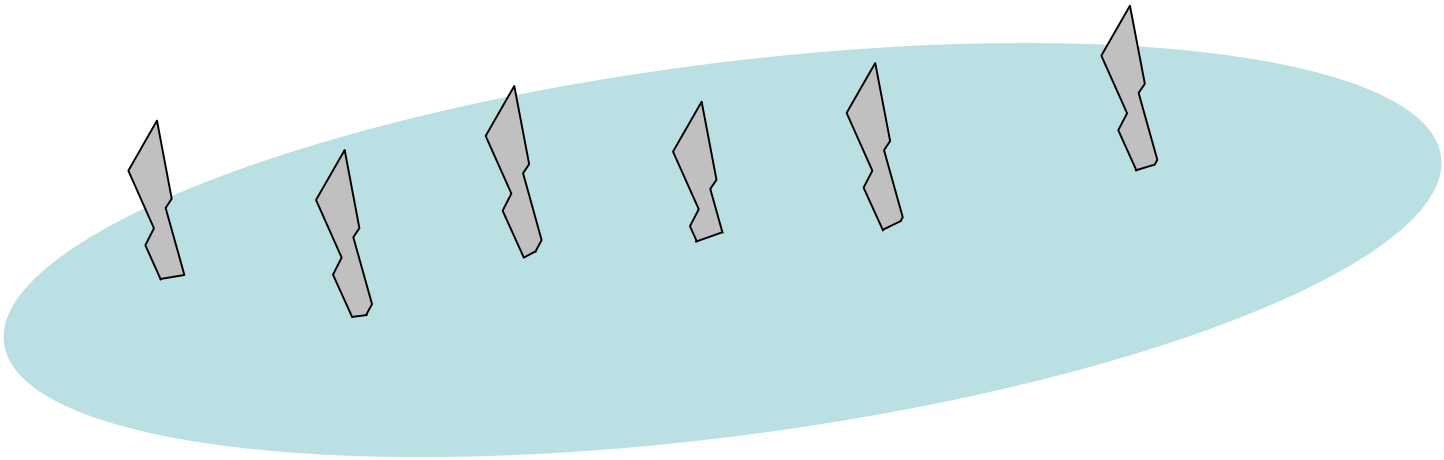
Programmer Testing



Code awkward? Fix it now.
Tests continue to pass
(Refactoring)

Martin Fowler, *Refactoring*

Programmer Testing

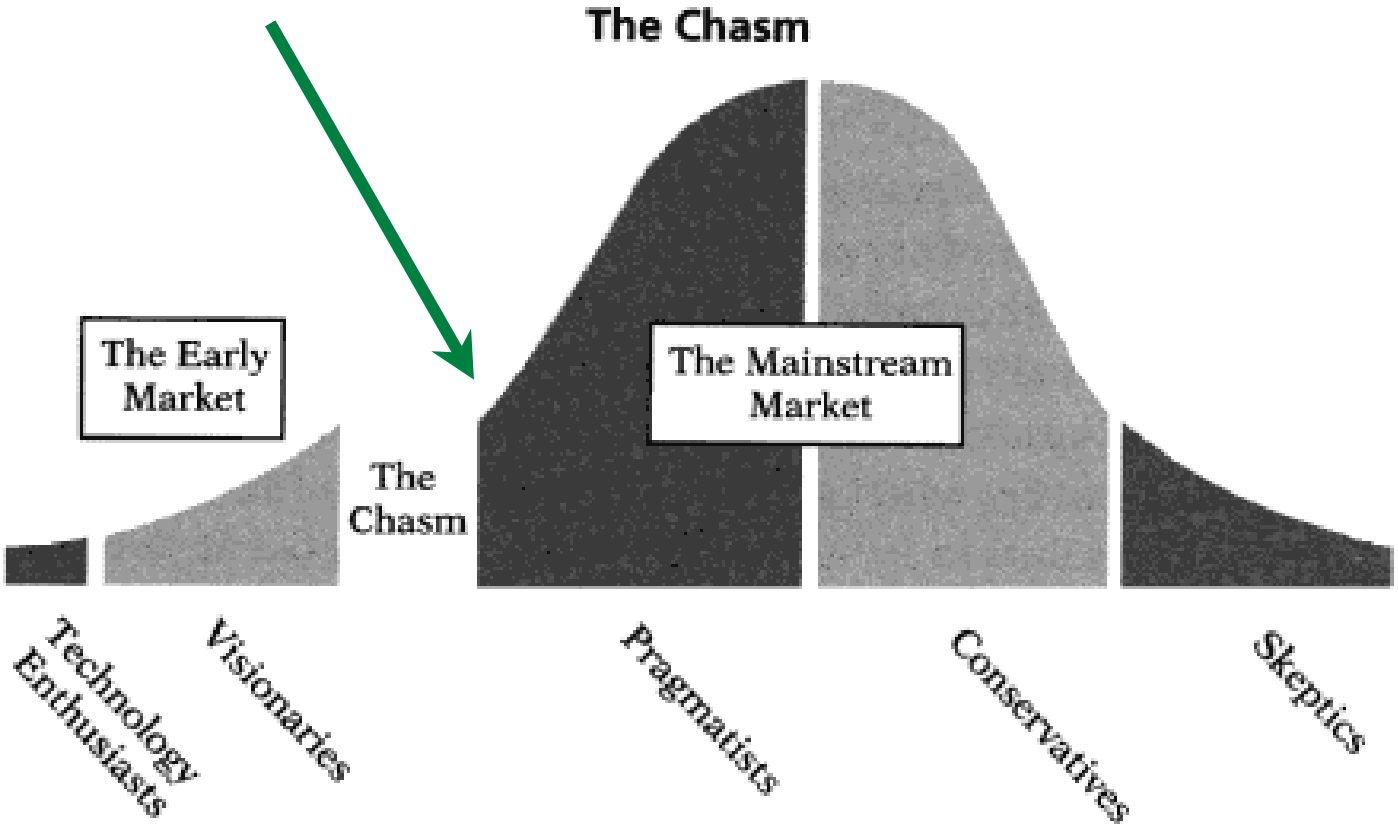


Eventually, the jelly is cooked, nailed down, and ready for further change

The Scary Part

- **There are no explicit requirements or specifications**
  - so tests cannot check code against them
- **Tests serve same goal as requirements or specifications**
  - they provoke programmers to write the right program

Status


The Chasm
The Early Market
The Mainstream Market
The Chasm
Technology Enthusiasts
Visionaries
Pragmatists
Conservatives
Skeptics

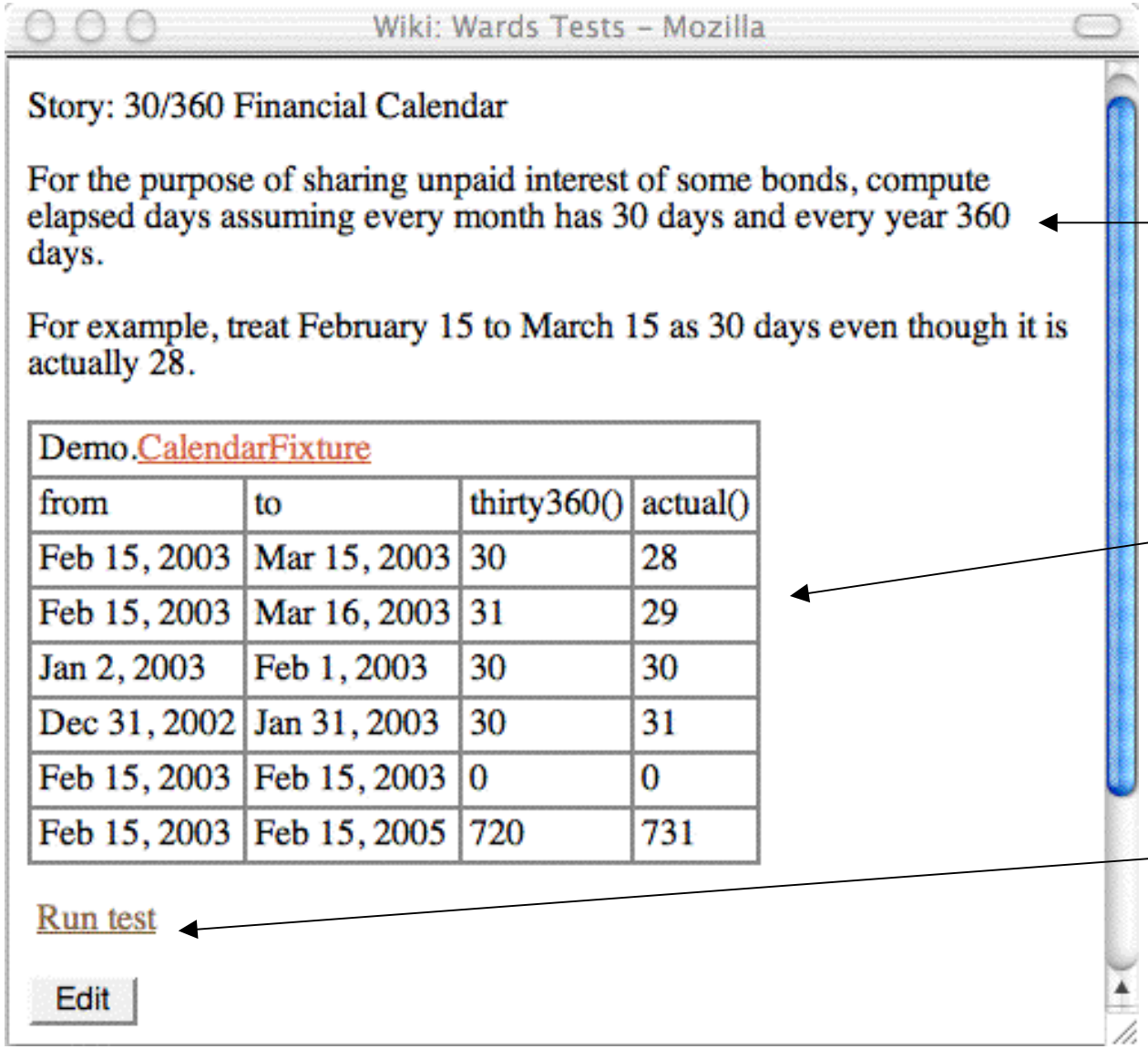
I need an object that records each time segment

Why was this decision made?

Because of a Business Expert

…I want to scribble notes about what I'm doing…

FIT Tests



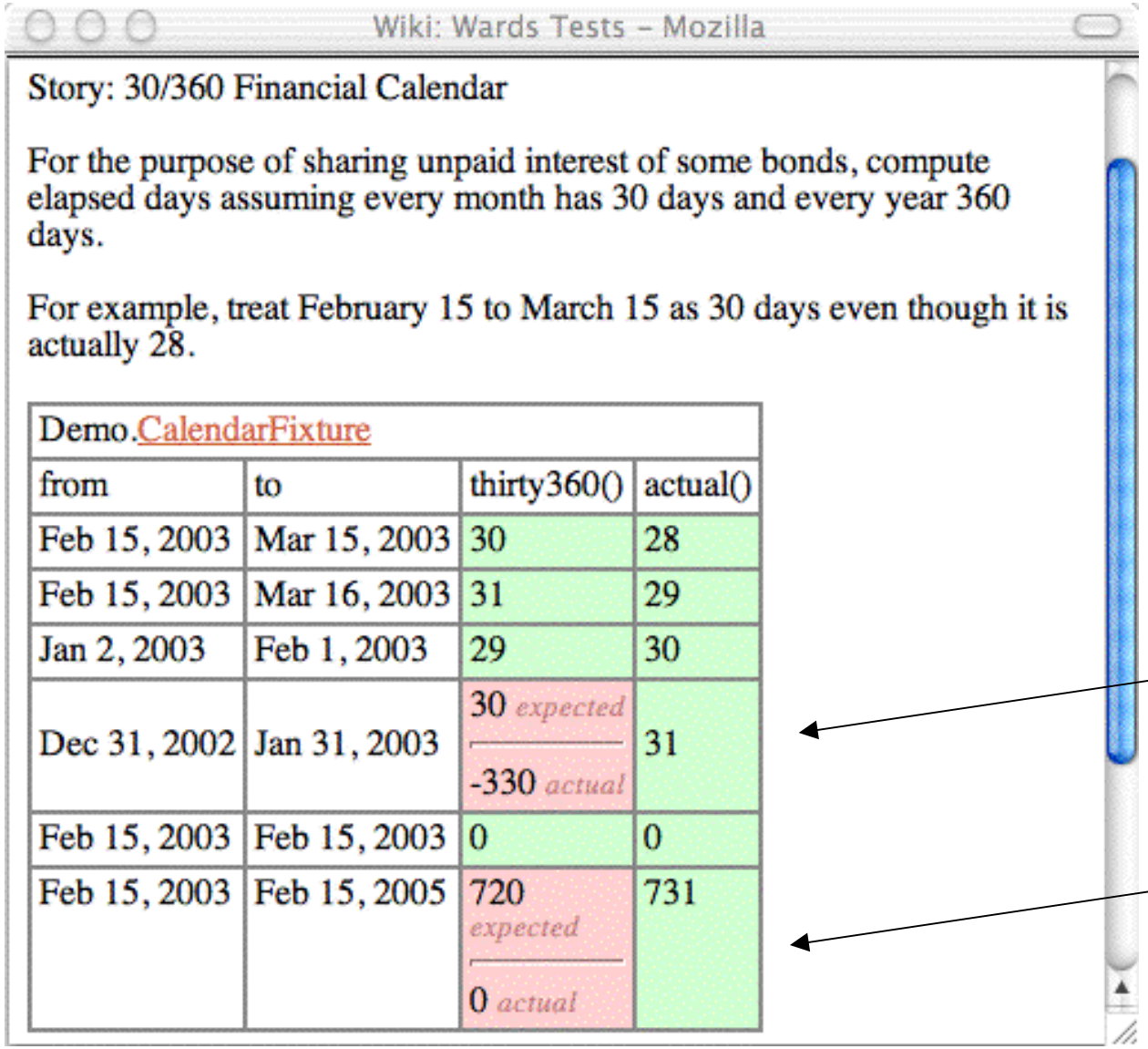
Reminder, not requirement

Customer-checkable

Anyone can run tests

fit.c2.com

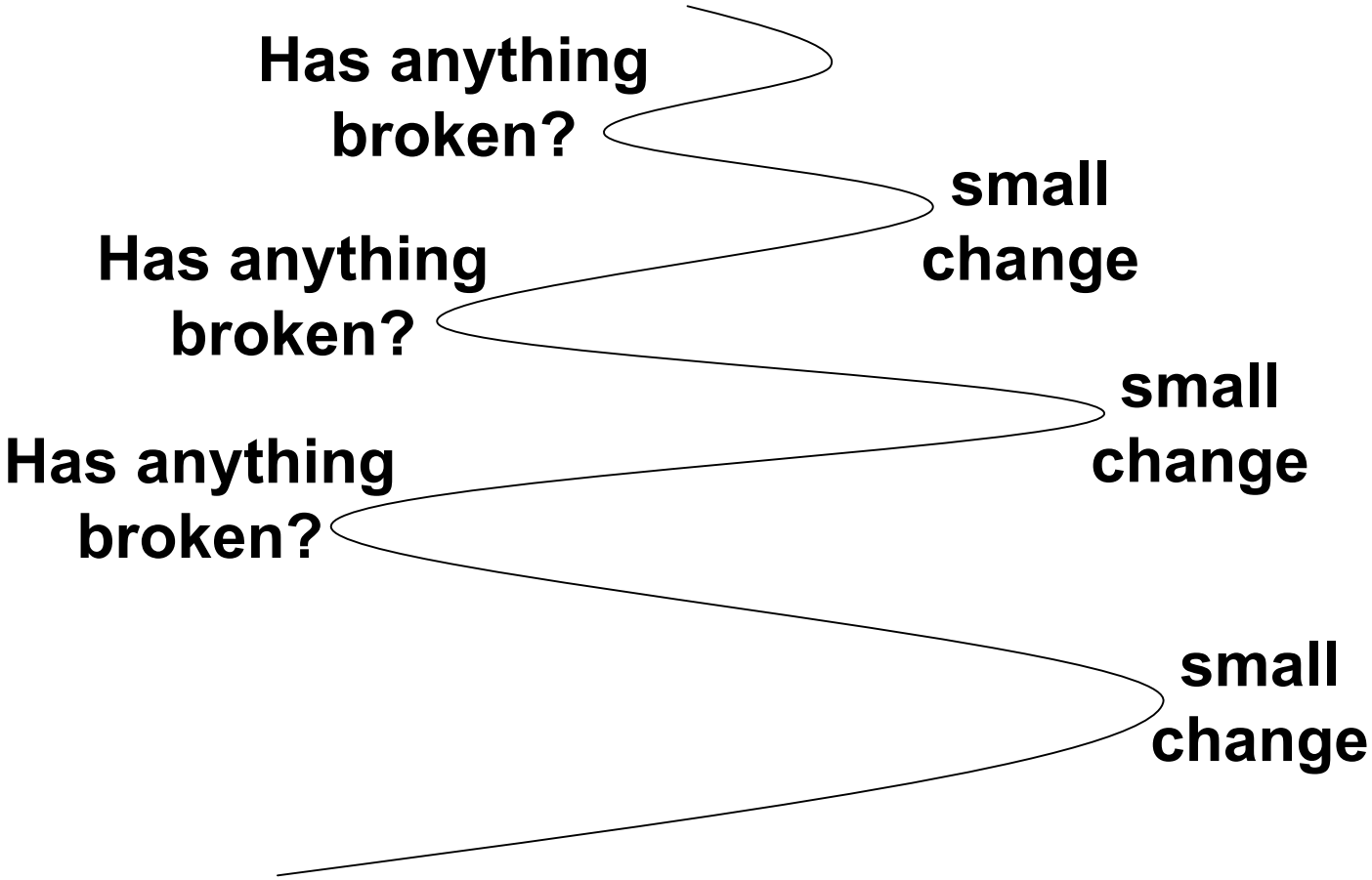
Test Results



Browser-Friendly

Rapid Feedback

Tests Make Change Smooth
(in addition to informing programmers)



Requirements for Test Notation

- **Provoking the right code**
- **Improving product conversation**
  - tests are something to talk *about*
  - ground conversation in the concrete
  - forging a common vocabulary
- **Making possibilities more noticeable**
  - explaining to someone else supplements trying out working software
  - concreteness sparks ideas

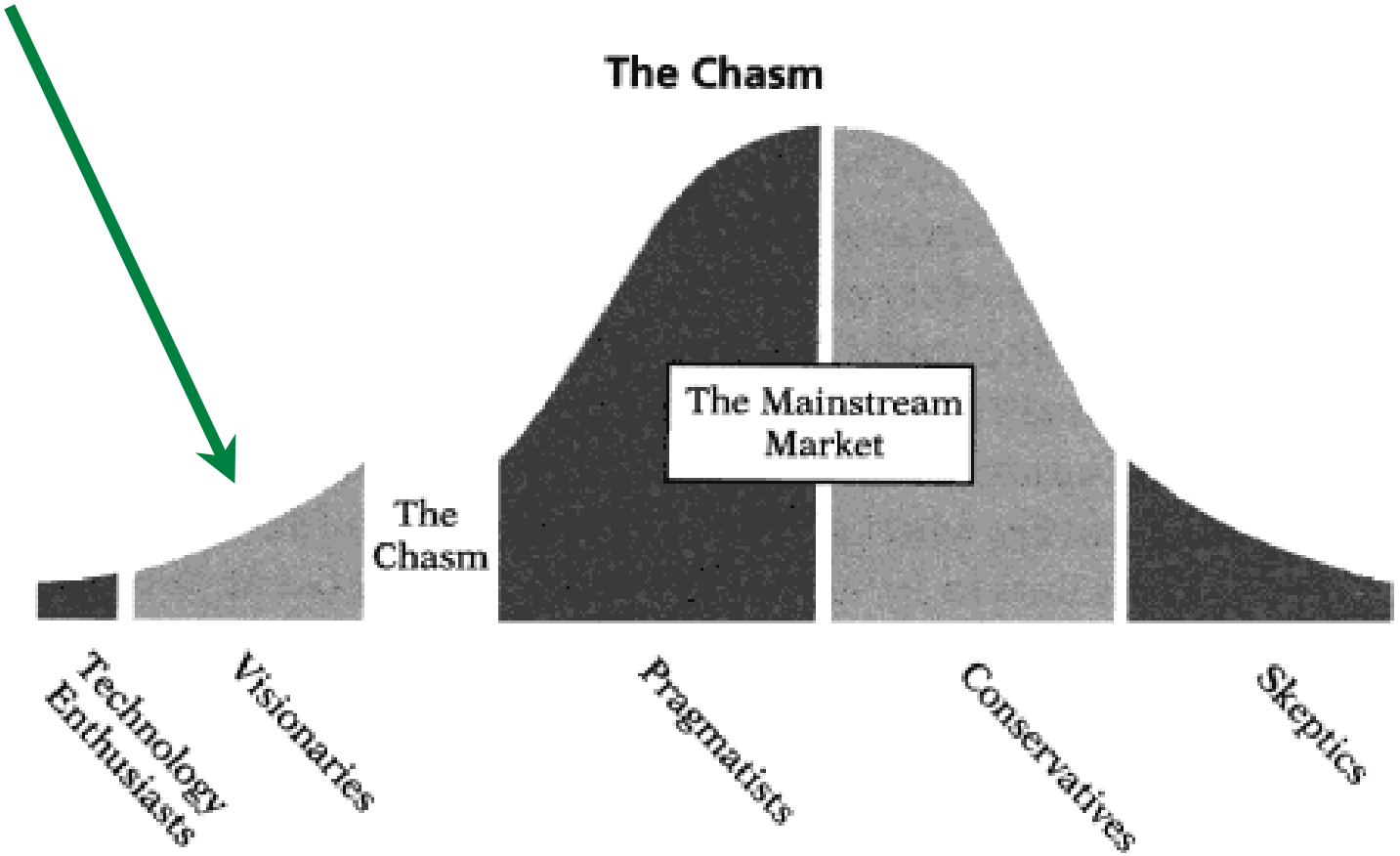
The Tester As Participant

How can we best be concrete?
What "goes without saying"?
What should the product *not* do?
Who's being overlooked?
What bugs seem likely?

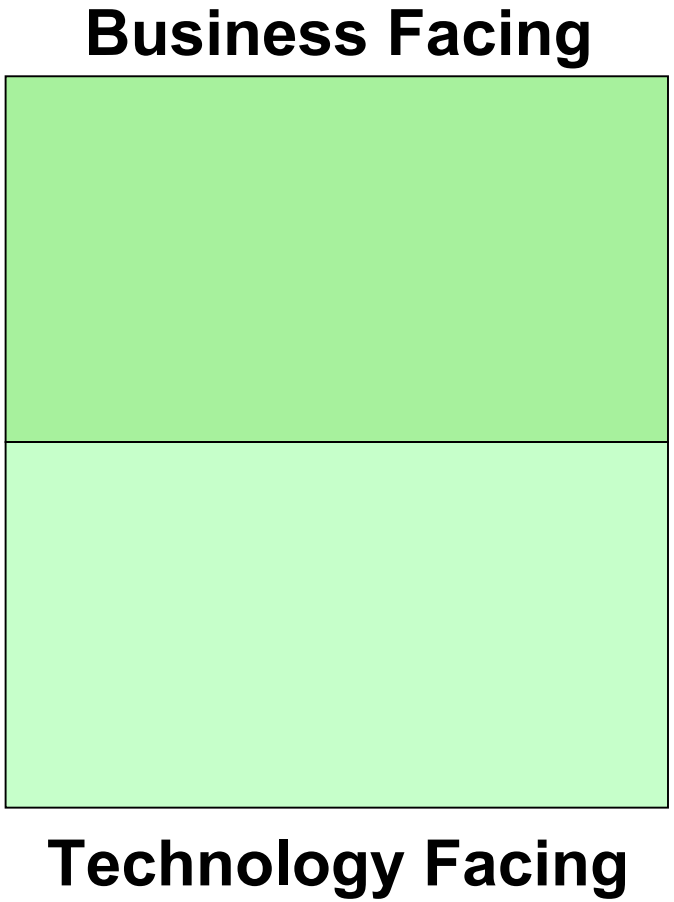
“Tester” As Job Title

Status

acceptance tests, customer tests, whole-product tests, business-facing tests…



Mid-Course Observation



These tests *primarily* support programming (as well as the whole team's understanding)

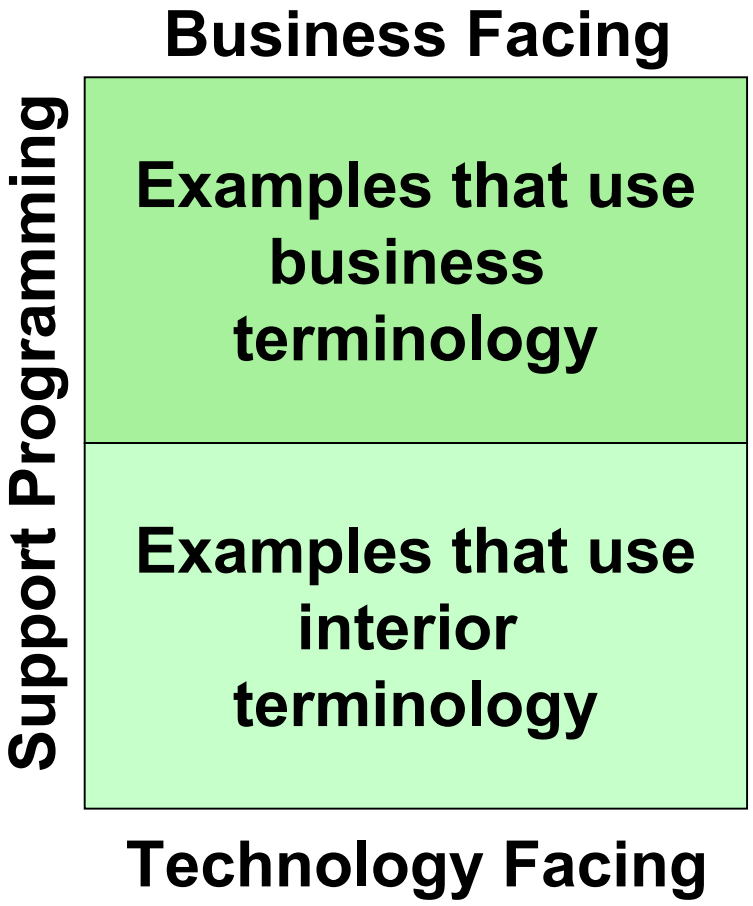
Is This Really Testing?

- **Checked examples**
  - for discussion
  - for confident implementation
  - where are the bugs?
- **Change detectors**
  - for confident implementation
  - where are the bugs?

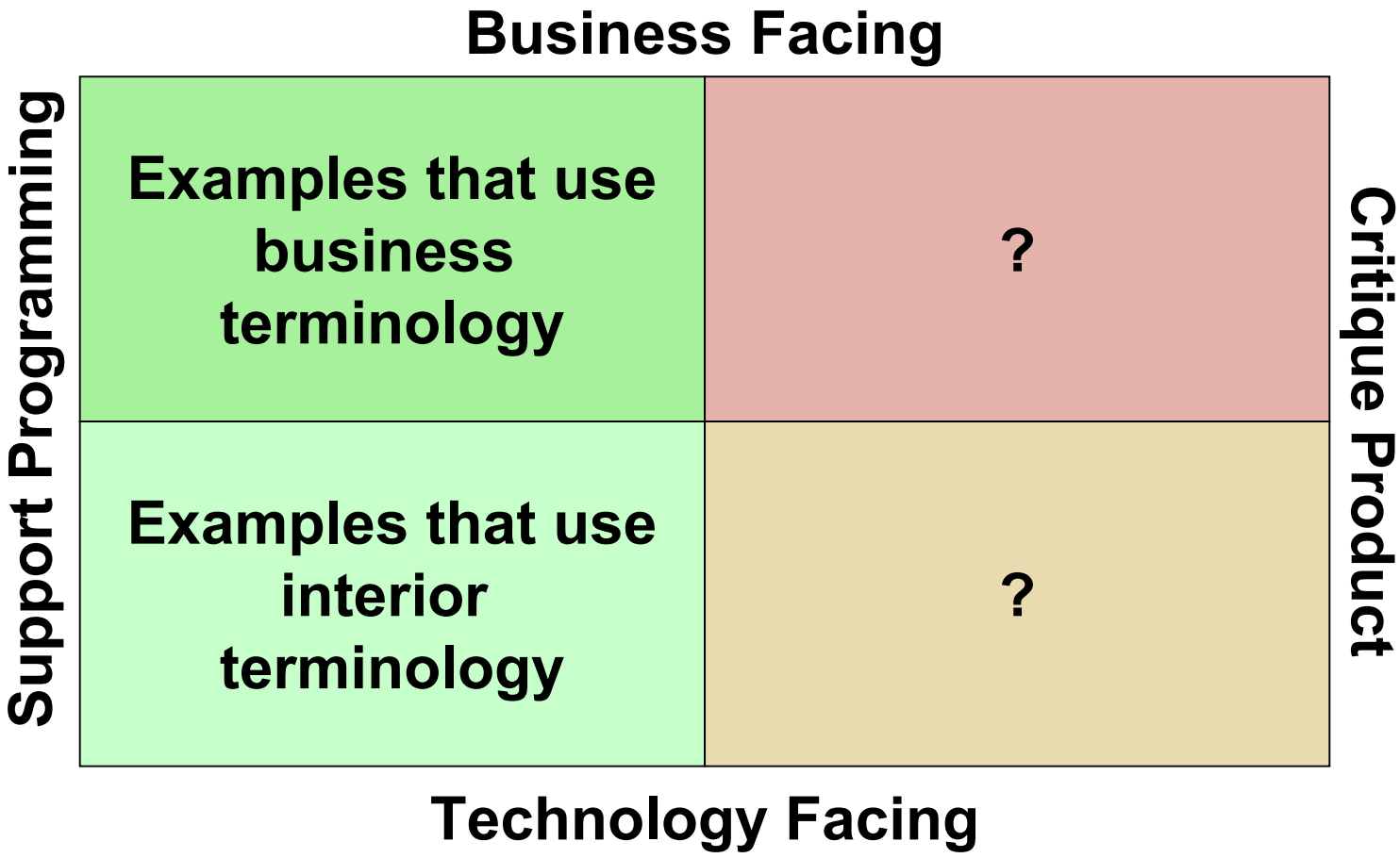
What About When the Examples Are Bad Examples?

(incomplete, misleading)

Extending the Model



Extending the Model

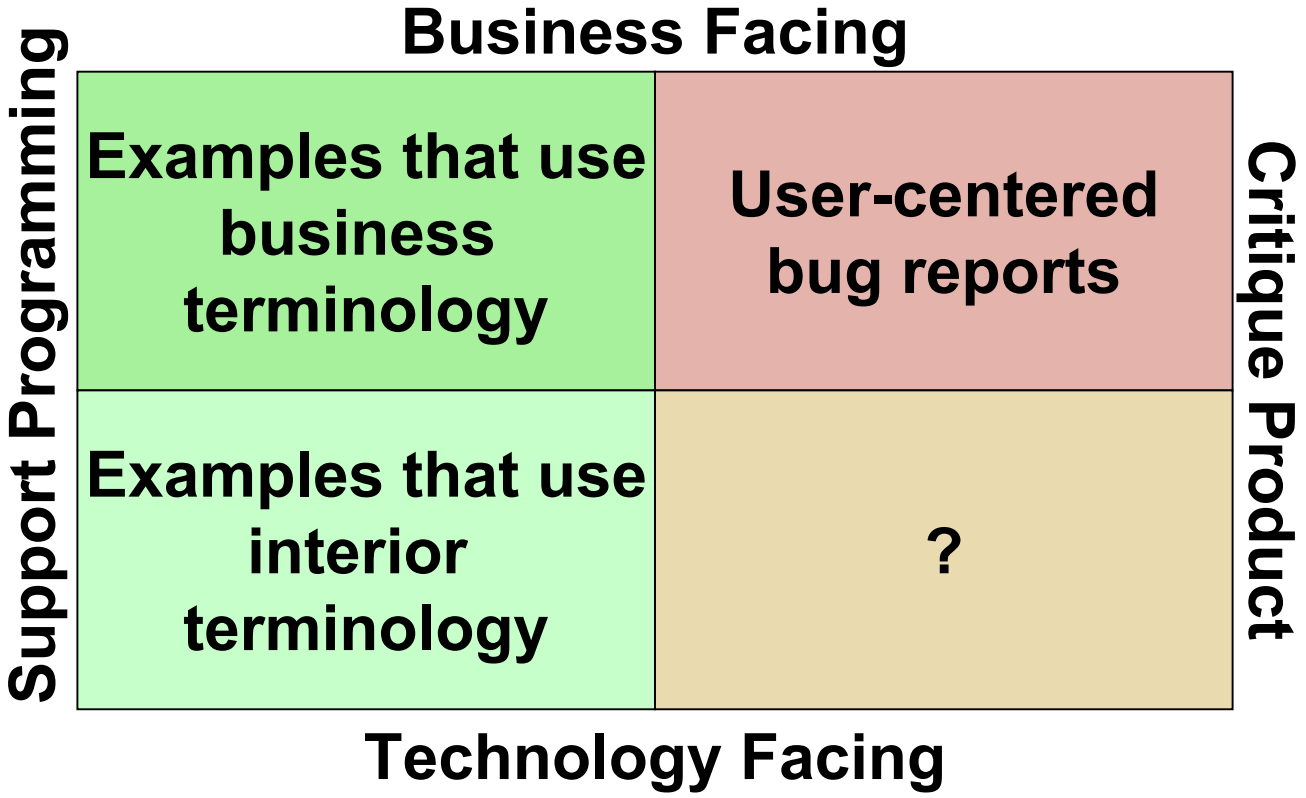


Critiquing the Product

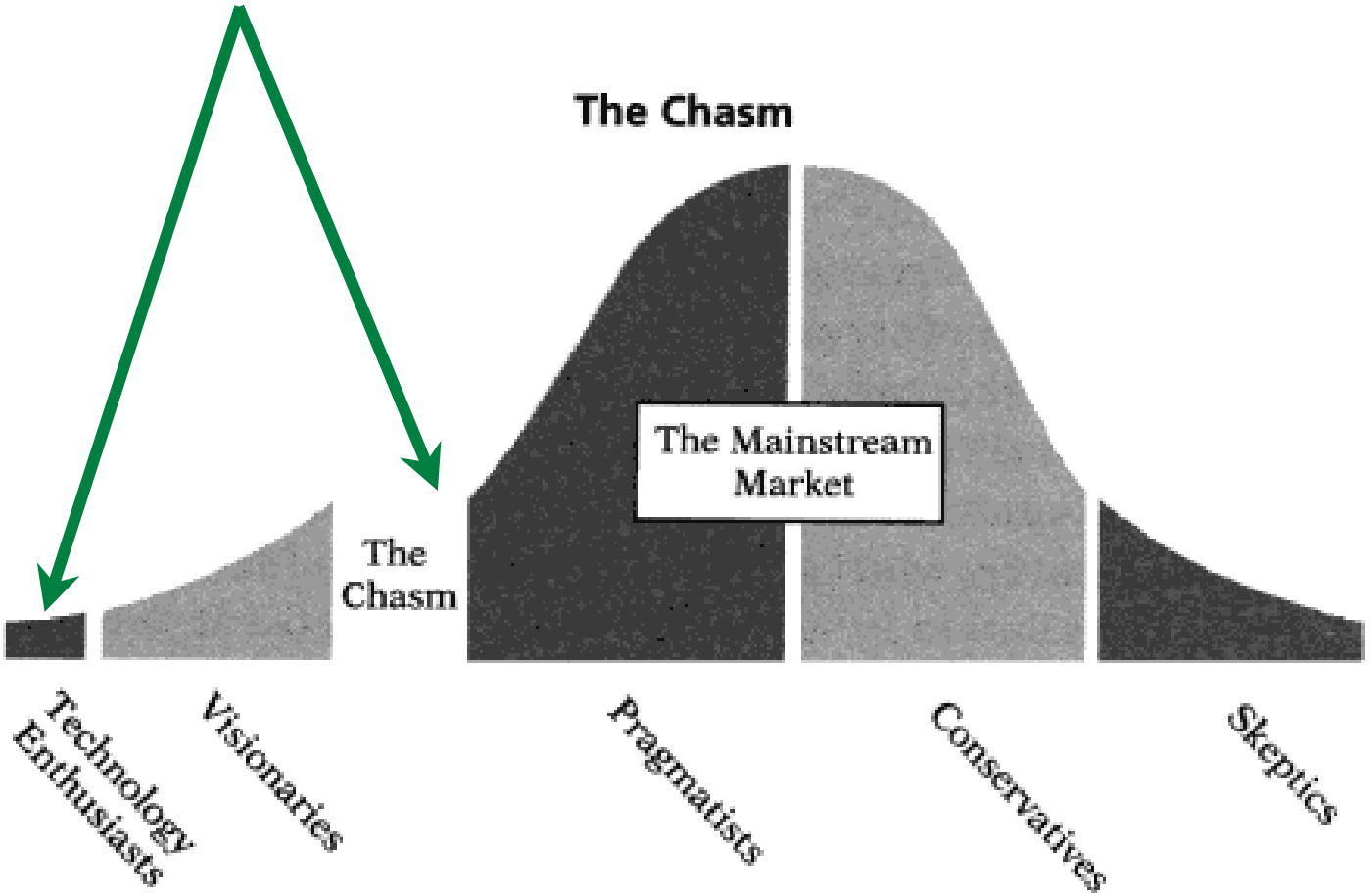
- **What resource do we newly have?**
  - the working product, including new code
- **Exploratory testing**
  - "simultaneous learning, test design, and test execution" - James Bach
- **Doing what?**
  - diverse users and their scenarios
  - imaginative end-to-end testing
  - some opportunistic feature testing

Exploratory Bug Finding

A Quadrant Entry



Status

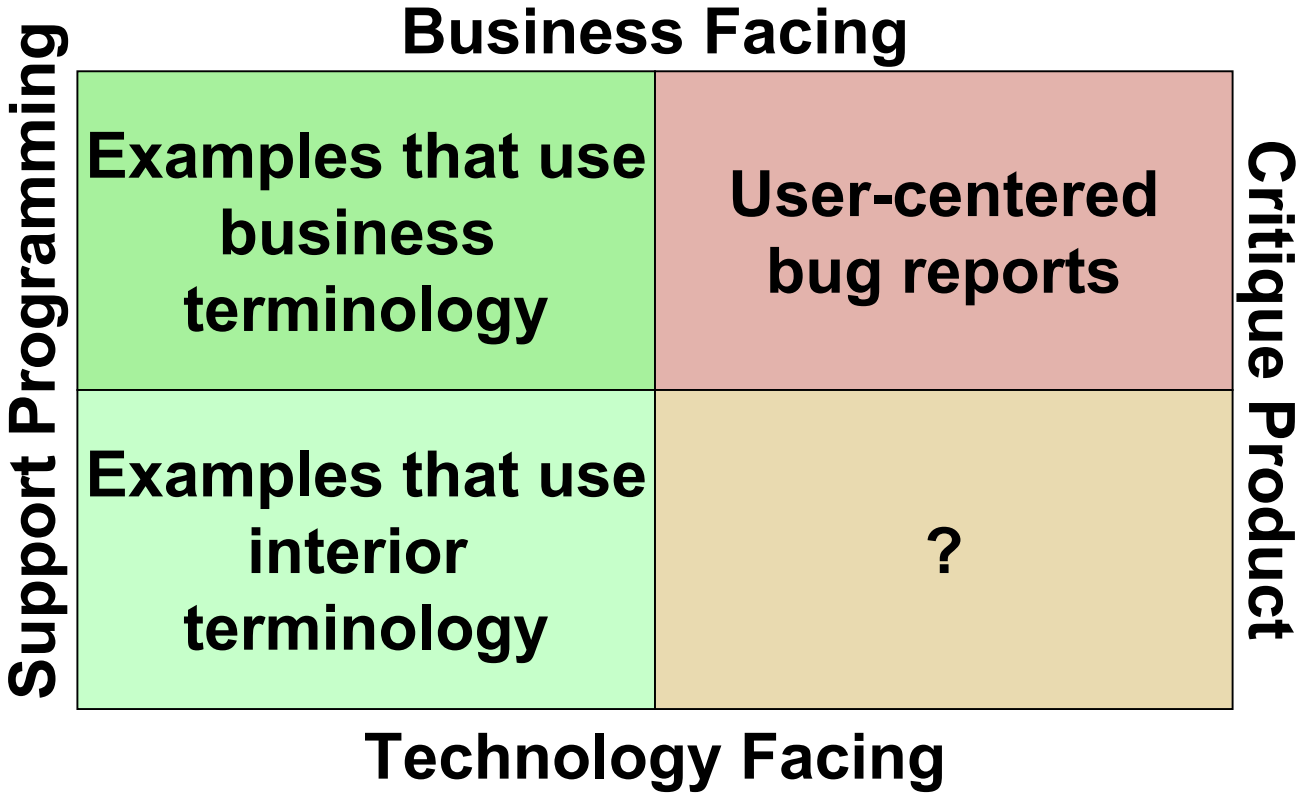

The Chasm
The Mainstream Market
The Chasm
Technology Enthusiasts
Visionaries
Pragmatists
Conservatives
Skeptics

Still Not Addressed

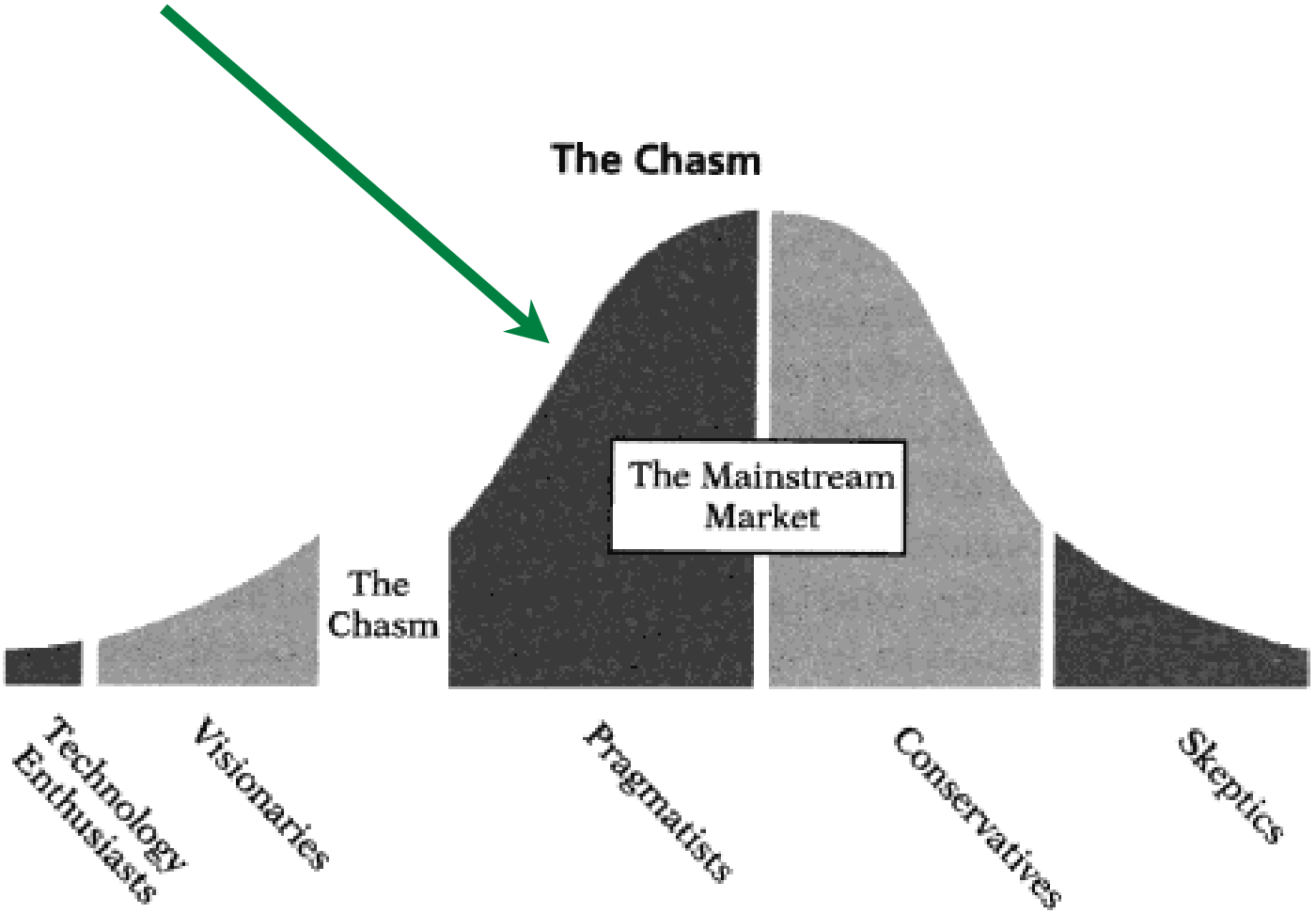
- **What about security bugs, configuration bugs, performance problems, bugs revealed under load, usability problems (like suitability for color-blind people), etc. etc. etc.?**
  - difficult to specify by example
  - whole-product, but not central to domain

These Are Technology Issues

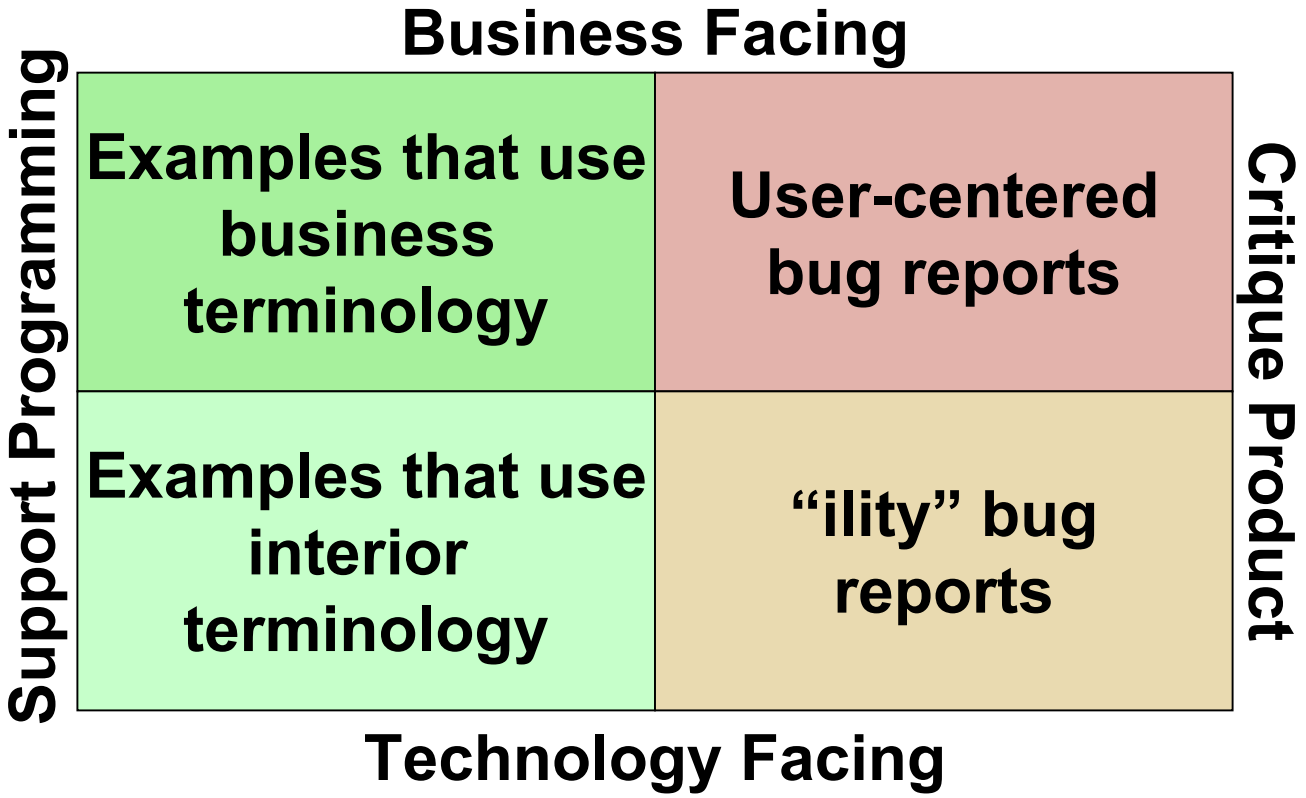
- **Understanding of implementation more important than understanding of a particular domain**



The Good News



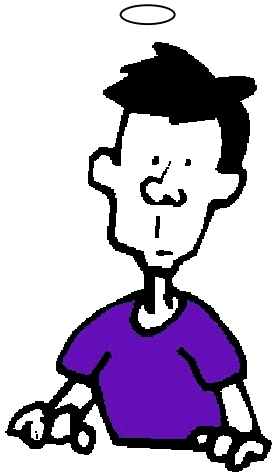
My Take on the State of the Practice



A Problem

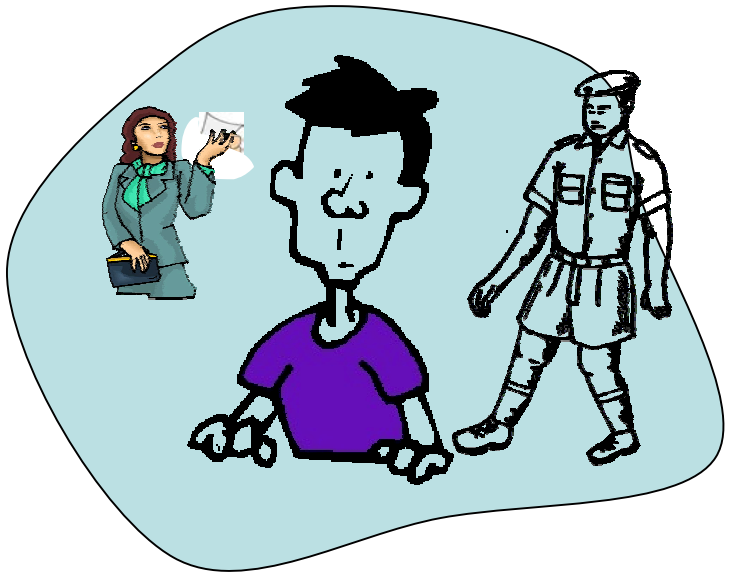
I am Tester, hear me roar!

… generalist … team…
… trust …



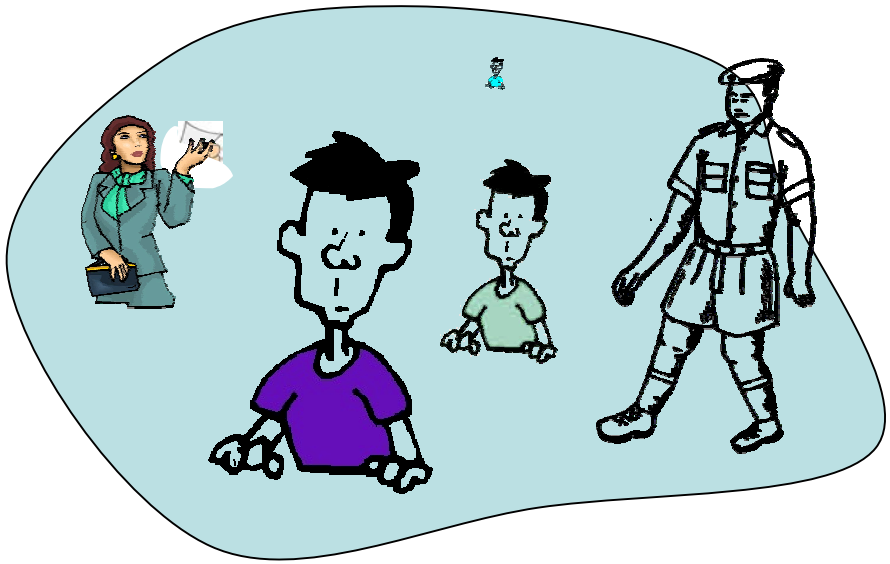
People As “Skill Bundles”

BEM

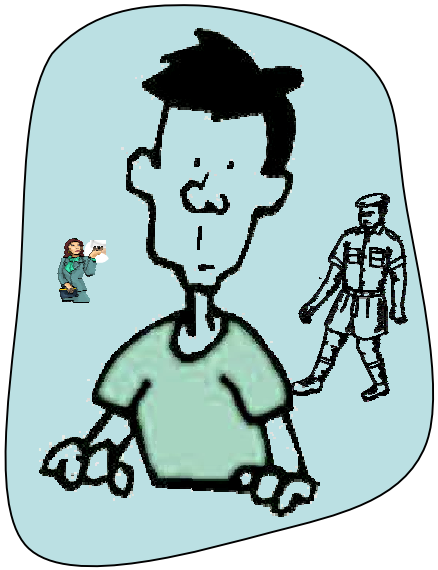


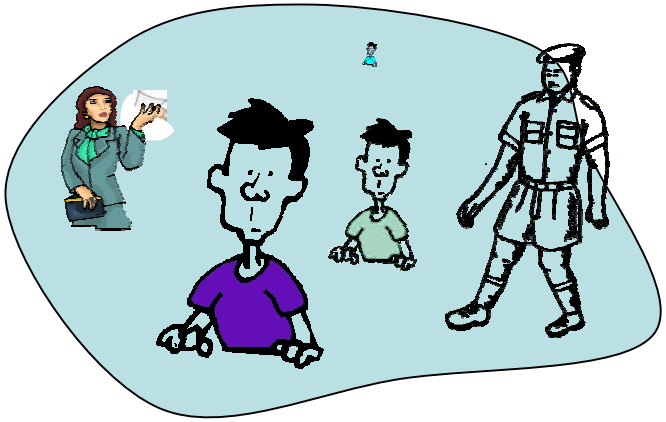
People As “Skill Bundles”

BEM

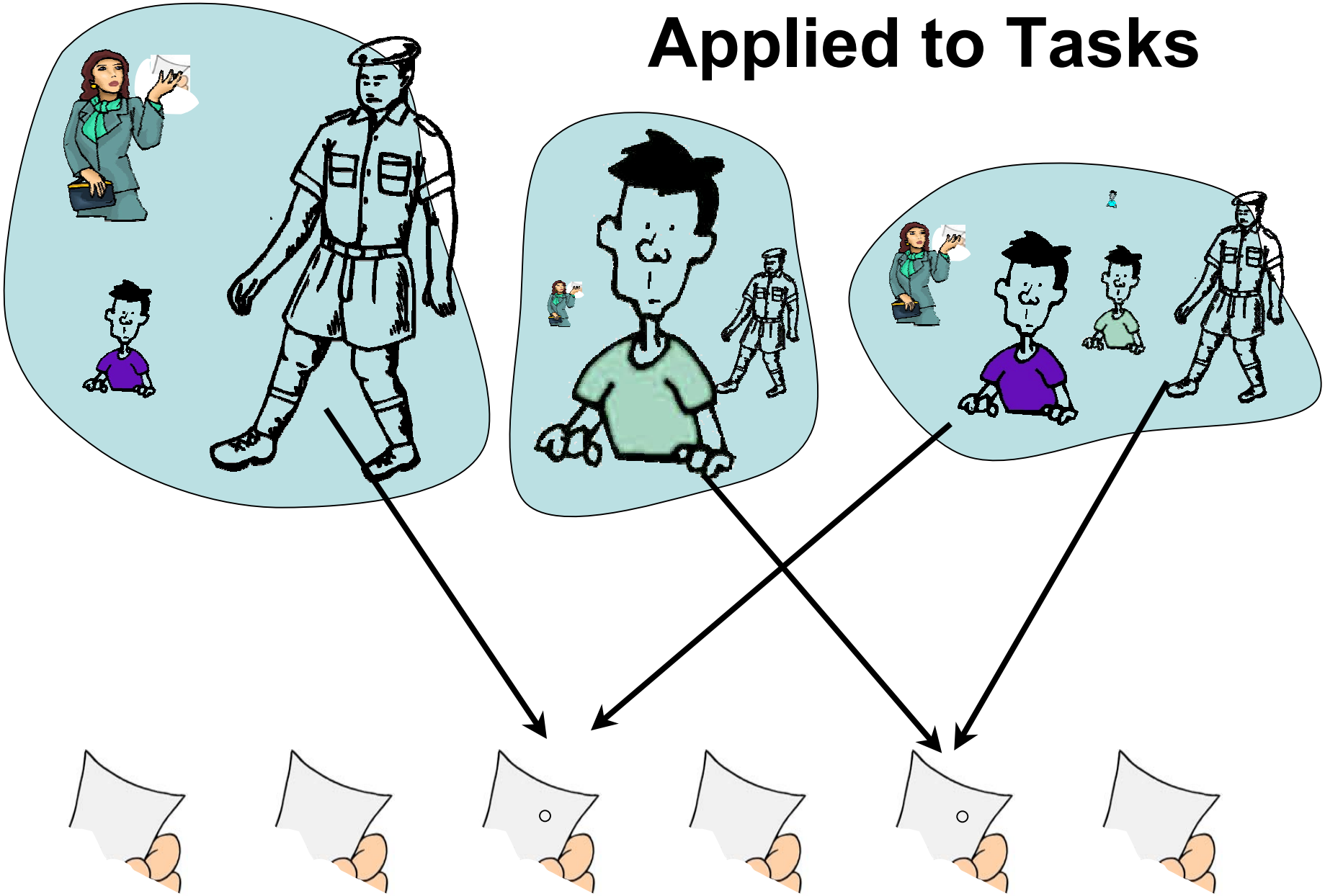


A Team of Skill Bundles

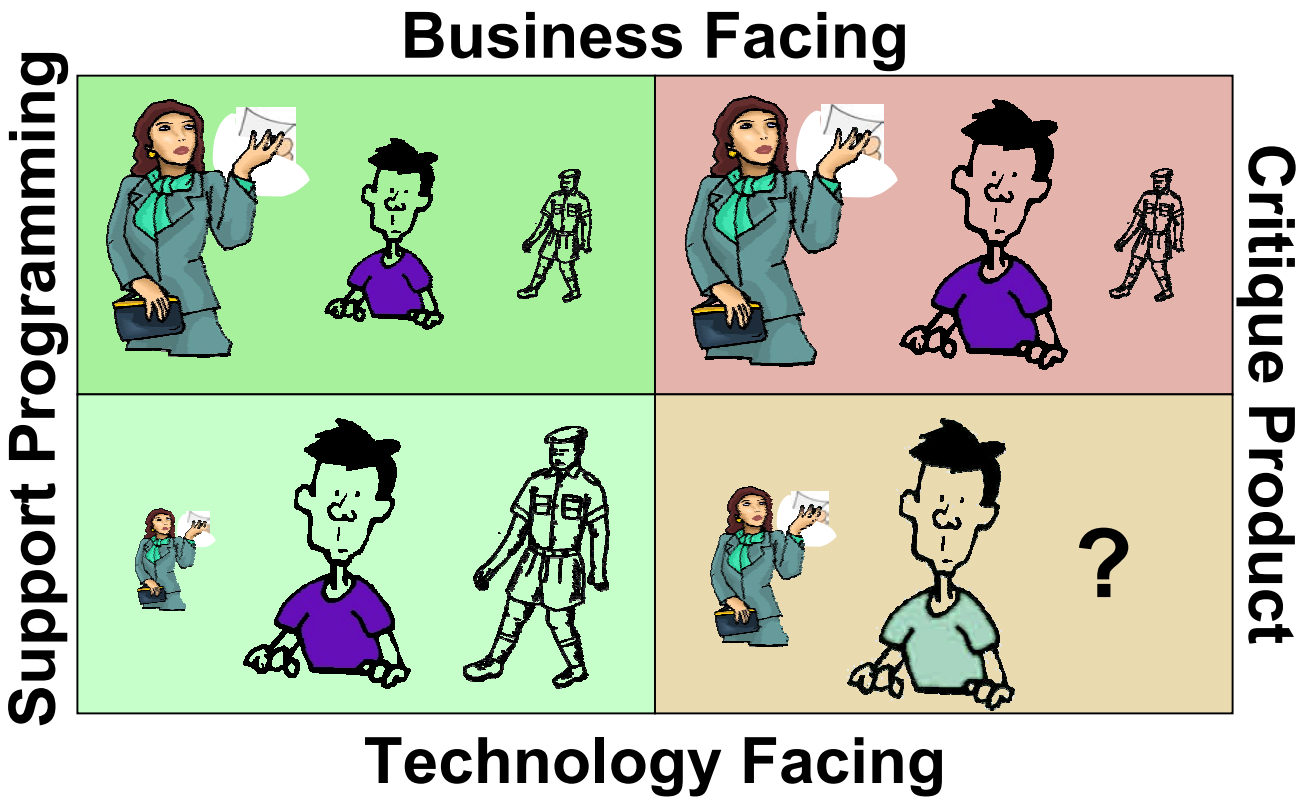




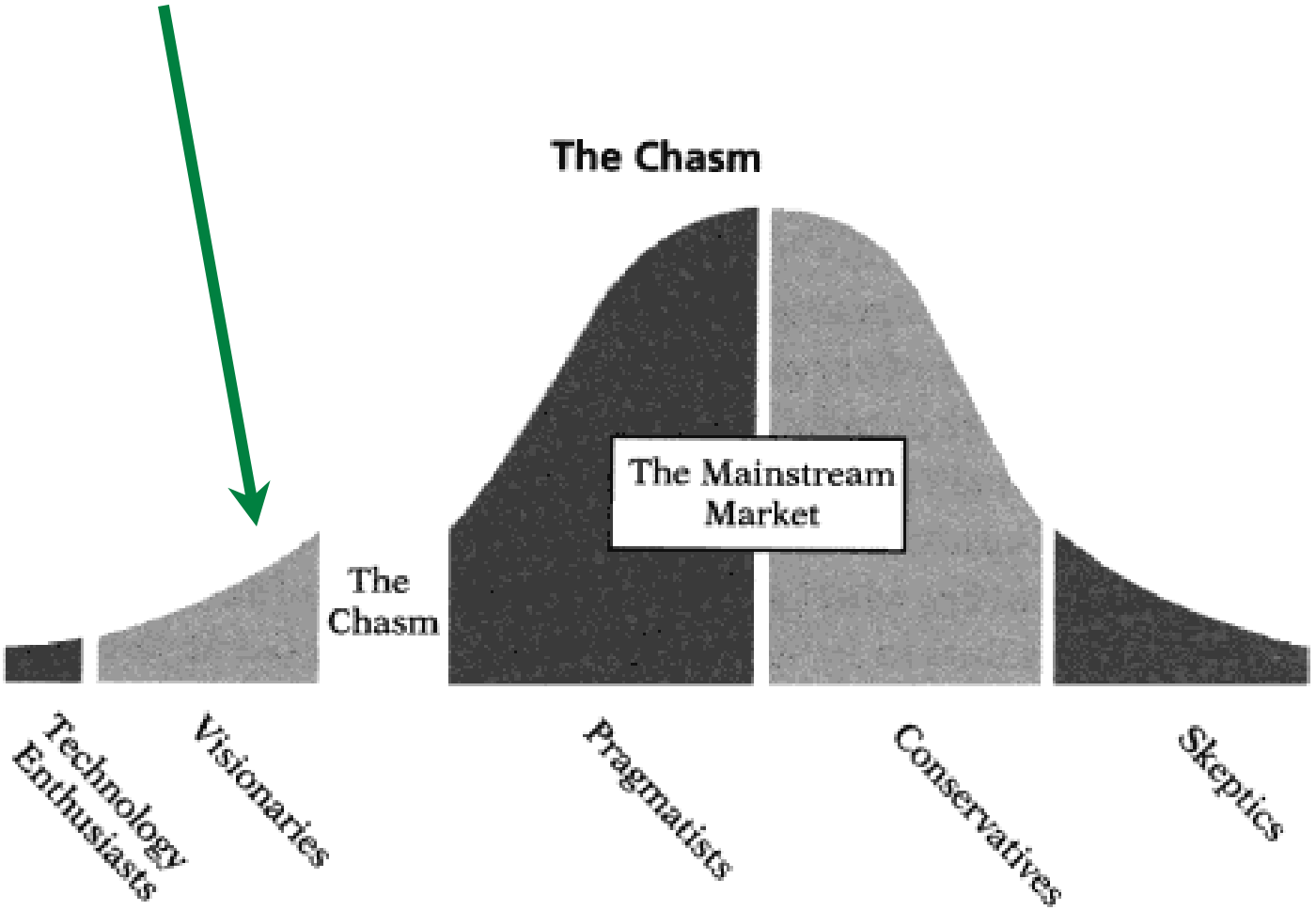
Applied to Tasks



Identity


Business Facing
Support Programming
Critique Product
?
Technology Facing

Summary: Testing in Agile Projects



Reading (1)

- **Agile development**
  - www.agilealliance.org
  - *Agile Software Development*, Alistair Cockburn
  - "Agile methods, the Emersonian worldview, and the dance of agency", Brian Marick, www.visibleworkings.com/papers/agile-methods-and-emerson.html
- **Agile from a tester's point of view**
  - www.testing.com/agile
  - *Testing eXtreme Programming*, Lisa Crispin and Tip House
  - agile-testing@yahoogroups.com
  - www.testing.com/blog
- **Programmer testing**
  - *Test-Driven Design by Example*, Kent Beck
  - *Test-Driven Development: A Practical Guide*, Dave Astels
  - testdrivendevelopment@yahoogroups.com

Reading (2)

- **Exploratory testing**
  - www.satisfice.com/articles.shtml
  - www.testingcraft.com/exploratory.html
- **Context-driven testing**
  - *Lessons Learned in Software Testing*, Kaner, Bach, and Pettichord
  - www.context-driven-testing.com/wiki/scribble.cgi
- **Miscellaneous**
  - FIT: fit.c2.com (see also fitnesse.org)
  - *Crossing the Chasm*, Geoffrey Moore
  - *Refactoring*, Martin Fowler (et. al.)

Learning Styles and Exploratory Testing

Andy Tinkham
Florida Institute of Technology
andy@tinkham.org

Cem Kaner, J.D., Ph.D
Florida Institute of Technology
kaner@kaner.com

Abstract

Exploratory testing is widely done in software testing. However, it is performed in many different ways. This paper discusses our initial work into examining a tester's learning style as an indication of the types of actions she might use while doing exploratory testing. We use the Felder-Silverman learning styles model as a basis for presenting our hypotheses about how this aspect of one's personality affects how she performs exploration.

Bios

Andy Tinkham is a graduate student at Florida Tech, pursuing a Ph.D. in computer science. The main focus of his research is on exploratory software testing and the ways in which people can be trained to be good explorers. He has 8 years of experience in the field of software testing, where he worked both as a tester for companies whose main business was the production of software and as a consultant, helping companies set up successful automated testing efforts. Mr. Tinkham holds a B.S. in Computer Engineering.

Cem Kaner, J.D., Ph.D., is Professor of Software Engineering at the Florida Institute of Technology. His primary test-related interests lie in developing curricular materials for software testing, integrating good software testing practices with agile development, and forming a professional society for software testing. Before joining Florida Tech, Dr. Kaner worked in Silicon Valley for 17 years, doing and managing programming, user interface design, testing, and user documentation. He is the senior author of *Lessons Learned in Software Testing* (with James Bach and Bret Pettichord) and *Testing Computer Software* (2nd Edition) (with Jack Falk and Hung Quoc Nguyen) and of *Bad Software: What To Do When Software Fails* (with David Pels).

Dr. Kaner is also an attorney whose practice is focused on the law of software quality. Dr. Kaner holds a B.A. in Arts & Sciences (primarily Math & Philosophy), a Ph.D. in Experimental Psychology (Human Perception & Performance: Psychophysics), and a J.D. (law degree).

This paper was originally prepared for and presented at the Pacific Northwest Software Quality Conference (PNSQC) 2003

This research was partially supported by NSF grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

Introduction

Exploratory testing has attracted a great deal of interest in the software testing community[1]. This approach to testing typically involves testing, learning, and designing new tests as interacting activities, all occurring simultaneously. Almost all testers explore at times as they perform their jobs, whether they acknowledge it or not. For example, consider regression testing of bugs. To check whether a bug was fixed, the tester might *start* with the exact steps listed in the bug report, but after the program passes this simple test, the tester will probably try additional tests to check if the bug is completely fixed. These tests might vary the conditions that initially exposed the problem, or they might explore possible side effects of the fix. The details of these additional tests are not recorded in any test plan. The tester invents them as she works. In this case, she is being an explorer – she is exploring.

One of the puzzles of exploratory testing is that there appear to be different exploration styles. That is, it appears to us that there are many different available strategies for creating and using exploratory tests, that these are not mutually exclusive, and that individual testers (or explorers) seem to adopt a subset of these strategies.[2] This makes it difficult to characterize exploratory testing in terms of a narrow set of techniques, or a set of mental models or behaviors shared by all explorers.

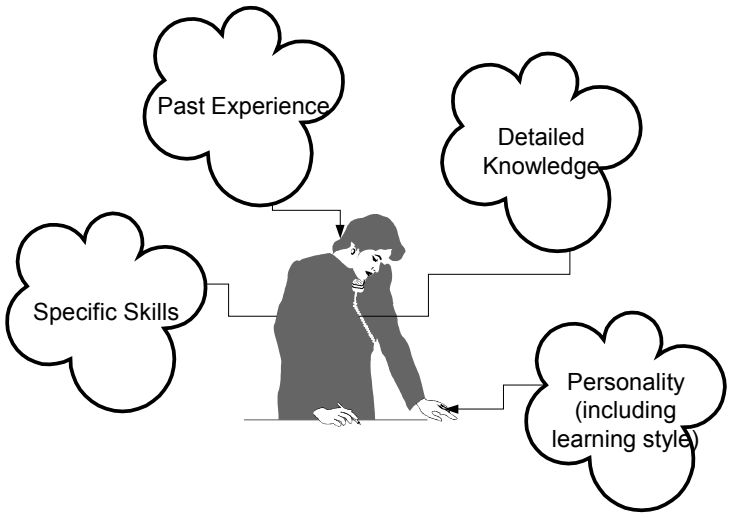
While doing exploratory testing, the tester generally has an overall purpose in mind that he is trying to achieve. Bach makes these purposes explicit and calls them charters[3]. An example charter might be to "check the UI against Windows interface standards". This charter serves to narrow the tester's focus and help him to better test the application by removing distractions and tangents that he might otherwise encounter. The charter does not, however, specify how he should accomplish the task. The specific actions required to satisfy the charter are left to the discretion of the tester, who determines them on the fly as he tests the application.

---

[1] For descriptions of exploratory testing, see (Tinkham & Kaner, 2003), (Bach, 2003), (Kaner, Bach & Pettichord, 2001), and (Agruss & Johnson, 2000)

[2] Kaner presented this idea in a set of 56 slides at the 7th Los Altos Workshop on Software Testing. His course slides (Kaner, 2002) are a more widely available, slightly updated source for this material.

[3] (Bach, 2003)



Many factors contribute to a tester's choice of exploration style

There is thus a large creative element to exploratory testing. Two different testers given the same charter may take two entirely different paths to achieve the stated goal. Neither path is necessarily better or worse than the other (though either one may find bugs that the other did not). How each tester develops her style (her pattern of adoption of exploratory test types) is probably based on several factors, including past experience, specific skills and detailed knowledge, and perhaps also aspects of her personality, such as learning style (the preferences the tester has for the ways in which she learns information).

This paper looks at how testers' learning styles could influence how they perform exploratory testing. Learning style seems like a relevant variable to us because a core activity of exploratory testing is learning about the software, including its weaknesses, potential failure modes, potential applications, market, configuration variability, and so on.

Several models of learning styles have been proposed. In this paper, we examine the Felder-Silverman model for insights that it might provide into the exploration process. We believe this model can provide guidance both in how an individual explorer approaches testing and how that explorer can expand his repertoire of techniques and thereby be more successful in exploration. We start by describing exploratory testing, and then discuss the Felder-Silverman model in more detail and apply its continua to exploratory testing. We have performed an informal survey of a small number of testers. While this survey was not statistically representative enough for us to be able to draw specific conclusions, it does indicate that there is a strong likelihood of patterns of correspondence between learning style tendencies and exploration technique choices occurring in a larger sample of the testing population, as well.

The research described in this paper is still in the early stages. While we believe the learning styles model makes sense and the points we make apply, we have not yet had the chance to validate the ideas. We are *very* interested in comments from the field to either support or refute the points we make. We are designing empirical research to study our hypotheses: we will make

our research results available at our lab's web site[4]. This research will involve observing and interviewing testers, in-depth study of the various learning style models, and the development and testing of exercises (each geared towards a particular learning style or combination of learning styles) to be used in training exploratory testers.

We present the hypotheses we have at this point because we believe the insights provided by learning style models can be immediately helpful to those doing exploratory testing. Our work might have implications for some other approaches to testing, but we are most interested in applying this to exploration because all of the investigative choices are under the explorer's control. Approaches centered on a specific technique, such as domain testing, are more constrained and so their application is less influenced by learning style.

Exploratory Testing

Exploratory testing is one of the most widely used approaches in the field of software testing[5]. It is also the least understood approach despite being used by most testers as part of their day-to-day activities. For example, any tester who deviates from a scripted process to characterize a defect he found or to verify a bug fix is doing exploratory testing. However, some testers are reluctant to describe their work as exploratory because they think the term connotes random work or that it is merely an excuse for testers to avoid the documentation and planning called for by traditional testing[6].

Competently done, exploratory testing is neither an excuse to avoid work nor a waste of time due to randomness. The essence of exploration is an active, risk-focused investigation of the product. Rather than designing tests early, when he is just beginning to learn what the product is, how it can fail, and who will use it to do what, on what platforms, the exploratory tester continually seeks out more information about the product and its market, platform, and risks – and designs tests to exploit the knowledge he has just gained. The explorer might use any test technique, and will probably use several different ones, in his quest to learn more about the product and its weaknesses.

Exploration Styles

Given a specific charter, individual testers will take substantially different approaches to fulfill that charter. One tester may start by creating models of the application. These models might be bare bones, quickly sketched on a piece of paper. Another tester might begin by brainstorming a list of different tests that could be executed and then work his way through the lists. A third tester might work from a list of failure modes (ways that we can imagine the product *might* fail), designing tests to determine whether the program actually does fail in those ways. While each approach could meet the charter of the testing being done, they will probably yield different bugs and different information about the product. Choosing a style of testing to achieve a specific

---

[4] http://www.testingeducation.org

[5] Those readers who may have less knowledge of what exploratory testing is are encouraged to read our earlier paper "Exploring Exploratory Testing" (Tinkham & Kaner, 2003) and James Bach's paper, "Exploratory Testing Explained" (Bach, 2003) for a more extensive explanation of the subject.

[6] While most articles do not explicitly describe exploratory testing in these terms, the author's attitudes can be inferred. See for example, the treatment of exploratory testing in the Software Engineering Body of Knowledge (SWEBOK, 2003) and papers extolling testing as a "discipline" (like (Drabick, 1999)).

charter and then developing effective tests within the style illustrates the creative demands of the exploratory testing process.

At the core of each style is the idea of questioning[7]. We think of each test case as a question asked of the application under test. Test design is a matter of developing good questions and asking them well. Exploratory test design is informed by the answers to previous questions.

Kaner identified nine different styles of exploration (most of which have several subsets) that he had seen exploring testers use[8]. Each style involves asking a different set of questions. The differences stem from the focus of the questions, and the skills and knowledge needed to ask the questions and interpret the answers. These styles involve using

- "hunches" (past bug experiences, recent changes),
- models (architecture diagrams, bubble diagrams, state tables, failure models, etc.),
- examples (use cases, feature walkthroughs, scenarios, soap operas, etc.),
- invariances (tests that change things that should have no impact on the application),
- interference (finding ways to interrupt or divert the program's path),
- error handling (checking that errors are handled correctly),
- troubleshooting (bug analysis (such as simplifying, clarifying, or strengthening a bug report), test variance when checking that a bug was fixed),
- group insights (brainstorming, group discussions of related components, paired testing), and
- specifications (active reading, comparing against the user manual, using heuristics (such as Bach's consistency heuristics[9])).

Why does one explorer rely on one (or a few) of these approaches while another adopts a different one? We think that part of the answer lies in the explorer's learning style.

The Felder-Silverman Learning Styles Model

A learning style is a person's "characteristic strengths and preferences in the ways they take in and process information"[10] These characteristics vary from person to person, and "may be strong, moderate, or almost nonexistent, may change with time, and may vary from one subject or learning environment to another"[11] for a given student. In their 1988 paper[12], Felder and Silverman proposed a model of learning styles that indicates a person's predilections on five continua: Sensory/Intuitive, Visual/Verbal, Inductive/Deductive, Active/Reflective, and Sequential/Global.[13]

---

[7] Kaner talks more about questioning strategies in his black-box testing course notes (Kaner, 2002)
[8] (Kaner, 2002)
[9] See (Bach, 1999)
[10] (Felder, 1996)
[11] (Felder, 1993)
[12] (Felder & Silverman, 1988)
[13] Readers who wish to determine their own learning style can visit http://www.ncsu.edu/felder-public/ILSpage.html for web-based and pencil-and-paper versions of Felder's *Instrument of Learning Styles*

The 5 continua of the Felder-Silverman model

A person's placement on the five continua might help her gain insight about why and when learning has been more pleasant and perhaps more effective, what blind spots she might have, and what techniques she might be underutilizing, that could help her learn more in situations that don't match her preferred environments.

Please note that a person's placement on the various continua is descriptive, not normative. There is no best pattern of results, no best learning style, no inherent superiority of any placement. Do not feel that you have to "be" a particular learning style. A person who prefers visual descriptions to verbal ones might still learn (if less enthusiastically) how to work through a strictly verbal specification. As Felder points out in his "Matters of Style" article[14], the model "provides clues, not infallible labels".

Felder presents a list of five questions that can be used to define (in part) a student's learning style:

1. "What type of information does the student preferentially perceive: *sensory*—sights, sounds, physical sensations, or *intuitive*—memories, ideas, insights?
2. Through which modality is sensory information most effectively perceived: *visual*—pictures, diagrams, graphs, demonstrations, or *verbal*—sounds, written and spoken words and formulas?
3. With which organization of information is the student most comfortable: *inductive*—facts and observations are given, underlying principles are inferred, or *deductive*—principles are given, consequences and applications are deduced?

[14] (Felder, 1996)

4. How does the student prefer to process information: *actively*—through engagement in physical activity or discussion, or *reflectively*—through introspection?
5. How does the student progress toward understanding: *sequentially*—in a logical progression of small incremental steps, or *globally*—in large jumps, holistically?"[15]

In the next sections, we explore the continua addressed by each question.

Sensory/Intuitive[16]

Felder defines a person who preferentially perceives sensory information as one who relies more on the information he receives through his external senses, while a person with a preference for intuitive information relies on his internal information (generated from memory, conjecture, and interpretation) and intuition. These preferences are equivalent to the sensing and intuiting types on the Myers-Briggs Type Indicator[17], and are derived from Carl Jung's theory of personality types.

According to Felder, strongly sensory learners are generally attentive to details. They are usually observant, and tend to favor facts and observable phenomena. They are apt to prefer problems with well-defined standard solutions and dislike surprises and complications that make them deviate from these solutions. Sensory learners can be patient with detail and are normally good at memorizing. They are generally good experimentalists.

Strongly intuitive learners, on the other hand, may be bored by details. They can easily handle abstraction, and are good at grasping new concepts. Often, intuitors strongly dislike repetition and they may be careless when performing repetitive tasks. Instead, they like innovation and are often imaginative and insightful. They respond best to thought problems, and like to emphasize fundamental principles and mathematical models. Intuitive learners often make good theoreticians, designers and inventors.

Visual/Verbal

Visual and verbal learners differ in how they best receive information. Visual learners retain more information they get from visual images such as pictures, movies, diagrams or demonstrations, and may have problems remembering information they simply hear. Verbal learners retain more information they hear (or read) such as lectures, written words, and mathematical formulas. "Most people (at least in western cultures) … are visual learners."[18]

Inductive/Deductive

The inductive/deductive dimension deals with how a learner organizes information. An inductive learner prefers to work from specifics and derive the generalities, while a deductive learner starts with the generalities and applies them to the specific situations they encounter. Thus, an

---

[15] (Felder, 1993)
[16] For examples of engineering students illustrating the sensory and intuitive aspects, see Felder's character sketch of two students, one of whom (Stan) is a sensory learner and the other (Nathan) is an intuitive learner, in his "Meet Your Students: 1. Stan and Nathan" article (Felder, 1989) available at http://www.ncsu.edu/felder-public/Columns/Stannathan.html
[17] (Felder & Silverman, 1988)
[18] (Felder, 1993)

inductive learner generally likes to be given a set of facts, observations, or an example and then to tease out the fundamental principles that support the example. Deductive learners prefer to learn the basic principles and then determine how best to apply these principles to situations they encounter.

Induction is described by Felder and Silverman as "the natural human learning style. Babies don't come into life with a set of general principles but rather observe the world around them and draw inferences… Most of what we learn on our own (as opposed to in class) originates in a real situation or problem that needs to be addressed or solved, not in a general principle…"[19] Inductive learners will often need to see the motivation for learning a piece of information before they can learn it and they need to see an event before they can understand the underlying theory about it.

At the other end of the continuum, Felder and Silverman describe deduction as "the natural human teaching style, at least for technical subjects at the college level. Stating the governing principles and working down to the applications is an efficient and elegant way to organize and present material *that is already understood*." Deductive learners learn best by starting at the fundamental principles and then learning the applications of these principles to real life and the problems they encounter.

Active/Reflective

People differ in how they process information once they have received it. Some people need to use the information right away for it to stick in their memories, while others need to think about the information and figure out how it fits into their mental framework before they can use it. The Active/Reflective dimension in the Felder-Silverman model covers this difference.

Active learners want to do something with information as soon as they get it. They might discuss it with others, either as peers or by explaining it to someone else, or they might experiment with the information they have received. They tend to like to work in groups and like to find solutions that work and in general are the people who design and carry out the experiments. If active learners had a trademark phrase, it could well be "Let's try it out and see what happens."

Reflective learners prefer to think about information before they use it. They prefer to work alone or with at most one other person who they trust. They need time to mentally manipulate the information to see what they can get from it. In general, reflective learners are the people who define the problems that need to be solved. The trademark phrase for reflective learners could be "Let's think it through first".

Sequential/Global[20]

The final dimension in the Felder-Silverman model is that of sequential versus global learning. This dimension deals with how learners "get" the information they are learning. Sequential

---

[19] (Felder & Silverman, 1988)

[20] For examples of engineering students illustrating the sequential and global aspects, see Felder's character sketch of two students, one of whom (Susan) is a sequential learner and the other (Glenda) is an intuitive learner, in his "Meet Your Students: 2. Susan and Glenda" article (Felder, 1990) available at http://www.ncsu.edu/felder-public/Columns/Susanglenda.html

learners learn material in a logically ordered progression, learning little bits as they go, and incrementally building on the knowledge they have already learned. Global learners, however, tend to learn in chunks. They will spend some time being lost, then suddenly everything will come together and they will understand the concept.

For sequential learners, each piece of information builds logically on the previous ones. Sequential learners are strong in convergent thinking and analysis, bringing ideas in together. They follow "linear reasoning processes" when they solve problems, and their solutions are often the sort that make sense to other people. Sequential learners often have little trouble in school, as they learn best when material is presented with increasing complexity and difficulty and they can work with material that they only partially or superficially understand.

Global learners instead tend to see the big picture. They spend a period of time not understanding the material, but then a critical piece of information arrives and everything falls together for them. Global learners tend to be more apt to see connections beyond those presented (often to completely different disciplines than the one they are learning in at the moment). When a global learner is solving a problem, she may seem to leap directly to the solution (possibly skipping intermediate steps) and be unable to explain how she got there to other people. Global learners tend to need to be able to fully understand the material before they can work with it, however, and this can lead to problems in school. Once they have this understanding, they can very quickly assimilate additional related information and often are strong in divergent analysis and synthesis.

Since the publication of their original paper in 1988, Felder has made two changes to the model. The first change was the dropping of the "Inductive/Deductive" continuum. We believe that this continuum provides insight for exploratory testers and so will treat the model as if it still contained this continuum for our purposes. The second change Felder made to the model was a word change for the Visual/Verbal dimension. Originally, this continuum was called Visual/Auditory. We will use the current terminology (Visual/Verbal) throughout the rest of this paper.[21]

Applications to Exploratory Testing

Now that we have reviewed the Felder-Silverman model, we can apply this to exploratory testing, looking at the potential exploratory styles of someone who has a strong preference for a given aspect (and ignoring, for now, the interaction of aspects from different continua). Again, it should be stressed that there is no superiority or inferiority implied in a specific aspect or the lack thereof. Each aspect brings strengths and weaknesses to exploratory testing, and the well-balanced test team will have members whose learning styles complement each other.

Sensory/Intuitive

So, how would a person who was strongly sensory-based or strongly intuitive-based approach exploratory testing? The sensory-based person (who you will remember likes details and well-defined solutions to problems, while preferring information gained from his senses) might focus on his actual observations of the software. The intuitor (with her preference for internally

---

[21] Readers interested in the reasoning behind these changes should read the preface to (Felder & Silverman, 1988)

generated information) might then focus instead on her internal model of the software she is testing.

The two testers will probably also vary in their approach to performing their testing. The stereotypic sensor will generally apply "rules and tools" – solutions that have worked in the past for specific bugs that he can apply in his current testing to determine whether a particular bug exists. His testing would take the form of a series of experiments on the application. These experiments will tend to be of the form "Does this specific bug exist". A strongly sensing tester may also be more likely to begin testing the product before he creates any models of the software (mentally or otherwise). He is more likely to consult the specification and other reference material, and to experiment with the conformance of the documentation and the product. The learning done by a strong sensor is apt to be more based on experiencing the product. Given the sensor's preference for well-defined standard solutions, he is probably going to be more inclined to develop a standard pattern for approaching exploratory testing. This pattern can develop into a mental script, shifting the tester's focus from exploratory testing to scripted testing.

The stereotypic intuitor is more likely to approach the problem instead by applying different theories of error to the software. She will take a risk-based approach to her testing, thinking of ways in which the software can fail and then thinking of tests which will show whether the software actually does fail in that manner. This is different from the sensor's testing for specific bugs in that the intuitor is not focusing on bugs she has seen in the past. Instead, she is using her experience and understanding of the application she is testing to think about all the different failure modes of the application. It is a subtle difference between the two—a difference primarily of level of thought. The sensor is taking specific examples of bugs and checking for them. The intuitor is looking for more general possibilities of failure (each of which may have multiple bugs associated with it) and then deriving tests that could trigger a particular failure mode. She will usually like it when her mental model of the software is proven to be incorrect, often viewing the act of bringing her model back in line with reality as a challenge to be tackled with great relish.

The intuitor is also more likely to begin testing by building a model of the software. This model could be a state-chart, a mapping of the software to its market, or some other representation of some portion of the system. While the sensor is perhaps doing research designed to predict the behavior of the system, the intuitor might instead be doing research to define and then refine her models, with the intention of then evaluating the model against the product.

Finally, the two testers will most likely find different exploration styles. The sensor could find the various attacks described by Alan Jorgensen[22] and James Whittaker[23] appealing, while the intuitor might be more drawn to the modeling techniques described by Elisabeth Hendrickson[24].

---

[22] (Jorgensen, 1999)
[23] (Whittaker, 2002)
[24] (Hendrickson, 2002)

Visual/Verbal

The major difference between exploratory testers with a strong preference for visual learning and those with a preference for verbal learning might reflect the internal mental model that the testers use. Visual learners will tend to work off an internal model that is picture-based. This model could be a set of UML diagrams, flowcharts, or even mental screenshots. They will also tend to work off visual portrayals of the steps in the tests they are executing. These portrayals may run like a movie in the visual learner's head. Alternatively, visual learners may make diagrams and pictures for their notes as they explore.

Verbal learners would instead use a textual model for their testing. These models might take the form of a textual description of the system (perhaps a textual use case format) or they may take the form of a remembered conversation. The model will be based around words – the tester will use words to describe the system to themselves and words to describe each step in the process.

In addition to their internal models, these testers may also differ in the types of specification documents they try to get from their analysts and developers. Visual testers probably will be more comfortable working with visual models of the system – the state charts, UML diagrams, flowcharts and other representations the people designing and building the software use to help clarify things for themselves. Verbal learners, on the other hand, are likely to be happier taking the textual specification for the system and wading through it, learning as they go.

Inductive/Deductive

An inductive learner may adopt an approach to testing where she gathers as many specifics (such as techniques, potential defects, changes made to the application, and application history) as possible and generalize them to the application. A deductive learner might instead approach testing by keeping a collection of general principles and heuristics and find ways to specifically apply these generalities.

The inductive learner will likely take advantage of historical data – looking at the available defect reports, the technical support database, published articles about the software being tested and about similar programs, and any other historical documents that she can get a hold of. From these documents, the inductor will derive a set of specific guidelines that she then can use to guide their testing. For example, the application being tested may have a history of defects in one particular area. An inductor would take the specific fact of the large number of defect reports and generalize it to show that there could still be a large number of defects remaining in that application area, and thus focus more attention on that area than on another area which has had no defects reported historically. While an inductive learner is using heuristics for this approach, it is less apt to be a deliberate usage than we believe the deductive learner will have.

The deductive learner starts with a collection of general heuristics and guidelines and then consciously applies them to the application. Many of the traditional techniques of software testing are deductive – the tester learns the basic skill (such as equivalence partitioning) and then determines how to apply it in the specific situation of her testing.

We expect there will also be differences in how deductive and inductive testers use bug taxonomies[25] and risk lists. We expect the deductive tester to gain familiarity with the categories and then be able to come up with new examples within those categories. The inductive tester, on the other hand, is expected to gain an understanding of the various list elements and then be able to see new ways of categorizing them.

Active/Reflective

Active and reflective testers differ most in how they execute tests. An active tester will usually do very hands-on testing. She often will perform many test cases rapidly and will view each test case as an experiment, asking, "What happens if I do this?" each time. An active tester will also tend to be more visibly a part of a testing group, often bouncing ideas and results off other members of the group to solicit their feedback.

A reflective tester, on the other hand, is apt to do far fewer tests. A thought process will precede each test case where the tester is thinking through the test. Reflective testers make up for their lack of speed in test execution by executing the "good" tests that are most likely to find bugs, however. Most reflective testers will probably tend to prefer to work alone or with at most one other person, and so may seem anti-social or outside the group. This isolation and thinking should give them the time to develop more complex tests and scenarios to apply to the application, and thus they should be encouraged to take the time they need.

Sequential/Global

The last pair of aspects we have to consider is the sequential and global aspects. A sequential learner builds information and knowledge in a logical progression, while a global learner needs critical pieces of information in order to get the understanding of the subject.

The sequential tester will seem to get off to a faster start. He will build test plans as he goes, step by step. Not having a piece of information will not normally prove to be a problem for a sequential tester, as he will work with the information that he does have. He also will be able to explain his tests clearly to people after he has performed them. In general, a sequential tester's test cases will grow in complexity over time as he builds a deeper understanding of the system.

A global tester will get off to a slower start. He may have problems understanding the point of the application (or their area within it) and need to be shown how to use the application in order to have any idea how to test it. Once he gets the piece of information that brings it all together for him, however, he quickly becomes able to create detailed, complex tests that often draw on connections that other people on the testing team have not seen.

Wrap-up

We are interested in the Felder-Silverman model of learning styles because it gives us hints about why different testers adopt different exploratory strategies. If those hints are validated, we will be better able to create effective sets of lesson plans and exercises to train explorers. Testers with exploratory testing experience might be more able to discover their blind spots or identify

---

[25] A taxonomy is a listing with the elements broken up into categories. For more information, see (Vijayaraghavan, 2002, 2003)

techniques that would not otherwise occur to them. Test managers will be more able to determine where there may be gaps in their testing teams and what kinds of skills they need to bring in to balance the team.

Much work remains to be done before the impact of learning styles on exploratory testing is completely understood. Each individual aspect needs to be explored in more detail for connections. The aspects must be looked at in combination (for example, how is someone who is strongly intuitive AND strongly verbal going to differ from someone who is just strong in one of those two aspects?). Experimentation must be performed to validate the claims. We believe it will be an exciting process; one that we hope will attract the interest of many people in the field. As we further our research, we will be documenting our research in two places – papers on the lab web site (http://www.testingeducation.org) and in our weblogs (at http://blackbox.cs.fit.edu/blog/andy and http://blackbox.cs.fit.edu/blog/kaner).

References

Agruss, C., & Johnson, B. (2000). Ad Hoc Software Testing: A perspective on exploration and improvisation. (*http://www.testingcraft.com/ad_hoc_testing.pdf*)

Bach, J. (1999, August 26, 1999). *General Functionality and Stability Test Procedure*. Retrieved July 29, 2003, from http://www.satisfice.com/tools/procedure.pdf

Bach, J. (2003). Exploratory Testing Explained. (*http://www.satisfice.com/articles/et-article.pdf*)

Drabick, R. (1999, October 27, 1999). *Growth of maturity in the testing process*. Retrieved July 29, 2003, from http://www.softtest.org/articles/rdrabick3.htm

Felder, R. M. (1989). Meet Your Students: 1. Stan and Nathan. *Chemical Engineering Education, 23*(2), 68-69 (*http://www.ncsu.edu/felder-public/Columns/Stannathan.html*)

Felder, R. M. (1990). Meet Your Students: 2. Susan and Glenda. *Chemical Engineering Education, 24*(1), 7-8 (*http://www.ncsu.edu/felder-public/Columns/Susanglenda.html*)

Felder, R. M. (1993). Reaching the Second Tier: Learning and Teaching Styles in College Science Education. *Journal of College Science Teaching, 23*(5), 286-290 (*http://www.ncsu.edu/felder-public/Papers/Secondtier.html*)

Felder, R. M. (1996). Matters of Style. *ASEE Prism, 6*(4), 18-23 (*http://www.ncsu.edu/felder-public/Papers/LS-Prism.htm*)

Felder, R. M., & Silverman, L. K. (1988). Learning and Teaching Styles in Engineering Education. *Engineering Education, 78*(7), 674-681 (*http://www.ncsu.edu/felder-public/Papers/LS-1988.pdf*)

Hendrickson, E. (2002). A Picture's Worth a Thousand Words: How to create and use diagrams and maps to improve your testing. *STQE, 4*(5), 26-32

Jorgensen, A. A. (1999). *Software design based on operational modes*. Unpublished PhD., Florida Institute of Technology, Melbourne, FL.

Kaner, C. (2002, Summer). *A Course in Black Box Software Testing (Professional Version)*, 2003, from http://www.testingeducation.org/coursenotes/kaner_cem/cm_200204_blackboxtesting/

SWEBOK. (2003, May). *Software Engineering Book of Knowledge*. Retrieved July 29, 2003, from http://www.swebok.org

Tinkham, A., & Kaner, C. (2003, May 15). *Exploring Exploratory Testing*. Paper presented at the STAR East 2003 Conference, Orlando, FL, USA (*http://www.testingeducation.org/articles/exploring_exploratory_testing_star_east_2003_paper.pdf*)

Vijayaraghavan, G. (2002). *Bugs in Your Shopping Cart: A Taxonomy*. Retrieved July 30, 2003, from http://www.testingeducation.org/articles/bugs_in_your_shopping_cart_a_taxonomy_paper_isqc_sep_2002_paper.pdf

Vijayaraghavan, G. (2003). *Bug Taxonomies: Use Them to Generate Better Tests*. Retrieved July 30, 2003, from http://www.testingeducation.org/articles/bug_taxonomies_use_them_to_generate_better_tests_star_east_2003_paper.pdf

Whittaker, J. A. (2002). *How to Break Software*. Boston: Addison Wesley.

Acknowledgements

The authors would like to thank the following people for their assistance with this paper:

- James Bach
- Kit Bradley
- Sue Carlson
- Elisabeth Hendrickson
- James Lyndsay
- Jean Marchant
- Brian Marick
- Max McNaughton
- Barb Nolan
- Erik Petersen
- Alan Richardson
- Dianne Runnels
- Melissa Strader
- Ruku Tekchandani
- Heather Tinkham

Common Mistakes in Test Cases:
Separating the Weak from the Strong

Claudia Dencker
Software SETT Corporation
233 Oak Meadow Drive
Los Gatos, CA 95032
408-395-9376
cdencker@softsett.com

Abstract
Test cases are often the weak link or Achilles heel in a tester's knowledge kit. Why? Most individuals haven't been trained in how to write really good tests. They don't know the difference between a good or bad (strong/weak) test case. This paper outlines five common mistakes made in test case writing. Through the use of examples, this paper provides a full explanation of each mistake, how it can be corrected and some additional tips to consider.

With practice, designing test cases goes quickly and intuitively and represents one of the most creative aspects of a tester's responsibilities. Avoiding these five basic mistakes will go a long way towards fostering discipline in the software testing profession as well as providing a fantastic test effort.

Author Bio
Ms. Dencker is President of Software SETT Corporation, established in 1987 and specializing in SQA services for client/server applications, embedded systems, and web applications. She has taught classes worldwide through the IEEE and to major Silicon Valley companies. She also participates on software testing projects in either an advisory or hands-on role. Ms. Dencker received a Bachelors degree and a teaching credential from San Jose State University.

Introduction

Years ago I was involved in a project where my team and I wrote an extensive set of test cases. These tests reflected the output of a small group of senior QA professionals who did a fairly good job of covering the product and ensuring depth of testing. As part of our project conclusion, we stored all test files on the client's project server and handed off documentation to the test lead.

Several months passed and we were asked to return and help with the testing of a follow-on release with the same client. When we came back and as part of our project ramp-up, we were informed that our test files from months back couldn't be located. So the team (of untrained test designers) created test cases themselves. After quickly finding the test files that we had so carefully stored and handed off, we then reviewed the more recent efforts. The difference in coverage and quality was shocking.

Some of the mistakes that the follow-on team did were unique to their experience. Whereas many of their mistakes have been made consistently by others. As such, we have come to the conclusion that test cases are often the weak link or Achilles heel in a tester's knowledge kit. Why? Most individuals haven't been trained in how to write really good tests. They don't know the difference between a good or bad (strong/weak) test case. So, let's go back to basics.

What is software testing? According to Glenford Myers in his book, *The Art of Software Testing*, he defines testing as "the process of executing a program with the intent of finding errors." Another excellent source, *Testing Computer Software* by Cem Kaner, Jack Falk, and Hung Quoc Nguyen, defines software testing as follows: "Finding problems is the core of your work. You should want to find as many as possible; the more serious the problem, the better." Both sources identify problem finding as the key focus of software testing. As such our test cases should be built with the same focus of finding problems in the software.

But, what is a test case? A test case according to the IEEE Glossary of Terms is defined as follows: "A test case is a specific set of test data and associated procedures developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement."

A test case consists of four basic elements:

- ***Test intent.*** A test intent is a statement of what to test. It answers the question, what am I trying to prove?
- ***Test data.*** Test data is a specification of the data elements, values or set that define how to satisfy the test intent. It answers the question, what do I need? Exact data points do not need to be identified at the test case design stage. Rather, the specification should be clear enough to allow the testers to make their own determination at the time of the test run.
- ***Brief action steps.*** Brief action steps are the basic instructions required to perform the test. It answers the question, what do I have to do? These should be fairly high level as the software is typically changing during test case design. This will help to minimize extensive modifications later.
- ***Expected results.*** Expected results are a list of results that you expect to happen. They answer the question, what can I expect as a likely result? How do I know that the test has passed?

So what is a good test case? Let's focus for a moment on what comprises good test cases. Good design doesn't happen in a vacuum. Rather test design is preceded by test planning and test strategy definition. Without these the test designer cannot create tests that have a high likelihood of finding a problem. Therefore, a good test case is one that finds a bug. Here is a set of characteristics that I look for when reviewing tests:

- Interesting test objectives and test data ranging from simple to complex to ensure a robust test, not just simple and easy.
- Expected results that are extensive and broad and not incomplete.

- Tests that are free from unnecessary interdependencies, but necessary dependencies that are clearly stated to ensure smooth test execution.
- Broad mix of invalid tests to enhance the likelihood of finding bugs.

In conclusion, a good test set is one that is well documented to enhance maintainability and well defined (including well written) to enhance reusability at a later time.

What are the problems with test cases? Why is it that some tests find lots of bugs, whereas others do not? The answer to this question lies in the fact that many test designers make common mistakes that can easily be avoided. If we, as software test professionals, want to avoid being accused of inadequate testing, then we should look at our test set. This paper discusses the following five common mistakes that compromise a test case and lists several others that the reader should note.

- Test intent/objective is an action step, not an objective.
- Test data drives the test intent.
- Expected result is incomplete or too cryptic.
- Test cases have unrealistic interdependencies.
- Not enough invalid tests in the test set.

Mistake #1: Test intent/objective is an action step, not an objective.

On the surface, this mistake appears to be minor. Many test intents are written as action steps, "Test this, test that." and not as objectives. But, by just changing the sentence from an imperative to a declarative one, we make a monumental shift in our thinking.

For those of you who have forgotten your English grammar and structure, a declarative sentence is one that makes a statement that is true or false and can be verified as such. "The sun shines 24 hours a day, 7 days a week." or "A message in the Inbox can be deleted." An imperative sentence (one that is most typically found in test intents) gives a command or makes a request. "Delete a message." Or "Check the weather." There are two other types of sentences, interrogative and exclamatory, but these don't have much bearing in test cases so we'll ignore them for now.

The reason why we want to write our test intents in a declarative manner is so that the tester will understand the purpose of the test. If the test intent is written as an imperative statement ("do this, do that") indicating action to be taken, the tester will not understand the purpose nor will he or she be able to derive the purpose. Worse yet, the tester may resort to guessing the purpose, through the action steps, which is error prone and compromises the test effort.

Understanding the purpose of the test is extremely important, because if the software changes (which it likely will), the actual result will not match the expected result. Also, if other test case components are foggy in the tester's mind, he or she will be unable to correctly execute the test case as the test designer envisioned it.

Here is an example of a poorly written test case from an e-mail test:

Test Intent	**Action Steps Required**	**Test Data**	**Expected Results**
Delete a message.	Delete a message.		Delete option grayed.

There are a number of things wrong here:

- ***Lack of detail in intent.*** Not only is the test intent written as an action step (imperative sentence) and not as a declarative sentence, it is missing a lot of detail. For example, other questions come to mind as the test intent is currently written:

1. What particular message should be deleted: big/small, first/last, attachments/no attachments, etc.?
2. Does it matter in what object the message is found: Inbox, Sent Items, Trash, Drafts?
3. How should the deletion be executed: off the toolbar, keyboard shortcuts, etc.?
4. What is the state of the software at the time of the message deletion: after upgrade, after start-up, after continuous use?
5. Are there any special user settings that should be active at the time of the deletion?

- ***Mismatch between the intent and expected result.*** The intent in our example implies that a message is present that can be deleted; yet the expected result implies that there are no messages. How will the tester know whether the intent or expected result is correct based on how the intent is written? **Hint:** Westerns read left to right and as such, will assume that the test intent is correct and the expected result is wrong. Remember, it could be the other way around.

- ***Simplistic test intent.*** The test intent in our example is rather simple. In my review of this case, I would look for other interesting conditions, such as checking boundaries, large messages versus simple ones, etc. This test case might be expanded into several others or be rewritten to include a more interesting condition.

- ***Missing test data.*** Where is the test data? Better yet, how can the test data be properly selected or created if the purpose of the test is unclear?

By writing your test intents as imperative, action sentences, there is also the notion that testing is all doing and not thinking. Again, the distinction is important. Testing is the process of finding problems. Tests should be designed to probe the software so that we find the failures before the software is released, not after. Some thought needs to go into this process to ensure that we meet our goals. All action and no planning (i.e., thinking) results in poorly executed projects.

Let's look at our bad test intent and see how it can be improved.

Test Intent	**Action Steps Required**	**Test Data**	**Expected Results**
A message can be deleted from the Inbox using a variety of interface options.	• Select a message. Click Delete from the Edit menu. • Repeat the test only press the Delete key. • Repeat the test using the Delete icon button on the toolbar.	Converted legacy messages should appear in the Inbox.	After each deletion, the message is deleted from the Inbox. It appears in the Deleted Items. **NOTE:** To restore a deleted message, open the Deleted Items folder and then move the message to the Inbox.

Here are some quick tips on good test intents:

- Write the intent as a declarative sentence, much like a scientific hypotheses, indicating what you are trying to prove true or false.
- A test intent could start out with "Verify that …" to help you get started, because what follows is a declarative sentence.
- A well-written test intent could be enough for a seasoned tester to do his or her testing. This technique is useful if you are looking for some structure during an ad hoc or exploratory test session.
- A test intent is the "right" size. This means that the intent is robust and interesting, but not so complex that it is hard to validate and maintain. Conversely, the test intent is not so cryptic that its meaning is lost over time or so simple that the chance of finding a problem is remote.

Mistake #2: Test data drives the test intent.

When reviewing the components that comprise a test case, only one takes precedence over others, the test intent. Occasionally, this rule is forgotten, and we find ourselves writing or reviewing tests that are driven by the test data where the test data takes precedence. When this happens, our tests become too small or "thin" and loaded with repetition.

Here is an example of test cases that are data driven.

Test Intent	Action Steps Required	Test Data	Expected Results
A message with NORMAL delivery and no attachments can be created and saved.	Create e-mail message. Upon message completion, save the message. • Be sure to designate the Msg ID as the Subject. • Message length does matter. • For internal addressees, refer to the destinations in the Global Data Setup. Across all the messages, vary the destinations. • For external addressees, refer to the destinations in the Global Data Setup. Across all the messages, vary the destinations. • Be sure to include yourself on any one of the To, Cc, Bcc destinations. • Set the message priority to normal.	One internal addressee	A message with the specified header information **and NORMAL** delivery can be created and saved in the Drafts folder.
Ditto above	Ditto above	One external addressee	Ditto above
Ditto above	Ditto above	Many internal addressees	Ditto above
Ditto above	Ditto above	Many external addressees	Ditto above

There are a number of things wrong here:

- ***All four tests are essentially the same*** as evidenced by the same expected results, the same action steps and the same test intents. Designers sometimes have difficulty capturing the variations of an intent. One inefficient way to do this is by repeating the test intent to capture the nuances in the test data. This leads to the following additional problems:
    1. This wastes precious testing time and doesn't enhance the tester's ability to find more defects.
    2. This bloats our test set unnecessarily, which leads to incorrect metric measurements and reporting. Management will be very impressed when you run a large number of tests in a short amount of time. But, your test set may not hold muster under more careful scrutiny.

- ***Data specification missing.*** At the time of test case creation, it is unwise to point to a specific data item that is to be used. Your test data may not be available yet, so it is premature during test design to identify the exact element. Rather, a better practice is to specify the characteristics of the data that

you feel best satisfies the test intent. In fact, it may be better to create a data specification for a range of tests so that the tester is guided appropriately during test execution.

- ***Action steps repeated throughout.*** When action steps are repeated for each test case or are very similar, you should consider placing the information outside the test case itself and into a general section called General Instructions, and then reference this section where appropriate. In fact, this applies to any other assumptions, setup information or other information that applies to more than one test case.

Let's look at our bad test data example and see how it can be improved:

Test Intent	Action Steps Required	Test Data	Expected Results
Internal and external addressed messages of various lengths, NORMAL delivery, and without attachments can be created and saved.	Create each message as specified in the next column following the criteria below. Upon message completion, Save the message. • Be sure to designate the Msg ID as the Subject. • Message lengths should vary from one-liners to long messages that overflow a screen. • For internal addressees, refer to the destinations in the Global Data Setup. Across all the messages, vary the destinations. • For external addressees, refer to the destinations in the Global Data Setup. Across all the messages, vary the destinations. • Be sure to include yourself on any one of the To, Cc, Bcc destinations. • Set the msg priority to normal.	• **Msg 1** = one internal To, Cc, Bcc addressee, no attachments • **Msg 2** = one external To, Cc, Bcc addressee, no attachments • **Msg 3** = many internal To, Cc, Bcc addressees, no attachments • **Msg 4** = many external To, Cc, Bcc addressees, no attachments	A message with the specified header information **and NORMAL** delivery can be created and saved in the Drafts folder.

As you can see, this example took the four, standalone test cases from our bad example and with very slight modification to the test intent, created one test case to which our test data of four messages applies. This example was aided by putting together a short data specification on the types of e-mail messages that could be written (see below). We analyzed an e-mail's characteristics of header information with internal (behind the firewall) and external (outside the firewall) addressees, delivery speed of normal and attachments. This data specification can be expanded to include other characteristics, such as message size, default user settings, other delivery speeds (urgent, low) and more, but has been restrained for explanation purposes.

NORMAL	**TO**				**CC**				**BCC**			
delivery	one addressee		many		one addressee		many		one addressee		many	
	Int.	**Ext.**	**Int.**	**Ext.**	**Int.**	**Ext.**	**Int.**	**Ext.**	**Int.**	**Ext.**	**Int.**	**Ext.**
No attach-ments	Msg1	Msg2	Msg3	Msg4	Msg1	Msg2	Msg3	Msg4	Msg1	Msg2	Msg3	Msg4
With attach-ments	Msg5	Msg6	Msg7	Msg8	Msg5	Msg6	Msg7	Msg8	Msg5	Msg6	Msg7	Msg8

Here are some quick tips on developing good test data:

- Test data sources can be many, but try not to create your own test data manually. Look for existing sources in production; create it in an automated fashion even if you require the assistance of development, or leverage off the developer's test data.
- Always review the test data to ensure that it supports the test intent, not the other way around. Remember that the test data doesn't need to be a single element, but it can be part of a dataset.

Mistake #3: Expected result is incomplete or too cryptic.

Another common mistake can be found in expected results. Typically most tests contain rudimentary expected result information. Where they fall short is in their robustness. Quite often expected results are "thin;" they expect the tester to look just "one" way or in "one" place rather than to look in many different places or in many different ways. By not extending the expected results, we miss out on opportunities to continue our testing and find additional defects that are worth reporting.

So, how do we improve our expected results? This can be an uphill climb because there is a natural tendency to do a quick job of the expected results for the following reasons:

- The design effort can be rushed in favor of testing itself, i.e., feeling the pressure to act or test before all design or thinking is done.
- The tester allows himself to be convenienced, thinking that it isn't necessary to be thorough; the developer will catch the multiple instances where a bug might be exhibited during the fix process.
- The software is not done or the specification is unclear so the test designer may not know what is supposed to happen.

Let's look at our first example again.

Test Intent	**Action Steps Required**	**Test Data**	**Expected Results**
Delete a message.	Delete a message.		Delete option grayed.

There are more things wrong here:

- ***Lack of depth in expected result***. The current expected result is very brief and leaves open more questions than it answers.
    1. Are there other state or timing factors in the validation process?
    2. Is there anything else that should happen even though the option is grayed?
    3. Can the tester look anywhere else to ensure that the deletion should not take place?
    4. Should other deletion mechanisms not work as well: DEL key, keyboard shortcut, etc?

- ***Mismatch between test intent and expected result.*** As mentioned in the first mistake, the intent implies that a message is present, yet the expected result implies that there is no message present. There is a mismatch here that is confusing. How will the tester know which one is correct, the intent or expected result? One must change so the test case makes sense, but if the test designer is unavailable, the tester will have to make the adjustment without proper guidance. Guessing someone else's intentions during test execution is unfortunate and does not support good practices.

  Well-written expected results benefit the test as follows:
    1. Provide more information and insight into the overall expectations of the test case, which leads to better testing.
    2. Serves as a "book-end" to the test intent providing closure on the test case.

- ***Missing test data.*** If the test data specification is missing, the tester doesn't know where to look. If it were present, the data specification could potentially shed some light on how the expected results could be more properly validated.

Let's look at our bad expected result example and see how it can be improved:

Test Intent	Action Steps Required	Test Data	Expected Results
All messages can be deleted from the Inbox. **NOTE:** Be careful here, you may lose all your Inbox msgs.	Select all messages and click Delete icon button in the toolbar.	Converted legacy messages should appear in the Inbox.	All messages are deleted from the Inbox and the messages appear in the Deleted Items folder.

As you can see, more information is provided that points the tester to the objects (Inbox and Deleted Items) in which to verify the result. Also, the test case has been improved to include an upper boundary test; all messages can be deleted.

Years ago when I actually ran this test, I encountered an overflow condition. No matter how hard I try, I tend to over-accumulate messages in my Inbox. I knew that when I delete messages, I could Undo the action or retrieve the deleted messages from the Deleted Items folder (major mistake – just enough knowledge to get into trouble). Unfortunately, when I checked this test out on my own, live e-mail account (another major mistake – testing on production), I had over 500 various sized messages in the Inbox. I deleted all of them, overflowed the buffer and essentially lost about 85% of the messages.

As currently written, this test is a valid, upper boundary test, but it uncovered a colossal failure.

Moving on, here are some quick tips on developing good expected results:

- Expected results should ensure that the tester validates in as many places and ways as makes sense. The three "w" questions should be addressed:
    1. ***Where*** should the tester look? The expected result should indicate the screen, report, database query, log file or other location where the tester should look to ensure that the test has met expectations and therefore, passed.
    2. ***What*** to look for? The tester should ensure that a specific value or string, condition or state or lack thereof is present (or not present) as directed by the test case.
    3. ***When*** to look? At what specific point in the process should the tester conduct their validation? This final criterion only plays a role if the test involves timing of some kind. For example, in one of our payroll tests, we were unable to validate the results of our tests until after two specific batch runs had occurred. This led to some interesting validation efforts, because we would often have a delay of up to three days before we could close our tests and determine whether they passed or failed.
- Map the expected results very closely to the test intent. In fact, when following good test case practices as outlined in this paper, sometimes the intent and expected results can be interchangeable. (See good example above.) I often think of the test intent and expected results as my test "bookends."
- Make sure that expected results don't just check the obvious. Look for hidden opportunities and interactions during the final checking of a test case.

Mistake #4: Test cases have unrealistic interdependencies.

When designing a set of tests, it can be easy to develop tests that are overly complex and interdependent which leads to difficulty during test execution and during maintenance after release. One of the basic guidelines to keep in mind when designing tests is to keep the cases as standalone as possible. There are some benefits to this:

- Tests can be executed in any sequence during subsequent test cycles, which makes the test set flexible.
- Tests are easier to maintain because all the information is in one or two places rather than in multiple locations, which could result in obsolete pointers.
- Tests have an increased likelihood of being closer in weight or size, which lends greater confidence to the schedule when scheduling by execution time per test case.

Some of the interdependencies in test cases that should be avoided are as follows:

- A test case requires a state change that only a prior test case can generate.
- Tests that must run first do not appear first in the list. Let me explain. While tests should be standalone items, occasionally one or more tests need to run first because of a unique software state. Humans tend to read top/down and as such, will run tests that start at the beginning of the list on down. If there is a requirement to run the tests in a certain sequence, then the execution plan should present the standalone items in their desired sequence.
- Tests are not returned to their correct starting state, thus causing the follow-on tests to start incorrectly.
- Test data is not returned to its original starting state, which will cause the follow-on tests to fail.
- Test data is changed in one test and acts as input for another test. This is a big no-no and should definitely be avoided. If you must do this, be sure to clean-up the test data as soon as possible so other tests can run properly.

Here is an example of tests that are unnecessarily dependent.

Test Intent	**Action Steps Required**	**Test Data**	**Expected Results**
With the message from test case 4 open, change the font.	Change the font.	Msg from case 4	Message text should change from a size of 10 to 12.

There are a number of things wrong here:

- ***Presence of a dependency.*** This test case assumes that data from test case 4 is accessible and that its state is currently present (or has to be run in order to generate the state). This may be difficult for the tester to achieve for the following reasons:
    1. Test #4 is run by someone else
    2. The state is difficult to create (some states can be nearly impossible to create)
    3. Running test 4 causes problems for other tests. The result or state created by test case 4 should be spelled out for the tester so he or she can determine the best way to create the pre-state.

- ***Lost pointers.*** What happens if tests are renumbered, and test case 4 is now #6?

Let's look at how our test case example can be improved:

Test Intent	Action Steps Required	Test Data	Expected Results
With a message open in edit mode, change the font size through any available means being sure to include min/mid/max values.	• Select a font from the drop down menu in tool bar. • Repeat test for other means: 1. Key a new value in the font field. 2. Select from the format/font dialog box.	• Smallest visible size (5) • Largest visible size (60) • Common size (10, 12)	Message font size should change as directed by the test data using both interface methods.

The only dependent state for this test case was that the message had to be open. This is an easy condition to reproduce, and as such, has been folded into the test intent. As you can also see, this improved test case has a data specification with suggested values, action steps that embrace two methods for executing the test intent and boundary checks (min/middle/max).

Here are some quick tips on developing standalone test cases:

- Test cases are read top/down and left to right if you use a table or matrix format. If you use a document-style template, then cases are typically read top/down.
- Test cases in a larger test set should start and stop at the same state so they can be used in any sequence. Remember, test cases are design documents as well, and don't always lend themselves to test execution. Though, I might add, as a seasoned QA professional I have done excellent testing from design documents when test intents are well written.
- Test cases should be self-contained with the only pointer referencing the general set-up section that precedes the list of cases.
- If cases need to be numbered, keep numbering unique to each section/sub-section so new tests can be added easily or others removed without "ugly" gaps. Though, many test designers like to keep obsoleted tests around for archival purposes and use the font strikeout feature or row "graying" of the test cases to show that they are no longer active.

One final note: There may be times when referencing another test case for how to establish a desired state is reasonable. If the setup is complex and has already been described, it might be better to define the desired starting state in general terms and refer the reader to the previous details. This avoids duplication, and if the details ever need to be changed, avoids problems with not all sets of details being updated.

Mistake #5: There are not enough invalid tests in the test set.

Tests typically fall into two types: valid (positive) and invalid (negative). A valid test is one that is exercised by valid inputs, actions or conditions. The expectation is that the test will pass. An invalid test is one that is exercised by invalid inputs, action or conditions. The expectation is that the test will fail, but it should fail gracefully with an error message or some feedback mechanism that informs the user or underlying program that the invalid input was unacceptable. An invalid test will fail when the invalid condition is not trapped with error handling, but rather causes data loss, loss of program control or some other unacceptable result.

Many tests suffer from a lack of invalid tests. While it is easier to create valid tests, it is far more interesting and challenging to create invalid ones. In fact, invalid testing is the heart and soul of testing

because the focus is on conditions that are unexpected, unacceptable and in my book, fun. If we return to our original definition of testing, "testing is finding problems", the best way of finding them is to design invalid tests.

Designing invalid test cases is not easy, but you have some good sources available to you:

- Error message log. You can either pull it directly from the source code or have the developers provide it.
- Troubleshooting section in the reference guide or in help. If the software under test has been around for a while, user documentation should be available. Reference the Troubleshooting section for interesting failure ideas.
- Outside upper boundaries. If you know the valid range of boundaries for the software under test, then test outside the boundaries. A good bet for failure is directly outside the upper boundary (max+1) or way outside the boundary (max+n), such as when you are doing stress testing. Remember that not all boundaries are explicitly documented. Many boundaries are hidden or implied. It will take careful understanding of the software under test and reading of the specification to uncover these gems.
- User profiles/use cases. These profiles and use cases can shed light on how the software should be used and where likely failure points could be. Through a unique sequence of actions that cause invalid inputs we may cause failures.
- Outside immediate area of test. Don't forget to take the "big" view on your area for test and look for unintended ripple effects.

Here is an example of a test set that lacked invalid tests:

> Out of a set of 12 tests that tested rudimentary message preparation (deletion and viewing) of an e-mail system, only three invalid tests were written. Two of the tests dealt with an outside lower and upper boundary condition, and the third one tested against a null condition.

The main problem with this test set is that ***the ratio of invalid tests is too small.*** I like to aim for at least a 50/50 ratio, though this can be challenging and begs a larger question, are more invalid conditions even available or likely for the area of test represented by the set? One of the ways for increasing the number of invalid tests is a cross-functional brainstorming session. We did this several years ago by involving a user representative, business partner (super user), test consultant, developer, and system architect. Over two, back-to-back six-hour sessions we not only devised our test strategy, but we identified the tests that we needed. The test consultant then took all the notes and fleshed out the tests into full test cases. The document was reviewed with the outcome of tests organized into logical groups that were then assigned to team members for set-up and execution.

Here are some quick tips for developing invalid test cases or doing invalid testing:

- Think "outside the box." Coming up with invalid test cases requires creative and devious thinking that can lead to interesting conditions that may or may not have a basis in reality. Be careful with spending too much time on way-out corner cases that may be fun, but not likely.
- If you are tight on time and have to choose, focus on invalid test cases. By trying to break the software, you are more likely to find problems than when you run valid tests. Also, you have to typically set up the software correctly in order to get to the condition that allows you to generate the error or failure. Thus, you are covering some of the ground of the valid tests. I have written, executed and managed many test strategies along these lines to great success.
- Invalid conditions can be very difficult to design, let alone implement. So, brainstorm with others (cross-functionally is best) to identify your invalid conditions.
- Remember that developers, when given the time, do a fairly good job of valid testing, but they fall short on invalid testing. This can be easily explained in that it is difficult to check and test one's own

work. Developers are just too close to their code and blinded by their own experience. This further reinforces the need for independent testing that focuses on the invalid conditions.

Conclusion

In conclusion, this paper has attempted to point out basic mistakes in test cases as well as identifying other minor ones that, by avoiding them, will help you to create a strong and robust set of tests. With practice, designing test cases goes quickly and intuitively and represents one of the most creative aspects of a tester's responsibilities. Avoiding these five basic mistakes will go a long way towards fostering discipline in our profession as well as providing a fantastic test effort that will have a good chance of "living" long after the release.

How Fast is Fast Enough?

By Scott Barber, NobleStar
sbarber@noblestar.com

This presentation is targeted for all members of the project team, but will hold particular value to the Performance Test Engineers and the individuals responsible for collecting Performance Related Requirements. Collecting performance related requirements is often one of the most difficult parts of a performance testing project. Even if those requirements exist, converting them into meaningful, quantifiable and testable requirements is not a simple task. This highly interactive session will take the audience through the actual process of collecting requirements and converting them into performance test cases by building our own requirements for a demo application.

Scott Barber consults on and teaches practical performance testing and engineering for NobleStar, where he has completed dozens of engagements. These engagements have been evenly split between engineering performance for complex systems and mentoring organizations in the development of customized performance testing approaches based on his performance engineering methodology. His methodology has been widely adopted not only through client engagements, but also as the basis for his recognized article series' "User Experience, not Metrics" and "Beyond Performance Testing." Scott's speaking engagements for 2003 include PNSQC, PSQT/PSTT, the Rational User's Conference and local users' groups. He earned his master's degree in information technology, and was invited to be a guest lecturer at MIT based on his article "Automated Testing for Embedded Devices." Scott makes himself available to others by participating as a discussion leader for QAForums.com and the Rational Developer's Network. He is a member of the Context-Driven School of Software Testing and a signatory to the Manifesto for Agile Software Development. You can view Scott's resume and review his methodology and publications at http://www.perftestplus.com.

(This paper has been adapted from *"How Fast is Fast Enough",* Part 3 of the made for Rational Developers Network article series *"Beyond Performance Testing"*. The remainder of the currently completed articles in this series are available at www.rational.net and www.perftestplus.com)

How Fast is Fast Enough?

This presentation is targeted for all members of the project team, but will hold particular value to the Performance Test Engineers and the individuals responsible for collecting Performance Related Requirements. Collecting performance related requirements is often one of the most difficult parts of a performance testing project. Even if those requirements exist, converting them into meaningful, quantifiable and testable requirements is not a simple task. This highly interactive session will take the audience through the actual process of collecting requirements and converting them into performance test cases by building our own requirements for a demo application.

Scott Barber consults on and teaches practical performance testing and engineering for NobleStar, where he has completed dozens of engagements. These engagements have been evenly split between engineering performance for complex systems and mentoring organizations in the development of customized performance testing approaches based on his performance engineering methodology. His methodology has been widely adopted not only through client engagements, but also as the basis for his recognized article series' "User Experience, not Metrics" and "Beyond Performance Testing." Scott's speaking engagements for 2003 include PNSQC, PSQT/PSTT, the Rational User's Conference and local users' groups. He earned his master's degree in information technology, and was invited to be a guest lecturer at MIT based on his article "Automated Testing for Embedded Devices." Scott makes himself available to others by participating as a discussion leader for QAForums.com and the Rational Developer's Network. He is a member of the Context-Driven School of Software Testing and a signatory to the Manifesto for Agile Software Development. You can view Scott's resume and review his methodology and publications at http://www.perftestplus.com.

Scott Barber
2207 Cocquina Dr.
Reston, VA 20191

How Fast is Fast Enough?

(This paper has been adapted from *"How Fast is Fast Enough",* Part 3 of the made for Rational Developers Network article series *"Beyond Performance Testing"*. The remainder of the currently completed articles in this series are available at www.rational.net and www.perftestplus.com)

"You thought that was fast? I thought it was fast. Well, was it?"

– *Jodie Foster as Annabelle in the movie Maverick (1994)*

As a moderator of performance-related forums on QAForums.com, I've seen questions like this one posed numerous times:

"I'm desperately trying to find out what the industry standard response times are. What are reasonable benchmarks that are acceptable for Web sites at the moment? Is 1.5 seconds a reasonable target????"

My answer to questions like this one always starts with "It depends on . . . " My friend Joe Strazzere addressed this question particularly well, as follows:

There are no industry standards.

You must analyze your site in terms of who the customers are, what their needs are, where they are located, what their equipment and connection speed might be, etc., etc.

I suspect 1.5 seconds would be a rather short interval for many situations. Do you really require that quick of a response?

The bottom line is that what seems fast is different in different situations. So how do you determine how fast is fast enough for your application, and how do you convert that information into explicit, testable requirements? Those are the topics this article addresses.

Considerations Affecting Performance Expectations

Let's start by discussing the leading factors that contribute to what we think fast is. I believe these considerations can be divided into three broad categories:

- user psychology
- system considerations
- usage considerations

None of these categories is any more or less important than the others. What's critical is to balance these considerations, which we'll explore individually here, when determining performance requirements.

User Psychology

Of the three categories, user psychology is the one most often overlooked – or maybe a better way to say this is that user psychology is often overridden by system and usage considerations. I submit that this is a mistake. User psychology plays an important role in perceived performance, which is the most critical part of evaluating performance.

Consider this example. I recently filled out my tax return online. It's a pretty simple process: you navigate through a Web application that asks you questions to determine which pages are presented for you to enter data into. As I made a preliminary pass through my return, I was happy with the performance (response time) of the application. When I later went back to complete my return, I timed the page loads (because I almost always think about performance when I use the Internet). Most of the pages returned in less than 5 seconds, but some of the section summary pages took almost a minute!

Why didn't I notice the first time through that some pages were this slow? Why didn't I get frustrated with this seemingly poor performance? I usually notice performance as being poor at between 5 and 8 seconds, and at about 15 seconds I'll abandon a site or at least get frustrated. There's no science behind those numbers; they're just my personal tolerance levels. So what made me wait a minute for some pages without even noticing that it was slow? The answer is that when I requested a section summary page, an intermediate page came up that said:

"The information you have requested is being processed. This may take several minutes depending on the information you have provided. Please be patient."

When I received that message, I went on to do something else for a minute. I went to get a drink, or checked one of my e-mail accounts, or any of a million other things, and when I came back the page was there waiting for me. I was satisfied. If that message hadn't been presented and I had found myself just sitting and waiting for the page to display, I would have become annoyed and eventually assumed that my request wasn't submitted properly, that the server had gone down, or maybe that my Internet connection had been dropped.

So, getting back to the initial question of how fast is fast enough, from a user psychology perspective the answer is still "it depends." It depends on several key factors that determine what is and isn't acceptable performance.

The first factor is the response time that users have become accustomed to based on previous experience. This is most directly tied to the speed of their Internet connection. My mother, for example, has never experienced the Internet over anything other than a fuzzy phone line with a 56.6-kilobytes-per-second modem. I'm used to surfing via high-speed connections, so when I sign on at my mother's house I'm frustrated. My mother thinks I'm silly: "Scott, I think you're spoiled. That's as fast as we can get here, and a lot faster than we used to get! You never *were* very patient!" She's right – I'm not very patient, so I have a low tolerance for poor Web site performance, whereas she has a high tolerance.

Another factor is activity type. All users understand that it takes time to download an MP3 or MPEG video, and therefore have more tolerance if they're aware that that's the activity they're performing. However, if users don't know that they're performing an activity like downloading a file and are just waiting for the next page to load, they're likely to become frustrated before they realize that the performance is actually acceptable for the activity they're performing.

This leads us to the factor of how user expectations have been set. If users know what to expect, as they do with the tax preparation system I use, they're likely to be more tolerant of response times they might otherwise think of as slow. If you tell users that the system will be fast and then it isn't, they won't be

happy. If you show users how fast it will be and then follow through with that level of performance, they'll generally be pretty happy.

The last factor we should discuss here is what I call surfing intent. When users want to accomplish a specific task, they have less tolerance for poor performance than when they're casually looking for information or doing research. For example, when I log on to the site I use to pay bills, I expect good performance. When I'm taking a break from work and searching for the newest technology gadgets, I have a lot of tolerance for poor performance.

So with all of these variables you can see why, as Joe Strazzere said, "There are no industry standards." But if there are no industry standards, how do we know where to start or what to compare against? I'll describe some rules of thumb later, when we get to the topic of collecting information about performance requirements.

System Considerations

System considerations are more commonly thought about than user psychology when we're determining how fast is fast enough. Stakeholders need to decide what kind of performance the system can handle within the given parameters. "Fast-enough" decisions are often based purely on the cost to achieve performance. While cost and feasibility are important, if they're considered in a vacuum, you'll be doomed to fielding a system with poor performance.

Performance costs. The cost difference between building a system with "typical" performance and building a system with "fast" performance is sometimes prohibitive. Only by balancing the need for performance against the cost can stakeholders decide how much time and/or money they're willing to invest to improve performance.

System considerations include the following:

- system hardware
- network and/or Internet bandwidth of the system
- geographical replication
- software architecture

Entire books are dedicated to each of these considerations and many more. This is a well-documented and well-understood aspect of performance engineering, so I won't spend more time on it here.

Usage Considerations

Usage considerations are related to but separate from user psychology. The usage considerations I'm referring to have to do with the way the Web site or Web application will be used. For example, is the application a shopping application? An interactive financial planning application? A site containing current news? An internal human resources data entry application? "Fast" means something different for each of these different applications. An application that's used primarily by employees to enter large volumes of data needs to be faster for users to be satisfied than a Web shopping site. A news site can be fairly slow, as long as the text appears before the graphics. Interactive sites need to be faster than mostly static sites. Sites that people use for work-related activities need to be faster than sites that are used primarily for recreational purposes.

These considerations are very specific to the site and the organization. There really isn't a lot of documentation available about these types of considerations because they're so individual, depending on the specific application and associated user base. What's important is to think about how your site will be

used and to determine the performance tolerance of expected users as compared to overall user psychology and system considerations. I'll say more about this in the next section.

Collecting Information About Performance Requirements

So how do you translate the considerations described above into performance requirements? My approach is to first come up with descriptions of explicit and implied performance requirements in these three areas:

- user expectations
- resource limitations
- stakeholder expectations

In general, user and stakeholder expectations are complementary and don't require balancing between the two. For this reason, I start by determining these requirements. Once I've done this, I try to balance those with the system/financial resources that are available. Truth be told, I generally don't get to do the balancing. I usually collect the data and identify the conflicts so that stakeholders can make decisions about how to balance expectations and resources to determine actual requirements.

Determining the actual requirements in the areas of speed, scalability, and stability and consolidating these into composite requirements is the final step. I'll describe that process in detail later on, but first let's look at each of the three areas where you'll be collecting information about requirements.

User Expectations

A user's expectations when it comes to performance are all about end-to-end response time, as we touched on earlier in our look at user psychology. Individual users don't know or care how many users can be on the system at a time, how the system is designed to recover in case of disaster, or what the cost of building and maintaining the system has been.

When a new system is replacing an old one, it's critical from the user's perspective for the requirements of the new system to be at least as stringent as the actual performance of the existing system. Users won't be pleased with a new system if their perception is that its performance is worse than the system it's replacing – regardless of whether the previous system was a Web-based application, client/server, or some other configuration.

Aside from this situation, there's no way to predict user expectations. Only users can tell you what they expect, so be sure you take the time to poll users and find out what their expectations are before the requirements are set. Talk to users and observe them using a similar type of system, maybe even a prototype of the system to be built. Remember that most users don't think in terms of seconds, so to quantify their expectations you'll have to find a way to observe what they think is fast, typical, or slow.

During my tenure as a performance engineer, I've done a lot of research in the area of user expectations. I believed at first in the "8-second rule" that became popular in the mid-1990s, simply stating that most Web surfers consider 8 seconds to be a reasonable download time for a page. But since then I've found no reliable research backing this rule of thumb, nor have I found any correlation between this rule of thumb and actual user psychology. I'm going to share with you what I *have* found, not to suggest these findings as standards but to give you a reasonable place to start as you poll your own users.

I've found that most users have the following expectations for normal page loads when surfing on high-speed connections:

- no delay or fast – under 3 seconds
- typical – 3 to 5 seconds
- slow – 5 to 8 seconds
- frustrating – 8 to 15 seconds
- unacceptable – more than 15 seconds

In my experience, if your site is highly interactive or primarily used for data entry you should probably strive for page load speeds about 20% faster than those listed. For mostly static or recreational sites, performance that's about 25% slower than the response times listed may still be acceptable.

For any kind of file download (MP3s, MPEGs, and such):

- If the link for the file includes a file size and the download has a progress bar, users expect performance commensurate with their connection speed.
- If users are unaware they're downloading a file, the guidelines for normal pages apply.

For other activity:

- Unless user expectations are set otherwise, the guidelines for normal pages apply.
- If users are presented with a notice that this may take a while, they'll wait significantly longer than without a notice, but the actual amount of time they'll wait varies drastically by individual.

When users are made aware of their connection speed, their expectations about performance shift accordingly. Part of my experiments with users was to create pages with the response times in each of the categories above over a high-speed connection, then to throttle back the connection speed and ask the users the same questions about performance. As long as I told them the connection rate I was simulating, users rated the pages in the same categories, even though the actual response times were very different.

One final note: There's a common industry perception that pages that users find "typical" or "slow" on high-speed connections will be "frustrating" or "unacceptable" to users on slower connections. My research doesn't support this theory. It does show that people who are used to high-speed connections at work but have slower dial-up connections at home are often frustrated at home and try to do Internet tasks at work instead. But people who are used to slower connections rate the same pages as fast, typical, slow, and unacceptable over their typical connection as people who are used to high-speed connections and try to load these pages over their typical connection.

Resource Limitations

Limitations on resources such as time, money, people, hardware, networks, and software affect our performance requirements, even though we really wish they didn't. For example, "You can't have any more hardware" is a resource limitation, and whether we like it or not, it will likely contribute to determining our requirements.

Anecdotally, there's a lot to say about the effects of resource limitations on performance requirements, but practically all it really comes down to is this: Determine before you set your performance requirements what your available resources are, so that when you're setting the requirements, you can do so realistically.

Stakeholder Expectations

Unlike user expectations, stakeholder expectations are easy to obtain. Just ask any stakeholder what he or she expects.

"This system needs to be fast, it needs to support ten times the current user base, it needs to be up 100% of the time and recover 100% of the data in case of down time, and it must be easy to use, make us lots of money, have hot coffee on my desk when I arrive in the morning, and provide a cure for AIDS."

OK, that's not something an actual stakeholder would say, but that's what it feels like they often say when asked the question. It's our job to translate these lofty goals into something quantifiable and achievable, and that's not always an easy task. Usually stakeholders want "industry standards as written by market experts" to base their expectations on. As we've already discussed, there are no standards. In the absence of standards, stakeholders generally want systems so fast and scalable that performance becomes a nonissue . . . until they find out how much that costs.

In short, stakeholders want the best possible system for the least possible money. This is as it should be. When it comes to stakeholders, it's our job to help them determine, quantify, and manage system performance expectations. Of the three determinants of performance requirements that we've been discussing, the stakeholders have both the most information and the most flexibility. User expectations very rarely change, and resource limitations are generally fairly static throughout a performance testing/engineering effort. Stakeholder expectations, however, are likely to change when decisions have to be made about tradeoffs.

Consider this. Recently I've been involved with several projects replacing client/server applications with Web-based applications. In each case, the systems were primarily data entry systems. Initially, stakeholders wanted the performance of the new application to match the performance of the previous client/server application. While this is in line with what I just said about user expectations, it's not a reasonable expectation given system limitations. Web-based systems simply don't perform that fast in general. And I've found that even users who are accustomed to a subsecond response time on a client/server system are happy with a 3-second response time from a Web-based application. So I've had stakeholders sit next to users on the prototypes (that were responding in 3 seconds or less) and had those users tell the stakeholders how they felt about performance. When stakeholders realize that users are satisfied with a "3-second application," they're willing to change the requirement to "under 3 seconds."

Speed, of course, isn't the only performance requirement. Stakeholders need to either inform you of what the other requirements are or be the final authority for making decisions about those requirements. It's our job to ensure that all of the potential performance requirements are considered – even if we determine that they're outside the scope of the particular project.

Determining and Documenting Performance Requirements

Once you've collected as much information as possible about user and stakeholder expectations as well as resource limitations, you need to consolidate all of that information into meaningful, quantifiable, and testable requirements. This isn't always easy and should be an iterative process. Sending your interpretation of the requirements for comment back to the people you gathered information from will allow you to finalize the requirements with buy-in from everyone.

I like to think of performance in three categories:

- speed

- scalability
- stability

Each of these categories has its own kind of requirements. In the sections that follow, we'll discuss how to extract requirements from expectations and limitations, then consolidate those requirements into composite requirements – or what some people would refer to as performance test cases.

Speed Requirements

If you've gone through the exercise of collecting expectations and limitations, I'm sure that you have lots of information about speed. Remember that we want to focus on end user response time. There may be individual exceptions in this category for things like scheduled batch processes that must complete in a certain window, but generally, don't get trapped into breaking speed requirements down into subcomponents or tiers.

I like to start by summarizing some speed-related information I've collected verbally – for example:

- normal pages – typical to fast
- reports – under a minute
- exception activities (list) – fast to very fast
- query execution – under 30 seconds
- nightly backup batch process – under an hour

You'll see that some of that information is fairly specific, while some isn't. For this step, what's important is to ensure that all activities fall into one of the categories you specified. You don't want every page or activity to have a different speed requirement, but you do want the ability to have some exceptions to "typical" performance.

Now we must assign values to the verbal descriptions we have and extrapolate the difference between goals and requirements. You may recall from the Performance Engineering Strategy document that performance requirements are those criteria that must be met for the application to "go live" and become a production system, while performance goals are desired but not essential criteria for the application. Table 1 shows the speed requirements and goals derived from the descriptions above.

Activity type	Requirement	Goal
Normal pages	5 sec	3 sec
Reports	60 sec	30 sec
Exception activities (listed elsewhere)	3 sec	2 sec
Query execution	30 sec	15 sec
Nightly backup	1 hour	45 min

Table 1: Speed requirements and goals example

Of course, speed alone doesn't tell the whole story. Even getting agreement on a table like this doesn't provide any context for these numbers. To get that context, you also need scalability and stability requirements.

Scalability Requirements

Scalability requirements are the "how much" and "how many" questions that go with the "how fast" of the speed requirements. These might also be thought of as capacity requirements. *Scalability* and *capacity* are used interchangeably by some people.

Here's an example of scalability requirements that go with the speed requirements above:

- peak expected hourly usage – 500 users
- peak expected sustained usage – 300 users
- maximum percentage of users expected to execute reports in any one hour – 75%
- maximum percentage of users expected to execute queries in any one hour – 75%
- maximum number of rows to be replicated during nightly backup – 150,000

As you can see, we now have some context. We can now interpret that the system should be able to support 300 users with about a 3-second typical response time, and 500 with an under-5-second typical response time. I'm sure you'll agree this is much different t from single users achieving those results.

Another topic related to scalability is user abandonment. We'll discuss user abandonment in detail in Part 4 of this series; for now, suffice it to say that a general requirement should be to minimize user abandonment due to performance.

Stability Requirements

Stability covers a broad range of topics that are usually expressed in terms of "What will the system do if . . . ?" These are really exception cases; for instance, "What is the system required to do if it experiences a peak load of double the expected peak?" Another, broader term for these types of requirements is *robustness requirements*. Ross Collard, a well-respected consultant, lecturer, and member of the quality assurance community, defines *robustness* as "the degree of tolerance of a component or a system to invalid inputs, improper use, stress and hostile environments; . . . its ability to recover from problems; its resilience, dependability or survivability." While robustness includes the kind of usage stability we're focusing on, it also includes overall system stability. For our purposes we'll focus on usage stability and not on topics such as data recovery, fail-over, or disaster recovery from the system stability side.

Some examples of stability requirements are as follows:

- System returns to expected performance within five minutes after the occurrence of an extreme usage condition, with no human interaction.
- System displays a message to users informing them of unexpected high traffic volume and requests they return at a later time.
- System automatically recovers with no human interaction after a reboot/power down.
- System limits the total number of users to a number less than that expected to cause significant performance degradation.

Now, let's put these together into some real requirements.

Composite Requirements

The types of requirements discussed above are very important, but most of them aren't really testable independently, and even if they are, the combinations and permutations of tests that would need to be performed to validate them individually are unreasonable. What we need to do now is to consolidate

those individual requirements into what I term composite requirements. You may know them as performance test cases. The reason I shy away from calling these performance test cases is that my experience has shown that most people believe that once all the test cases pass, the testing effort is complete. I don't believe that's always the case in performance engineering, though it may be for performance testing.

Meeting the composite requirements simply means the application is minimally production-ready from a performance standpoint. Meeting the composite goals means that the application is fully production-ready from a performance standpoint *according to today's assumptions.* Once these composites are met, a new phase begins that's beyond the scope of this series – capacity planning, which also makes use of these composite requirements but has no direct use for test cases.

Let's look at how the individual requirements we came up with map into composite requirements and goals.

Composite Requirements

1. The system exhibits not more than a 5-second response time for normal pages and meets all exception requirements, via intranet, 95% of the time under an extended 300-hourly-user load (in accordance with the user community model) with less than 5% user abandonment.
2. The system exhibits not more than a 5-second response time for normal pages and meets all exception requirements, via intranet, 90% of the time under a 500-hourly-user load (in accordance with the user community model) with less than 10% user abandonment.
3. All exception pages exhibit not more than a 3-second response time 95% of the time, with no user abandonment, under the conditions in items 1 and 2 above.
4. All reports exhibit not more than a 60-second response time 95% of the time, with no user abandonment, under the conditions in items 1 and 2 above.
5. All reports exhibit not more than a 60-second response time 90% of the time, with less than 5% user abandonment, under the 75% report load condition identified in our scalability requirements.
6. All queries exhibit not more than a 30-second response time 95% of the time, with no user abandonment, under the conditions in items 1 and 2 above.
7. All queries exhibit not more than a 30-second response time 90% of the time, with less than 5% user abandonment, under the 75% report load condition identified in our scalability requirements.
8. Nightly batch backup completes in under 1 hour for up to 150,000 rows of data.
9. The system fully recovers within 5 minutes of the conclusion of a spike load.
10. The system displays a message to users starting with the 501st hourly user informing them that traffic volume is unexpectedly high and requesting that they return at a later time.
11. The system limits the total number of users to a number less than that expected to cause significant performance degradation (TBD – estimated 650 hourly users).
12. The system automatically recovers to meet all performance requirements within 5 minutes of a reboot/power down with no human interaction.

Composite Goals

1. The system exhibits not more than a 3-second response time for normal pages and meets all exception requirements, via intranet, 95% of the time under a 500-hourly-user load (in accordance with the user community model) with less than 5% user abandonment.
2. All exception pages exhibit not more than a 2-second response time 95% of the time, with no user abandonment, under the conditions in item 1 above.
3. All reports exhibit not more than a 60-second response time 95% of the time, with no user abandonment, under the conditions in items 1 and 2 above.

4. All reports exhibit not more than a 30-second response time 95% of the time, with no user abandonment, under the conditions in item 1 above.
5. All reports exhibit not more than a 30-second response time 90% of the time, with less than 5% user abandonment, under the 75% report load condition identified in our scalability requirements.
6. All queries exhibit not more than a 15-second response time 95% of the time, with no user abandonment, under the conditions in item 1 above.
7. All queries exhibit not more than a 15-second response time 90% of the time, with less than 5% user abandonment, under the 75% report load condition identified in our scalability requirements.

These requirements may be more detailed than you're used to, but I hope you can see the value of insisting upon explicit composite requirements such as these.

Summing It Up

No matter how many people ask for one, there's no industry standard for Web application performance. In the absence of such a standard we must depend on our own best judgment to determine just how fast is fast enough for our application. This article has discussed how to determine how fast is fast enough and how to convert that information into explicit, testable requirements. While explicit requirements based on reasonable performance expectations don't ensure project success, they do ensure the ability to evaluate the performance status of a system throughout the development lifecycle, and that alone can be invaluable.

About the Author

Scott Barber consults on and teaches practical performance testing and engineering for NobleStar, where he has completed dozens of engagements. These engagements have been evenly split between engineering performance for complex systems and mentoring organizations in the development of customized performance testing approaches based on his performance engineering methodology. His methodology has been widely adopted not only through client engagements, but also as the basis for his recognized article series' "User Experience, not Metrics" and "Beyond Performance Testing." Scott's speaking engagements for 2003 include PNSQC, PSQT/PSTT, the Rational User's Conference and local users' groups. He earned his master's degree in information technology, and was invited to be a guest lecturer at MIT based on his article "Automated Testing for Embedded Devices." Scott makes himself available to others by participating as a discussion leader for QAForums.com and the Rational Developer's Network. He is a member of the Context-Driven School of Software Testing and a signatory to the Manifesto for Agile Software Development. You can view Scott's resume and review his methodology and publications at http://www.perftestplus.com.

BUGS IN THE BRAVE NEW UNWIRED WORLD

A Failure mode catalog of risks and bugs in mobile applications

ABSTRACT

In the research reported in this paper, we present a failure mode catalog of wireless applications running on handheld devices. We populate the failure mode catalog with a broad range of actual and potential problems that can occur in mobile applications running on wireless networks. Entries made within the catalog include hypothetical failures (some published, some based on our own analyses), examples of real-life failures reported in the trade press, bug databases, magazine articles and other relevant sources, plus some causal analyses. These entries clarify the scope and meaning of failure categories. Testing is challenging in the handheld wireless world because it is new and some of the problems are new (or at least they show up in new ways). A failure mode catalog of the type we present here helps testers do risk-based testing. The risk-based tester creates a list of risks (ways the program could fail), assigns priorities to each, and performs tests that explore the risks of interest. We believe that a failure mode catalog designed to help testers generate test ideas will be particularly useful for an experienced tester who lacks domain expertise in this class of applications. It gives him ideas about how failures arise, or at least, what to test for, in this new breed of applications. The resulting set of the tests should be broader, targeted towards more risks, and probably often more powerful because they are designed to detect a plausible error. Additionally, even very experienced testers have blind spots -- bugs they haven't encountered or don't often think about. A broad taxonomy helps the tester recognize blind spots and get past them by using the list as an idea generator.

ABOUT THE AUTHORS

Ajay K. Jha, ajha@fit.edu is a Software Engineering graduate student at Florida Institute of Technology, specializing in software testing. He has almost three years of experience in testing and troubleshooting electro-mechanical, embedded, and general computer-based systems. His current interests include heuristic risk-based testing of mobile applications and exploring agile methods for testing and development of software.

Cem Kaner, J.D., Ph.D., kaner@kaner.com, www.kaner.com, is Professor of Computer Sciences at Florida Institute of Technology. He is senior author of *Testing Computer Software*, of *Lessons Learned in Software Testing*, and of *Bad Software: What To Do When Software Fails.*

MAILING ADDRESS

Florida Institute of Technology
Department of Computer Science
150 West University Blvd
Melbourne, FL – 32901

This research was partially supported by NSF grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers", www.testingeducation.org

INTRODUCTION

Handheld devices are evolving and becoming complex with the addition of newer features and functionalities. One of the drivers of this change is the rapid proliferation of the IP-based wireless networks and the maturation of the cellular technology. In a recent issue of *Wired* magazine, Arthur C. Clarke suggested that it was high time to listen to the technology and change the name of the magazine from *Wired* to *Unwired.* Wireless and mobile applications are a natural extension to the current wired infrastructure. Traditional mobile applications like e-mail and PIM have been widely adopted in the enterprise and consumer arenas and wireless data services enabling B2B and B2C transactions are rapidly making inroads. In the consumer arena, Japan leads the world with adoption of 3G and introduction of a plethora of applications for the mobile users.

Testing is challenging in the handheld wireless world because some of the problems are new (or at least they show up in new ways). To facilitate risk-based testing in this area, we present and organize a broad set of failure modes (publicly reported or potential failures) associated with wireless applications running on handheld devices.

WHAT IS A MOBILE APPLICATION?

Definitions of *mobile application* vary. In this report a *mobile application* is any application that runs on a handheld device like a PDA, smart phone, tablet PC or similar device. Five different types of applications can be developed for handheld devices, applications that:

- Stand alone, running on the handheld device itself;
- Connect to the back-end through synchronization software;
- Work with a back-end through packet switched wireless connectivity;
- Work with the back-end through circuit switched wireless connectivity;
- Vertical, using special networks like SMR (Specialized Mobile Radio) or paging networks.

We are most interested in mobile applications that connect with the back-end using either circuit switched or IP based wireless connectivity. This includes applications used to provide access to databases or application servers, to enhance classroom interaction, or to provide computer capabilities on smart phones. As we see with communications-enabled PDAs and smart phones, computing and communication technologies are converging.

One challenge associated with testing mobile applications is the difficulty of reproducing the production environment. Most testing must be performed using simulators and emulators. Even if we can simulate some aspects of the application, the handset for example, we can't be sure what happens when we try it over a real wireless network. This results in many field failures.

Another challenge is that, as the technology emerges, the market (developers and customers) is still figuring out what makes mobile applications great or less-than-great. Quality criteria are in flux and will stay that way until the market matures. Despite the uncertainty, testers must look carefully at the application under test, asking whether this is as good a product as it should be. One of our goals in developing a broad failure mode catalog is to broaden the risk analysis that testers use to guide their testing. A catalog provides a wider range of examples (and categories of risk) than any one person is likely to think of while designing her or his tests.

FAILURE MODE AND EFFECTS ANALYSIS

Failure mode and effects analysis (FMEA) is a traditional hardware engineering technique. It is less often and less formally applied in software, but FMEA-like analyses are the foundation of risk-based testing.

We will briefly describe the FMEA discipline here, but focus the paper on failure modes for wireless mobile applications, setting aside the rest of the FMEA analysis.

A *failure mode* is, essentially, a way that the product can fail. A failure mode's *effect* is the probable result of that particular type of failure.

In failure mode and effects analysis (FMEA), the analyst identifies the failure modes of a system, component by component. S/he evaluates the likely effects of each type of failure and prioritizes work accordingly. For the higher priority failure modes, s/he determines what could cause the failure, how to test for the failure, and how to eliminate or reduce the chance of the failure.

For more information on FMEA in general (http://www.fmeca.com/ffmethod/methodol.htm) is a good summary. For ideas on applying the full FMEA discipline to software, see the discussions in (http://www.stuk.fi/julkaisut/tr/stuk-yto-tr190.pdf). There is no explicit standard for software FMEA (SWFMEA), but the standard IEC 60812 published in 1985 is often referenced when carrying out FMEA for software-based systems.

RISK BASED TESTING AND FMEA

Amland (1999) explains risk-based test management in his paper where testing is prioritized to concentrate on the areas to test next.

James Bach (1999) explains risk analysis for the purpose of finding software errors. The steps followed in his approach are:

- Make a prioritized list of risks;
- Perform tests to explore each risk;
- As risks evaporate and new ones emerge, adjust your test effort to stay focused on the current risk set.

A risk points you to a way (or ways) that the software could fail. The challenging part in building a prioritized risk list is figuring out all the ways the software could fail, i.e. the failure modes.

Over the past year, we've been using 25 to 50 broad categories to classify failures that can occur in wireless applications. Our current category list is in Figure 2. This is a continuously evolving list. I will publish the final version in my M.Sc. thesis. For each category, we provide examples of publicly reported failures (with links to the press reports) and potential failures (problems we expect to happen, and would want to look for as testers, but that haven't been discussed in public bug reports). The public and potential failures together make up a set of *failure modes* within a category.

One of our goals in developing a broad failure mode catalog has been to broaden the risk analysis that testers use to guide their testing. A well-structured catalog provides a wider range of examples (and categories of risk) than any one person is likely to think of while designing his or her tests. Another of our goals has been to provide training material, to help testers new to

wireless mobile applications start from a pre-structured risk profile and so come up to speed more quickly.

TAXONOMIES

The American Heritage® Dictionary of the English Language: Fourth Edition 2000, defines a taxonomy to be:

> **"1.** The classification of organisms in an ordered system that indicates natural relationships.
>
> **"2.** Division into ordered groups or categories."

A taxonomy is an effective way of structuring and classifying information or data.

A few examples of well-known taxonomies are:

- The science of systematics, which classifies animals and plants into groups showing the relationship between each;
- Bloom's taxonomy (Bloom, 1956) of levels of knowledge, which educators use to develop teaching goals and assessments (such as exam questions);
- Beizer's (1990) taxonomy of software errors, which classifies bugs in terms of the lifecycle, phase in which they were introduced (design, implementation, etcetera)

For more examples of interest to software readers, see Vijayaraghavan (2003). As Vijayaraghavan pointed out, there are disputes over whether the structured risk lists (e.g. in security), structured bug lists, and many other structured lists can be called "taxonomies" if they do not provide orthogonal classifications. We're sidestepping that issue by calling our structured list a "catalog" instead of "taxonomy."

Kaner, Falk and Nguyen (1993) published a catalog of common software errors in their book's Appendix A. Some of our thinking about how testers can use a failure mode catalog came from experience with many readers' reports of their uses of this bug list.

Vijayaraghavan (2003) published a useful taxonomy of e-commerce failures.

Bach (2003b) points out that testers work more effectively with risk catalogs whose structure is immediately obvious to the reader. Testers especially benefit from clear structure when trying to decide what area(s) to focus on at a given point in a project.

We think that Bach's (2003a) *Heuristic Test Strategy Model* (Figure 1) provides a clear structure that we can fit failure modes into. Accordingly, we have reworked Vijayaraghavan's taxonomy into this structure and are extending it to mobile and handheld applications. We have adapted the model slightly by splitting Bach's qualitative failure categories into operational and developmental subcategories.

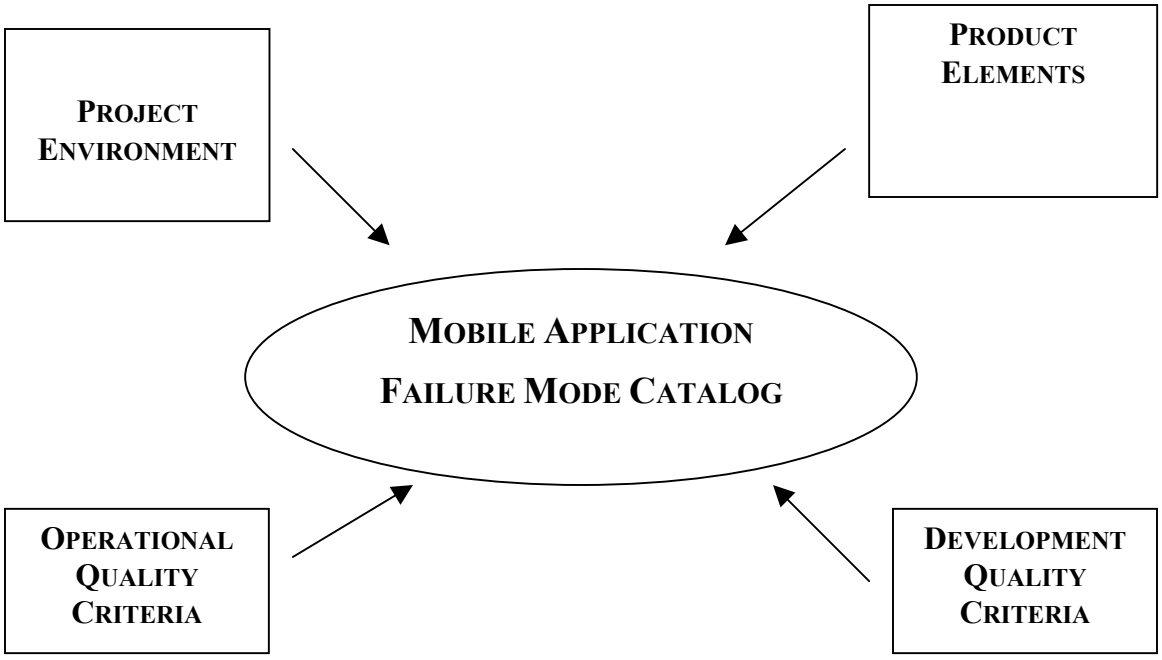


Figure 1. Bach's Heuristic Test Strategy Model

HOW TO USE THE CATALOG

We provide a top-level outline of our catalog in Figure 2, and more detail in the Appendix.

A catalog like this is intended to help the tester generate a list of risks. No catalog can be a complete list of failure modes for every application, not even for every PDA-based wireless application. The idea is to provide a source of ideas that can inspire a risk-focused tester. The tester reads a description of a category of potential failures, reads some examples within the category, and asks herself:

- Could this application fail in some way that would fit in this category?
- Could it misbehave like any of this category's examples?
- What about other, similar ways?

Exploration by analogy generates the list the tester actually uses. Our list is merely, but usefully, a starting point.

We believe that a catalog designed to help testers generate test ideas will be particularly useful for an experienced tester who lacks domain expertise in this class of applications, because it suggests problems to test for in this new breed of application. The resulting set of tests should be broader, covering more risks, and may be more powerful because each is designed to detect a plausible error. Additionally, even very experienced testers have blind spots—bugs they haven't encountered or don't often think about. A broad taxonomy helps the tester recognize blind spots and get past them by using the list as an idea generator.

FIGURE 2: OVERVIEW OF THE MOBILE / HANDHELD FAILURE MODE CATALOG		
OPERATIONAL QUALITY CRITERIA	**Functionality**	• Suitability • Compliance • Accuracy • Interoperability
	Usability	• Efficiency • Memorability • Satisfaction • Learnability • Error Messages
	Quality of service	
	Performance	
	Dependability	• Maturity • Fault tolerance • Recoverability
	Security	• Authentication • Wireless network security • Data integrity • Privacy and confidentiality • Access control • Availability
DEVELOPMENT QUALITY CRITERIA	**Maintainability**	• Analyzability • Changeability • Stability
	Testability	• Field Failures
	Portability	• Adaptability • Installability • Conformance • Replacability
	Scalability	
PRODUCT ELEMENTS	**Structure**	• WAP gateway failures
	Functions	• Navigation • Calculation • Mobile middleware interface
	Data	• Real time failure • Data instance failure
	Platform	• Third party software failures • Hardware failures • Micro-browser failures • Wireless network failures • Host location register / visitor location register / location database • Mobile database
	Multi-function operations	• Mobility management • Location management • Software upgrade errors • Transaction errors • Data handling
	Synchronization	• Software interface • Hardware interface • Wireless Synchronization
	Memory management	• Memory leaks
PROJECT ENVIRONMENT	**Customers** **Information** **Team** **Equipment and tools** **Schedules** **Test Items** **Deliverables**	

A Sample Application: Mobile Computing in Education

One sector where mobile computing has proved useful is education. Handhelds can be integrated into the curriculum to deliver notes or assignments to students, to help the teacher monitor what the student is doing, and to collect results (work-in-progress or completed assignments) and feedback from the students.

Many kinds of wireless networks, handheld devices and mobile applications are used for this kind of application. Typical wireless networks are the 802.11-family and infrared-based beaming stations. Among the handhelds Palm (http://www.palm.com/us/education/), Pocket PC (http://www.microsoft.com/education/?ID=PocketPC) and Blackberry (http://www.blackberry.com/products/handhelds/index.shtml) devices are most commonly used. There are some specialized handhelds meant exclusively for educational use like the Texas Instruments TI 83 and TI 89 (http://education.ti.com/educationportal/index.jsp).

Examples of applications in use are Cells, Handysheets, and PicoMap from University of Michigan's Center for Highly Interactive Computing in Education (2003a), and CellSheet and LearningCheck from Texas Instruments (2003a). Teachers can integrate these applications, and interact with students' devices, using wired or wireless access, with products like the Palm OS Application Assessment Manager (PAAM) (GoKnow, 2003a) and the TI-Navigator (Texas Instruments, 2003b).

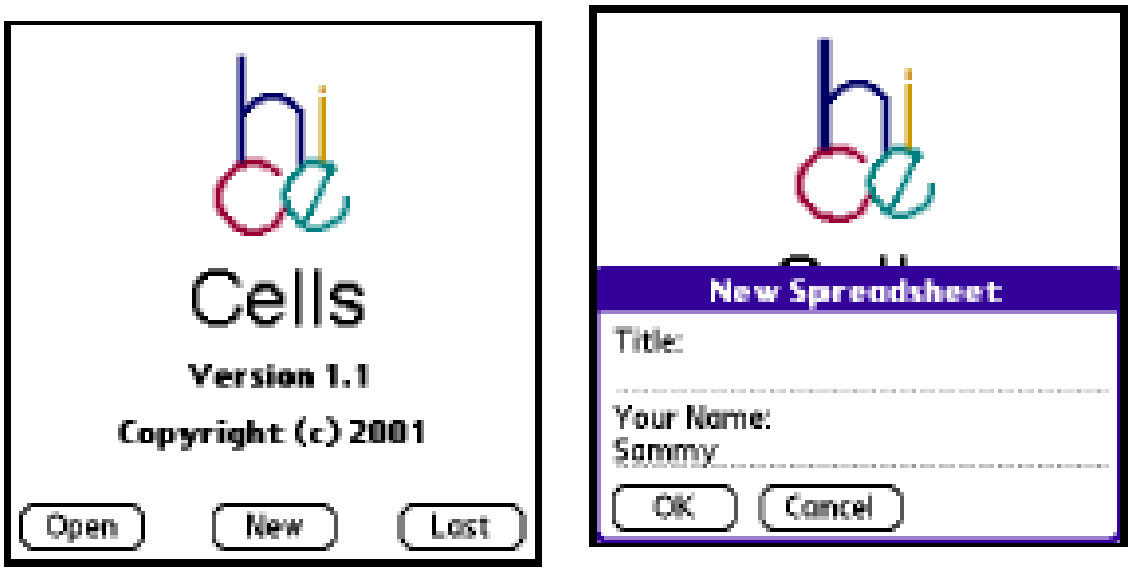


Figure 3: Source: Cells 1.1 Quick Start Guide (Center for Highly Interactive Computing in Education, 2003c)

In this paper, we use a sample application called Cells (Center for Highly Interactive Computing in Education, 2003b) which enables students to enter data into a spreadsheet on their PDA. We demonstrate how to use the failure mode catalog to generate test cases for this application and its integration into PAAM.

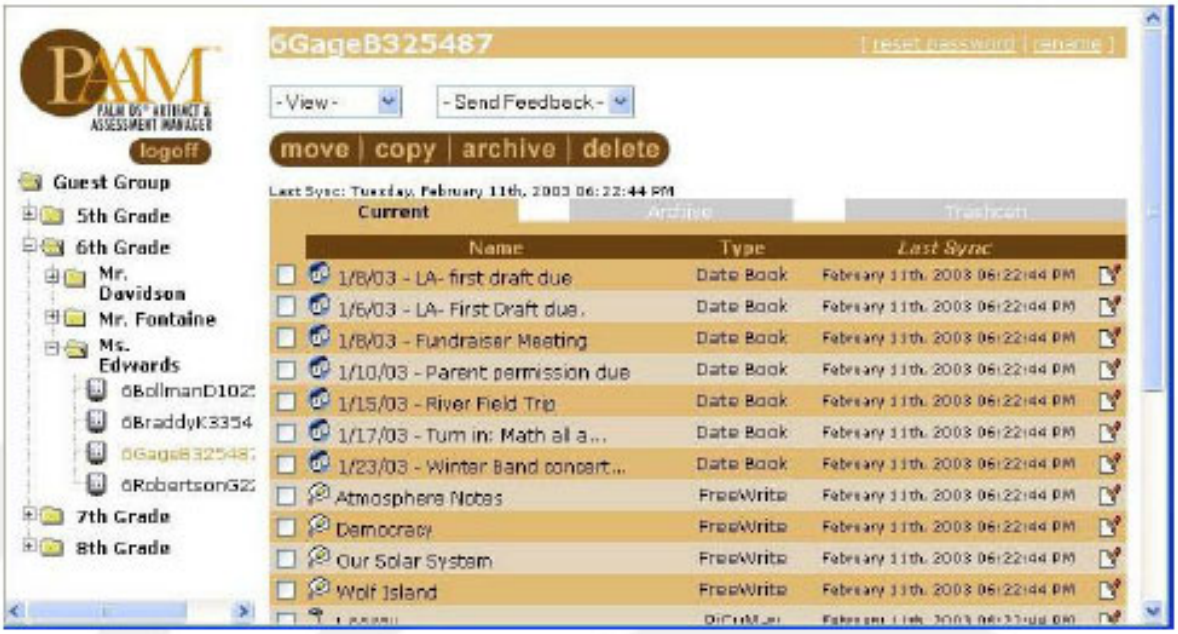


Figure 4: Source: Walkthrough: PAAM™-Palm OS Artifact and Assessment Manager (GoKnow, 2003b)

APPENDIX 1: FAILURE MODES FOR WIRELESS / MOBILE APPLICATIONS

Let's set some expectations. This is a work in progress. It is incomplete. The complete version will be Ajay's M.Sc. thesis, which will be published at www.testingeducation.org later in the year. The catalog is most thoroughly worked, at this point, in the sections on security, usability, functionality, synchronization and performance.

This appendix lists a set of failure modes that are specific to wireless / mobile applications. We intentionally do *not* list generic examples of failure modes. We expect to eventually create a broad list that includes generic examples by merging Vijayaraghavan's (2002) thesis work, Ajay's thesis work, relevant portions of the Kaner, Falk, Nguyen (1993) Appendix and other failure (as opposed to fault, such as are available from NIST, 2000) collections. If you're interested in offering help (such as data, funding, or editorial review/writing) for that broader project, please contact Cem Kaner at kaner@kaner.com.

The overall structure of this catalog matches Bach's (2003a) *Heuristic Test Strategy Model* and we include several of his descriptions, including his heading captions, with his permission. We have modified the second-level structure in some cases to conform to other published standards or well-known discussions. Eventually, we expect that we'll reorganize to match a revision of Bach's model after he reorganizes some of the details of his model in response to our suggestions.

OPERATIONAL QUALITY CRITERIA

"*Quality Criteria* are the rules, values, and sources that allow you as a tester to determine if the product has problems. Quality criteria are multidimensional, and often hidden or self-contradictory." (Bach 2003a, p. 1) "A *quality criterion* is some requirement that defines what the product should be. By looking or thinking about different kinds of criteria, you will be better able to plan tests that discover important problems fast. Each of the items on this list can be thought of as a potential risk area. For each item below, determine if it is important to your project, then think how you would recognize if the product worked well or poorly in that regard." (Bach 2003a, p. 4)

Operational quality criteria are criteria that relate to the product in use. We distinguish them from *development* criteria, which relate to the product as a static object under development.

1. FUNCTIONALITY: *Can it perform the required functions?*

ISO 9126 defines *functionality* as, "A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs. This set of attributes characterises what the software does to fulfill needs, whereas the other sets mainly characterise when and how it does so."

(Source: http://www.issco.unige.ch/ewg95/node14.html)

ISO 9126 further subdivides Functionality into Suitability, Accuracy, Interoperability and Compliance. These subcategories are discussed as individual categories in this qualitative fault modelling of mobile applications.

SUITABILITY

"Attributes of software that bear on the presence and appropriateness of a set of functions for specified tasks." (ISO 9126, 1991)

Failure Modes:

- No implementation or malfunction of the renaming of the cells spreadsheet;
- Unable to beam the Cells spreadsheet;
- File size too big to be served and viewed on a handheld.
- No "admin" button to carry out administrative tasks in PAAM.
- Failure to add an individual handheld to the existing subgroup of PAAM.
- Failure to delete, add or move files from one subgroup to another.
- Failure in the "filter" function of PAAM
- Failure to archive files in PAAM

Related Bugs and Links

- Incomplete functional implementation in Pocket EXCEL http://www.earthv.com/articles.asp?ArticleID=579
- Pocket PC 2002: Contact Does Not Beam

  http://support.microsoft.com/default.aspx?scid=kb;en-us;323011
- Pocket PC Calendar Does Not Correct Time Zone Changes

  http://support.microsoft.com/default.aspx?scid=kb;en-us;268249

ACCURACY

"Attributes of software that bear on the provision of right or agreed results or effects." (ISO 9126: 1991)

Failure Modes:

- Data that is beamed is inaccurate due to transmission problem
- Data not stored in the appropriate folder
- Not able to add a student's name in the subgroup of PAAM
- Application designed without taking screen size into consideration
- Misalignment of image on the screen
- Multiple copies of a file with the same name but different content exist on the handheld.

Related Bugs and Links:

- Known Issues in Pocket Excel on a Handheld PC
  http://support.microsoft.com/default.aspx?scid=kb;en-us;189502
- Known Issues in Pocket Word on a Handheld PC
  http://support.microsoft.com/default.aspx?scid=kb;en-us;188782
- Graphics Issues in Pocket PowerPoint Presentations
  http://support.microsoft.com/default.aspx?scid=kb;en-us;186757

INTEROPERABILITY

"Attributes of software that bear on its ability to interact with specified systems." (ISO 9126: 1991)

Failure Modes:

- Specific make of handheld not able to send information to PAAM
- Problem running the software due to incompatible version of the operating system of the handheld. For example certain applications developed for Palm OS requires earlier version of Palm OS.
- Application not available for all the leading handheld platforms like Palm OS, Pocket PC, Blackberry and Symbian OS.
- Application not able to run on dirty configuration. A dirty configuration is any configuration that is not supported but is very common. (Collard, 2003)
- Application can run only on one kind of network. For example, if CDMA or 1XRTT is used for voice and data, it can only work in North America and places where CDMA is in use.
- Beaming of file or data between handhelds having different operating systems not possible.
- Failure to interoperate between differing screen resolutions. Some of the common resolutions available are 160x160, 320x240, 95x65, 120x130, 1024x768, 1024x768
- Failure to interoperate between differing colors. Some of the common color schemes to be found on the handheld devices are 2-bit, 16-bit, 32-bit and varying gray levels.
- Application compatible across different microbrowsers available.
- Application complies with the differing graphics format.

Related bugs and links:

- Incompatibility bugs wireless
  http://australianit.news.com.au/articles/0,7204,6459892%5e15397%5e%5enbv%5e,00.html

- Problems When You Convert Files Between Excel and Pocket Excel

  http://support.microsoft.com/default.aspx?scid=kb;en-us;185921

COMPLIANCE

"Attributes of software that make the software adhere to application related standards or conventions or regulations in laws and similar prescriptions." (ISO 9126: 1991)

Failure Modes:

- Non compliance with the wireless network standard
- Application does not comply with the UI guidelines of development leading development environments like Palm OS, MSDN or BREW.

2. USABILITY: *How easy is it for a real user to use the product?*

Usability is the "effectiveness, efficiency and satisfaction with which a specified set of users can achieve a specified set of tasks in a particular environment." (ISO 9241) According to Jakob Neilson (1993), usability subsumes the notions of learnability, memorability, efficiency, error rate / recovery and satisfaction. ISO 9126 divides usability into understandability, learnability and operability.

Usability issues are highly pronounced on handheld devices due to the limitations of wireless handheld devices (Passani, 2000). They have a limited form-factor and the display units are smaller than their desktop counterparts. Designing for such small screen size needs more thinking and better navigation structures. Another factor worth taking into consideration is the data input in a PDA or smart phone. The keyboard or the soft input panel of these devices is not very spacious. This warrants new and innovative ways of reducing the amount the data that a user is made to enter. In this paper, Usability failure modes and risks are divided into learnability, efficiency, memorability, error recovery and satisfaction.

LEARNABILITY

The system should be easy to learn so that the user can rapidly start getting some work done with the system. (Neilson 1993)

Failure Modes:

- Inconsistent layout
- Information not arranged in hierarchical or tree-like structure

Related bugs and links:

- Inconsistent user interface

  http://www.cewindows.net/commentary/userinterface.htm

EFFICIENCY

The system should be efficient to use, so that once the user has learned the system, a high level of productivity is possible. (Neilson 1993)

Failure modes:

- No importance given to the main activities of the portable users and all the functionalities implemented
- Main activities of the user not implemented in the fastest possible manner
- Verbose text on a small screen
- No implementation of "Back" functionality
- Application directly transcoded to any wireless markup language from HTML
- Users not given proper feedback when they commit errors
- Users do not understand what to do when they commit mistakes

SATISFACTION

The system should be pleasant to use, so that users are subjectively satisfied when using it. Users should like the system (Neilson 1993)

Failure Modes:

- Users found it difficult to use the application.
- Application needs a lot of data entry

MEMORABILITY

The system should be easy to remember, so that the casual user is able to return to the system after some period of not having used it, without having to learn everything all over again (Neilson 1993)

Failure Modes:

- No customization of the application for the specific micro browser. This might result in content not being rendered properly on a micro browser.
- Difficulty in remembering actions needed to perform a task

3. ACCESSIBILITY. *Can it be used by everyone?*

A system is said to be accessible when it can be used by anyone irrespective of their physical or technical capabilities. With respect to people with physical disabilities, accessibility means

providing supporting tools or assistive technologies like screen reader, adapted keyboards or head-mounted pointers to enable the use of product.

Failure Modes:

- Red / green color blindness not taken into consideration
- User not able to press a button on the handheld device due to improper placement of the button
- User not able to use biometric security feature of the handheld due to strict requirement of the hand movement.

4. QUALITY OF SERVICE. *Can you configure and depend on the network's service?*

A network provides high Quality of Service (QoS) if it delivers traffic consistently across a network, provides high transmission rates, low error rates and supports designated usage patterns. "*Quality of Service* (QoS) refers to the capability of a network to provide better service to selected network traffic over various technologies, including Frame Relay, Asynchronous Transfer Mode (ATM), Ethernet and 802.1 networks, SONET, and IP-routed networks that may use any or all of these underlying technologies. The primary goal of QoS is to provide priority including dedicated bandwidth, controlled jitter and latency (required by some real-time and interactive traffic), and improved loss characteristics. Also important is making sure that providing priority for one or more flows does not make other flows fail. QoS technologies provide the elemental building blocks that will be used for future business applications in campus, WAN, and service provider networks." (Cisco, 2003, at http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/qos.htm#1020563)

Failure Modes:

- Inability to predict the network capabilities of the wireless network. Due to this exchange of parameters between the application and network is not possible.
- Loss of bandwidth or attenuation of the signal strength due to movement of the mobile node.
- Loss of communication during handover between cells. This is not significant for the voice application but even milliseconds of broken link can result in undesirable consequences in case of data application.
- Cost per unit data not very economical for customers.
- High cost to establish a connection or to access a resource.
- High bit error rate due to mobility of the node connected to the wireless network.
- Low quality of multimedia application due to limited bandwidth.
- Limitation imposed by the requirement of portability of the system. Battery life is limited and mobile computing technology requires significant power for effective transmission.

- Alteration of the QoS parameters in the wireless networks not taken into account while designing the system.
- Context management or the user environment not taken into consideration.

5. PERFORMANCE: *How speedy and responsive is it?*

Mobile applications run on wireless links that have high latency and low bandwidth. A data packet needs multiple hops before communicating with another device or application. Special consideration is required while designing the system to enhance the performance of such applications.

Failure Modes:

- Usage of HTTP/1.0 instead of HTTP/1.1 (Cheng, 1999)
- Problem in the WML (wireless markup language) timer function
- Time taken to load a page on a microbrowser very high.
- Time taken to beam a file to another handheld very high.
- No caching mechanism used to enhance the performance.
- Performance problems arising after loading third party software applications.
- Firmware problems resulting in low performance.
- Performance degrades when loading multiple pages.
- Problems arising because of poor mobile client server architecture. Fat client vs thin client approaches (Yang, 2003)
- Usage pattern of the wireless network not taken into consideration.
- Performance critical features or functions of an application not identified and optimized.
- Response time after a complex calculation very high.
- Throughput of the system very low when many users log in and try to use a feature.
- Occurrence of buffer overflows and memory leaks after exposing the system to load for an extended period of time (Collard, 2002)
- Weak point in the system detected after exposing load at a specific portion of the system considered being not so robust. This is also known as hot spot testing (Collard, 2002)
- System's performance hampers when varying the load from low to high and following some pattern of load fluctuation (Collard, 2002)
- Problem occurs when exposing the system to abrupt load. This is also known as spike or bounce testing. Load balancing and resource reallocation problems surface during such tests (Collard, 2002)

- Breakpoint of the system found to be less than expected or designed. Breakpoint of a system is defined as the load or mix of loads at which the system fails and the manifestation of the failures (Collard, 2002)
- Reduced performance when using a network for data transfer in conjunction with data transmission.
- Delay in loading bitmaps and other graphics more than expected.
- Wireless link "stalls" resulting in spurious TCP timeouts (Chakravorty, 2002)
- Delay caused due to high RTT and slow startup phase for the system to utilize the wireless link (Chakravorty, 2002)
- Excessive queuing over the downlink results in higher probability of timeouts during the initial requests for connection (Chakravorty, 2002)

Related bugs and links:

- Microsoft Active Sync 3.x slows down the system
- Dell Halts Axim Shipments Over Software Problem http://stickyminds.com/news.asp?Function=NEWSDETAIL&ObjectType=NEWS&ObjectId=6549

6. DEPENDABILITY: *Will it work well and resist failure in all required situations?*

Dependability is a term encompassing many notions like reliability, recoverability, availability and safety within itself (Malloy 2002). ISO 9126 divides reliability into three separate categories: Fault Tolerance, Maturity and Recoverability. Applications running on wired infrastructures have very high dependability because of the nature of such infrastructures. Current and emerging wireless networks and hence, the applications running on them, do not offer such levels of dependability. It is thus extremely important to concentrate on this category while testing any wireless or mobile application.

Reliability within the mobile context could be defined as the "Ability of the wireless and mobile networks to perform their designated set of functions under certain conditions for a certain operational time." (Malloy 2002) I have included the listing of failure modes with respect to Fault Tolerance, Maturity and Recoverability in this paper as that seemed to be the most appropriate way to deal with the issues concerning dependability of a wireless application and networks.

FAULT TOLERANCE

"Attributes of software that bear on its ability to maintain a specified level of performance in cases of software faults or of infringement of its specified interface." (ISO 9126: 1991)

Failure Modes:

- Unable to resume transmission of data or corruption of data due to failure of an access point.

- No auto configuration of addressing based on context (Sternbenz, 2002)
- No auto configuration of signaling and routing based on mission (Sternbenz, 2002)
- Problem in channel allocation algorithm for the cellular network
- Failure in borrowing a channel due to delay in communication (Cao, 2000)
- Failure to establish a channel due to network congestion.
- Failure to establish a channel due to communication link failure.
- Failure in establishing a channel due to mobile switching center failure.

MATURITY

"Attributes of software that bear on the frequency of failure by faults in the software." (ISO 9126: 1991)

Failure Modes:

- Mean time between Failures (MTBF) in carrying out a transaction very high.
- Frequent "stalls" in transmissions.
- Low power of transmission of the radio waves.

RECOVERABILITY

Recoverability is the capability of a system or application to maintain services during attack or when all the resources are not available.

Failure Modes:

- Application does not switch to offline mode when there is a loss in network connectivity.
- Delayed recovery after timeouts. Normally the recovery should be in milli or microseconds. GPRS networks have the recovery time in seconds after timeouts due to link "stalls" (Chakravorty, 2002)
- No adaptability of the power of transmission of the signals.
- No reconnection attempt after the device fails to establish the wireless link.

7. SECURITY: *Is the system secure?*

Security issues could be subdivided into six subcategories: privacy and confidentiality, access control and authorization, authentication, data integrity, wireless network security and availability.

PRIVACY AND CONFIDENTIALITY

Confidentiality and privacy means the protection of the information about a user or process.

Failure Modes:

- Disclosure of passwords.
- Disclosure of private data like credit card details.

Related Bugs and Links:

- Yahoo! Mobile service discloses random sensitive information to unauthorized users.

  http://xforce.iss.net/xforce/xfdb/11352
- Stolen credit card information

  http://www.cewindows.net/commentary/userinterface.htm
- Hand-helds are a hacker's delight

  http://www.ciol.com/content/developer/2003/103080301.asp

ACCESS CONTROL AND AUTHORIZATION

Access control implies as to who may have access to the physical device or data.

Failure Modes:

- Physical access to a stolen device

  http://www.kb.cert.org/vuls/id/789985

Related Bugs and Links:

- Linux PDA Security Hole

  http://www.pdacenter.net/news/static/102643633846024.shtml
- Palm Password bypass error

  http://www.securityfocus.com/bid/2429
- World readable permission in Palm desktop

  http://www.securityfocus.com/bid/2398/discussion/
- Palm Desktop For MacOS X Hotsync Insecure Backup Permissions Vulnerability

  http://www.securityfocus.com/bid/3863
- Mouse hole in device security

  http://pcw.vnunet.com/News/1123233

- Unauthorized access due to stolen device. http://www.computerworld.com/securitytopics/security/story/0,10801,78127,00.html

AUTHENTICATION

Authentication implies establishing identity of users, process or hardware components.

Failure Modes:

- No authentication mechanism used.
- Weak passwords that can be easily broken.

Related Bugs and Links:

- PalmOS Authentication Bypass Vulnerability
  http://www.securityfocus.com/bid/5538/info/
- Palm OS Weak Encryption Vulnerability
  http://www.securityfocus.com/bid/1715/info/

DATA INTEGRITY

Integrity means that the system should not corrupt the data as compared to their original state.

Related Bugs and Links:

- Corruption of file system using Zaurus vulnerability
  http://www.internetnews.com/dev-news/article.php/1402491

WIRELESS NETWORK SECURITY

This category targets the security failures that could occur in different wireless networks. Networks covered under this category are the Bluetooth, 802.11 and the cellular network. Wireless network as of now is the biggest security hole in the IT infrastructure and exposes even the wired infrastructure to attacks. A brief explanation of the networking technologies and links to relevant literature could be found in appendices at the end of the paper. Many of these failure modes are explained in detail in a special publication by NIST (Karygiannis, 2002). A new security specification called WPA (Wi Fi Protected Access) is being adopted as it fixes most of the fundamental flaws in WEP. The key features of of WPA are Extensible Authentication Protocol (EAP), Temporal Key Integrity Protocol (TKIP), Message Integrity Check (MIC), and 802.1X for authentication and dynamic key exchange. WPA is a standards based solution and offers higher level of interoperability.

Bluetooth Failure modes:

- No external boundary protection

- Set Bluetooth devices to the lowest sufficient power level to ensure transmissions within bounds
- Unit keys used instead of combination keys
- Weak PINS chosen
- No alternative protocol used for the exchange of PINS
- No established "minimal key size" for any key negotiation process
- No application level (on top of Bluetooth stack) encryption used for highly sensitive data
- Security patches and upgrades not up-to-date
- Man in the middle attack
- Break of stream cipher - DES/RC4 are weaker than 3DES and AES

- Unit keys used instead of combination keys
- Weak PINS chosen
- No alternative protocol used for the exchange of PINS
- No established "minimal key size" for any key negotiation process
- No application level (on top of Bluetooth stack) encryption used for highly sensitive data
- Security patches and upgrades not up-to-date
- Man in the middle attack
- Break of stream cipher - DES/RC4 are weaker than 3DES and AES

802.11 Failure Modes:

- Eavesdropping from the parking lot within the AP range
- Radio interference resulting in DoS
- No encryption mechanism followed by the AP
- Shared key authentication

- MAC address could be spoofed even if WEP is enabled allowing access to the WLAN
- Security patches and upgrades not up-to-date
- Algorithms using shorter keys used
- Inherent weaknesses in the WEP—could be easily cracked
- If the plaintext message is known and attacker has the copy of the cipher text— key could be obtained by getting the IV and using a dictionary attack.
- Cipher stream reuse - key stream could be recovered from the WEP packet
- Fluhrer-Mantin-Shamir exploit of the weakness in the KSA—resulted in two tools: WEPcrack and air snort
- Static noise attack—resulting in DoS
- Weak AP password
- If reset option is enabled—default encryption settings could revert to no encryption
- No physical barriers—first line of defense
- Network name or the SSID is stored in clear text
- No authentication mechanism in place—EAP not enabled
- Man in the middle attack
- ARP poisoning
- Cache poisoning
- Rogue access points
- No access control list (ACL) or VPN in place
- Access to the wired network not protected
- IPSec or VPN or secure shell traffic not used

Related Bugs and Links:

- Problems with WEP and adoption of WPA

InformationWeek Wireless Fidelity Deploying WPA Today June 30, 2003

- Wi Fi users do not use security features

  SecurityFocus HOME News Study Wi-Fi users still don't encrypt

- Problem in the wireless router

  SecurityFocus discussion SMC Wireless Router Malformed PPTP Packet

  SecurityFocus HOME Vulns Info Buffalo WBRG54 Wireless Broadband Router

- Configuration problem

  SecurityFocus HOME Netgear FM114P ProSafe Wireless Router Rule Bypass

- Information disclosure due to configuration error

  SecurityFocus HOME Netgear FM114P ProSafe Wireless Router UPnP Inform

- Input validation error

  SecurityFocus HOME Netgear FM114P Wireless Firewall File Disclosure V

- Microsoft Wireless and mobile security resources

  Wireless and Mobile Security Technical Resources

- New WLAN Attacks Identified

  http://www.wi-fiplanet.com/news/article.php/2246081

Cellular network Failure Modes:

This subcategory lists the most common attacks on a cellular system. Some of the failure modes in this category are inspired by Ko's (1996) paper on attacks on cellular systems.

- Phone cloning resulting due to the ESN (Electronic serial number) and MIN (Mobile identification number) being read by attackers.
- Cell phone not equipped with PIN (personal identification number)
- Hijacking of the voice channel by increasing the power level of the cellular phone.
- No encryption of voice while transmission
- Denial of service occuring due to jamming of the RF channel using RF attack.

Related Bugs and Links:

- Handspring VisorPhone vulnerable to DoS via SMS image transfer

  http://www.kb.cert.org/vuls/id/222739

- SMS denial of service on Seimens

  SecurityFocus HOME Siemens Mobile Phone SMS Denial of Service Vulnera

- Verizon Wireless bug allows SMS tapping
  http://www.threezee.com/modules.php?op=modload&name=Sections&file=index&req=viewarticle&artid=6&page=1
- Cellular companies fight fraud

  http://www.decodesystems.com/mt/97dec/

AVAILABILITY

System availability is duration of time a system is available for use by its intended users. Opposite of availability is denial of service (DoS), when the system is not available with its services either fully or partially.

Failure Modes:

- DoS attack due to firmware problems.
- Failure initiating and maintaining the wireless link due to interference with external devices.

Related Bugs and Links:

- Airborne Viruses
  http://www.networkmagazine.com/article/NMG20001130S0001
- Palm OS - Phage virus
  http://doc.advisor.com/doc/07194
- New email virus bombards mobile phone users

  http://news.com.com/2100-1023-241489.html?legacy=cnet
- Palm HotSync Manager Remote Denial of Service Vulnerability
  http://www.securityfocus.com/bid/6673/info/
- PalmOS TCP Scan Remote Denial of Service Vulnerability
  http://www.securityfocus.com/bid/3847
- Mobile virus threat looms large

  http://news.bbc.co.uk/2/hi/technology/2690253.stm

DEVELOPMENT QUALITY CRITERIA

Development quality criteria focus on the product under development and its relationship to the development organization rather than to the user.

8. MAINTAINABILITY: *Will it be easy to maintain?*

Maintainability is defined as the ease with which changes can be made to a software system. These changes may be necessary for the correction of faults, adaptation of the system to a meet a new requirement, addition of new functionality or removal of existing functionality. Maintainability can be either a static form of testing, i.e. carried out by inspections and reviews, or a dynamic form i.e. measuring the effort required to execute maintenance activities.

(Source: http://www.testingstandards.co.uk/maintainability_guidelines.htm)

ISO 9126 divides maintainability into analyzability, changeability, stability and adaptability.

ANALYZABILITY

"Attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified" (ISO 9126: 1991)

Failure Modes:

- Components like mobile middleware, programming language etcetera used in the development of the application not very clear or poorly chosen.
- Improper documentation of the design and architecture.

CHANGEABILITY

"Attributes of software that bear on the risk of unexpected effect of modifications." (ISO 9126: 1991)

Failure Modes:

- Unable to change the network configuration or type with ease.
- Unable to add more features to the application.
- Failure to personalize the application.

STABILITY

"Attributes of software that bear on the effort needed for validating the modified software." (ISO 9126: 1991)

Failure Modes:

- Addition of features makes the application difficult to use.
- Application becomes unstable after changes in the language or tool used.

9. TESTABILITY: *How easy will it be to test?*

Testability could be defined as *visibility* and *control*. *Visibility* is our ability to observe the states, outputs, resource usage and other side effects of the software under test. *Control* is our ability to apply inputs to the software under test (Pettichord, 2002)

FIELD FAILURES

These are the failures that escape the unit testing stage or any other kind of testing done using the simulators or emulators. Errors and failures are encountered when the application runs on the actual device or on the actual wireless network in the production environment.

Failure Modes:

- Application fails to connect to the back-end on the real network.
- Application fails to load and function on the actual device.

10. PORTABILITY: *How easy will it be to change the environment?*

"The ease with which a system or component can be transferred from one hardware or software environment to another" (Institute of Electrical and Electronics Engineers, 1990)

ISO 9126 sub-categorizes Portability into Adaptability, Installability, Conformance and Replaceability.

ADAPTABILITY

"Attributes of software that bear on the opportunity for its adaptation to different specified environments without applying other actions or means than those provided for this purpose for the software considered." (ISO 9126: 1991)

Failure Modes:

- Application not customizable to the user environment.
- Application does not behave well on limited bandwidth.
- Application does not adapt to network latency.

INSTALLABILITY

"Attributes of software that bear on the effort needed to install the software in a specified environment" (ISO 9126: 1991)

Failure Modes:

- Failure to install the application locally using synchronization software.
- Failure in online installation. Many applications give the benefit of wireless online install using internet or local area network.

- Failure to install the application on the emulator or simulator therby resulting in inadequate unit testing.
- Failure to install the software due to interference with another application.
- Failure to customize the installation if user has the option to do that.
- Failure to install the application at the desired location inspite of the location being valid.
- Failure to install the application on all the supported configurations.
- Unwanted results when user aborts the installation midway.
- Problems in the input field while installing the software.

Related Bugs and Links:

- Services may not start after installation

  http://www.securityfocus.com/bid/7282/discussion/
- Palm install could not handle a valid path

  http://www.faughnan.com/palm.html#Installation
- "Cannot Find Pocket Streets" Error Message When You Try to Install Pocket Streets

  http://support.microsoft.com/default.aspx?scid=kb;en-us;319689

CONFORMANCE

"Attributes of software that make the software adhere to standards or conventions relating to portability" (ISO 9126: 1991)

Failure Modes:

- Non conformance with the carrier (cellular service provider) standards and guidelines.

REPLACEABILITY

"Attributes of software that bear on the opportunity and effort of using it in the place of specified other software in the environment of that software."
(ISO 9126: 1991)

11. SCALABILITY: *Can I increase the capacity with ease?*

"The ease with which a system or component can be modified to fit the problem area" (Institute of Electrical and Electronics Engineers, 1990)

Failure Modes:

- Problems due to large number of users.

12. LOCALIZABILITY. *Can I adapt the application to serve bigger market?*

Internationalization (sometimes shortened to "I18N, meaning "I - eighteen letters -N") is the process of planning and implementing products and services so that they can easily be adapted to specific local languages and cultures, a process called localization. The internationalization process is sometimes called translation or localization enablement. (Source: http://whatis.techtarget.com/definition/0,,sid9_gci212303,00.html)

PRODUCT ELEMENTS

"Product Elements are things that you intend to test. Software is so complex and invisible that you should take special care to assure that you indeed examine all of the product that you need to examine." (Bach 2003a, p. 1) "Ultimately a product is an experience or solution provided to a customer. Products have many dimensions. So, to test well, we must examine those dimensions. Each category, listed below, represents an important and unique aspect of a product. Testers who focus on only a few of these are likely to miss important bugs." (Bach, 2003a, p. 3)

13. STRUCTURE: *Everything that comprises the physical product.*

WAP GATEWAY FAILURES

WAP gateway is a server that is responsible for converting a WTP request made by a smart phone to an HTTP request to be processed by a web server. WAP gateway also translates an HTML web page to WML if it is required.

Failure Modes:

- Failure in transcoding HTML to WML resulting in errors serving a page to the cell phone.
- Problems arising due to cards / fonts of WML not supported on a device.
- Problems arising due to deck size of WML exceeding the device limit.

14. FUNCTIONS: *Everything that the product does.*

NAVIGATION

Failure Modes:

- Non intuitive placement of navigation buttons on the screen.
- Users reaching a deadlock while navigating. Only way to proceed further is to return to the main screen.

CALCULATION

Failure Modes:

- Improper calculation of the arithmetic functions like average, minimum, maximum etcetera
- Improper calculation at the boundary values.
- ASCII values calculated in case user enters a character other than a number.

MOBILE MIDDLEWARE INTERFACE FAILURES

A middleware is defined as: **"**An enabling layer of software that resides between the business application and the networked layer of heterogeneous (diverse) platforms and protocols. It decouples the business applications from any dependencies on the plumbing layer, which consists of heterogeneous operating systems, hardware platforms and communication protocols." (Source: International Systems Group)

A mobile middleware is an enabling layer of software that is used by application developers to connect their applications with disparate mobile (wireless and wired) networks and operating systems. This category targets the failures that could occur in a mobile middleware.

15. DATA: *Everything that the product processes.*

REAL TIME FAILURE

Real time applications or services not only have to carry out the right computation but the time taken to carry out a specific task or providing a service in a specified amount of time is also of considerable importance. Real time applications could be broadly categorized as hard real time applications or soft real time applications depending on the real time constraints they have to satisfy (Dreamtech, 2002)

Failure Modes:

- Delay in sending localized / personalized content to a user

DATA INSTANCE FAILURE

Related Bugs and Links:

- Problem due to number of records – system stop

  http://support.microsoft.com/default.aspx?scid=kb;en-us;Q317698

16. PLATFORM: *Everything on which the product depends.*

MOBILE SWITCHING CENTER FAILURES

In the case of cellular wireless network, establishing a communication session means making a wireless channel available between a *mobile host* (cell phone) and mobile support station. Mobile host sends a request to the *mobile support station* in its cell. There are many different channel allocation algorithms to avoid channel interference and efficiently utilize the limited frequencies. All these functions are carried out at the mobile switching center. (Cao, 2000)

THIRD PARTY SOFTWARE FAILURES

Mobile Application Architecture utilizes variety of third party software applications. In the case of location based services, mapping is carried out with the help of content providers having the geographic data. Potential problems in the third party applications may lead to failures in the application under test.

HARDWARE FAILURES

Related Bugs and Links:

- Problems with Casio Cassiopeia Pocket PC 2002

  http://www.pdastreet.com/forums/showthread.php?threadid=779
- Problems in Compaq hardware

  http://www.cewindows.net/bugs/pocketpc2002-compaq.htm
- Problems in HP hardware

  http://www.cewindows.net/bugs/pocketpc2002-hewlettpackard.htm
- Problems in Toshiba hardware

  http://www.cewindows.net/bugs/pocketpc2002-toshiba.htm
- HP Jornada with Pocket Internet Explorer for Windows CE Saves Cookies When "Save my password" Is Not Selected

  http://support.microsoft.com/default.aspx?scid=kb;en-us;303676

MICRO-BROWSER FAILURES

A micro-browser offers the same basic functionality as a desktop browser. It is used to submit user requests, receive and interpret results and allow the users to surf web pages using their handheld (Nguyen, 2003)

Failure Modes:

- Application not tested for correct functionality on text only browser.
- Application fails to work on palm based browser.
- Microbrowser does not support the security mechanism like SSL
- Microbrowser does not support tables and images.
- Microbrowser not supporting cHTHL therby not able to serve i-mode pages.
- AvantGo specific problems. AvantoGo supports web channel formatted pages.

Related Bugs and Links:

- Known Issues in Pocket Internet Explorer on a Handheld PC

  http://support.microsoft.com/default.aspx?scid=kb;en-us;190307
- Pocket Internet Explorer Quits When You Connect to an SSL Site with the DES56 Cipher

http://support.microsoft.com/default.aspx?scid=kb;en-us;320894

- PRB: You Receive an Unknown Error When You Call a Method of the MFC ActiveX Control

  http://support.microsoft.com/default.aspx?scid=kb;en-us;310566

- Error Message: 500 java.lang.IllegalArgumentException: [object]

  http://support.microsoft.com/default.aspx?scid=kb;en-us;258910

- Pocket Internet Explorer: "Bad MIME Format" Viewing JPEG Images

  http://support.microsoft.com/default.aspx?scid=kb;en-us;187608

- Cannot Download .wav Files in Pocket Internet Explorer 1.1

  http://support.microsoft.com/default.aspx?scid=kb;en-us;167923

- Unable to Personalize a User Who Is Using Pocket Internet Explorer

  http://support.microsoft.com/default.aspx?scid=kb;en-us;284151

WIRELESS NETWORK FAILURES

This category targets the failures in the typical wireless networks.

802.11 Failure Modes:

- Configuration problem in the 802.11 BSS (Basic Service Set)
- Configuration problems in 802.11 ESS (Extended Service Set)
- Location of access point not appropriate leading to fading of signal
- Interference with another access point leading to loss of signal

Bluetooth Failure Modes:

- Failure in setting up a Bluetooth piconet.
- Failure to build a Bluetooth scatternet from piconets.

Cellular Network Failure modes:

- Loss of signal.
- Coverage issues.
- Loss of bandwidth due to mobility.

HOME LOCATION REGISTER / VISITOR LOCATION REGISTER / LOCATION DATABASE

In cellular systems home location register (HLR) is the database that has the permanent registry for the service profile i.e. information about the subscriber. Visitor location register (VLR) on the other hand serves as a temporary repository for the profile information. Many different algorithms are in use to manage the mobility. The most common strategy in use in North America is IS-41 that utilizes a two tier system of HLR and VLR to keep track of the mobile node.

Failure Modes:

- Failure to update the HLR on the status of the mobile host after it enters a new VLR.
- Excessive load on the network signaling resource due to the mobility of the mobile hosts.
- Excessive load on the database due to frequent updates needed resulting due to mobility of the node.

MOBILE DATABASE

There are two different approaches for the database connectivity on a handheld wireless device. There is a thin client model where technology like WAP enables users to view information that has been extracted from the database and displayed as a Web page with the help of a micro browser. Since availability of the wireless network has still some issues, this model is not very suitable for data intensive applications. The alternative model makes the significant data reside on the handheld (a local relational database on the handheld)

DATABASE SERVER

A database server is software that manages data in a database. Database management functions such as locating the actual record being requested, updating, deletion and protection of the data is performed by the database server. It also provides both the access control and concurrency control. So, while testing a mobile application that connects to the database, if there is some erratic data encountered, the database server could be the culprit and should be tested.

17. MULTI-FUNCTION OPERATIONS. *How the product will be used.*

MOBILITY AND RESOURCE MANAGEMENT FAILURES

This category targets the failures that occur due to the mobility of the node and improper resource management to offer uninterrupted wireless connectivity to the user.

Failure modes:

- Frequent disconnection due to the mobility of the node.
- Disruption during hand-off between different networks.
- Depletion of the IPv4 addresses.

LOCATION MANAGEMENT

Location management is an extremely important functionality in location based mobile applications. A location based mobile application utilizes the knowledge of the location of the mobile node to serve location specific information. It is used in telematics, route directions, call routing, billing and several other applications.

Failure modes:

- Change in the logical identity of the device or the owner. A logical identity could be MAC address, IP address or anything else used to identify a mobile node.
- Problems arising due to not updating the Location database server.
- Problems arising due to mobile node not re-registering with the base station.
- Failure in receiving GPS data, in case of GPS being used to locate mobile nodes.
- Failure to translate the geocode (latitude, longitude) into a map by the content provider.
- Failures in the triangulation mechanism to determine location.

SOFTWARE UPGRADE ERRORS

Failure Modes:

- Application does not work with the upgraded operating system.
- Application or device freezes due to firmware upgrade.

Related Bugs and Links:

- Problem faced due to firmware upgrade on Samsung T series:

  http://www.reviewcentre.com/post64347.html

TRANSACTION ERRORS

These are the errors that occur in carrying out a typical transaction. Transaction will depend on the functionality that the application offers and is highly context dependent.

Failure Modes:

- Failure in transmitting information to the handhelds by the base unit.
- Failure in completing a task assigned due to missing information on how to transmit data.

DATA HANDLING

Failure Modes:

- Varying treatment of characters by different mobile content delivery technology not taken into consideration.
- Application vulnerable to failures at the boundary values.

18. SYNCHRONIZATION: *How the data will be synchronized*

Synchronization is a feature that enables exchanges, transforms and synchronizes data between two different applications or data stores. Synchronization could be either cradle-based or wireless. The SyncML Consortium is on a mission to get mobile application developers and handheld device makers to use a common, XML-based data synchronization technology. This category lists the different failure that could be encountered while synchronizing data between two applications.

SOFTWARE INTERFACE:

Failure Modes:

- Corruption of data files during hotsynch
- Problem in the synchML server
- Active synch problems in case of Pocket PC devices
- Problems encountered while establishing partnerships with more than one machine.
- Failure in synchronizing data due to interference with another application.

Related Bugs and Links:

- Office- Palm link
  http://www.earthv.com/articles.asp?ArticleID=579
- Intellisynch error
  http://www.pdastreet.com/forums/showthread.php?threadid=779
- Installation of another software over active synch
  http://support.microsoft.com/default.aspx?scid=kb;en-us;263450
- Problem with the USB driver of Mac
  http://www.palminfocenter.com/view_Story.asp?ID=703
- Problem with hotsynch synchronizing datebook
  http://www.geocities.com/Heartland/Acres/3216/faq_pg7.htm
- Hot synch problem with some Windows XP
  http://www.computing.net/pda/wwwboard/forum/278.html
- Database access errors during synchronization
  http://support.microsoft.com/default.aspx?scid=kb;en-us;294213
- Synchronization failure with outlook and active synch
  http://support.microsoft.com/default.aspx?scid=kb;en-us;276563
- Problem synchronizing third party applications and software
  http://support.microsoft.com/default.aspx?scid=kb;en-us;271980
- Problem with AvantGo synchronization
  http://support.microsoft.com/default.aspx?scid=kb;en-us;259938
- Problems in continuing message interchange over synchML server
  http://lists.axialys.net/pipermail/syncml/2003-April/000010.html

- Problems with multiple inboxes with single parternership
  http://support.microsoft.com/default.aspx?scid=kb;en-us;269217
- Palm hotsynch troubleshooting
  http://www.palm.com/support/hotsync.html
- Problem with hotsynch due to discrepancy in the registry entries
  http://www.geocities.com/Heartland/Acres/3216/faq_pg6.htm
- Error due to CDO collaboration object model not installed
  http://support.microsoft.com/default.aspx?scid=kb;en-us;299625
- Error in Activesynch due to missing files
  http://support.microsoft.com/default.aspx?scid=kb;en-us;281598
- Soft Reset
  http://www.mobile-and-wireless.com/Articles/Index.cfm?ArticleID=27164
- Problem in the synchronization port - COM / USB of the computer
  http://support.microsoft.com/default.aspx?scid=kb;en-us;185750
- Problem synchronizing application like money by unexpected actions during the synchronization process
  http://support.microsoft.com/default.aspx?scid=kb;en-us;263988
- No reconnection after log off and logging on again
  http://support.microsoft.com/default.aspx?scid=kb;en-us;q321935&
- Synchronization problem due to error in the server
  http://support.microsoft.com/default.aspx?scid=kb;en-us;318450
- Corrupted Data File May Prevent Mobile Application From Opening Up
  http://www.filemaker.com/ti/108092.html
- ActiveSync Reports Unresolved Items When Device Resources Are Low
  http://support.microsoft.com/default.aspx?scid=kb;en-us;295001

HARDWARE INTERFACE

In case of local synchronization, a cradle that is connected to either the serial or USB port is used to mount the mobile device to synchronize the files and other data between a desktop and the mobile device. This category lists the failures that could occur in interfacing the cradle with a desktop due to failures in the serial or USB port of the desktop. For more information on interfacing

Failure Modes:

- Failure in the USB driver installed on the desktop.

- Failure in the serial port communication.
- BIOS errors in the desktop resulting in breakdown of communication between the external device and desktop.
- Failure in the infrared port of the mobile device or partner machine.

Related Bugs and Links:

- Problem with the USB driver of Mac

  http://www.palminfocenter.com/view_Story.asp?ID=703

WIRELESS SYNCHRONIZATION

Wireless modems are also in use to synchronize data. In this wireless synchronization method, mobile device connects to the proxy server using a wireless modem and preformatted web pages stored on the web server is transferred to the device. User can then view the pages offline (Nguyen, 2003). Infrared synchronization is also in use where devices are synchronized locally using line of sight.

Failure Modes:

- Problems in the proxy server.
- Non compliance with the guidelines for preformatting the web pages for mobile devices.
- Failure in infrared synchronization.

18. MEMORY MANAGEMENT: *Symptoms of memory-related errors.*

Mobile devices are highly resource constrained with respect to the amount of primary and secondary storage. Special attention is required while developing and testing to avoid memory leaks and wild pointers.

MEMORY LEAKS

Failure Modes:

- Failures due to memory leaks on the client device
- Failures due to memory leaks on the server side.

Related Bugs and Links:

- Memory Leak in Pocket Internet Explorer

  http://support.microsoft.com/default.aspx?scid=kb;en-us;315028

- Memory leak due to SOAP exception

  https://www.alphaworks.ibm.com/forum/wstkmd.nsf/0/001C8DD95F6E7CF2633A6F05F9275483?OpenDocument

- Crash when establishing connection with dead Web Services

  http://www.alphaworks.ibm.com/forum/wstkmd.nsf/0/896C1B509F0130669C5E7D90F1D9812E?OpenDocument

PROJECT ENVIRONMENT

Elements of the *project environment* define how the project is being run, rather than what it contains. There are really two projects for a tester to consider, the broad development project and the testing sub-project. Aspects of either can be a source of constraints, problems, or opportunities for the tester.

We've included these categories for completeness, relative to Bach's model, but these are not failure modes and so we will not elaborate on them further in this paper. This section is taken verbatim from Bach (2003a, p.2), with his permission. The only area populated in this broad category is equipment and tools.

Project Environment includes resources, constraints, and other forces in the project that enable us to test, while also keeping us from doing a perfect job. Make sure that you make use of the resources you have available, while respecting your constraints. (Bach 2003a, p. 1) "Creating and executing tests is the heart of the test project. However, there are many factors in the project environment that are critical to your decision about what particular tests to create. In each category, below, consider how that factor may help or hinder your test design process. Try to exploit the resources you have available while minimizing the impact of constraints.

1. **CUSTOMERS:** *Anyone who is a client of the testing and development project*

   Stakeholders of the project determine as to what kind of tests they want to run. Usage of the failure mode catalog will depend on the expectations of the clients.

2. **INFORMATION:** *Information about the product or project that is needed for testing*

   Mobile application is used in a variety of horizontal and vertical industries. Some of the vertical applications are stock trading, airline reservation, healthcare solutions, and warehouse inventory solutions etcetera. Among the horizontal applications, the most prominent ones are wireless e-mail and personal information management, wireless office data solutions and sales force automation etcetera. Testing will depend on the context in which the application will be used.

3. **TEAM:** *Anyone who will perform or support testing and development*

   Experience, skills and expertize in special test techniques of the people responsible for carrying out testing should be considered while formulating a test strategy.

4. **EQUIPMENT AND TOOLS:** *Resources required to administer testing and development*

   This category lists the problems and links specific to the leading software development environments for mobile application development.

   - Wireless JAVA

     http://wireless.java.sun.com/j2me/index.html

- Bugs in J2ME

  http://search.java.sun.com/search/java/index.jsp?qt=%2Bcategory%3Amid-profile+%2Bstate%3Aopen&nh=10&qp=&rf=1&since=&country=&language=&charset=&variant=&col=javabugs
- Palm OS Application Development

  http://www.palmos.com/dev/start/
- Microsoft Mobile development
  http://www.microsoft.com/windowsmobile/information/devprograms/default.mspx
- Problems in BREW

  http://www.qualcomm.com/brew/developer/resources/ds/faq/techfaq14.html
- List of known bugs in OpenWave SDK

  http://developer.openwave.com/support/bug_form.html

5. **SCHEDULES:** The sequence, duration, and synchronization of events

6. **TEST ITEMS:** *The product to be tested*

7. **DELIVERABLES:** *The observable products of the test project*

APPENDIX 2: AN OVERVIEW OF WIRELESS APPLICATIONS, NETWORKS AND HANDHELD DEVICES

The software / hardware architecture of a typical mobile application could be best visualized in a layered framework for strategizing the testing process. There are typically four levels of abstraction that could be envisioned. Many different types of devices, wireless networks and content delivery technologies are in use. Some of the wireless networks and client side applications are briefly explained in this appendix. Third appendix lists down some of the content delivery technologies and associated acronyms.

- Mobile applications
- Wireless networking infrastructure
- Client-side devices
- Mobile content delivery and middleware

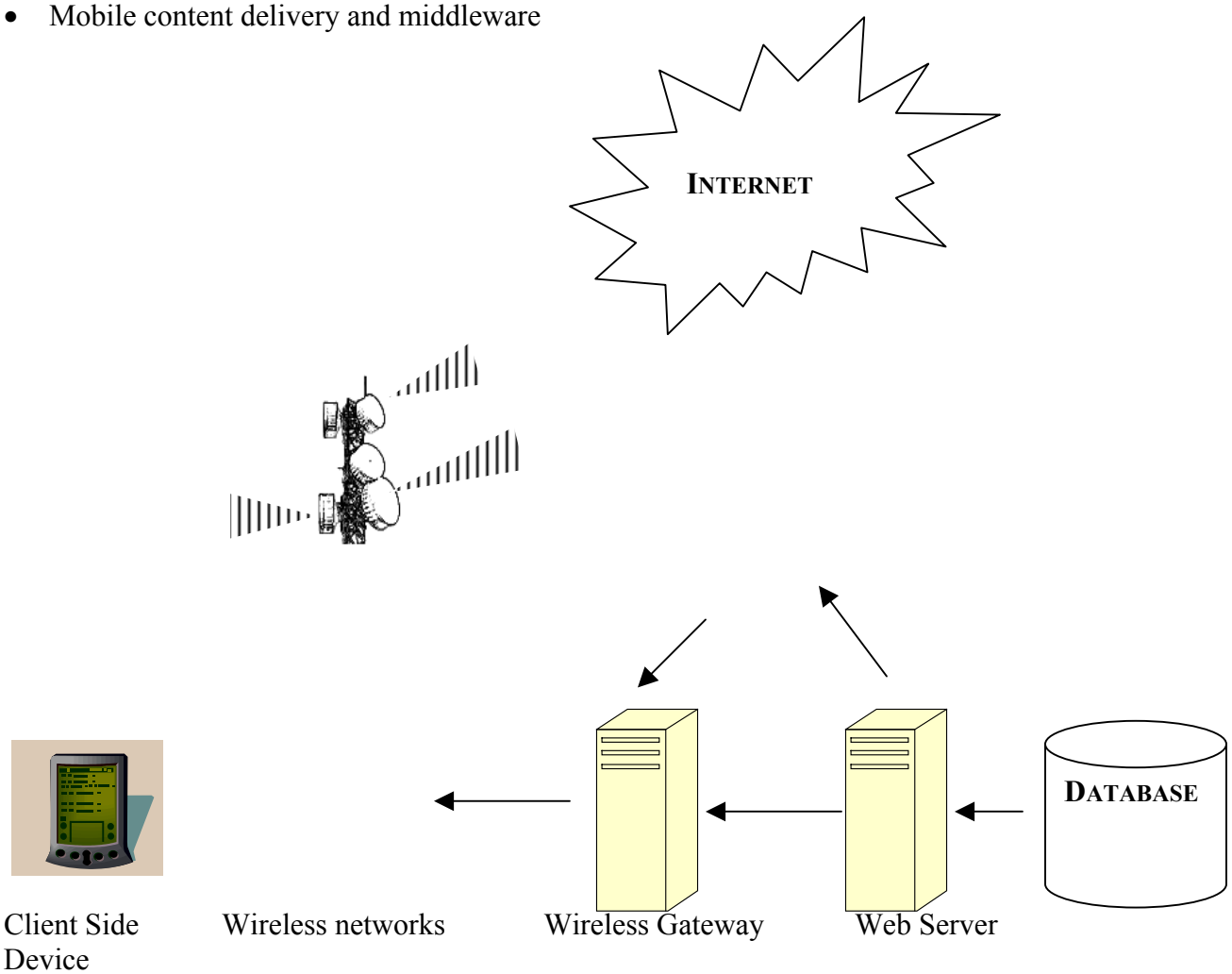


Figure 6: A Typical wireless system

I thank Scott Barber for suggesting me to include the schematic of the architecture of a wireless system and Dr. Kostanic, kostanic@fit.edu (Florida Tech) for providing me with some useful pictures. Above mentioned layers along with the technology in use for each layer are described in detail as follows:

MOBILE APPLICATIONS

Many of these mobile solutions are already in use in the vertical industry like Healthcare, Education and delivery services. Some of the commonly encountered mobile applications are Mobile e-mail and PIM. Other applications that are in use and emerging are mobile financial applications like banking and stock trading applications, mobile advertising which is location specific, mobile entertainment services and games, mobile office products like Pocket Word/Excel, mobile education software, enterprise wireless data applications and mobile healthcare solutions. (Varshney, 2002)

WIRELESS NETWORKING INFRASTRUCTURE

Wireless Networks could be broadly divided into wide area network (WAN), Local area Network (LAN) and the personal area network (PAN) based on coverage.

WIDE AREA NETWORK (WAN)

Cellular Networks: This is the network with maximum coverage area. It is a licensed public wireless network used by Web cell phones and private radio frequency digital modems in handhelds. WAN cellular towers come in three different power configurations: macro cell, micro cell, and pico cell. There are two network architectures for the communicating devices: the circuit-switched and the packet-switched. A circuit-switched network builds up circuit for a call and establishes a dedicated connection of circuits between points. Examples of circuit-switched devices are the telephones, cellular phones, web phones and dial-up modems. In a packet-switched network, the IP-addressed data packets are routed between points on demand. Packet-switched networks exchange variable amounts of data or voice packets. Data can be transferred almost immediately as the network is always on. WAN could be then further subdivided into voice-oriented network and data oriented network. Some of the most widely used technologies for data transmission are GSM/GPRS, 1XRTT CDMA, and Edge etcetera. GPRS and EDGE overlay packet based air interface on the existing circuit-switched voice network. A new generation of wireless wide area technology known as 3G is deployed in Japan and Europe. It offers data speeds starting from 2Mbps in the fixed wireless environment, 384 Kbps at low mobility and 128 Kbps while moving in a car. 3G systems operate in 2GHz frequency band and are intended to provide a wide range of services including telephony, paging, messaging, Internet and broadband data.

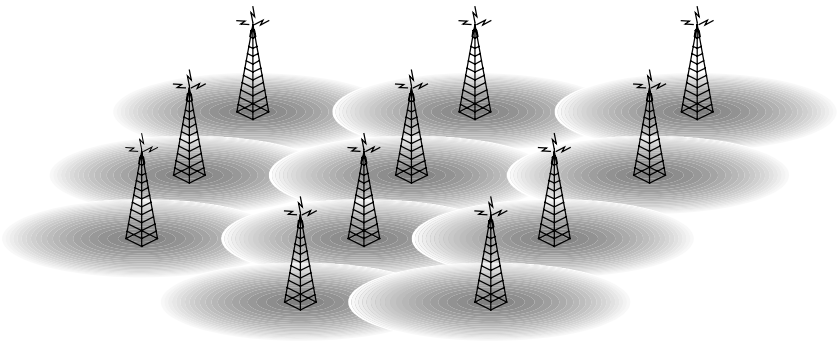


Figure 7: Cellular network

Paging network: Paging networks are of two types: one-way paging and two-way paging. They are the earliest form of networks used to send messages to mobile workers.

LOCAL AREA NETWORK

IEEE 802.11 family: These are the wireless local area networks operating in unlicensed spectrum. In 1997, the IEEE adopted IEEE Std. 802.11-1997, the first wireless LAN (WLAN) standard. This standard defines the media access control (MAC) and physical (PHY) layers for a LAN with wireless connectivity. It addresses local area networking where the connected devices communicate over the air to other devices. 802.11b working on 2.4 GHz spectrum has the physical layer data rate at 11Mbps. 802.11g, is a new IEEE standard for wireless LANs. It is backward compatible with 802.11. It can support 54Mbps raw data rate. The backward compatibility comes from the fact that it works in the same 2.4 GHz band and OFDM

(Orthogonal Frequency division multiplexing) enables the high data transfer rate. Another standard of interest is 802.11a. More information is available at: http://www.wi-fi.org/OpenSection/index.asp?TID=1

HIPERLAN: In European countries the set of wireless network communication standards is known as HiperLAN. There are two specifications adopted by ETSI (European Telecommunications standards Institute), HiperLAN/1 and HiperLAN/2.

PERSONAL AREA NETWORK

Bluetooth: Bluetooth is a wireless technology used in the personal connectivity market by linking mobile computers, mobile phones, portable handheld devices, and connectivity to the Internet. It is a low power, personal, wireless voice and data network having a range of 10 meters. A Bluetooth network called Pico net can connect eight Bluetooth devices. It works in the 2.4 GHz frequency band and does not suffer from the interference by obstacles like a wall. It supports both point-to-point wireless connections without cables between mobile phones and personal computers, as well as point-to-multipoint connections to enable ad hoc local wireless networks (source: http://bluetooth.com/tech/works.asp).

Infrared: Infrared got standardized in 1994 with the publication of Infrared Data Association's standard. Infrared devices use line of sight, exchanging data by lining up their infrared lenses and have a typical range of 2 meters. They are mainly used to manually exchange information using point-to-point connection.

CLIENT-SIDE DEVICES

The first handheld computing device that acquired a significant market share was the Apple Newton. Since then, the handheld space has evolved to the point where there are literally thousands of different combinations of hardware devices, software capabilities and wireless networking features. These are some of the devices that are in use in the commercial, industrial and personal sectors.

Smart phones: They are cellular phones with the display hardware and software for the wireless Internet connectivity. They have a micro browser and some memory that is continuously being expanded. There are many different names for such phones depending on the technology used for the Internet services and information. In Japan it is known as imode phone; in Europe it's called a WAP phone, and in many places it is known as a web phone. (Beaulieu, 2002)

PDA: It is a miniature computer with special OS, storage, a keyboard or the soft input panel and a display. In general they have much more computing power than a smart phone. They again are called with different names like handheld, palm-top, communicator etcetera. There are two different kinds of handheld: the industrial and the consumer handheld (Beaulieu, 2002). The main difference is in the packaging. The PDAs used in the consumer market are mostly based on Palm OS, Microsoft Pocket PC OS and Blackberry OS. Some manufacturers of industrial handheld are: Symbol, Intermec, Itronix, Husky and others. The industrial handhelds mostly connect to the wireless LAN rather than WAN.

Pagers: A pager is a handheld wireless device that uses a paging network for data communication (Beaulieu, 2002). Pagers could be one-way, two-way or uplink. An uplink pager

is used to transmit telemetry or location information, normally used for asset management. Pagers are more cost effective, time sensitive and have more battery life than a cell phone.

Appliances: iAppliances is the generic name for the class of devices with a specialized purpose and limited Internet or wireless data connectivity. Some examples of such devices are e-book readers, e-mail stations, Internet radios, et al.

The hybrids: Series of handheld compatible phones are rolled out. They could be called as the communication devices that could compute or the computing devices that can communicate. They can run high-level applications and still work as cellular phones. Java phones are the early devices in this category that delivers voice as well as data. Trend is towards development of a Swiss army knife kind of device that combines all the benefits of the above-mentioned devices into a single ideal handheld device. (Beaulieu, 2002)

MOBILE CONTENT DELIVERY AND MIDDLEWARE:

Numerous different technologies are available for content delivery and the supporting middleware. Some of these are listed in the third appendix. A more comprehensive glossary is available at http://www.devx.com/wireless/Door/11271, (last accessed August 19, 2003)

APPENDIX 3:

GLOSSARY OF MOBILE COMPUTING TERMS AND ACRONYMS

WAP / WML: Wireless markup language and Wireless Application protocol are closely tied. They are used to display information on narrowband wireless clients like cell phones and pagers. WML is used for creating web pages for handheld devices. WAP is the application communication protocol used to access services and information. A consortium consisting of Unwired Planet, Motorola, Ericsson, and Nokia was responsible for the creation of WAP and WML. More information can be obtained at http://www.wapforum.org/

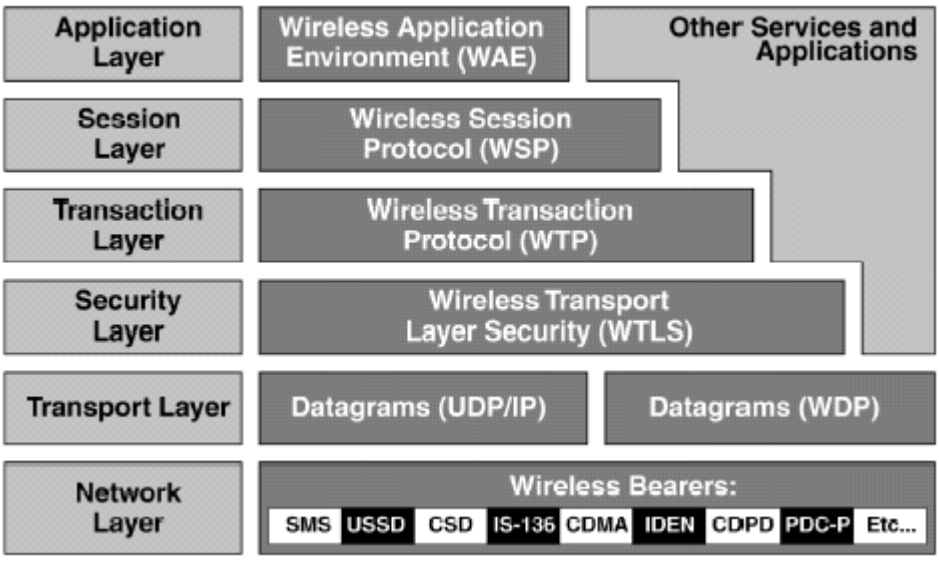


Figure 5. WAP protocol stack (WAPforum, 2000)

HDML: HDML stands for hand-held device markup language. HDML and HDTP, which is the accompanying protocol, were created by Unwired Planet in 1997. There was also a micro browser that was introduced called the "UP.browser" that runs on cell phones and similar devices.

cHTML: Compact HTML is a subset of HTML 2.0, 3.2 and 4.0. The goal of the language is quite similar to that of WML. The cHTML standard exists only as a W3C note rather than a well-established standard. Compact HTML strips down the normal HTML to the barebones making it suitable for narrowband and constrained devices. It uses normal HTTP for data transfer making it easier to serve up content for the handheld devices that support it.

VoiceXML: VoiceXML is an application of XML, so it possesses the same structure, restrictions and benefits of XML. It is designed for creating audio dialogues with human beings. It allows for a combination of synthesized speech and digitized audio (output from the server side), recognition of spoken and DTMF key input, and recording of spoken input. VoiceXML minimizes the client/server interactions by specifying multiple interactions per document. The major goal is to bring the advantages of web-based development and content delivery to interactive voice response applications.

Simplified HTML: This is a simplified version of HTML. PQA (Palm Query Application) uses a subset of HTML and is one of the main browsing languages in the Palm handheld market.

XHTML: XHTML is the replacement of HTML as the web browser language as recommended by W3C. XHTML 1.0 was a reformulation of HTML 4.01 in XML. XHTML Basic is defined as

proper subset of XHTML for mobile application presentation including web phones. More information is available at http://www.w3.org/TR/xhtml-basic/

J2ME: Sun Microsystems offers a highly optimized runtime environment targetted towards the handheld devices having limited resources. J2ME provides some core APIs, classes and technologies for wireless programming. More information at is available at Sun Microsystems' website http://java.sun.com/j2me/

iMode: imode is wireless data service developed by CoCoMo. It is packet based as opposed to circuit switched voice systems. cHTML is used to write imode pages. More information is available at http://www.ai.mit.edu/people/hqm/imode/

SynchML: is a mobile data synchronization protocol that synchronizes data between a network / desktop and a mobile device. It offers support for a variety of transport protocols and applications thereby enhancing interoperability.

ACRONYMS

CDPD: Cellular Digital Packet Data

GPRS: General Packet Radio Service

PIM: Personal Information Management

B2B: Business to Business

B2C: Business to Customer

CDMA: Code Division Multiple Access

TDMA: Time Division Multiple Access

FDMA: Frequency Division Multiple Access

GPS: Global Positioning System

GSM: Global System for Mobile Communication

SMS: Short Message Service

MMS: Multimedia Message Service

OFDM: Orthogonal Frequency Division Multiplexing

WISP: Wireless Internet Service Provider

WTP: Wireless Transaction Protocol

IDEN: Integrated Digital Enhanced Network

REFERENCES

- Amland, Stahle (1999), "Risk-Based Testing and Metrics" *EuroSTAR99 Proceedings.*
- Bach, James (1999) "Heuristic Risk-Based Testing," *Software Testing and Quality Engineering, 1* (6) 22-29, http://www.satisfice.com/articles/hrbt.pdf, last accessed July 20, 2003.
- Bach, James (2003a) "Heuristic Test Strategy Model", http://www.satisfice.com/articles.shtml, last accessed 07/17/2003.
- Bach, James (2003b) "Troubleshooting Risk-Based Testing", *Software Testing and Quality Engineering*, 5 (3), 28-33, http://www.satisfice.com/articles/rbt-trouble.pdf, last accessed July 20, 2003.
- Beaulieu, Mark (2002), *Wireless Internet: Applications and Architecture*, Addison-Wesley, 2002
- Bloom, B.S. (1956), *Taxonomy of Educational Objectives Handbook 1: Cognitive Domain,* New York: Longman, Green & Company
- Beizer, Boris (1990), *Software Testing Techniques* (2nd Ed.), Van Nostrand Reinhold.
- Biaz, Saad; and Vaidya, Nitin, H. (1997); "Tolerating Location Register Failures in Mobile Environments", Technical Report 97-015, Computer Science, Texas A&M Univ., December 1997.
- Biaz, Saad; and Vaidya, Nitin, H. (1998); "Tolerating Visitor Location Register Failures in Mobile Environments", 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98), October 1998.
- Cao, G (2000), "Designing Efficient Fault-Tolerant Systems on Wireless Networks", Proc. of the Third IEEE Information Survivability Workshop (ISW-2000), October 2000.
- Center for Highly Interactive Computing in Education (2003a), home page, http://www.research.umich.edu/proposals/proposal_dev/UM_Resources/CHICE.html, last accessed July 21, 2003.
- Center for Highly Interactive Computing in Education (2003b), downloads page, http://www.handhelds.hice-dev.org/beta.php, last accessed July 21, 2003.
- Center for Highly Interactive Computing in Education (2003c), downloads page, http://www.handheld.hice-dev.org/beta/nb/Cells%20Quick%20Start.pdf, last accessed July 21, 2003.
- Chakravorty, R. and Pratt I. (2002), "Performance Issues with General Packet Radio Service (GPRS)", Journal of Communications and Networks (JCN), pages 266-281, Vol. 4, No. 2, December 2002 (ISSN 1229-2370)
- Chalmers, D; Sloman, M (1999); "A Survey of Quality of Service in Mobile Computing Environments", IEEE Communications Surveys, **2**(1) 1999

- Cheng, Stephen; Lai, Kevin; Baker, Mary (1999); "Analysis of HTTP/1.1 Performance on a Wireless Network", Technical Report: CSL-TR-99-778, Computer systems laboratory, http://mosquitonet.stanford.edu/index.html, last accessed August 10, 2003.
- Cisco (2003) *Cisco Documentation,* http://www.cisco.com/univercd/home/home.htm, last accessed July 22, 2003.
- Collard, Ross (2002); "Performance, Load and Stress Testing", Collard and Company, Version 4.4, April 2002.
- Collard, Ross (2003); "Techniques and Processes for Reliability Testing", International Conference On Software Testing Analysis & Review May 12-16, 2003 Orlando, FL
- Dreamtech software team (2002), *Programming for Embedded system: Cracking the code,* 2002, Wiley Publishing, Inc.
- GoKnow, Inc (2003a), home page, http://www.goknow.com, last accessed July 20, 2003.
- GoKnow, Inc (2003b), "PAAM™-Palm OS Artifact and Assessment Manager", http://paam.goknow.com/files/PAAMWalkthrough_021403.pdf, last accessed o7/21/2003
- Institute of Electrical and Electronics Engineers (1990); *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries;* New York, 1990.
- ISO 9126 (1991), *International Standard ISO/IEC 9126. Information technology — Software product evaluation — Quality characteristics and guidelines for their use*, International Organization for Standardization, International Electrotechnical Commission, Geneva, Switzerland.
- ISO 9241-11(1998), *Ergonomic requirements for office work with visual display terminals: Guidance on Usability*, (1998).
- Kaner, Cem; Bach, James; and Pettichord, Bret (2001), *Lessons Learned in Software Testing*, Wiley.
- Kaner, Cem (2001); Academic course notes for SWE-5410 at Florida Institute of Technology, Fall-2001, http://www.testingeducation.org/coursenotes, last accessed July 18, 2003.
- Kaner, Cem; Falk, Jack; Nguyen, Hung Quoc (1993), *Testing Computer Software* (2nd Ed.), Van Nostrand Reinhold (reprinted by John Wiley & Sons, 1999).
- Karygiannis, Tom; and Owens, Les (2002), "Wireless Network Security", NIST Special Publication 800-48, November 2002, http://www.csrc.nist.gov/publications/nistpubs/800-48/NIST_SP_800-48.pdf, last accessed August 10, 2003
- Ko Hai-Ping (1996), "Attacks on Cellular Systems," GTE Laboratories Incorporated, http://seclab.cs.ucdavis.edu/cmad/4-1996/pdfs/Ko.PDF, last accessed August 5, 2003.
- Malloy, Alisha D; Varshney, Upkar; and Snow, Anrew P (2002), "Supporting Mobile Commerce Applications Using Dependable Wireless Networks", Mobile Networks and Applications, 7, 225-234, 2002.

- National Institute of Science & Technology (NIST, 2000) "Project: Error, Fault and Failure Data Collection and Analysis", http://hissa.nist.gov/project/eff.html, last accessed July 22, 2003.
- Neilson, Jakob (1993); *Usability Engineering*, Academic Press, Inc, 1993.
- Nguyen, Hung, Q.; Johnson, Bob; and Hackett, Michael (2003), *Testing Applications on the Web* (2nd Ed), Wiley.
- Passani, Luca (2000) "Building 'Usable' WAP Applications", http://www.topxml.com/conference/wrox/ wireless_2000/lucatext.pdf,last accessed July 18, 2003.
- Pettichord, Bret (2002); "Design for Testability", Pacific Northwest Software Quality Conference, Portland, Oregon, October 2002.
- Sterbenz, James P. G.; Krishnan, Rajesh; Hain, Regina, Rosales; Jackson, Alden W.; Levin, David; Ramanathan, Ram; and John Zao (2002); "Survivable Mobile Wireless Networks: Issues, Challenges, and Research Directions", ACM *Workshop on Wireless Security (WiSe),* Atlanta, GA, USA, September 28, 2002.
- Swaminatha, Tara M; Elden Charles R (2003); *Wireless Security and Privacy,* Addison Wesley 2003.
- Texas Instruments (2003a), "Handheld Software Applications (Apps) for the TI-89 and Voyage™ 200," http://education.ti.com/us/product/tech/89/apps/appslist.html, last accessed July 21, 2003.
- Texas Instruments (2003b), "TI-Navigator: Classroom Learning System," http://education.ti.com/us/product/tech/navigator/features/features.html, last accessed July 21, 2003.
- Varshney, Upkar; and Vetter, Ron (2002), "Mobile Commerce: Framework, Applications and Networking Support", Mobile Networks and Applications 7, 185–198, 2002.
- Vijayaraghavan, Giridharan; and Kaner Cem (2002), "Bugs in Your Shopping Cart – A Taxonomy", 15th International Software Quality Conference, San Francisco, USA, September 2002.
- Vijayaraghavan, Giridharan (2003), "A Taxonomy of e-commerce Risks and Failures", M.Sc. thesis, submitted to Department of Computer science at Florida Institute of Technology, May 2003.
- WAPforum (2000), "Wireless Application Protocol, white paper," http://www.wapforum.org/what/WAP_white_pages.pdf, last accessed August 2, 2003.
- Yang, S. Jae; Nieh, Jason; Krishnappa, Shilpa; Mohla, Aparna; and Sajjadpour, Mahdi (2003), "Web Browsing Performance of Wireless Thin-Client Computing", Proceedings of the Twelfth International World Wide Web Conference (WWW 2003), Budapest, Hungary, May 20-24, 2003.

Bugs in the Brave New Unwired World

© Dreamworks SKG

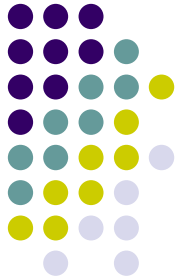
Ajay K Jha

Florida Institute of Technology

ajha@fit.edu

This research was partially supported by NSF grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

Overview



- Introduction
- What is a Mobile Application?
- FMEA and Risk Based Testing
- Using Heuristics
- Organization of the Failure Mode Catalog
- How To Use the Failure Mode Catalog
- Using the catalog with a sample Application

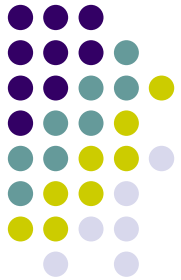
This research was partially supported by NSF grant ETA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

What is a Mobile Application?

- **Application that runs on a**
  - Personal Digital Assistant
  - Smartphone
  - Or any other handheld device
- **Using**
  - 802.11 network
  - Infrared
  - Bluetooth
  - Cellular
  - Or any other wireless network

This research was partially supported by NSF grant ETA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

Types of Mobile Applications

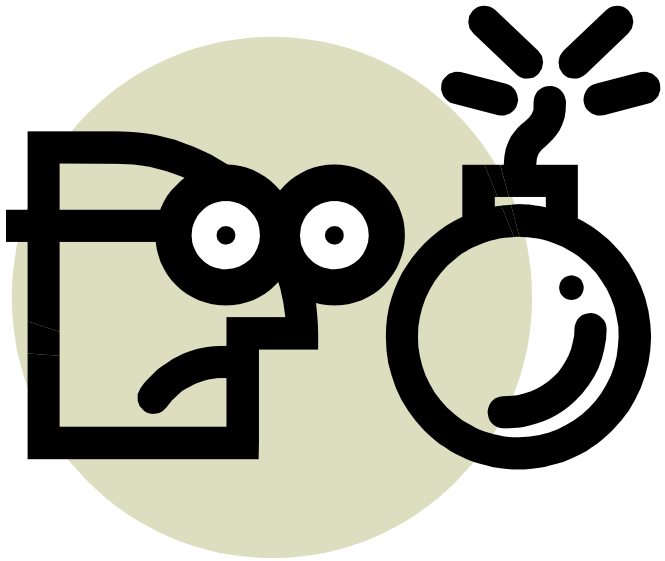


- Stand alone, running on the handheld device itself
- Connect to the back-end through synchronization software
- Work with a back-end through packet switched wireless connectivity
- Work with the back-end through circuit switched wireless connectivity
- Vertical, using special networks like SMR (Specialized Mobile Radio) or paging networks

This research was partially supported by NSF grant ETA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

Failure Mode and Effects Analysis

- A ***failure mode*** is, essentially, a way that the product can fail.
- A failure mode's ***effect*** is the probable result of that particular type of failure.



This research was partially supported by NSF grant ETA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

Risk Based Testing

- Amland (1999) explains **risk-based test management** in his paper where testing is prioritized to concentrate on the areas to test next.
- James Bach (1999) explains **risk analysis for the purpose of finding software errors**. The steps followed in his approach are:
  - Make a prioritized list of risks;
  - Perform tests to explore each risk;
  - As risks evaporate and new ones emerge, adjust your test effort to stay focused on the current risk set.

This research was partially supported by NSF grant ETA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

Using Heuristics

- "Involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods" from http://m-w.com/cgi-bin/dictionary?heuristic#, last accessed Aug 15th 2003
- Thumb rules for making decisions

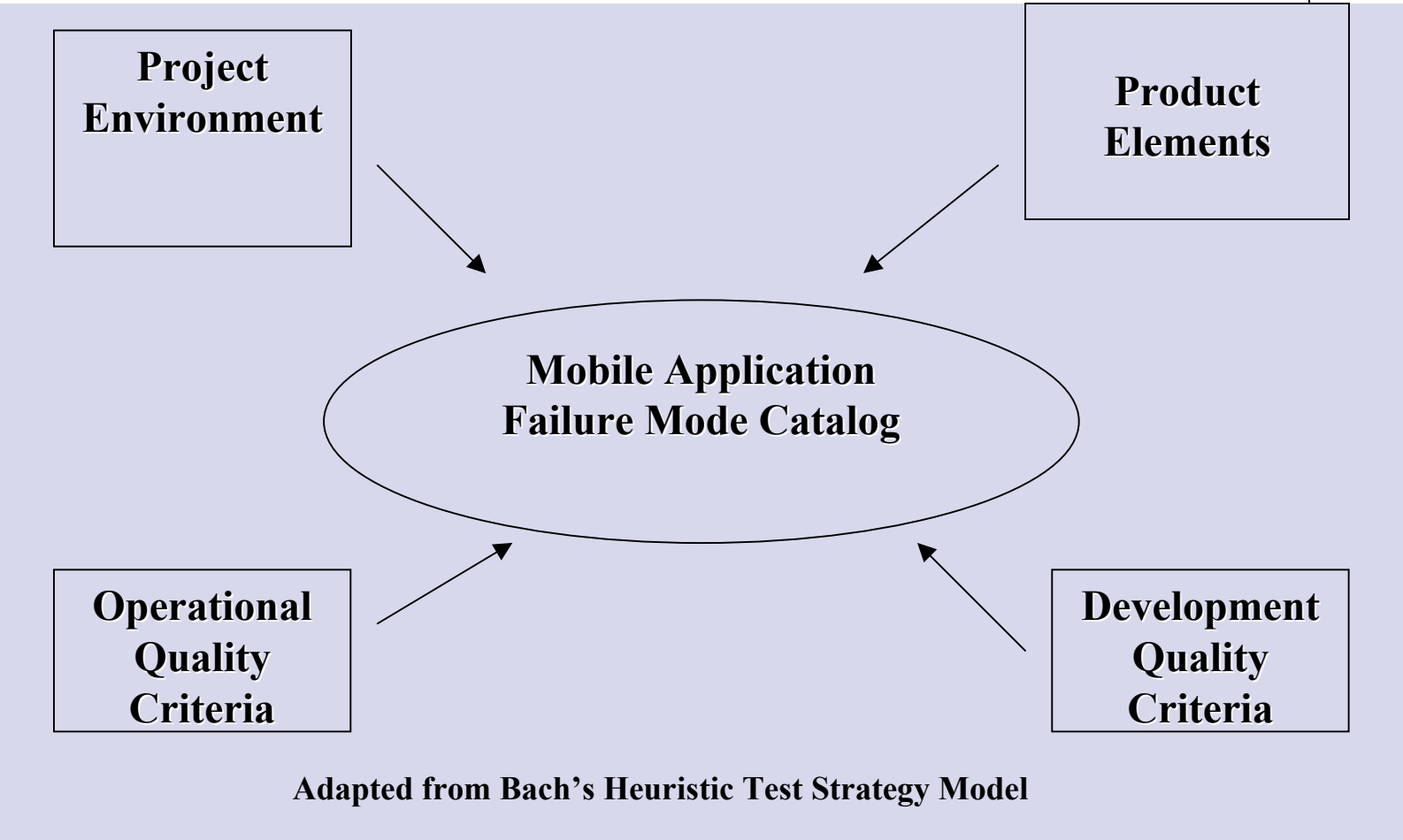
This research was partially supported by NSF grant ETA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

Structure of the Failure Mode Catalog

- Bach (2003b) points out that testers work more effectively with risk catalogs whose structure is immediately obvious to the reader.
- Bach's (2003a) *Heuristic Test Strategy Model* provides a clear structure that we can fit failure modes into.

This research was partially supported by NSF grant ETA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

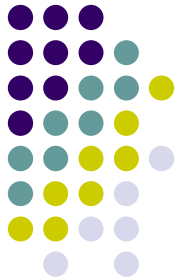
Heuristic Test Strategy Model



Adapted from Bach's Heuristic Test Strategy Model

This research was partially supported by NSF grant ETA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

How To Use the Failure Mode Catalog

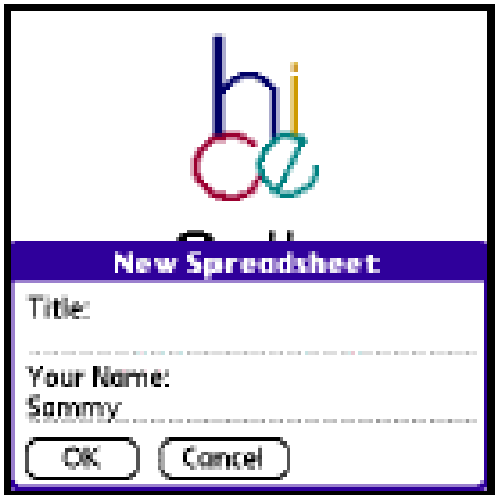


- The tester reads a description of a category of potential failures, reads some examples within the category, and asks herself:
  - Could this application fail in some way that would fit in this category?
  - Could it misbehave like any of this category's examples?
  - What about other, similar ways?

This research was partially supported by NSF grant ETA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

Sample Application under Test

- Mobile Computing in Education
- *Cells*: Spreadsheet Application for handhelds



Source: Cells 1.1 Quick Start Guide (Center for Highly Interactive Computing in Education, 2003)

This research was partially supported by NSF grant EIA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

Sample Application under Test

- *PAAM*: Palm OS Artifact and Assessment manager

Source: Walkthrough: PAAM™-Palm OS Artifact and Assessment Manager (GoKnow, 2003)

This research was partially supported by NSF grant ETA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

Using the Failure Mode Catalog as a Test Idea Generator

- **Demonstrated at the presentation**

This research was partially supported by NSF grant ETA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

Useful Links

- Please download the presentation (available by October, 2003) and a more exhaustive failure mode catalog (available by the end of 2003) at www.testingeducation.org
- James Bach's Heuristic test strategy model at www.satisfice.com

This research was partially supported by NSF grant ETA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

References

- Amland, Stahle (1999), "Risk-Based Testing and Metrics" *EuroSTAR99 Proceedings.*
- Bach, James (1999), "Heuristic Risk-Based Testing," *Software Testing and Quality Engineering, 1* (6) 22-29, http://www.satisfice.com/articles/hrbt.pdf, last accessed July 20, 2003.
- Bach, James (2003a), "Heuristic Test Strategy Model", http://www.satisfice.com/articles.shtml, last accessed 07/17/2003.
- Bach, James (2003b), "Troubleshooting Risk-Based Testing", *Software Testing and Quality Engineering*, 5 (3), 28-33, http://www.satisfice.com/articles/rbt-trouble.pdf, last accessed July 20, 2003.
- Center for Highly Interactive Computing in Education (2003), downloads page, http://www.handheld.hice-dev.org/beta/nb/Cells%20Quick%20Start.pdf, last accessed July 21, 2003.
- GoKnow, Inc (2003), "PAAM™-Palm OS Artifact and Assessment Manager", http://paam.goknow.com/files/PAAMWalkthrough_021403.pdf, last accessed Aug 10th, 2003

This research was partially supported by NSF grant ETA-0113539 ITR/SY+PE: "Improving the Education of Software Testers"

Test-first Programming by Example

Brian Marick
Ward Cunningham

Test-first programming has become popular recently, even trendy. Programmers may wonder why so many programmers are so enthused about testing, something programmers are supposed to hate. Testers may wonder how this kind of testing is different from the testing they do.
Ward Cunningham is a founder of Cunningham & Cunningham, Inc. He has also served as Director of R&D at Wyatt Software and as Principle Engineer in the Tektronix Computer Research Laboratory before that. Ward is well known for his contributions to the developing practice of object-oriented programming, the variation called Extreme Programming, and the communities hosted by his WikiWikiWeb

Brian Marick has been a tester and programmer since 1981 and a testing consultant since 1992. He specializes in programmer testing, test design techniques, and context-driven test strategies. He is the author of The Craft of Software Testing (Prentice-Hall, 1995), a coauthor of the Manifesto for Agile Software Development, and a technical editor for Software Testing and Quality Engineering Magazine. Many of his writings can be found at www.testing.com. He can be reached at marick@testing.com.

Ward Cunningham is a founder of Cunningham & Cunningham, Inc. He has also served as Director of R&D at Wyatt Software and as Principle Engineer in the Tektronix Computer Research Laboratory before that. Ward is well known for his contributions to the developing practice of object-oriented programming, the variation called Extreme Programming, and the communities hosted by his WikiWikiWeb

The Power of Retrospectives

Esther Derby,
Esther Derby Associates

"Experience is not what happens to a man; it is what a man does with what happens to him." A. Huxley.
The same is true for software teams. Too often, we don't do much — if anything — to squeeze learning out of experience. Retrospectives are a way to take "what happens" during a software development project and use it to build understanding and capability. Esther will tell us the stories of three projects and what they learned from their retrospectives and talk about how retrospectives fit in through out the life of a project.

Esther Derby helps her clients start projects on a solid footing, assess the current state of projects, and generate knowledge from project experiences. In 2001, Esther was a convener, along with Diana Larsen and Norm Kerth, of the first international Retrospective Facilitators Gathering at Newport, OR. Esther writes a regular column on Stickyminds.com and Computerworld.com. Her articles on managing software and project have appeared in STQE, Software Development, Cutter IT Journal, and Cutter Executive Update.

Esther is a Technical Editor for STQE magazine and publishes a quarterly newsletter, insights, for people who manage in software organizations. You can reach her at derby@estherderby.com or by visiting www.estherderby.com or www.estherderby.com/weblog/blogger.html.

Adding Bits of Precision to Usage Models

David Gelperin, dave@livespecs.com
LiveSpecs Software, www.livespecs.com

Abstract

For interactive systems, usage models are an important component of the requirements specification. In the spirit of agile methods and "just enough" written communication, well-understood user activities might be communicated by XP user stories and subsequent discussions. In cloudier areas, more details should be written to reduce the risk of misunderstanding.

This paper describes nine techniques that you can use to increase the precision of your usage models. Each technique can be used alone or in combination to supplement your current practice. ***Readers are assumed to have a basic understanding of use cases.***

Dave's Bio –

David is CTO & President of LiveSpecs Software in Minneapolis, MN. He has more than 35 years experience in software engineering with an emphasis on software quality, verification, and test (SQVT) including software process engineering.

Dave has been a SQVT consultant/mentor and instructor (20 yrs), quality support manager (5 yrs), verification lead (2 yrs), project lead (2 yrs), and programmer (5 yrs). He has consulted for both commercial and in-house software development organizations, including some with safety and mission critical applications. Most consulting assignments entailed assessments of software engineering or test practices and recommendations for improvement.

David received a Ph.D. (1973) and MS (1970) in Computer Science from the Ohio State University, after majoring in math at Carleton College (1964)

Adding Bits of Precision to Usage Models

David Gelperin
LiveSpecs Software
dave@LiveSpecs.com

1. Introduction

For interactive systems, usage models are an important component of the requirements specification. Consider a defined set of stakeholders, their functional needs, and a set of system functions sufficient to satisfy those needs. Usage models can be used to demonstrate how the functions can be combined to satisfy the stakeholder needs and to demonstrate the operational completeness of the set of functions relative to the needs. Usage models enable users to see if the system functions fit their work styles. Such models also support user guide development, user interface design [5, 6], system and acceptance test design [4], the usage-based reading strategy for reviewing specifications [18], and the estimation of test and development effort [17].

In the spirit of agile methods and "just enough" written communication, well-understood user activities might be communicated by XP user stories and subsequent discussions. In cloudier areas, more details should be written to reduce the risk of misunderstanding. Even if a few sentences or paragraphs are sufficient to manage misunderstanding risk, more details can provide the benefits described above. The project team should decide how much usage information should be written. Sometimes this decision will be incremental as discussion reveals unanticipated complexity or misunderstanding.

Software development that produces code always gets to precision. So the question is not if precision, but when it first appears. Usage models with more details reduce ambiguity and therefore guesswork by software engineers. This does not mean that precision is necessary everywhere. The challenge is to provide "just enough" detail in "just the right" places to maximize effective communication and enable the success of those using the requirements information. Requirements users include customers, project managers, testers, technical writers, interface designers, software engineers, and specification reviewers.

In [3], Alistair Cockburn, introduces the notion of "levels of precision in functional requirements". In a subsequent wiki discussion [2], he writes about 1-bit through 6-bit precision corresponding to increasing information content. In his scheme for use cases, 1-bit precision names the goal, 2-bit adds the success course, 3-bit adds failure conditions, 4-bit adds failure courses, 5-bit adds object (data) descriptions, and 6-bit adds stakeholder models.

This paper describes nine additional candidates for details that you can add to increase the precision of your usage models. The first four candidates are additions to a use case specification while the other five candidates describe facts about the application domain. Each candidate can be used alone or in combination to supplement your current practices.

The remainder of the paper is organized as follows: Section 2 contains a sample use case that illustrates several of the addition candidates; Section 3 describes the four case condition candidates; Section 4 describes the five types of facts about the application domain; Section 5 contains conclusions.

In the following, readers are assumed to have a basic understanding of use cases [1].

2. Use case example

The following example describes the reserve a seat functionality of a Web-based Airline Reservation System [10] and illustrates some of the candidates for use case addition. We present the entire example without comment. Then we describe individual candidates and examine specific parts of the example.

Case Name: Get [new] Seat on Reserved Flight

Risk Factors: Frequency of occurrence: 0 to 2 times per reservation
Impact of failure: *likely case* – **low**, open seating is a workaround
worst case – **medium**, open seating in a large plane with many expensive seats is likely to anger important passengers

Case Conditions:
Invariants:
None
Preconditions:
For reservation system, status is active
For passenger, system access status is signed on

Interactions

Success Course:

Passenger	**Web-based Airline Reservation System**
1. requests seat assignment	2. requests a reservation locator
3. provides a (corrected) reservation locator alternative	4. searches for reservation
Until (reservation located or all reservation locator strategies tried), actors repeat 3 to 4	
	5. offers seating alternatives, **unless** a. reservation not located or b. seat previously assigned or c. no seats are available or d. no seats are assignable
6. selects a seating alternative	7. assigns selected seat **unless** a. no seating alternative selected
	8. If (For reservation, seat previously assigned)

	returns previous seat to inventory *Post-conditions* -- For flight, previously assigned seat is available Endif
	9. confirms assignment SUCCESS EXIT

Success Course Conditions

Invariants:

For reservation system, status is active

For passenger, system access status is signed on

For passenger & flight, a reservation exists and can be located

Preconditions:

For flight, some seats are assignable

Passenger wants to get or change seat assignment

Post-conditions:

For flight, seating alternative was selected

For reservation, selected seat is assigned

Alternative Courses:

Exception Handlers (EH)*:*

EH 1 - 5a (reservation not located) – handler description omitted

EH 2 - 5b (seat previously assigned)

EH2 Invariants:

For reservation system, status is active

For passenger, system access status is signed on

For passenger & flight, a reservation exists and can be located

For reservation, seat previously assigned

Passenger	**Web-based Airline Reservation System**
	1. offers to change seat assignment
2. wants a. to change seat assignment b. no change	3. If **(**Passenger wants to change seat assignment), CONTINUE Else, provides help and SUCCESS EXIT

Figure 1. Example of the reserved seat functionality in a web-based airline reservation system

3. Use case additions

We now describe the four candidate additions to a use case specification. They are (1) risk factors, (2) course conditions, (3) invariant conditions, and (4) input alternatives.

3.1. Risk factors

To fully understand a user goal and the importance of supporting it in the context of a set of goals and support functions, one needs to know how often the user pursues the goal and the consequences of failing to achieve it. These factors should influence product interface and architectural design as well as verification and validation strategies.

Risk information, including both frequency of occurrence and failure impact (likely and worst case), should associated with each use case. This risk information should represent the post system reality, which may be quite different from the situation before the system exists. Note that all of this risk information reflects user activity in the application domain.

Failure likelihood has also been proposed for inclusion with use cases. While it measures an important form of risk, it is not risk associated with user activity, but risk associated with a particular system and its supporting technology. We therefore choose to package it with system design rather than use cases. Note that a use case reviewer that encounters failure likelihood values will require system design information to assess its accuracy.

In the precise use case example above, risk information appears as:

> **Risk Factors**: Frequency of occurrence: 0 to 2 times per reservation
> Impact of failure: *likely case* – **low**, open seating is a workaround
> *worst case* – **medium**, open seating in a large plane with many expensive seats is likely to anger important passengers

We specify frequency in a range per event or per unit of time. Additional information such as average or mode values may also be included.

Failure impact is assessed as high, medium, or low unless techniques such as Failure Modes & Effects Analysis [7] or Hazard Analysis [16] yield more refined assessments. The rationale for each estimate is provided to make accuracy assessment easier.

If frequency or impact have multiple modes, then each mode and its conditions should be described. For example, tax-reporting functions may have extremely heavy use in November and February, medium use in December, March and April, and much lighter use in the other seven months. Averaging across modes does not provide useful information.

3.2. Course conditions

Most use case formulations suggest the inclusion of preconditions and post-conditions for each case. Unfortunately, case conditions alone do not usually contain much information, because they can contain only those conditions common to all paths through the case. Most cases contain paths representing accomplishment of the goals and others representing failure. There are few conditions in common between success and multiple forms of failure.

Pre and post conditions are useful ways to explicitly define the guard conditions and effective consequences of a sequence of actions. We propose that such conditions not only be associated with the case, but with each course of action in the case. This means associating conditions with the success course and with each alternative course.

Note that this does not mean explicitly associating conditions with each path through the case, because this array of paths is not explicitly modeled. Since a path is composed of a set of complete or partial courses, including course conditions enables the conditions for each path to be derived [13].

In the precise use case example above, the preconditions and post-conditions associated with the second exception handler would be:

EH2 Preconditions

For reservation system, status is active
For passenger, system access status is signed on
For passenger & flight, a reservation exists and can be located
For reservation, seat previously assigned

EH2 Post-conditions

For reservation system, status is active
For passenger, system access status is signed on
For passenger & flight, a reservation exists and can be located
For reservation, seat previously assigned

3.3. Invariant conditions

Invariant conditions are those that are TRUE throughout a sequence of actions. We recommend that case and course conditions be specified using invariant as well as pre and post conditions. The explicit specification of invariant conditions removes the redundancy that must occur without them, i.e., an invariant condition will occur in both the pre and post conditions.

In addition, factoring out invariants strengthens the semantics of both pre and post conditions. Using invariants, each precondition must have the potential to be FALSE at the end of some course of actions and each post-condition must have the potential to be FALSE as the start of some course of actions. Being clear about what may change and what may not has value in its own right.

In the use case example above, using invariants for the second exception handler results in:

EH2 Invariants

For reservation system, status is active
For passenger, system access status is signed on
For passenger & flight, a reservation exists and can be located
For reservation, seat previously assigned

Using invariants to specifying the course conditions for success results in:

Success Course Conditions

Invariants:

For reservation system, status is active
For passenger, system access status is signed on
For passenger & flight, a reservation exists and can be located

Preconditions:

For flight, some seats are assignable
Passenger wants to get or change seat assignment

Post-conditions:

For flight, seating alternative was selected
For reservation, selected seat is assigned

3.4. Input alternatives

During use case interaction, user selections may be required between:

- available application choices, e.g. seat locations
- alternative ways to enter the same information, e.g. scan or key a credit card number
- alternative forms of the same information, e.g. see example below

Precision is increased by explicitly listing the user options in any of these situations. If the choices are common knowledge, then an explicit list is not necessary. Otherwise, explicit specification should be seriously considered.

Alternatives can be listed directly as in:

2. passenger wants
   a. to change seat assignment
   b. no change

Alternatives can also be named and then listed in an application domain glossary. For example:

3. passenger provides a (corrected) *reservation locator* alternative

with glossary entries:

reservation locator
a. reservation confirmation number

b. **flight id** and **passenger id**

flight id
a. flight date, departure airport, and flight number
b. flight date, departure airport, and destination city

passenger id
a. frequent flyer number
b. passenger name

Note that there are 5 valid forms of reservation locator.

The selection of directly listed or named alternatives depends on complexity.

4. Application domain facts

We now describe the other five candidates. Each is a type of fact about the application domain. These candidates are (1) derived values, (2) derived conditions, (3) function limits, (4) domain invariants, and (5) condition dependencies. As with named alternatives, these candidates might also be placed in an application domain glossary. Some readers will recognize these candidates as business rules [14, 15].

4.1. Derived values

The values of some attributes may be derived from the values of other attributes. Derived values explicitly show the calculation, which may include any mathematical function or operation.

For example in a library management system, the total number of copies might be specified as follows:

Total # of copies (i.e., current inventory)
= (# of available copies + # of open reserve copies
+ # of closed reserve copies + # of onloan copies
+ # of inrepair copies)

4.2. Derived conditions

Derived conditions are analogous to derived values. The truth-value of the condition is derived from an expression containing other underlying conditions. Examples include (applicant is eligible) and (order is valid). Derived conditions increase readability without sacrificing precision.

A derived condition is shorthand for a logical expression containing underlying conditions joined by logical *AND*s and *OR*s. Understanding the full meaning of the derived condition requires understanding the underlying conditions and their logical relationships. (order is valid) can be

defined by "*AND*"ed sets of valid underlying conditions that are "*OR*"ed together, i.e. conjunctive normal form.

(platform has failed) could be defined as ((power has failed) OR (hardware has failed) OR (system software has failed) OR (communication has failed)). Any of the underlying conditions might be further decomposed.

4.3. Function Limits

There are often limiting or boundary values that control the behavior of a function. These limiting values should be associated with explicit attributes of individual transaction objects or aggregates of these objects. Examples include aggregate reservations limit for an aggregate of flight reservations, purchase limit for a single purchase, loan limit for a single loan, and loan type aggregate limit for all loans of a particular type. We recommend the explicit specification of function limits because too often such rules only exist implicitly in the processing code.

As with other attributes, these values might be bounded or derived. For example, in many airline reservations systems, there is a derived value rule like the following:

aggregate reservations limit:
for each flight, the number of active reservations must not be greater than (100 + oversell percentage) x (total number of usable seats).

This limit constrains the reservation booking function.

4.4. Domain Invariants

In an application domain, each class has attributes and definitions for valid values of these attributes. In addition, there may be additional (business) rules that define invariant relationships between attribute values within a class or across classes. These rules may be conditional, i.e. TRUE under some conditions, or unconditional, i.e. always TRUE.

Domain invariants are unconditional rules defining invariant relationships between attribute values across classes. For example:

For every book:
no patron borrows more than one copy
only librarians can reserve, fix, or remove copies
For every flight:
no available seat is assigned
no assigned seat is available
no seat is assigned to more than one passenger

Although domain invariants could appear among the invariant conditions in every use case in the application domain, we recommend that they be specified with class definitions or in an application domain glossary.

4.5. Condition Dependencies

Sometimes there are dependencies between conditions that should be explicitly documented. For example, in the success course of the use case above, we find:

5. *Web-based airline reservation system* offers seating alternatives, **unless**
   a. reservation not located, or
   b. seat previously assigned, or
   c. no seats are available, or
   d. no seats are assignable

There is a dependency between the last two conditions.

> If "no seats are available" then "no seats are assignable".

This means that if the third condition is TRUE, then the fourth condition must also be TRUE. If seats are available however, seats may or may not be assignable.

We recommend that inter-condition dependencies be explicitly specified because, as with domain invariants, it increases their visibility and supports their effective review.

5. Conclusions

One or more of the nine detailing candidates may be used to help avoid recurrent or anticipated misunderstandings or to increase the benefits of usage modeling.

6. References

[1] Cockburn, Alistair **Writing Effective Use Cases** Addison-Wesley 2001

[2] Cockburn, Alistair "Comments on User Story and Use Case Comparison" wiki page Available at http://c2.com/cgi/wiki?UserStoryAndUseCaseComparison)

[3] Cockburn, Alistair "PARTS: Precision, Accuracy, Relevance, Tolerance, Scale in Object Design" Available at http://members.aol.com/acockburn/papers/precisio.htm

[4] Collard, Ross "Developing Test Cases from Use Cases**"** Software Testing & Quality Engineering July/August 1999 pp. 30-37

[5] Constantine, Larry and Lockwood, Lucy **"**Structure and Style in Use Cases for User Interface Design**"** Available at www.foruse.com/Resources.htm#style

[6] Constantine, Larry and Lockwood, Lucy **Software for Use** ACM Press Addison Wesley 1999

[7] Failure Mode and Effect Analysis (FMEA) Information Centre (www.fmeainfocentre.com)

[8] Gelperin, David "A Q&A Intro to Precise Usage Modeling" Available for download at www.LiveSpecs.com

[9] Gelperin, David "Just Enough Precision" Available for download at www.LiveSpecs.com

[10] Gelperin, David "Precise Use Cases" Available for download at www.LiveSpecs.com

[11] Gelperin, David "Precise Use Case Examples" Available for download at www.LiveSpecs.com

[12] Gelperin, David "Precise Usage Model of a Library Management System" Available for download at www.LiveSpecs.com

[13] Gelperin, David "Modeling Alternative Courses in Detailed Use Cases" Available for download at www.LiveSpecs.com

[14] Gottesdiener, Ellen **Business Rules show Power, Promise** Applications Development Trends March 1997 [www.adtmag.com/pub/mar97/softeng.htm]

[15] Guide **Business Rules Project – Final Report** October 1997 Available at www.essentialstrategies.com/publications/businessrules/index.htm

[16] Leveson, Nancy G. and Peter R. Harvey, "Analyzing Software Safety," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, September 1983.

[17] Schneider, Geri and Winters, Jason P. **Applying Use Cases** Addison Wesley 1998

[18] Thelin, Thomas, Runeson, Per, and Regnell, Bjorn "Usage-based reading – an experiment to guide reviewers with use cases" Information and Software Tech. 43 (2001) pp. 925-938

The Clear Requirements Challenge

David Gelperin, dave@livespecs.com
LiveSpecs Software, www.livespecs.com

The Challenge

1. Specify the merchandise return capability (described below) using **your current techniques** or any others.
2. Have the Clear Requirements specs (below) and your specs judged by a panel of stakeholders who need the requirements information (i.e. project managers, product architects, product testers, and technical writers)

The Example

Consider the functional requirements for the merchandise return capability within a sales support system. There is a spectrum of specification precision from Title to Detail as follows:

Title	Merchandise Return Support Capability
Summary	Sales system must support merchandise returns. Missing receipts and manager overrides should be addressed.
Detail (natural language)	Sales system must support returns of unopened, unused, or defective merchandise for up to 14 days from date of purchase. With a valid sales receipt, returns must be accepted at any store. Without a valid sales receipt, returns must be accepted only at the store where purchased, and then only if the customer's sales record is located in the system and the customer provides a valid photo-id. Any store manager can override these rules and accept a return. [Etc.]
Detail (semi-formal language)	[Below, we show how an analyst might express these details using *clear requirements* techniques, starting with application glossaries and action contracts, followed by precise use cases and then using manual test procedures.]

[Please read the glossary examples in the appendix so the following description will be clearer.]

When specifying this Merchandise Return (MR) capability, many of the glossaries would already be partially populated, because MR is unlikely to be the first capability described. These populated glossaries include stakeholders, non-stakeholder sources, actors, application tasks, essential classes, and system functions, along with several of the terminology glossaries.

The analyst would add *return days limit* to the **function limits glossary** defined as the (arbitrary) constant 14. Also, because there are six decision factors, the analyst would add *valid return* and *invalid return* to the **derived conditions glossary** as follows:

For the function – Merchandise Return,
valid return is any of:

After sale interval	**Receipt status**	**Return location**	**Sales record status**	**Photo-id status**	**Manager override**
=< return days limit	valid				not given
=< return days limit	invalid or missing	purchase location	found	valid	not given
					given

invalid return is any of:

After sale interval	**Receipt status**	**Return location**	**Sales record status**	**Photo-id status**	**Manager override**
> return days limit					not given
=< return days limit	invalid or missing	not purchase location			not given
=< return days limit	invalid or missing	purchase location	not found		not given
=< return days limit	invalid or missing	purchase location	found	invalid or missing	not given

The analyst would add *after sale interval* to the **derived values glossary** as:
For the function -- Merchandise Return,
after sale interval = (current date – sale date)

Then the behavior of the Merchandise Return capability could be specified with **action contracts** as follows:

Invariants		Post-conditions
Return status	**Receipt status**	**Customer information**
valid	invalid or missing	recorded

Invariants		Post-conditions
Return status	**Compensation request**	**Compensation**
valid	refund	payment refunded
valid	replacement	item replaced or payment refunded

Invariants		Post-conditions
Return status	**Returned item type**	**Returned item "to be" disposition**
valid	defective	returned to manufacturer
valid	unopened	reshelved
valid	unused	repackaged, put on sale, or returned to manufacturer

Invariants	Post-conditions
Return status	**Return information**
valid	details recorded

Invariants	Post-conditions
Return status	**Response**
invalid	apologized for inconvenience

Invariants					Post-conditions
Return status	**After sale interval**	**Return location**	**Sales record status**	**Photo-id status**	**Suggestion**
invalid	=< return days limit	not purchase location			try to find sales slip or return at purchase location
invalid	=< return days limit	purchase location	not found		try to find sales slip
invalid	=< return days limit	purchase location	found	invalid or missing	try to find sales slip or bring valid photo-id

Then, the analyst might specify usage of this capability with a **precise use case** as follows:

Case Name: Return purchased items

Risk Factors: Frequency of occurrence: 0 to 3 per 100 transactions (average)
10 to 15 per 100 transactions (peak)
Impact of failure: *likely case* – **low**, if receipt, save copy for entry when function available
worst case – **low**, if no receipt, ask to return when function available

Case Conditions:
Invariants: None
Preconditions: Customer has items to be returned

Interactions

Success Course:

Customer	**Sales System**
1. requests merchandise return	2. requests sale info
3. provides sales a. receipt or b. date and location	4. accepts return, **unless** a. return is not valid
	5. requests compensation preference
6. selects a. refund or b. replacement	7. provides compensation, records return details and thanks customer for their business SUCCESS EXIT

Success Course Conditions
Invariants:
For sales system, status is active
For sale, return is valid
Pre-conditions:
Customer has items to be returned
Post-conditions:
For sale, items returned
compensation provided
return details recorded

Alternative Courses:

Outcome-Changing Action (OCA)

OCA1. After 3, **manager overrides rules**

Invariants:
For sales system, status is active
Customer has items to be returned
Preconditions:
Return is invalid
Customer or clerk requests a manager
Manager decides to override return rules
Post-conditions:
For sale, return is valid
Override is recorded

Customer	Sales System
	1. records override

OCA2. Before 4, **system records customer information**

Invariants:
For sales system, status is active
Customer has items to be returned
For sale, return is valid
Receipt is missing or not valid
Post-conditions:
Customer info is recorded

Customer	Sales System
	1. requests customer info
2. provides name, address, phone, and email	3. records customer info

Exception Handlers (EH):

EH 1 - 4a (**return is not valid**)

Invariants:
For sales system, status is active
For sale, return is not valid

Customer	Sales System
	1. Apologizes for inconvenience
	2. If (after sale interval > return days limit) FAILURE EXIT Endif
	3. Suggests trying to find the sales receipt
	4. If (not purchase location) also suggests returning to purchase location Else If (sales record found) also suggests returning with a valid photo-id Endif FAILURE EXIT

Finally, the analyst might specify **test procedures** for the capability as follows:

Id: ATP 2.10 - Successful Merchandise Return

Objectives & Scope: To acceptance test successful merchandise return scenarios.

Special setup steps: None

Invariants:
For sales system, status is active
For sale, return is valid
Pre-conditions:
Customer has items to be returned

Run &/or Check steps: Customer --
1. requests merchandise return
2. provides sales information
SO2.2. provides customer information, if asked
6. selects compensation

Post-conditions:
For sale, items returned
compensation provided
return details recorded
customer information recorded, if receipt not valid
override recorded, if manager decided to override

Special wrap-up steps: None

Scenario parameter values:

Scenario	Sales info	Mgmt override	Form of compensation
1	valid receipt	no	replacement
2	date & location	no	refund
3	date & location	yes	replacement
4	valid receipt & > return days limit	yes	refund

Appendix

Types of Glossaries

1. Application Domain Elements

   a. **Stakeholders**
      (name, contact info, description, goals/objectives, values, system/product roles, comments)
   b. **Non-stakeholder sources** e.g., systems or documentation
      (name, access info, description, comments)
   c. **Actors**
      (name, type, description, responsibilities, relationships, access rights, comments)
   d. **Application tasks**
      (name, description, occurrence frequencies, comments)
   e. **Essential classes**
      (name, attributes, values, facts, states, transitions, comments)

2. Application Domain Facts -- facts relating to multiple classes can be specified in the following glossaries, facts related to a single class should be specified with the class

   a. **Derived values** i.e., value calculated from other values
      (classes or function name, value name, defining arithmetic expression, comments)

      E.G. For copy and borrower, in a Library Management System
      Average # of copies per active borrower =
      (# of onloan copies) / (# of borrowers, for which loans > 0)

   b. **Derived conditions** i.e., Boolean functions calculated from other Boolean functions
      (classes or function name, condition name, defining logical expression, comments)

      E.G. For platform, power, hardware, system software, and communication,
      platform has failed =
      a. power has failed or
      b. hardware has failed or
      c. system software has failed or
      d. communications has failed

c. **Function limits**
(function name, limit name, defining expression for min. or max., comments)

E.G. For flight and reservation, in an airline reservation system
The aggregate reservation limit is
(# of active reservations not >
((100 + oversell percentage) x (total # of usable seats)))

d. **Domain invariants** i.e., application domain conditions that should always be TRUE
(classes, invariant condition, comments)

e. **Condition dependencies** i.e., truth value dependencies between two conditions
(classes, dependency expression, comments)

3. System Elements

a. **System/product functions**
(function name, function group name, description, invariant, pre, & post conditions, comments)
b. **Function groups**
(group name, description, access rights, comments)
c. **Trigger events**
(event name, description, comments)
d. Input alternatives
(input name, alternatives, comments)

E.G. Input alternatives can be named and then listed in a glossary. For example, locator information for a reservation can be provided in several ways.

passenger provides a (corrected) *reservation locator* alternative

The associated alternative input entries are:

reservation locator
a. reservation confirmation number
b. flight id & passenger id

flight id is flight date, departure airport, and
a. flight number
b. destination city

passenger id

a. frequent flier number
b. passenger name

Note that there are 5 valid forms of reservation locator.

4. Terminology

a. Application terminology
(term, definitions, comments)

b. **Application acronyms**
(acronym, full phrase, definitions, comments)

c. System terminology
(term, definitions, comments)

d. **System acronyms**
(acronym, full phrase, definitions, comments)

5. Trigger Words

a. **Risky words** – weak, vague, or ambiguous words
b. **Technical words** – words that should not appear in use cases
c. **Ignorable words** – words to be skipped during analysis e.g., the

Requirements Elicitation for "Customer Delighting" Product Families

Orhan Beckman, Ph.D.
Bhushan Gupta
Steve Sheffels
Hewlett-Packard Company
orhan@hp.com, bhushan_gupta@hp.com, steve_sheffels@hp.com

Abstract

This paper emphasizes the importance of guiding the development of a family of products using a common set of cross-product requirements. Lack of such requirements introduces challenges related to meeting customer expectations for consistency and interoperability after release. To further establish this concept, examples are cited from research and development at Hewlett-Packard, supporting a methodology that leads to a common set of requirements. The methodology is based on application of a set of quality monitors to the product lifecycle, COILUSD (Compare, Own, Install, Learn, Use, Support, Dispose). Additional monitors can be included for supporting special product environments. Once requirements are established, an abstraction is adopted across the organization to drive the requirements for an entire product family.

Requirements Elicitation for "Customer Delighting" Product Families

Orhan Beckman, Ph.D.
Bhushan Gupta
Steve Sheffels

Abstract:

This paper emphasizes the importance of guiding the development of a family of products using a common set of cross-product requirements. Lack of such requirements introduces challenges related to meeting customer expectations for consistency and interoperability after release. To further establish this concept, examples are cited from research and development at Hewlett-Packard, supporting a methodology that leads to a common set of requirements. The methodology is based on application of a set of quality monitors to the product lifecycle, **COILUSD** (**C**ompare, **O**wn, **I**nstall, **L**earn, **U**se, **S**upport**, D**ispose). Additional monitors can be included for supporting special product environments. Once requirements are established, an abstraction is adopted across the organization to drive the requirements for an entire product family.

Introduction:

Requirements elicitation has been a topic of interest to software developers in the past few years. Communication throughout the spectrum of software engineering communities is ongoing. Forms of communication include books [1], journal articles; templates assembled by industry experts [2] and everyday discussions at software development sites. There are a variety of ways to classify and represent requirements, such as use case scenarios [3], functional requirements, non-functional requirements, UML diagrams [4], requirements language, and other related approaches. Multiple techniques including prototyping, use case modeling, UML modeling; quality monitors, FLURPS (Hewlett Packard), and CUPRIMDO (IBM) are used to derive these representations [6], [5]. These are all based on user interaction and perception to achieve intended functionality when the software is being used in the field. Aspects of product interaction across the product lifecycle, such as comparing products in the marketplace, ordering, installing, learning to use, fixing, disposing once obsolete, and replacing, can all play an important role in enabling the software to deliver its value to the customer. The significance of the product lifecycle stages is often missed, and the implications of missing the requirements associated with these stages are often realized in the most expensive environments, systems testing and end user software problems in the field.

Companies typically have a family or multiple families of products. Products in a family share common requirements, whether understood or not, tacit or vocalized, acted upon or reacted upon. Customers need and expect a level of consistency for products that are intended to coexist with or replace other products. This is particularly true of product lifecycle stages in

which the desired customer experience is to navigate the stage expending a minimum of effort, time, money and emotion. This applies to the setup phase of choosing, ordering, and installing, the maintenance phase of servicing and supporting, and the transition phase of disposing and replacing. Customers also like to utilize the effort, time, money and emotion spent in navigating one stage of a particular product in a family when navigating similar stages of other products in the family. This second customer expectation, to be able to transfer learning and investment made in one product to another, applies to all aspects of the product lifecycle.

Products within a family share actors, use cases, objects, and actions. The process of developing requirements to "delight" customers begins by identifying shared elements both within and between products in the family. Areas of overlap represent potential opportunities for building consistency and interoperability into the customer experience. A family of products developed in isolation faces a challenge to meeting customer expectations for consistency and interoperability after release. A product team designing for a segregated product line behaves more as an individual company with a narrow definition of product success than as a member of a larger team contributing to cross-product solutions in the marketplace. A narrow focus on individual product value, or quality, often comes at the expense of the value each product could provide by working in concert with others in the family. Even when cross-product design goals are desired in a company, product family characteristics are often not considered until well into the lives of both the individual products and the product families, resulting in products that are more difficult to market together as a solution, more frustrating to operate in concert and more costly to support.

Driving product design to meet customer expectations benefits the organization by both increasing customer satisfaction and reducing expenses associated with marketing and supporting products. A two-tier approach to product requirements is essential for meeting these requirements in a family of products. The first tier is driven by the standards applied consistently for every product in the family, and the second tier is unique to individual products. Developing requirements and associated design implications helps to define what must be achieved throughout the product family in order to meet customer expectations for cross-product consistency and operability.

Example:

An example based on the Hewlett Packard printer family shows that when a myopic view on product quality is taken, the result is a negative impact to the customer in regards to cross-product consistency and interoperability. Until relatively recently (1998), LaserJet and DeskJet printer engineering teams independently developed and shipped their installation software. The installer was not considered a core component of the printer software solution, and dedicated

resources were assigned for each product. The teams that developed the installer software worked in relative isolation, paying attention only to the printer they were developing an installer software for without regard to the common usability aspects shared with other HP printer products. The unfortunate result was inconsistent product behavior for HP's customers, not to mention the costly and redundant development of non-core software. An intentional effort initiated across the product family allowed Hewlett Packard to streamline processes, cut costs and provide a consistent usability experience to meet customer expectations.

Basic Assumptions:

A number of assumptions about product families are made in this paper. These assumptions help lay the groundwork for the theoretical constructs explored and guidelines provided. The assumptions are:

- A single customer is likely to use two or more products within a product family developed by different teams in a software company.
- The communication between product development teams, although not restricted, does not naturally flow without formal intervention or structure.
- There is likely to be a common or shared customer support infrastructure for two or more products within the product family.
- The value that a family of products delivers in the marketplace is greater than the sum of the value of the individual products alone.

Methodology:

The invented methodology for the requirements elicitation process captures the total customer experience for a family of products, is scalable and can be applied at different levels of abstraction. *Elicitation* of individual product requirements is the first step in the process of deriving cross-product requirements. These attributes are later elaborated, in the context of the product lifecycle, COILUSD, to capture the total customer experience , qualified, to serve as design guidelines and quantified,to serve as release criteria. Major steps in the methodology, described later in more detail, are:

STEP 1: Define bounds of product family and identify stakeholders
STEP 2: Appoint and gather stakeholder-representatives for each product
STEP 3: Enter requirements for each product on a COILUSD-FLURPS grid
STEP 4: Compare and contrast the product grids to identify areas of commonality
STEP 5: Communicate and validate the defined requirements with stakeholders

STEP 1: Define bounds of product family and identify stakeholders

To begin the process, the scope of the requirements definition effort is defined. The first step in scoping is to define the members of the product family. In general, products that will be used or supported together are good candidates for belonging to the same product family. Family members may include hardware, software and service offerings.

The second step in scoping is to define the stakeholders for each of the products in the product family. A product stakeholder is defined as someone who invests in the product and expects an acceptable level of return on his or her investment. An obvious stakeholder is the end user of the product. The end user invests money, time and/or effort in the product to reach a goal and expects an acceptable level of performance or return on the investment made. Other stakeholders for a given product may include those who finance the product, develop the product, test the product prior to release, market the product, install the product, train customers to use the product, and service and support the product. At the end of step 1, the product family and the stakeholders for each product in the family should be defined.

STEP 2: Appoint and gather stakeholder-representatives

The goal of step 2 is to identify a representative for each stakeholder group. This is likely to entail working with different functional areas of the organization tasked with bringing the product to market. Overrepresentation is to be expected, as each product in the family may have a representative for each functional area. Note that having End User stakeholder representatives is critical to the success of the requirements gathering exercise. While personnel from the Marketing Department can fill this role, it is desirable to have actual End Users of legacy or competitive products participate in the exercise. At the end of step 2, representatives for each of the groups of stakeholders should be identified.

STEP 3: Enter requirements for each product on a COILUSD-FLURPS grid

Step 3 identifies stakeholder requirements for each product in the product family. Dimensions of quality: Functionality, Localization, Usability, Reliability, Performance, Supportability, Security and Scalability are intersected with product lifecycle stages: Install/Setup, Learn, Use, Maintain and Dispose, to create a FLURPS x COILUSD Quality/Lifecycle (QL) matrix (see example, Table 1). The QL matrix defines the requirements problem space to be evaluated and documented. Quality dimensions may have zero or more requirements at each stage of the product lifecycle. The quality dimension (FLURPS) is widely known at HP and we extended it to FLURPSS to accommodate dimensions important to internet applications. The technique described in this paper is capable of accommodating variations in both lifecycle and quality dimensions required to meet the needs of different contexts.

	Install/Setup	Learn	Use	Maintain	Dispose
Functionality	By non-technical users	Single usage self learning	Order print material at a desired PSP	Web version update	No client delete required
Localization	English, German, French, Spanish & Italian (EGFSI)	EGFSI	EGFSI	EGFSI	EGFSI
Usability	Single Screen	On line help	Simulate Print Service Provider workflow	New versions must have consistent Graphical User Interface.	2 Click uninstall
Reliability	Single Attempt	All help available via links	No loss of assets.	Accurate data migration	Uninstall leaves no files behind.
Performance	Under 30 minute install	No formal training necessary	Average job upload time 45 seconds	Upgrade down time 30 minutes	Uninstall downtime 30 minutes
Scalability	Multiple environment support	Consistent across various configurations	Similar experience across configurations	Control over all configurations	Flexible dispose of configurations
Security	Encrypted passwords	Privacy of learners	Data and Asset confidentiality	Upgrades have same security level	No loss of data Confidentiality
Supportability	24x7 Support	Frequently Asked Questions	24x7 Support	24x7 Support	24x7 Support

Table 1. An abridged QL matrix with a sample requirement in each cell.

Stakeholder representatives for each product gather in individual groups to populate their product's QL matrix with requirements. Requirements should be statements and take the form of <stakeholder> must be able to <verb> <noun> <modifier>. For example the End User representative may write a statement in the Usability-Installation cell describing usability requirements for installation of the product as follows: *The customer must be able to install the product in less than 5 minutes*. The End User representative may add as many statements as needed to adequately define the requirements for each aspect of quality within the lifecycle. These statements should be succinct and should describe the desired experience only in terms of "what" must happen not "how" the product should be designed to meet the requirement. Requirement statements should be devoid of design details. This is a requirements elicitation exercise, and not a design exercise and therefore design details at this point in product development only serve to defocus the group from determining what requirements need to be

met. There may be many possible designs that can be employed to meet a single requirement and it is often difficult to determine if a single design actually meets a requirement if the requirement is not first specified and agreed upon.

Product stakeholders may work in parallel adding requirements to the QL matrix to save time, however experience suggests that the discussion that ensues by taking a more serial approach is rich and well worth the extra time it requires. The requirement definition process continues until each stakeholder representative is satisfied that the requirements are adequately documented. Stakeholders should capture where requirements redundancy occurs in the matrix (where redundancy means that one requirement is shared by two or more stakeholders of a product). By the end of Step 3 each product should have a QL matrix containing the requirements placed on the product by each of the product's stakeholders.

STEP 4: Compare and contrast the product grids to identify areas of commonality

The goal of Step 4 is to identify and use areas of commonality within and between the product requirement spaces to define product/family level requirements. The product stakeholder-group members gather together at this point to share and compare product QL matrices. Based on experience, the authors suggest that the QL matrices be hung next to each other on a wall where members from each product stakeholder group may 'walk the walls' to compare and contrast the requirements derived for each product, and discuss patterns as they emerge. Examining common rows (quality attributes), columns (lifecycle stages) and cells between product QL matrices will help stakeholders identify aspects of areas of commonality and distinction between requirements for different products.

Stakeholders should at this point consolidate requirements within and across products in three ways:

1) Identify redundancies between product requirements

2) Identify new or emerging themes from groups of requirements

3) identify where a group of requirements in one part of the matrix should be generalized to other parts of the matrix to reinforce the requirement. If, for example, the Support stakeholders after examining the support requirements products can identify a common goal or goals underlying the individual product requirements, then they would have the opportunity to define a *cross-product requirement*.

A *cross-product requirement* has advantages over individual product requirements where overlap exists. First, it is easier to communicate and manage a common requirement across products than it is to manage individual product requirements that differ in only minor ways. Second, common requirements help to support common metrics or criteria for success across a

family of products. Lastly, for many classes of requirements such as reliability, compatibility and usability, variance in the level to which those requirements are met across products in the product family reduces the significance of the overall value the inter-product solution can provide, detracting from the requirement.

The QL matrix is a convenient structure in which to house the product-family requirements. The individual product requirements that do not overlap or otherwise correlate with other product requirements should be left in the individual product's QL matrix as product-specific needs to be considered (second tier needs). At the end of Step 4, the product-family requirements should be identified and documented.

STEP 5: Communicate and validate the defined requirements with stakeholders

The goal of step 5, the final step in the requirements elicitation process, is to communicate and validate the product-family requirements with stakeholders outside the appointed requirements group. Further refinement of the product-family requirements is to be expected as the results are reviewed. The end goal is adoption of the product-family requirements by the members of the organization developing, testing and marketing the products.

On a practical note, the authors have found that the QL matrix that defines the product-family requirements is a convenient template that can be used to kickoff product elicitation efforts for new products in the family. Using this QL matrix as a starting point has the advantage of leveraging the requirements work that has already been done while helping to ensure that the product-family requirements are adopted by new product teams.

Once a complete family-level requirements definition has been identified, documented and communicated, the results are used to influence product design across the product family.

Application of the results:

An early and less rigorous version of the method described in this paper was applied in HP to the domain installer software development. This application yielded product-family requirements including ”consistent user interface”. This requirement was utilized to influence product development at different levels, in a more direct way than previous methods had produced. At the most cursory level, product development groups typically agreed on common user interface elements. A common user interface style guide covering terminology, use of color and user interface elements was developed and followed. The end result of meeting the “consistent user interface” requirement using this approach was products that appeared to be, but did not necessarily act like, members of the same family.

Using the QL matrix method described in this other groups in HP conducted a feature-by-feature analysis of products in a family to promote family requirements for greater consistency. The groups then redesigned common features in a consistent manner and proactively sought to

design new common features to be family, not just product, compatible. The result in meeting the requirement at the family level was products that both looked and behaved in a predictable manner, and that felt like members of the same product family.

Perhaps the deepest level of influence that the use of family-level requirements has had over product development is in the area of code sharing and, in one case, code reuse. Groups found it more efficient, where possible, to share software development for common elements rather than develop them separately, applying user interface styles to satisfy the "consistent user interface" requirement. The key end result of meeting the requirement at this design level was increased development efficiency. While the end result for the "End User" stakeholder was basically the same as with the previous method, this method enabled internal stakeholders — software engineers, quality engineers, documentation authors and human factors engineers — to achieve their development goals with more efficiency while meeting the "consistent user interface" requirement.

The authors have found that there is no one "right" way to meet a family-level requirement, however the previous example helps to illustrate an applied learning in terms of the costs and benefits associated with various methods.

Winning Advantages:

Areas of *requirements overlap* represent opportunities to drive quality across a family of products and highlight potential opportunities for increased development efficiencies. Using the requirements elicitation approach requires fewer resources for developing a common solution than developing independent solutions for each product in the family. Family-level requirements have been used in HP to drive a 'customer-pleasing' level of consistency across both DeskJet and LaserJet products. Focused attention on installation requirements across a family of products turned a problematic situation, in which driver installation from product to product was inconsistent and unpredictable, into an installation experience that was standardized, optimized and predictable. The end result had benefits for all of the various stakeholders, including those testing the products, selling the products, supporting the products, and using the products. Unforeseen benefits included increased opportunities for code reuse on common installation components and more efficient use of engineering, quality and usability testing resources. All of these benefits are derived from family level requirements. We believe the elicitation process described in this paper is an effective and efficient method for developing these requirements.

The methodology attempts to cover a broad swath of the customer experience and the requirements supporting it. The technique provides a level of abstraction where the common quality parameters of all products can be established and owned by different teams if desired. In situations where the methodology is being used to establish division-wide goals, it needs

sponsorship from upper management and support from key stakeholders. A periodic review of the quality goals is necessary to evaluate whether the goals can be realized within the resource constraints.

Conclusions:

Most recently the authors have used the methodology described in this paper to successfully establish the division level requirements for Digital Publishing Solutions (DPS). These requirements now influence the design of products and solutions across DPS. Since this methodology has potential for being scalable, it can be used to establish the common quality goals across the suite of products that are used to create customer-specific solutions. These products include Digital Presses, Large Format Printers, Remote Proofing Printers, Creative Application Plug-ins, Printer Drivers and Workflow Solutions. Each team developing a component is responsible for meeting the product family requirements. The very nature of the process provides an opportunity for all of the stakeholders, including a real or representative customer, to participate in setting the product quality goals and requirements, resulting in a greater buy-in from everyone. The authors hope this method will be more widely adopted enabling others in the industry to achieve greater customer satisfaction and development efficiency – a formula in which both parties win.

References:

1. Wiegers, Karl; Software Requirements, 2nd Edition, , Microsoft Press, 2003
2. Software Requirements Specification Template, http://www.processimpact.com/goodies.shtml
3. Kulak, Dary, & Guiney, Eamonn; Use Cases, Addison Wesley, 2000
4. Booch, Grady; Jacobson, Ivar & Rumbaugh, James; UML Distilled, Addison Wesley, 1998
5. Gatlin, Many & Hannon, Greg; Managing Software Product Quality, PNSQC 2000
6. Grady, Robert B.; Practical Software Metrics for Project Management and Process Improvement, 1992, Prentice Hall

Control Your Requirements Before They Control You

Les Grove, Doug Reynolds, David Suryan
Tektronix, Inc.
P.0. Box 500 M/S 39-548
Beaverton, OR 97077

les.grove@tek.com; douglas.f.reynolds@tek.com; david.suryan@tek.com

Abstract

A requirements specification should be the definitive source for how a product behaves; but requirements are living artifacts of a project and can be in a constant state of change. How well a team manages requirements can determine whether a project is a success or a failure. Well-managed requirements improve budgeting, moral, product quality, and time-to-market. But requirements don't control themselves and managing them can be difficult if the requirements are not well-written or organized for change.

Though there may be other solutions, this paper provides practical information and guidelines that minimize the negative effects of changing requirements through better-written and better-managed requirement specifications. This paper also covers how to perform requirements traceability and requirements change management.

The methods, guidelines, and results described in the paper represent the work of a group of engineers at Tektronix to improve requirements management practices company-wide. The goal was to find real-world, practical methods that were being used internally or externally and bring them into the Tektronix culture within one calendar year.

Biographies

Les Grove is a software engineer at Tektronix, Inc., in Beaverton, Oregon. Les has 18 years of experience in software development, testing, and process improvement. He has a Bachelors degree in Computer Science and is currently pursuing a Masters degree in Software Engineering.

Doug Reynolds is a Software Quality Engineer at Tektronix Inc., in Beaverton Oregon. Doug has worked for Tektronix for the past 11 years. He holds a Master of Computer Science and Engineering from Oregon Health & Science University (OHSU) and a Bachelors of Science in Electrical Engineering Technology from Oregon Institute of Technology.

David Suryan is a software engineer at Tektronix, Inc., in Beaverton, Oregon. David has 20 years of experience in software development, and testing. He has a Bachelors of Science in Electrical Engineering Technology from Oregon Institute of Technology.

This paper was developed with a team of dedicated people interested in improving requirements management practices at Tektronix: Kyle Bernard, Jeremiah Burley, Kathy Engholm, Lynne Fitzsimmons, Steve Follet, Perry Hunter, Badri Malynur, Ken Marti, Libby Mitchell,Vince Matarrese, Doug Muller, Mike Stevens, Eric Thums, and Ben Ward.

Introduction

Following an internal software process assessment, managing requirements was determined to be an area for

improvement at Tektronix. A working group that varied between 10 to 15 people was assembled, consisting of lead engineers and program managers from around the company. The working group met regularly for over 9 months to identify problems, uncover root causes, and develop solutions. The group also maintained an internal website using an open source model to distribute materials, post meeting minutes, track changes, and maintain a mail list.

The problems of managing requirements were many: traceability rarely performed, inconsistencies only sometimes corrected, specifications were not being kept up-to-date, little or no metrics were gathered, etc. Through investigation and a series of discussions, the group identified the following root causes:

- Many authors of requirements specifications did not know how to write and organize requirements.
- Traceability was considered an administrative task and there was no accepted method of doing it.
- The process to change requirements was incomplete and often informal.

By assembling people from various product groups and disciplines various current practices were discussed along with investigation of outside solutions. Best practices emerged and were documented into a guideline, which we are sharing in this paper. These techniques are focused on document-driven specification. The group is currently investigating automating the requirements processes through a commercial tool.

Putting together a consistent set of guidelines for a 4000+-employee company is not easy. It has to work across multiple disciplines (software, hardware, program management, etc.) and for various project sizes. Sometimes, multiple best practices have to be combined because of unwillingness to change for conformity reasons – leaving a guideline built by committee.

Overall, the key to the success of improving the requirements practices at Tektronix has been:

1) Management buy-in which included providing:
    a. Key engineering support and resources to work on improving the requirements engineering process.
    b. Resource and time to project schedules so engineers can generate good requirements specifications.
    c. A commitment to new tools to improve the requirements process.
2) A group of people who deeply felt that requirements practices needed to be improved and wanted to participate in that effort.

1. Purpose and Scope of this Guideline

A requirements specification needs to be the definitive source for how a product will behave. This paper provides guidelines for writing and managing requirements specifications. Requirements elicitation, requirement analysis, and requirements validation are outside the current scope of this paper.

Requirements Definitions

Requirements are a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property or attribute. They may be a constraint on the development process of the system. [17]

Capabilities needed by a user to solve a problem or achieve an objective. [10]

Stakeholders

The customers who buy and use the products are the most important stakeholders, also included are the people who depend on the requirements to do their job. These stakeholders need to have well-written requirements and need to be informed when requirements change. A partial list includes:

- Core/Product Definition Team
- Customer Support
- Customers
- End users
- Engineering (HW, SW, Mechanical)
- Manufacturing
- Marketing
- Program Management
- Quality Assurance
- Technical Communications
- Test Engineering
- Usability Engineering

Reasons for Writing Requirements

Collecting requirements enables a development team to better understand its market and the product/project constraints. By understanding user tasks and stakeholder values, rather than superficially attractive features, time is saved by not implementing features that aren't used.

By understanding the requirements up front, a development team can engineer an architecture to accommodate the requirements (rather than adding things to the product piecemeal). The team can develop a contextual model that supports user goals in context rather than providing a collection of features organized by the design or implementation structure.

Good requirements significantly reduce rework, especially during the later stages of development when the time to correct requirements-related defects could cause a slip in the schedule and increase costs.

System testing is facilitated by documented, unambiguous, verifiable, concise, traceable, consistent, etc. requirements, thus increasing chances of delivering a high quality product.

Requirements should:

- Specify the behavior and characteristics of the system.
- Support clear communication to all stakeholders.
- Provide a clear, referenceable basis for agreement.
- Provide a clear, referenceable basis for change.
- Provide a reference for product testing.
- Specify design constraints that will limit the developers' options.
- Relate to stakeholder (customer) needs, problems, or context.
- Establish a common understanding among the team members and stakeholders.
- Establish a basis for future development.

Risks from Inadequate Requirements

Often, time to market pressure causes a development team to abbreviate the requirements phase and start design and implementation early. This attitude arises from a presumption that the time spent writing requirements delays the product release by the same amount of time. It's been found that product teams improve their software development speed by spending more time and more effort in the requirements stage of the project and spend significantly less time in testing and integration[1]. Further, product quality is better when good requirements have been established first. Boehm[2] found that correcting a requirement error after the product was in operation cost 68 times as much as correcting an erroneous requirement during the requirements phase. When requirements are incomplete and poorly documented there are several associated risks:

- Minimal (incomplete) specifications lead to missing key requirements.
- Ambiguous requirements lead to product defects and rework.
- Creeping requirements contribute to schedule and cost overruns and degrade product quality.
- Product testing cannot be performed adequately, leading to defects in the released product.
- The requirements become a poor reference for product evolution. As a result the reference becomes the old product and it's difficult to discover all product features and behaviors by inspection.
- Incomplete requirements make accurate planning and tracking impossible.
- Requirements that lose their connection to customer needs run the risk of being dropped if the development time becomes short, development teams change, or other factors affect the project.

2. Requirements and the Development Life Cycle Process

Requirements are based on stakeholder needs, expectations, constraints, and interfaces. These should be clearly identified and understood through a collection/elicitation process such as observations, interviews, prototypes, market research, etc. These inputs are consolidated, documented, and analyzed. Product-level requirements come from this user input, along with other market information (available technology, competitive environment, standards, etc.). Functional requirements (software, hardware, etc.) are allocated from the product requirements.

Best practice life cycle processes are requirements-driven and support iteration and feedback, i.e., they are both top-down and bottom-up. Figure 1 shows a product life cycle flow. Other processes are possible (e.g., evolutionary, incremental, and follow-on projects) and would need to be defined as part of a program or project plan.

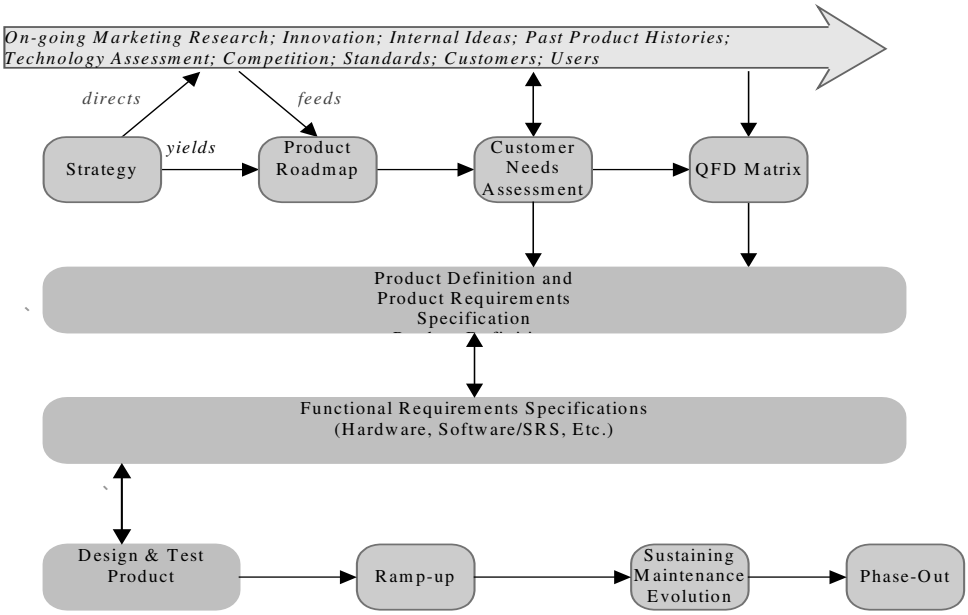


Figure 1 Requirements in the Life Cycle

2.1. Product Requirement Specification

The Product Requirement Specification (PRS) states the features, functions and characteristics of the product to be developed. The goal of the PRS is to identify the features and functionality of the product for program-level planning. There must be sufficient requirements detail so the product will satisfy customer and/or market needs. There should be a high correlation between the PRS and the product data sheet or brochure. Highly detailed requirements specifications that specify the behavior of all boundary conditions should be left to specific functional requirements specifications. Such a high level of detail could obscure the important characteristics of a feature and delay important project planning activities.

Product requirements need to state "what the product must do" and provide information to help with architecture, partitioning, design, and implementation. The product requirements should avoid stating "how it will do it".

2.2. Functional Requirements Specification

Functional requirements specifications (e.g. software, hardware, mechanical) define the parts of the PRS that have been allocated to each function. These specifications provide detail for all of the product's external behavior and characteristics. The product can be something sold to customers as well as something used for internal use such as a software module. Functional requirements serve as a basis for architecture, design, change, evolution and product testing. It can be argued that complete behavioral descriptions aren't necessary to design and implement a feature as many of the boundary conditions can be left to the implementer's judgment.

All behavioral requirements should be identified up front (as much as possible). The process of identifying and reviewing behavioral requirements helps identify missing requirements, issues, interactions, error, and boundary conditions. Thorough identification of behavior reduces the need for downstream re-work. This information provides the basis for product testing and a reference for future products. Using an existing product instead of the product's requirements as the reference for a new product is inadequate as it's difficult to discover all the behavior for boundary conditions.

3. Writing Good Requirements

There are several characteristics of good requirements and good requirement specifications. By keeping these characteristics in mind when writing and reviewing specifications better products will be developed.

3.1. Requirements Quality Criteria

Quality criteria are the properties that requirements need to poses; these are grouped into two sets. First, there are the characteristics that individual requirement statements should exhibit. Second, there are packaging (structure and style) characteristics that the requirements specification should contain. Each subsection below describes a quality criterion (with examples).

Characteristics of Individual Requirements Statements

Complete

Each requirement fully describes the functionality to be delivered and contains all the information necessary for the developer to design and implement that functionality. It is understood that not all requirements are fully known or understood early in the lifecycle, so it is best to flag missing information with "TBD" (and why it is such) so that incomplete requirements can be located and eventually completed.

A requirement for a Car Radio may state: "There will be a way to adjust the VOLUME." This requirement is incomplete – how is the volume to be adjusted, does it have limits, etc. A more complete requirement would be: " The system shall permit manual volume adjustment between 0 and 95 dBA (inclusive) in 1 dBA increments."

Correct

Each requirement does not conflict with any reference or source (customer or surrogate, project documentation, technical standard, etc.). Traceability makes this easier and less error prone.

A requirement for a Car Radio may state: "The On/Off button shall not function when the car is turned off". However, the QFD stated that users required the capability of using the radio with the car off. The requirement must be corrected.

Necessary

Each requirement documents something that a user needs, finds useful, or something that is required for conformance to an external system requirement or standard. Features that delight customers can provide differentiation from the competition. Do not add features that might be "cool" and have no basis in documented customer needs. Doing so complicates the user interface as well as increases development time and system complexity.

A Car Radio can have a feature that saves a set number of pre-selected channels. Since setting the channels can be performed easily and quickly is this something that users value? Is this feature just gold-plating (unnecessary embellishment)? Since there is no rationale explaining why the requirement is needed its hard to know for sure. Alternatively, there should also be a link that identifies the source of the requirement – where did it come from.

Prioritized

Each requirement statement needs to indicate how essential it is. This priority is usually a code that can be based on its relationship with program success, user importance, QFD score, and/or customer acceptance. This provides guidance to remove features to fit within the constraints of time, budget and resources.

See 3.2 Prioritizing Requirements

Identified

Each requirement should be marked with a unique, persistent identifier. This will assist in referring to individual requirements: "Requirement 'N' seems un-testable to me because…" An identifier should always be associated with its requirement and never be changed or reused if the requirement is removed at some point. For this reason, automated hierarchical headings in a word processor are dangerous to use as identifiers.

Unambiguous

All readers of a requirement should arrive at a single, consistent interpretation. This is often a significant problem but sometimes difficult to detect because the author might be unaware of the ambiguity, yet multiple readers will interpret the requirement differently. Have different stakeholders read and review the requirements to help remove ambiguity. See Appendix B for lists of words that make requirements ambiguous.

Consider this requirement: "The car radio shall provide up to six pre-selected channel settings". The ambiguity lies in the phrase "for up to six". Does it mean, "For up to and including six" or "for up to and excluding six"?

Verifiable

A finite cost-effective test can be devised to demonstrate that a requirement has been met. For example: "Easy to use," "usually," and "often" are not verifiable. Beware of the use of "Always" and "Never," for example: "The system will *never* crash" is not verifiable. Each requirement should be written so that you can verify all external behaviors and characteristics through demonstration, analysis, testing or inspection. See Appendix B for lists of words that make requirements unverifiable.

Consider this requirement from an application: "Make the Print function similar to typical Windows environment". This is not verifiable because "Similar" is subjective. Also, what is a "typical" Windows environment? This requirement could be re-worded as follows: "Make the dialog for Printing similar to the Windows NT environment so that 80% of users with Windows NT experience can find, identify and execute the function within 2 minutes".

"The (whatever) operation shall execute as fast as possible". This is not verifiable because we don't know when the point has been reached that the (whatever) task is as fast as possible.

Consistent

Each requirement should have no conflicts with other requirements with regard to behavior, terminology, characteristics, or temporal (timing) issues. Inconsistency can also be created when a requirement changes and other dependent requirements are not changed.

Behavior: A requirement may specify that two values are to be added and another may specify that they be multiplied.

Terminology: Is "volume" the same as "tone" or "amplitude"? Define terms in the specification.

Characteristics: A requirement may state that an indicator is green when another may state that all indicators are blue.

Temporal: A requirement may state that "A" must always follow "B", while another requirement may require that "A" and "B" occur within 10uS of each other.

Concise

Keep sentences and paragraphs short. A requirement statement should be succinct and deal with the issue(s) at hand and avoid verbiage that does not directly covey the requirement. Keep in mind that this can be taken to extremes, increasing ambiguity.

Try to understand this requirement after reading it once: "A graphical indication is available to the user which shows approximate input signal power level relative to the upper and lower bounds for proper operation of the radio."

Now try this: "Input signal power level in relation to the radio's upper and lower limits is graphically indicated with an accuracy of at least 10%."

Characteristics of Requirements Specifications

Complete

The specification should be complete enough to drive current product development activities at all times. The specification should not only specify functional requirements, but quality requirements, design constraints, documentation requirements, legal requirements, marketing requirements, and any other type important to the product. The specification should also capture dependencies, interfaces, constraints, and functional interactions between requirements, startup, shutdown, exception handling, and test cases when there are tricky non-obvious cases.

A defect may be reported against our radio because it did not save the selected channel settings across power cycles. This would clearly annoy users since the radio would have to be setup each time the car was started. The specification was not complete because it did not consider retaining the radio settings between power cycles.

Modifiable

Requirements specifications need to be easily revised and a change history needs to be maintained. Each requirement is uniquely labeled and expressed separately from other requirements so that it can be referred to unambiguously. Each requirement appears only once to avoid changing one instance without changing another. Labels on requirements must never be changed when other requirements are added, moved, or removed.

Traceable

It should be possible to link all requirements back to their stated customer needs or problems. It should also be possible to link all requirements to the elements that implement and test them (functional requirements, design, components, test cases).

See Section 4.1 on Traceability for more information.

Design Independent

Requirements clarify the problem to be solved and describe the behavior that the product should display. Engineering design solutions are not a part of a requirements specification. A possible design could be mentioned as a note but a design should never be a requirement unless the customer demands it. Exclude user interface design (menus, layout, etc.), as this is best documented in a User Interface Specification. However, it is appropriate and desirable to include UI requirements (which can include language conventions, naming conventions, look and feel, usability goals and metrics) to specify the user task(s) and the performance levels that must be achieved.

Consider the following requirement: "Provide a mathematical function (i.e. a decimation algorithm) to avoid handling high resolution graphics". So, why would a user want a decimation algorithm? How does this help to avoid handling high-resolution graphics? It turns out this is actually a design to help satisfy the following requirement: "The update rate of displaying high-resolution graphical pictures >= 1Meg shall be at least 10 pictures per second". Now does it make sense? Now that we know the real requirement we will find out if a decimation algorithm can achieve this or whether more effort is needed to meet the user's need.

3.2. Prioritizing Requirements

Each requirement should be marked with a priority code to indicate how essential it is to the product release. Typically, all requirements that relate to a product are not equally important; otherwise, there is a loss of freedom in responding to new requirements, change requests, resource changes, and schedule pull-ins. A prioritization scheme helps managers make important project decisions and developers make correct design choices and devote appropriate effort to different parts of the product. Whatever scheme developed needs to be documented and communicated so that all stakeholders of the specification understand it. Any prioritization scheme needs to provide guidance for the following:

- How important each requirement is to the success of a program/project?
- Which requirements do/don't get implemented in the next release (i.e., which affect the schedule)?
- Which requirements are only there for architectural/design consideration for future releases?
- Who approves changes/removals for each priority level?

Here are some example priorities with corresponding codes.

> ***C*** *- Critical. The product is not viable unless this requirement is met or this requirement is imposed by the Product-Line (i.e. all products within the product-line will have this feature). Changing such a requirement requires review by a change control board (CCB) and approval by a Product Line Manager.*
>
> ***E*** *- Essential. This requirement is important to the product's success and will be implemented. Changing such a requirement requires review by a change control board (CCB) and approval by a Program Manager.*
>
> ***D*** *- Desired. This is something that is thought to provide customer benefit or even delight, and will be implemented. Changing such a requirement requires review by a change control board (CCB) and approval by a Program Manager. This requirement should be implemented last and would be removed if time runs out.*
>
> ***O*** *- Optional. There is a good chance that this requirement will not be implemented and effort to implement this requirement is not included in any plan or schedule. If there is time to implement this requirement, a request needs to be made to the CCB so that stakeholders (testing, documentation, etc.) will be notified of the change.*
>
> ***X*** *- Specifically Excluded. This requirement was considered but will not be included in the product.*
>
> ***G*** *- Guideline. This is not a requirement, but something that needs to be remembered about the product or system to help provide guidance about the design choices that will have to be made. This can include customer preferences, design constraints, previous products, etc.*

Note: Be careful when sorting requirements based on priority codes because they often are not in alphabetical order.

3.3. Providing Rationale

Rationale provides the reason for and purpose of the requirement. There are two basic ways to include rationale with requirements. First, having traceability back to a stakeholder can provide the information necessary to understand the requirement. Second, including extra text around requirements can explain the rationale for anyone reading it. Providing rationale helps to remember the intent behind the requirements, clarify the requirements, resolve ambiguity, explain tradeoffs, and provides useful information when requirements changes are considered. Chances of incorrectly changing requirements are reduced. This is extremely important information but is often not easily available. Having rationale documented eases and improves decision-making and prioritization during the development process.

Consider this requirement:

"The clock settings (hours and minutes) on the car radio shall be adjustable whether the car radio is on or off."

Without any rationale one could assume this requirement is arbitrary or perhaps undesirable. It may become attractive in the future to change the requirements to allow the control to be inactive when the radio is turned off. Now, consider the following rationale for this requirement:

"If the clock settings are not always adjustable then the car radio would have to be turned on to set the time."

If these reasons provide a stronger case than the reasons for making the control inactive, then the work to change it and then change it back is avoided. It would be even worse if it were changed back after customers complained.

Please note that rationale is different from design. Design is concerned with satisfying requirements. Rationale describes *why* the requirement is present.

3.4. Style Guidelines

Though there is no one method to write requirements, keep the following recommendations in mind:

- Write with proper grammar, spelling and punctuation.
- Use the active voice (where the subject performs the action: "Force Trigger shall cause a trigger event") rather than the passive voice (where subject receives the action: "A trigger event is caused by Force Trigger"). A requirements specification needs a Table of Contents, a Glossary for consistent use of terminology, and a Revision History.
- Write at a level of detail that if the requirement is satisfied, the customer's need is met, but not so much detail as to unnecessarily constrain the design.
- Separate each requirement clearly from other supportive information such as rationale.
- Avoid long narrative paragraphs that contain multiple requirements. Conjunctions such as "and" or "or" suggest this. Avoid bulletized lists to specify separate requirements (among other things they provide implicit prioritization which may not be desirable).
- List "normal" cases of requirements first – then list exception cases.

3.5. Considering Reuse

Some of the areas where reuse can come into play for requirements specifications are:

1. Elements that will be reused from other products and projects need to be specified in the requirements specification: components, architectures, designs, tests, etc.
2. Requirements that come from other requirements specifications need to include the references.
3. Reusing all or part of the requirements specification on follow-on products/projects needs to be considered and specified.

4. Managing Requirements

The goal of managing requirements is to:

- Provide the program/project with a current, approved set of requirements during the life cycle
- Ensure the relationships between the requirements and other entities are captured bi-directionally
- Identify inconsistencies between the requirements and work products, making corrections when inconsistencies are discovered
- Manage all changes to the requirements

4.1. Traceability

Tracing requirements deals with documenting the links between individual requirements and other elements. Backwards traceability identifies the source of each requirement (customer need). Forwards traceability identifies the work products and deliverables that satisfy each requirement; this provides assurance that each requirement is fulfilled. When traceability is performed, the following are achieved:

- Inconsistencies between requirements and work products can be identified and corrections can be made.
- Impacts of requirements changes become are better realized.
- Originating customer needs can be referenced prior to changing/removing requirements.
- Proof exists that the delivered product actually met all requirements.
- Status of a project can be better determined as missing links indicate work products that have not been delivered.
- Reuse is better facilitated. When requirements are reused in future products, the associated work products are identified.

Traceability Matrix

A Traceability Matrix provides a cross-reference between requirements and other related artifacts. Creating a traceability matrix can be a tedious job, but the benefits outweigh the risk of finding a missed requirement late in a project. Building and maintaining a traceability matrix should be part of regular, productive work rather than and imposed near the end of a project or program in order to satisfy a quality requirement.

First, decide which work products to include in the matrix. At a minimum, requirements and test cases need to be included. The example in Figure 2 shows forward and backward traceability, as well as product and engineering requirements. Other work products can be added as appropriate: use cases, design elements, components, test plans, documentation.

	(1)	(2)	(3)	(4)
	QFD / Customer Need	**Product Requirements Spec**	**Functional Requirements Spec**	**Test Case**
(5)	<qfd a>	<prs 1>	<req i>	<case A>
(6)	<qfd b>	<prs 2>	<req ii>	<case B>
	<qfd c>			
	<qfd d>			
(7)	<qfd e>	<prs 3>	<req iii>	<case C>
	<qfd f>			<case D>
			<req iv>	<case E>
			<req v>	<case F>

Figure 2 Traceability Matrix (example)

(1) This column refers to customer needs as defined in the QFD or other source.

(2) This column refers to the product requirements specification that details the product's functions, features, specifications, quality requirements, cost requirements, and projections.

(3) This column refers to a functional requirements specification (software, hardware, mechanical). Each functional specification would have its own column.

(4) This column refers to the test cases that demonstrate that requirements are met.

(5) This row shows a straightforward example with single references of a customer need, product requirement, functional requirement, and a test case.

(6) This row shows that multiple customer needs can map into a single requirement.

(7) This row shows that multiple functional requirements can map into a single product requirement. In this case, an additional level of traceability is needed between each functional requirement and their associated test case(s).

Using Traceability to Discover Missing Items

The power of a traceability matrix comes in the ability to find missing items. When a column does not contain a reference item, this is an indication of a disconnect that needs to be investigated and resolved. Figure 3 shows some examples of disconnects that can be found in a matrix.

	QFD / Customer Need	Product Requirements Spec	Functional Requirements Spec	Test Case
(A)	<qfd g>	<prs 4>	<req vi>	?
(B)	<qfd h>	<prs 5>	?	?
(C)	?	?	<req vii>	<case G>
(D)	?	?	?	<case H>

Figure 3 Traceability Matrix w/Missing Items

(A) No test case has been identified for this requirement. This signals that one needs to be created.

(B) A product requirement has not been allocated to the functional specification. The functional spec needs to be updated.

(C) A functional requirement does not appear to be necessary as it is not traceable to a product requirement or customer need.

(D) This test case appears to be unnecessary.

Responsibility

Decide who will be responsible to do the tracing. Depending on the extent of the traceability, more than one person may be responsible (program manager, engineer, quality). However, it should be given to a single person who has visibility into the requirements and the other artifacts that will be included in the matrix. This person needs to identify the people that will supply the linkage information.

Management

The traceability matrix needs to be managed as the other documents and work products of the project. It should be possible to identify the versions of the items that are referenced in the matrix – either by calling out the versions of each element in the matrix or identifying all references and the matrix with the same version label in the configuration management system.

It is important that the traceability matrix is updated as items are implemented or requirements change. Periodically review the matrix to ensure that it is being kept up-to-date.

4.2. Change Management

Requirements change management deals with the process used to change baselined requirements specifications and the associated work products. The process should consider the sources that request changes (people, suppliers), how the requests come into the evaluation process (email, defect management system), how the requests will be evaluated (Change Control Board), and how the results of the evaluation will be communicated to the affected stakeholders.

Figure 4 Generic Requirements Specification Change Process

Sources

The sources of change requests include any project stakeholder as well as users, customers, and changes of technical standards.

Change Requests

There may be many channels by which a stakeholder can request a change: email, verbally, etc. Regardless of how the request is made, all changes need to be funneled through a single, documented change-reporting channel. At a minimum, the following should be documented or referenced in the change request:

- Purpose: Whether the change is a new requirement, change to an existing requirement, or the removal of a requirement.
- Source: Who/what is the origin of the change.
- Change: The detailed description of the change.
- Rationale: The reason this change has been requested.

Process

In order to evaluate all change requests a Change Control Board (CCB) is formed with the responsibility to evaluate every change request and make approve/disapprove decisions. The (CCB) represents the interests of customers (Marketing), the program (Program Manager), development (software, hardware, and mechanical engineering), and other key stakeholders (manufacturing, testing, etc.) who look at each change from their respective points of view. For smaller programs/projects, the CCB could be just a few people (or even one individual) who would have to be trusted to consult with other stakeholders when appropriate.

Addition information that should be included with the change request includes:

- Impact: The effect the change would have if implemented: structural, major, minor, etc.
- Location: The function(s) and/or component(s) that would be affected by the change: user interface, hardware, etc.

Here is an example evaluation process for a Change Request (CR):

1. The CR is checked to see that it is valid, well-written, not a duplicate, and received through the appropriate change-reporting system. If not, the CR is *rejected* and the submitter could modify the CR and resubmit. If it's not practical to reject (e.g., comes from a customer), the CR may be corrected by the CCB.
2. Requirements affected by the change are identified. Traceability is used to find related requirements.

3. Specific changes to requirements are proposed.
4. The CCB determines whether the authority to consider the change is beyond the scope of the CCB (e.g., a change to a product-line requirement). If so, the CR is *elevated* and evaluated at the appropriate organizational level.
5. The CCB determines whether the change will not be done (rejected). The change would need to be *archived* by marking it as such in the change-request system, adding it to a roadmap document, or adding it into current requirements specifications as "Specifically Excluded" or "Future Consideration".
6. The CCB determines which release to implement the CR.
7. Before actually approving the change, the costs of making the changes are estimated. In considering a change the CCB must consider the impact on cost and functionality, impact on customers and other stakeholders not on the CCB (component suppliers, customer service, etc.), and the potential of the change to destabilize the system. After the costs are calculated, the change is evaluated as in steps 5 and 6 and is *rejected*, *archived*, or *approved*.

Communications

It is extremely important that once a requirement changes (or a change request is denied) all stakeholders, including the originator of the change, are notified.

Stakeholders

Once a requirement change is approved and communicated, the following actions need to occur:

- The requirements specification(s) is/are updated, reviewed, and approved. It is possible that after the updated requirement(s) are reviewed that the change cannot be implemented as stated – in which case the CCB would be notified and the review process restarts.
- The traceability matrix is updated.
- Work to implement the change is assigned.
- The change is implemented and verified.

5. Measures and Metrics

"If you don't measure, then you're left with only one reason to believe that you are still in control: hysterical optimism." [5]

Developing and managing requirements is knowledge-based work that relies on the judgments of people. Without objective information, judgments rely on "gut feel" instincts – but the only subjective information by which to make these judgments is a specification, along with some conversations contained in emails and peoples' memories. Gut feel doesn't provide high-confidence answers to some very important questions such as: 'Is the implementation of requirements behind schedule?'; 'How many change requests have impacted the software?'; 'What components have changed the most?"; or "Is the rate of new change requests increasing or decreasing?". These can only be answered with objective, qualitative information using measures and metrics. Not only can qualitative information help with judgments for the current project or program, it can help plan future projects and programs by providing historical data.

This section provides examples of measurements that can be made during the requirements process and some guidance on how to use them.

Specification Measurements

The following measures can be generated from the requirements specification itself:

- Total number of requirements.
- Number of "TBD's" remaining.

- Following an inspection/review of a specification: number of defects and issues (from quality criteria and inspection checklist).
- Amount of time spent reviewing specifications.

Traceability Measurements

The following measurements can be generated from tracing requirements:

- Number of requirements traced to tests.
- Number of requirements tested successfully.

Change Management Measures

The following measurements can be generated from managing requirements changes:

- Number of change requests received, open, and closed (cumulative and per period).
- Number of changes made (cumulative and per period).
- Number of added, deleted, and modified requirements (cumulative and per period).
- Number of change requests per source.
- Number of changes proposed and made for each requirement since baselined.
- Total effort devoted to managing and making changes

Metrics

A metric is two or more measurements combined. If measurements described above are taken, then metrics can be derived to help make judgments about the requirements and the requirements process. Before using any metric effectively, a goal (or set of goals) must be established, followed by generating questions that need to be asked to determine whether each goal is met. Finally, analyze each question in terms of metrics you need to answer each question. This is known as the Goal-Question-Metric (GQM) approach.

Example:

Goal: Determine whether the requirements specification is ready to release.

Question: What is the quality of the requirements specification?

Metric: Defect Density. This would require two measurements: the number of requirements with defects by reviewing specification and the total number of requirements.

$$Defect\_Density = \frac{\#\text{ of requirements with defects}}{\text{Total }\#\text{ of requirements}}$$

Note: There are other questions that could be asked for this goal or other metrics that could be used to answer the question.

6. Glossary

Term	Definition
Change Management	The process of making or proposing changes on work products using judgments.
Customer	A person or organization that will receive (and usually pay for) the product when it is complete. Requirements are based on customer needs.
GQM	Goal-Question-Metric. A system to tie measurement to the overall goals of a project or process.
Life cycle process	A model of development that depicts the relationships among phases, major milestones, and deliverables that span the life of the product.
Measurement	A qualitative assessment of the degree to which a product or process possesses a given attribute.
Metric	A system of measurements.
QFD	Quality Function Deployment. A tool provides a visual link between the customer requirements and product solutions.
Quality Criteria	Properties that requirements and requirements specifications need to poses.
Requirement	A specification of what should be implemented; a description of how a system should behave, or of a system property or attribute; capability needed by a user to solve a problem or achieve an objective.
Requirements Traceability	The ability to describe and follow the life of a requirement in both a forward and backward direction.
Specification	A document that prescribes, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a product, system, or component.
Stakeholder	A person or organization that depends on the requirements. Stakeholders can be customers, but are usually internal to the company.
TBD	To Be Determined. An indicator that additional information is needed.

7. References

[1] Blackburn, Scudder, et al, *Improving Speed and Productivity of Software Development: A Global Survey of Software Developers* IEEE Transactions on Software Engineering, December 1996 (Vol. 22, No. 12).

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

[3] Conner, C. and Callejo, L., *Requirements Management Practices for Developers*, Rational Software White Paper, 2002.

[4] Davis, A., *Software Requirements: Objects, Functions and States.* Englewood Cliffs, NJ: PTR Prentice Hall, 1993.

[5] DeMaro, T. *Controlling Software Projects*. Yourdon Press/Prentice-Hall, 1982.

[6] Fenton, N., Pfleeger, S., *Software Metrics: A Rigorous & Practical Approach*, PWS Publishing Co., 1997.

[7] Freedman, F. and Weinberg, G., *Handbook of Walkthroughs, Inspections, and Technical Reviews*, Dorsett House Publishing, 1990.

[8] Hamlet, D., Maybee, J., *The Engineering of Software*, Addison-Wesley, 2001.

[9] IEEE Standard 830-1998, *Recommended Practice for Software Requirements Specifications*.

[10] IEEE Standard 1233-1998, *IEEE Guide for Developing System Requirements Specifications*.

[11] Kotonya, G. and Sommerville, I. *Requirements Engineering*, John Wiley & Sons, 1998.

[12] Leffingwell, D., Widrig, D., *Managing Software Requirements, A Unified Approach*, Addison Wesley, 2000.

[13] McConnel, S., *Code Complete*, Microsoft Press, 1993.

[14] Rosenberg, L, et al., *Testing Metrics for Requirement Quality*, 11th International Software Quality Week, 1998.

[15] Sarma, M. S., *A Framework for Managing Requirements Specification Process*, India Software Engineering Process Group Conference, 1999.

[16] SEI, *Capability Maturity Model Integration (CMMI) for Systems Engineering and Software Engineering, Version 1.1*, 2001

[17] Sommerville, I. and Sawyer, P., *Requirements Engineering: A Good Practice Guide.* John Wiley & Sons, 1997.

[18] Thayer, R., *Software Requirements Engineering, Second Edition*, IEEE Computer Society Press, 1997.

[19] Weigers, K., *Software Requirements*, Redmond, WA: Microsoft Press, 1999

Appendix AREQUIREMENTS INSPECTION CHECKLIST

Requirements Statements

Is each requirement:

- ❑ Pertinent to the problem and its solution?
- ❑ Providing an adequate basis for design?
- ❑ Truly necessary?
- ❑ Consistent with a customer need?
- ❑ Prioritized?
- ❑ Uniquely identifiable/labeled?
- ❑ Clear, concise and unambiguous?
- ❑ Verifiable by testing, or demonstration? Many words suggest a requirement is not verifiable (see Appendix B).
- ❑ Actually a requirement, not design or implementation solution?
- ❑ Written in the "active" voice?
- ❑ Not in conflict with or a duplicate of any other requirements?
- ❑ Singular. There are no multiple requirements contained within one requirement statement.
- ❑ Written at a consistent and appropriate level of detail? Should any requirement be specified in more (or less) detail?

Specification

- ❑ Do you understand the reason or purpose of each requirement?
- ❑ Are all the requirements needed to develop a solution included? Where information isn't available, are the areas of incompleteness specified?
- ❑ If the product satisfies every essential requirement (e.g., Priority = "C" or "E") will it be acceptable?
- ❑ Are you uneasy about any part of the specification? Are some parts impossible to implement within the known capabilities and limitations of the techniques, tools, and resources?
- ❑ Is the document organized well enough to easily locate requirements?
- ❑ Are there any missing or unspecified requirements? Is all expected behavior documented?
- ❑ Is the specification organized so that each item can be changed without excessive impact to other items?

Appendix B TERMS TO AVOID

Requirements that use ambiguous or subjective terms and phrases cannot be verified. These terms can mean something different to everyone. Avoid using of the following terms and phrases in any requirement (although these could be included in Guideline statements or Rationale):

Implied certainty:

All, *Always*, *As required*, *Every*, *None*, *Never*

Implied uncertainty:

Can, *May*, *Might*, *Should*

Persuasive:

Certainly, *Clearly*, *Obviously*, *Therefore*

Incomplete lists:

And so forth, *And so on*, *Etc.*, *Such as*

Vague:

Customarily, *Most*, *Mostly*, *Often*, *Optionally*, *Ordinarily*, *Some*, *Sometimes*, *Usually*

Vague verbs:

Handle, *Improve*, *Maximize*, *Minimize*, *Optimize*, *Process*, *Provide for*, *Reject*, *Skip*, *Eliminate*

Vague adjectives:

Able to, *Adequate*, *Appropriate*, *Capable*, *Easy*, *Effective*, *Efficient*, *Fast*, *Normal*, *Quick*, *Rapid*, *Simple*, *Sufficient*, *Superior*, *Timely*, *User-friendly*

Vague pronouns:

It, *Our*, *They*

Requirements Are Forever

Product Requirements Methods at Cadence

Mark Noneman
VP, Enterprise Quality
Cadence Design Systems, Inc.
555 River Oaks Parkway
San Jose, California 95134

Abstract

Does your software seem to live forever? Are you frustrated with project-centered methods that ignore the problems of modifying existing software for yet another release? This session describes a requirements management approach geared specifically to a multiple-release, hierarchical, software product environment.

The old Cadence Design Systems approach of single-release "Market Requirements Documents" and "Functional Specifications" were not allowing us to commit to long-term product roadmaps, improve product interoperability, or release products more frequently. Worse, our new "platform" strategy made it necessary to create and manage requirements across multiple products that are in different phases of maturity and market adoption. In addition, the document-centric approach led to cumbersome, outdated documents with poor visibility or usage. By shifting our focus from documents to data we made traceability a reality. Marketing and engineering have better insight into the impact changes in requirements will have on release scope, resources, and schedule. Furthermore, defining requirements over the life of products and platforms gave us better insight to release content and product roadmap planning. This paper presents the requirements management method and the way it was developed and deployed at Cadence. It also discusses lessons learned in its execution and plans for improvement.

Biography

Mark Noneman is Vice President of Enterprise Quality, Cadence Design Systems. His responsibilities include improving whole-product quality focused on software best practices, especially requirements methods. He has a total of 20 years of experience as an electronics engineer, system architect, project manager, and quality expert at TRW, Nokia Mobile Phones, Mentor Graphics, and Cadence Design Systems.

Introduction

In the past few years, Cadence Design Systems has undertaken a focused activity to improve the quality of its products and services across the company. The company had re-organized from a model based on centralized functions to a collection of business units responsible for product revenue and investment expenses. At the same time, a series of questionable business decisions resulted in several new product development projects being canceled after many millions of dollars of investment and many staff-years of effort. After a critical review by upper management, Cadence started to adopt the well-known Stage-Gate business decision model as a means of getting its important decisions under control as recommended through a consulting engagement. After some false starts that caused the company to bring the development of the Stage-Gate process wholly in-house, a set of gates, criteria, processes, and documentation templates were created and rolled out across all software product development activities.

After several years of successful use, we identified weaknesses that make the process less useful for immature products and products that serve early-adopter markets. Currently, Cadence actively supports more than 50 different products with many more that have reached obsolescence. Perhaps the most pervasive weakness we identified in the last year was in the development and management of requirements. This paper describes that situation, what Cadence is doing to improve requirements processes, and what we have learned to date.

Requirements Hierarchy in the Stage-Gate Model

The hierarchy of requirements defined by Stage-Gate started with the Marketing Requirements Document (MRD). From this high-level description of customer and business needs written by the marketing group, the Product Requirements Specification (PRS) was derived by product architects and subject matter experts. The PRS described the features to be implemented in enough detail for software developers to then create Functional Specifications (FSpecs) for each piece of functionality planned in the product. From the FSpec, test plans and technical documentation were created in addition to the detail design and implementation of the software code.

Products or Projects?

One issue we were struggling with was a mismatch between the details of the Stage-Gate process and our business. Cadence develops and sells application software for engineering automation tasks, particularly for integrated circuit and printed circuit designs. The lifespan of our products is measured in years, often decades. Our customers view our products as a sequence of update releases—sometimes those releases contain new functionality, sometimes bug fixes—for tools that help them automate complicated engineering design tasks.

On the other hand, our Stage-Gate process viewed the world from a project-centric perspective. The “project” was the primary focus of all the planning and document templates that were developed for the Stage-Gate process. In particular, our requirements documents were designed to live for the life of the project, but not the product. Each new release project started over again

with new requirements documents. (Granted, many overview paragraphs were copied from one generation of document to another.)

Figure 1 shows a high-level view of the Stage-Gate process as initially implemented at Cadence. The top-most row of "gates" is known as Investment gates—they are for business-level decisions throughout the product lifecycle. The Release gates, shown at the bottom of the figure, are not as rigid as the Investment gates and each sequence of Release gates can occur many times throughout the product's lifecycle. In later versions of the Stage-Gate process depiction, we attempted to further "soften" the Release gates by showing overlap in the earlier development tasks, somewhat akin to a modified waterfall model.

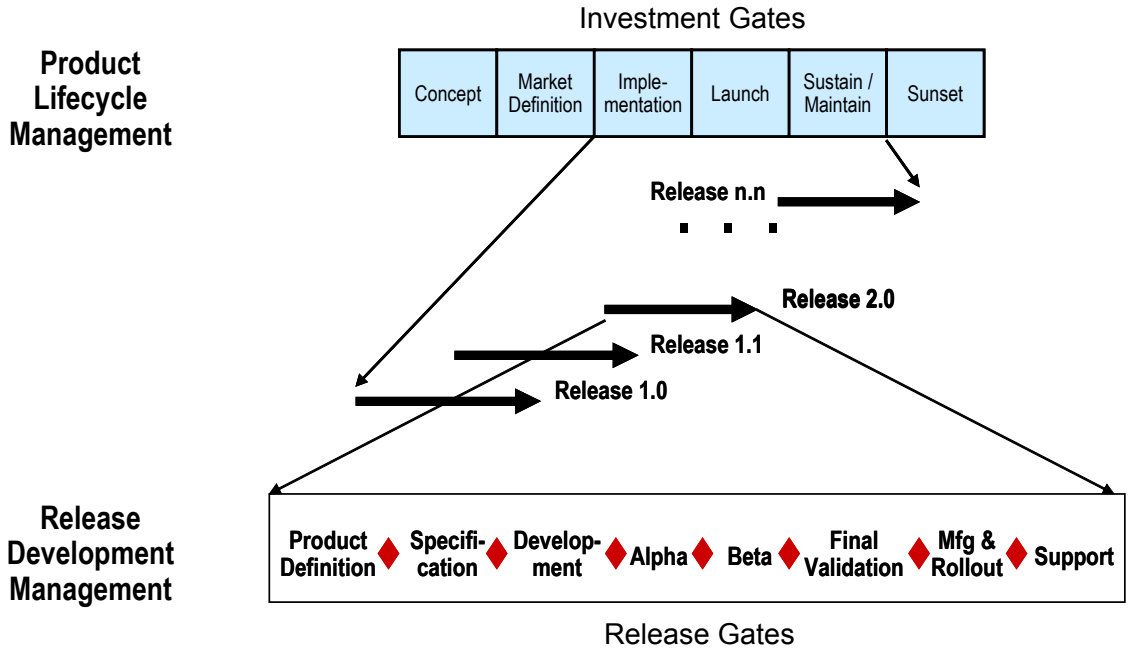


Figure 1. The Cadence Stage-Gate lifecycle uses key "gates" to review and approve important business decisions.

Of course, the waterfall nature of the Stage-Gate definition itself caused many problems. Gates with names such as "Requirements Complete" and "Planning Complete" did not accurately reflect the iterative nature of software development.

Since the requirements documents defined in Stage-Gate only lived for the duration of a project, it became increasingly difficult for marketing and engineering to accurately communicate the needs of customers on the one hand and the time it would take to implement on the other. Eventually, MRDs were written that did not adequately describe the interaction of new capabilities with already existing capabilities. As a result, the long development times that engineering estimated seemed out of line with the simplistic view of a single-release MRD. It got to the point where some engineering teams essentially ignored the MRD (especially from a prioritization perspective) and only implemented whatever they considered were the most important features to move the product in the direction they thought it was going. Worse yet, the

MRD was sometimes written in retrospect, after the requirements for the next release were already being implemented.

The single-release MRD and PRS also meant that we didn't have a repository for what these long-lived products were supposed to do and how we had planned to implement them. Without a record of the product's whole requirement set, it became impossible to know what we were testing. Test sets became collections of bug-fix test suites as opposed to verification that the product conformed to requirements.

Of course, these factors were a recipe for disaster. We released products, but product quality and our credibility with customers suffered along the way.

The Problem with Documents

The Cadence Stage-Gate process made heavy use of documents throughout the project lifecycle, as is traditional for waterfall-like development methods. Of course, there is nothing wrong with documents, per se. They are convenient repositories for very important information in defining the product. In particular, documents are a good way to record business planning information such as market segmentation, product vision, and other strategic plans.

As expected, even the labels we used to refer to requirements were document-centric; for example, the Marketing Requirements Document. Unfortunately, documents are not convenient for highly dynamic information. In a waterfall development model, all requirements are defined up front and are viewed as static in nature. Under that model, documents are a fine repository for requirements. But in the real world, especially commercial software development projects, requirements are highly dynamic by nature.

Communicating changes in documents, although difficult, can be done. It is a well-known and frequently practiced discipline in many organizations. The problem with requirements is that nearly all of them exist only in the context of other requirements. In particular, being able to trace a functional specification to its market needs is imperative to making good decision making and prioritization in the course of product development. To accommodate this need, we used traceability matrixes in our documents to indicate from where a requirement was derived.

Unfortunately, even organizations diligent enough to actually fill out the traceability matrixes early in the project typically could not keep up with the number of changes that occurred as the project unfolded. In short order, the traceability matrixes became worse than useless—you couldn't tell if they were up-to-date or not and untold hours were wasted tracing requirements to obsolete sources (which also change frequently in our environment).

To Make Matters Worse

Our problems with requirements came to a head at the end of 2002 as Cadence embarked on a product strategy to create what we call "platforms"—suites of products designed to work together to solve a customer whole-design problem. The marketing analysis of these platforms revealed new requirements for existing products (and, in some cases, the need for some new

products). In particular, we had a need to describe how our products would work together to solve the targeted customer problems, including platform features, interfaces, and behaviors.

There was now an additional level of requirements hierarchy to describe the platform definition. Our existing attempt to use requirements documents that defined a single project for a single product was completely unworkable. To successfully execute our platform strategy, a new requirements methodology for definition and management was desperately needed.

Working the Issues

To tackle this problem, we created two cross-functional groups: one to define how we would describe a platform and one to revamp our concept of requirements. These groups worked in parallel to speed up the process. Frequent collaboration between these teams ensured that their results would be compatible.

Each of these groups leveraged experts from across the company. To expedite our work and to avoid the well-known "design-by-committee" pitfalls, we relied on a small core team that evaluated existing approaches and interviewed a broad selection of subject matter experts and other thought leaders (whose buy-in we knew would be essential for our new requirements methods to be accepted). We collected hundreds of statements from dozens of people from which we extracted several key findings for what was right and what was wrong with our requirements methods.

The most important findings are summarized here:

- Customer needs were ineffectively described. Marketing was not writing customer needs in a way that engineers could figure out how to implement them correctly.

- The MRD was ineffective. Most groups went through the motions of creating a MRD only to have it sit on the shelf and collect dust. The PRS was the document on which product development teams focused their efforts

- The impact of changes to requirements was difficult to determine prior to making the change. A lack of up-to-date traceability meant that analyzing the effect of proposed changes was tedious and error-prone

- Testing did not verify that the product conformed to *customer* requirements. Test plans written in response to functional specifications were several levels removed from the customer need and could not specifically verify customer requirements

- Prioritization of requirements was either overlooked or done poorly. A typical PRS listed scores of requirements and all resulting functional specifications contained hundreds of requirements. It was nearly impossible to agree on which of those were most important in solving the customer's problem when we could not tell which collection actually solved the problem

To address these issues, we pulled key concepts from practices in the external software development community and adapted them to the Cadence environment. The rest of this paper describes those solutions.

Requirements Are Data Objects!

The most fundamental change we made was a shift in perspective: Requirements are really data objects. For those that have used requirements management tools, this statement may seem obvious. But for an organization steeped in the tradition of documents, requirements are thought of as sentences or paragraphs, not data elements. Once we could get people to grasp the essentials of the shift in perspective, the advantages of requirements as data objects became clear.

First, creating relationships between requirements becomes straightforward. The traceability problem is easier to manage, and that makes impact analysis of changes in higher-level requirements or lower-level implementation decisions simpler. Using relational database concepts, parent-child links, queries, filters, and reports allow us to talk about requirements in much more constructive and flexible ways.

Second, requirements stored as data objects allow us to address the issues caused by the project-centric nature of our previous document paradigm. That is, requirements can now live forever! We don't have to start over again at each release project. That means we could gradually build a complete requirements repository for a product as it matures. When requirements change across releases, we still have a record of the hierarchical relationship and, therefore, the impact of the change. In reality, most of our products have been developed for years using a document-centric approach. New requirements are addressed as "deltas" even in our new data-centric approach. Nevertheless, over time we are building a good basis for change management over the life of the product, not just within a project.

Third, a data object view of requirements allows us to talk about structured (that is, repeatable and predictable) attributes for requirements and their relationships. For example, top-level platform requirements have an attribute that records the source of the requirement: customer enhancement request, customer needs assessment, competition, etc. Similarly, low-level functional requirements have an attribute indicating by what method the requirements will be verified: inspection, analysis, demonstration, or test.

Finally, a database repository for requirements allows us to eliminate "documents" in the traditional sense altogether. Although this caused great consternation for some people, especially project managers, we could point to other examples of database repositories that do not have document paradigms. The most obvious example is our defect and enhancement request system. No one would even think to ask for a document-based approach to defect reports. Luckily, all modern database systems have filtering and reporting mechanisms that can be used to present requirements in a format that is easy to review. The systems were accepted due to the thorough and comprehensive reporting capabilities they have, providing the engineers and product managers the right view of the selected data they need in the right time. "Just in time" customized reporting added direct value to the product teams.

Requirements Flow

Once we began thinking about requirements as data objects, we could start working on their relationships given our new platform strategy.

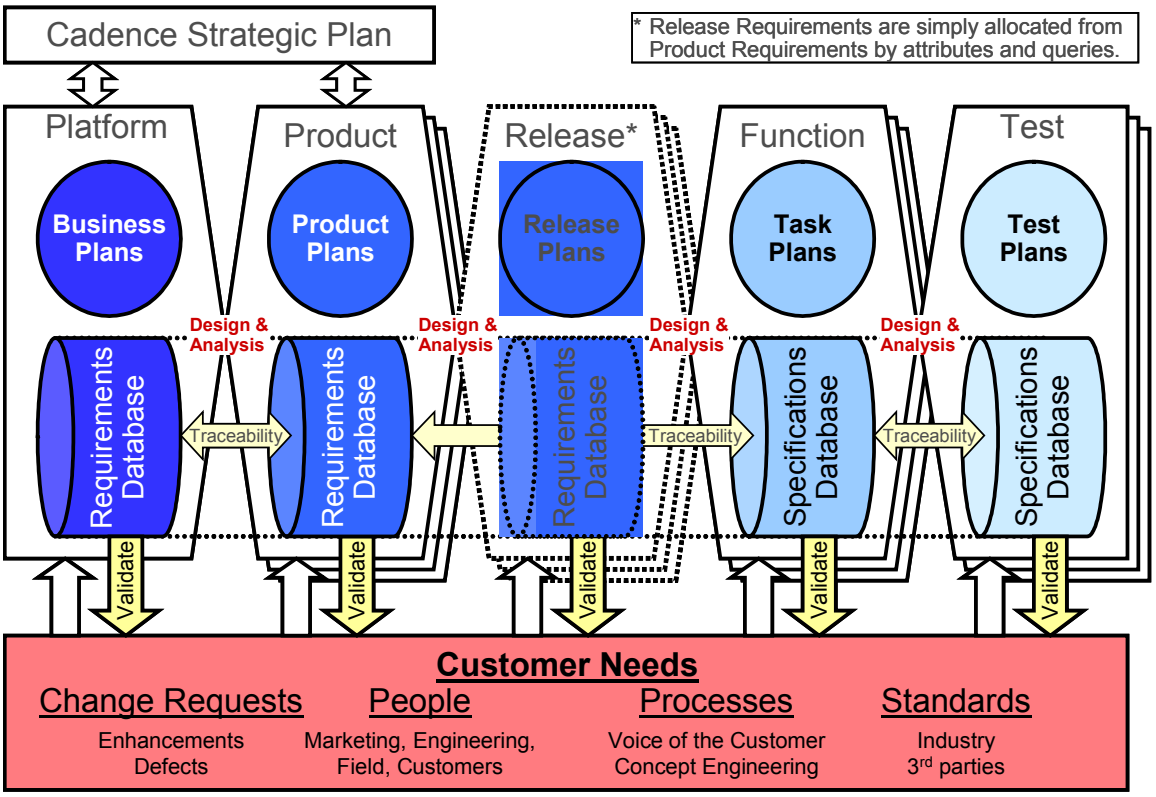


Figure 2. Conceptual requirements flow-down with traceability for Cadence "platforms" and products.

Figure 2 depicts our high-level requirements flow. Each level of hierarchy (shown horizontally, left to right, highest level to lowest level) consists of a set of plans and a set of associated requirements. The set is referred to as a *definition*. For platforms and products, the plans are highly business oriented and define the current vision, direction, and success criteria for that level. Product plans must incorporate both the platform plans that affect it as well as any other plans that may be unique to the stand-alone product. For release, function, and test levels, the plans are task oriented and relate to the more immediate needs of development and deployment.

The primary input to all definitions is represented at the bottom of the Figure 2. A variety of techniques are used to cull data sources for customer and market needs and priorities. We have found it very useful to maintain traceability at any level back to the original source. Debates continuously arise as to the intended meaning of a requirement or “why do they want this?” issues. When we can validate a requirement with a real customer, competitive, or business need, many issues are immediately resolved. When we can’t trace back to a customer need, it drives the research necessary to either properly validate the requirement (which gives us traceability for future issues) or to eliminate it. Furthermore, as lower levels of requirements are developed from

higher ones, traceability allows us to validate with customers (or customer proxies) that we're solving the real problem.

At the platform and product definition levels, important input also comes from the company's strategic plan—we also need capabilities because they will help customers in the future, as opposed to solving a current customer problem.

Figure 2 shows platform definitions relating to one or more product definitions, each of which relates to one or more release definitions. The act of relating one level to another is where the engineering effort takes place. Design of architectural elements, user interfaces, control and data interfaces, algorithms, code, and tests must ensure that all planned higher-level needs are implemented in the lower-level definitions. Conversely, we use the design and analysis activities to ensure that all lower-level definitions can trace to higher-level ones. That avoids "gold-plating" the product—including features that do not address a customer or strategic need.

Given a database view of requirements, we do not need to artificially differentiate between "documents." The dotted lines connecting the requirements database symbols in the figure imply that all requirements for the product are in a single physical database even though, through filtering and querying, we can view the requirements as separate logical (hierarchical) levels. This helps prevent the version control issues that often arose whenever we would "copy-and-paste" requirements between levels—changes at one level invariably did not get changed at other levels. A single database approach allows us to simply reference other requirements directly from one level to another.

Of final note in Figure 2 is the "release" definition (shown with a dotted outline). The product definition level lives for the life of the product (see Requirements Lifecycle)—so how do we know what features are planned for each release? A roadmap planning activity (which takes place at the start of each new release cycle and is updated as needed) defines which features we need in a release as well as two releases after that. The results of that planning activity are captured in the product requirements database. The release requirements are really only selected product requirements for that release. Changes in our roadmap planning automatically get reflected in the release definition once the attributes are updated. These, of course, are changes that can be tracked and analyzed for impact on schedule and resources.

Requirements Lifecycle

As discussed previously, some of our issues with requirements stemmed from the fact that our Stage-Gate definition was project-centric instead of product-centric. Our new requirements definition changes that. The platform and product levels are specifically defined to last the life of the platform and product, respectively. Through additions and changes, we are now able to build up a complete requirements repository for platforms and products over time. We record not only the requirement but the rationale (as an attribute) and its source (via traceability). This gives us the ability to better understand the scope of planned changes to requirements.

In fact, the use of an actual database makes all levels of requirements live as long as the product does. By keeping track of changes to data objects, we can even trace the various requirements

by release, function, and test. In this way, when a platform or product-level requirement changes, we can better estimate the impact to the actual source code, user manuals, and tests. Those changes that will "only take a few days" can now be evaluated more accurately and reliably.

Telling the Story

Even though our new perspective on requirements addresses many of the problems we were having, one difficult problem remained: How can we more effectively communicate customer and strategic requirements at an appropriate level of detail to engineering?

We turned to a form of requirement definition that is becoming quite popular: use cases. At the platform and product level, we use a simplified version of use cases somewhat akin to the user stories used in agile development methods. We have learned to carefully identify the interfaces involved (actors) because leaving one out accidentally can be disastrous. Starting with a list of actors and reviewing them with all stakeholders can avoid many fundamental problems that are very difficult to fix later.

At the platform and product level, we use only use cases to describe the customer problem we want to solve (whether identified by a customer or through strategic planning). By keeping requirements described only in a use case form, we keep the effort focused on solving a customer problem. Use cases are also more easily verified by key stakeholders. Otherwise, it is easy to implement only some of the features necessary for a complete solution in a release. Staying focused on the solution to a customer problem as the deliverable makes prioritizing features and functionality much easier. Why work so hard to deliver only part of a solution?

More elaborate use cases are often used for features and functionality when either user interaction or an external interface to another system is involved. Use cases are not useful for describing the requirements for algorithms or data structures not involved in interfaces. We have found that traditional requirement definitions work well in those cases.

Rolling It Out

Although we are still rolling out our new requirements methods, there seems to be two kinds of reactions: embrace it or fear it. Those that embrace it often want to rush to an automated requirements management system even before they have fully absorbed the meaning of a data-oriented requirements paradigm. They often want to have it both ways: a set of documents that is managed by a database system. Although most requirements management tools today have some kind of document-like interface (some even use Microsoft Word as the document user interface!), this often causes more problems than it solves. To get the full value of the data-centric requirements method, it appears best to go "whole hog" and entirely eliminate documents from the requirements part of the definition. Document interfaces can be useful in managing the links between the plans and the requirements of a given definition level but not for the requirements themselves.

Those that fear the new methods are reacting just as many people do when faced with any process change initiative. In this case, people usually will not say they do not agree with the

requirements approach we have developed. Instead, they talk about “pilot projects,” “risk reduction,” and “resource constraints”—potential issues to be sure but delay tactics nonetheless. The most important step here is to transition people into the data-oriented view of requirements. Instead of jumping to an automated requirements management database system, starting with index cards may be just the trick. Yes, index cards. On the front of the card, you can briefly write a simple use case or traditional requirement—the small size forces people to be concise. The back of the card can be used for attributes about that requirement. Traceability becomes an attribute much like the traditional traceability matrixes found in documents. Managing traceability with cards is not better than with documents, but some people need to crawl before they can walk.

Another approach that can appear cheaper than adopting a commercial requirements management solution is to use a spreadsheet. The tabular nature of a spreadsheet makes it easy to be concise, add attributes, and begin thinking about requirements as data objects. Traceability is semi-automated since links from one spreadsheet to another are possible with modern spreadsheet applications. Moving requirements around at one level will maintain the link without resorting to reference numbers. However, changes to requirements can be difficult to trace for impact analysis unless you start with the changed requirement. Also, other productivity features of a commercial requirements management system are not available. Nevertheless, a spreadsheet can be a tremendous help in managing complex requirements and their relationships. We do not recommend managing more than a hundred requirements or so with a spreadsheet since lifecycle costs can be more than adopting a commercial requirements management system.

Keep on Rolling!

Although our new requirements methodology is promising to improve effective communication and management of long-lived product requirements, there are still several areas that need to be improved.

For example, because a “capability” (something that solves a customer problem) can be described at both a platform and a product level, it can be confusing to determine where the derived “features” that implement that capability should go. Do they go where the capability is defined or do they all go at the product level, regardless of where the capability is defined? The question is not academic as it affects the level of detail at each level of specification. If there is too much detail at too high a level, then marketing ends up telling engineering how to implement the product.

Another area for improvement is the definition of the test level. Anyone that has been involved in testing will recognize that there needs to be some level of test for each level of definition of the platform. Keeping all test descriptions at one level is confusing. Should we have one test level for each level of definition? It appears so.

Conclusions

Adopting modern software requirements methods is both exhausting and rewarding. For companies faced with long-lived products, using a data-centric solution can help solve many

problems that project-centric approaches create. By involving key stakeholders in identifying the most important problems and focusing efforts on improving those areas, progress can be shown quickly for those that can embrace the new approach. But for those that may be reluctant, don't give up; there is always a way to help people crawl before they walk.

Bibliography

Alistair Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2000.

Dean Leffingwell, *Managing Software Requirements: A Unified Approach*, Addison-Wesley, 1999.

John Mansour, *Product Management University*, Course Manual, Zigzag Marketing, 2003.

Extreme Programming website, user stories,
http://www.extremeprogramming.org/rules/userstories.html

SOFTWARE STAKEHOLDER MANAGEMENT-

It's not all it's coded up to be

BY ROBIN DUDASH, CQManager, CQA, CQE, CRE, CSQE, QS-LA

Innovative Quality Products & Systems, Inc.
934 Whitestown Road
Butler, PA 16001

Abstract

Quality software is typically defined as 'defect-free' code. However, a more inclusive definition is meeting customer expectations, which is accomplished through applying a project stakeholder management model.

Robin Dudash has 18 years experience in computing from business mainframes, client server, to real-time process control platforms. Through her successful project management she has led major capital expenditures budgeting in excess of $2 million. Some of these projects were fully automated control systems and contracted to attain 99.9% system reliability.

Ms. Dudash has been a Senior ASQ member since 1994 and the ASQ Pittsburgh Education Chair for the last seven years. She has also taught the CSQE Refresher course based on the ASQ CSQE BOK (http://www.asqorg/cert/types/csqe/bok.html). This course has realized a pass rate of 95% for the last 5 years, and is now available on-line.

Ms. Dudash currently owns her a company - Innovative Quality Products and Systems - that conducts consulting for ISO/QS/TE-9000 quality system development, training services and internal quality auditing. She is also a subcontracted Lead Assessor.

Robin has degrees in Computer Science and Applied Mathematics, and an MBA, concentrating in finance, from the University of Pittsburgh. Ms. Dudash holds CQManager, CQA, CQE, CRE, CSQE, QS-LA, and QS-9000 certifications. She may be contacted at www.iqps.net.

The Project

It was time for the Project Team's unveiling of the new software system. A year had passed since they were first commissioned to develop a new order entry system. The Project Team was especially proud of the work they had done. They worked hard to make sure that the project was done on-time, on-budget, and performed exactly as they thought everyone said that it needed to function. "We finally did things right this time," said the Project Manager. Everyone was sent to the application's training class to learn how to use the new system. Today was the day for the new order entry system to be used by those who had been trained.

On the other hand, the Sales Representatives did not understand why the order entry system had to change; the old one worked just fine. The first anxious Sales Representative sat down at the sophisticated new computer station to enter the first order, and said, "This is too complicated. I can't use this!" Another Sales Representative picks up the rhetoric and says, "Yes, this will never do!" The Sales Department swells with groans of frustration. A concerned Sales Manager hears the complaints and thinks about all the money spent. The Sales Manager glares at the Project Manager and says, "This isn't what I wanted! This will never do!" These are the very same people trained a month and a half ago. The Project Team feels betrayed. How could they do this to people who have worked so hard to get them what they wanted?

The story may be exaggerated (just a little), but haven't we all heard of or been a part of a project like this in our lifetime? Unfortunately, some of us have been the user, the proud team member, the trainer, and yes, even the one that will take the fall – the Project Manager.

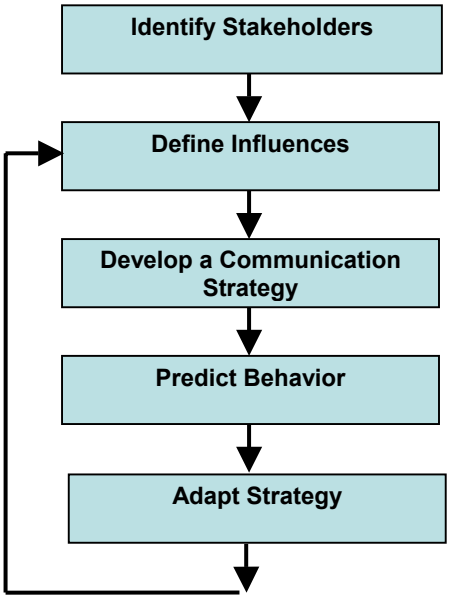
Project Stakeholder Process

This Project Manager failed to realize that not only is s/he responsible for delivering the software system, but is also responsible for managing stakeholder expectations. Perhaps not even realizing who are software system's stakeholders is the first mistake. We must recognize that "software stakeholders" extend beyond just the user.

A formal project stakeholder process assists in the:

- identification of these stakeholders;
- definition of their specific stake in the project;
- development of a communication strategy to specifically address stakeholder interests;
- prediction of stakeholder behavior to analyze project impact;
- and, adaptation of this strategy in the project implementation.

Project Stakeholder Process



Identifying Stakeholders

There are many 'software stakeholders' to be identified for a project. To identify all of the current and potential stakeholders, let us examine how software quality might be characterized with respect to stakeholder expectations. A typical software quality attribute and expectation is the lack of defects. However, this is a very narrow view of software quality. A more comprehensive view is defined by IBM, which measures the ability of its software products to satisfy CUPRIMDSO quality attributes -- capability, usability, performance, reliability, installability, maintainability, documentation/information, service and overall. Quality attributes such as these are what Juran called quality parameters for 'fitness for use'.[1] Subsequently, a software product that was thoroughly tested and "bug free" may not meet current (or even future) stakeholder (customer) expectations, such as easy to use, short response time, and easy to change, resulting in dissatisfaction.

Looking at this broader definition of software quality, the Project Manager in the project described can identify all of the "stakeholders" -- the Project Team, the Software Supplier, the User Department, and the IS Department Manager. Each of these stakeholders has a different reason for having an interest in the software system, which influences their behavior. Management of these "stakeholder interests" is referred to as Stakeholder Management. While this is important in every project, it is especially important for software development where the deliverables are not as tangible as constructing a building.

Identify Behavioral Influences

Projects are developed in an organizational environment within a company, consisting of functional departments with organization goals and objectives. These goals and objectives

[1] Stephen H. Kan, Metrics and Models in Software Quality Engineering, (Addison-Wesley, 1995), pg 5.

evolve as the organization reacts to market and other environmental impacts. Project managers need to identify and interact with key organizations and individuals within the project systems environment. This management process is necessary to determine how the stakeholders are likely to react to project decisions, what influence their reaction will carry, and how they might interact with each other and the project manager to affect the chances for project success. The impact of project's strategy and decisions on all the stakeholders must be considered in any rational approach to the management of a project.[2] The following Stakeholder Management Model is depicted for the described Order Entry project:

Stakeholder Management Model

Stakeholder	Behavior Influence/ Interest	Reason
Project Manager	Ultimate responsibility of the project execution and deliverables	Personal credibility
Project Team	Responsibility for completing assignments	Job performance Personal credibility
IS Manager	Ultimate responsibility for fulfilling the business need Responsible for maintaining the software in the future	Departmental worth
Sales Representatives	Responsibility for using the software to service customers	Job performance
Sales Manager	Ultimate responsibility for fulfilling business goals	Affects ability to service customers
Software Supplier	Responsibility for fulfilling requirements	Payment (commission) Future sales performance

[2] D. I. Cleland, Project Management-Strategic Design and Implementation, 1st ed., (TAB Books, 1990), pgs 94-95.

Develop a Communication Strategy

Once the stakeholders are identified and their interests understood, the most important activity as a project manager is to define the project goals, scope and end results. While organizational goals may have initiated the project, these goals may not have considered all of the stakeholders. The project manager must revisit the project deliverables with all of the stakeholders, and process the information received from the stakeholders. This understanding must then be articulated back to the stakeholders to obtain definition and agreement. In all cases, this definition and agreement must be documented.

Not only are these goals, scope and end results established at the start of a project, but they must also be communicated throughout the project lifecycle. Again, a project is developed in a dynamic organizational environment. The project manager must manage stakeholder expectations by listening to current business needs, addressing any yet unstated stakeholder requirements, and adjusting project deliverables to address those needs. What was perceived as a need a year ago when the original goals and scope were defined may not be what is needed now.

The project manager must also be sure that the project owner, the IS Manager in this case, is clear on project goals and objectives. The IS Manager can assist in managing business requirements, acquiring additional resources to meet changing needs, and breaking down organizational barriers to success.

With clear goals and objectives, the project manager also can direct the project team towards the agreed upon requirements. Given the most talented people in the company on the project, the Project Team still cannot reach the goal without a clear target.[3]

The goals and objectives must also be communicated to the end users of the software. In return, the project manager must listen to the needs and concerns of the users, and assure them that their concerns are understood. User 'buy-in' is key to managing their expectations. Methods to obtaining user 'buy-in' can include prototyping interfaces and conducting training early in the design phase to solicit user feedback. Obviously the more the user interface differs from the current system, the more resistance that will be expressed by the user. It is important to keep this aspect in mind when developing the end user communication strategy as it will mostly definitely impact their behavior.

If a software supplier is used, the project manager must also communicate goals and objectives to the supplier and make these part of the contract. In need of special attention, but often overlooked is the contractual training to be provided. Often, training is conducted once after the design is mostly complete, which may be too late to accommodate specific needs. Typical training provided by the software supplier is a 'show-and-tell' class, and may not address the users' deepest concerns, for instance 'How will I use it on my job?' While a good introductory class, this type of 'show-and-tell' class may present more questions for the users than answers, creating a stage for animosity. The project manager should use this opportunity to address

[3] Paraphrased from W. A. Randolph and B.Z. Posner, Effective Project Planning & Management, (Prentice-Hall, 1988), pg. 16.

concerns and specific ‘likes’ and ‘dislikes’ to build a communication channel, and/or present actual user prototype screens as more focused alternative.

A second training class scheduled close to system startup should train users using actual examples. It has been my experience that the closer training is conducted to actual system startup, the more successful the startup. A few individuals may even need one-on-one training. Of course, user manuals as much as we like to keep them in the drawer should be part of the project as they also are part of the end user communication strategy, perhaps in the form of a quick reference written from the users’ perspective. Of course, an ideal method of delivering user assistance would be an integrated on-line help, and throw away the manuals, but then this may not be cost-effective for smaller implementations. A widely distributed software application such as Microsoft Excel or Word would significantly benefit maintenance costs.

Thinking that all this training adds costs to the project? Yes, it does, however I have observed project after project which shortcuts training by not properly planning, and/or not soliciting user feedback costs 2-3 times their original estimate in the end (if there is an end). Without user “buy-in” throughout the project lifecycle, users will find every reason not to use the system, and the project will incur un-needed costs just to ease their complaints.

The following depicts a communication strategy based on quality parameters for the project manager to use with respect to each stakeholder:

Stakeholder	COMMUNICATIONS STRATEGY								
	Capability	**Usability**	**Performance**	**Reliability**	**Installability**	**Maintainability**	**Documentation/ Information**	**Service**	**Overall**
Project Manager	Define and document agreed-upon goals Document agreed-upon changes Assess training needs for project team	Assess training needs for users Readability of user docs	Define and document agreed-upon goals Address 'likes'and 'dislikes' about current system Document system test plan	Agree upon software development methodology	Define and document installation plan	Define and document resource requirements for future growth	Re-affirm goals throughout project lifecycle Project status Obtain approval of milestones Assign work breakdown packages	Design for system expansion Assess training needs for system maintenance	Re-affirm goals throughout project lifecycle Keep a global awareness to changing needs Communicate project status Obtain approval of milestones Assume empowerment of project management Keep positive attitude
Project Team	Understand goals	Comply with development procedures	Comply with development procedures	Comply with development procedures	Comply with development procedures	Comply with development procedures	Comply with development procedures Project status	Comply with development procedures	Project status Keep positive attitude

Stakeholder	COMMUNICATIONS STRATEGY								
	Capability	**Usability**	**Performance**	**Reliability**	**Installability**	**Maintainability**	**Documentation/ Information**	**Service**	**Overall**
IS Manager	Agree upon goals	Define and document standard user-interface	Agree upon acceptance criteria for testing	Define and document software development methodology	Agree upon installation plan	Define and document software development methodology Define requirements for future growth Provide software maintenance tools and training	Agree upon acceptance criteria Approve milestones	Conduct project compliance audit Prepare for ownership and maintenance of the system Assist in developing relationship with Software Supplier for future maintenance	Project status
Sales Reps	Understand goals	Provide 'show-and-tell' training class early Provide user prototype entry screens for review Provide training using actual test cases	Understand expectations and use	Communicate project expectations	Understand installation plan	Anticipate expectations and future use	Project status	Prepare for ownership	Project status
Sales Manager	Agree upon goals	Invite a Sales Rep to represent the users on the Project Team	Understand expectations and use	Communicate project expectations	Agree upon installation plan	Assess 5 year business goals	Project status	Prepare for ownership	Project status

Stakeholder	COMMUNICATIONS STRATEGY								
	Capability	**Usability**	**Performance**	**Reliability**	**Installability**	**Maintainability**	**Documentation/ Information**	**Service**	**Overall**
Software Supplier	Agree upon goals Provide evidence of capability	Assign through contract review	Assign through contract review	Assign through contract review	Assign through contract review	Assign through contract review	Project status	Assist in developing relationship with IS Manager for future maintenance contract	Contract milestone progress payments Project status

Predict Behavior and Adapt

Based on an understanding of stakeholder behavioral influences and a communication strategy, the project manager can proceed to predict stakeholder behavior in executing a project. Stakeholders with a high-vested interest should be studied carefully by the project manager, and their strategies and actions noted to see what effect such actions might have on the project's outcome. Once the potential effect is determined, then the project execution should be modified through resource reallocation, replanning, or reprogramming to accommodate or counter the stakeholder's actions through the stakeholder management model.

For example, perhaps the Sales Manager takes the attitude that the department is too busy to provide input in accepting project deliverables. Offer to have the project status meeting in the Sales area, communicate the decisions that need only Sales participation, and provide a meeting agenda with a time-frame. If a Sales Representative on the Project Team is not possible, then negotiate that yourself and/or a project team member observe a Sales Representative's different situations. It is important to establish a communication channel with the end user department to establish their project ownership.

Effective Stakeholder Management ➔ Stakeholder Satisfaction

A formal Stakeholder Management process ensures that multi-year projects which are subject to so much change are adequately managed. The typical reliance on informal or hit-or-miss methods for obtaining stakeholder information is ineffective for managing the issues that can come out of any type of project. By developing a Strategic Management Model for each project, the project manager has assembled adequate intelligence for the selection of realistic options in the management of stakeholders.

The body of knowledge for the Certified Software Quality Engineer (CSQE) recognizes that delivering quality software that satisfies stakeholder expectations is contingent on successful project management. Not only is the project manager responsible for the project definition and execution, but the successful project manager also accepts the responsibility for stakeholder management. In order to satisfy stakeholder expectations, their needs (stated and unstated) must be managed, and the ultimate responsibility is the project manager.

Enhancing the Total Customer Experience through HP-UX Patch Quality

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

Abstract

At Hewlett-Packard, we define Total Customer Experience (TCE) to be our customers' overall impression of HP based on their experiences with our people, products, services and solutions. The quality of the patches we produce for the HP-UX operating system and applications has a significant impact on customers' satisfaction with their HP-UX software. This paper explores the many dimensions of quality, explains HP's best practices for building quality into each HP-UX patch and how we arrived at them, and discusses how application of these best practices has improved TCE.

About the Author

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

Enhancing the Total Customer Experience through HP-UX Patch Quality

Introduction

At Hewlett-Packard, we define Total Customer Experience (TCE) to be our customers' overall impression of HP based on their experiences with our people, products, services and solutions. Positive TCE not only results in customer loyalty and repeat sales, but also influences customers to recommend HP to others. Negative TCE can translate directly to lost sales.

A patch is an incremental change to released software. Customers install HP-UX patches for a variety of reasons:

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

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

In the late 1990's, HP had not yet begun to use the phrase "Total Customer Experience," but it was clear that HP-UX patches were causing problems for customers, and so HP undertook to remedy the situation. Once the TCE initiative was announced, we quickly saw that our goals and methods were consistent with the new corporate-wide program, so it is appropriate for us to discuss the patch improvement efforts in that context.

HP-UX patch quality problems were at the root of many customer complaints and sometimes even caused telephones to ring in executive offices. Patches were consistently near the top of our support organization's issues list. Most of the complaints had common themes:

- HP released too many patches,
- too many of HP's patches were defective,
- it was too difficult to determine which patches to adopt, and
- consequently, too much time was required for system maintenance.

In the intervening years, with the solid backing of upper management in both R&D and support organizations, there have been some dramatic changes in how HP-UX patches are created, tested, and delivered. Customers have definitely noticed the improvements and have encouraged us to continue our efforts, sometimes even sharing their best practices for quality with us. The remainder of this paper looks at some of the lessons we have learned, some of the most important

changes we have made to date in the HP-UX Patch Program, and some of the work that is yet to be done.

Understanding the Problem

At the outset of the HP-UX patch quality improvement project, it was important for HP to build an accurate picture of the patch quality problem, not simply to begin to make changes that we thought would be useful and pleasing. Finding out what customers really want can be tricky. Often, when they voice their dissatisfaction, they present us with a possible solution rather than an actual problem statement. To complement the feedback we had from meeting with customers at trade shows, conducting surveys, and the like, we conducted a contextual inquiry (Beyer[1]) regarding HP-UX patch administration. This investigative process begins by watching a variety customers work, asking questions about what they are doing and why, and recording their actions and comments. All of the recorded information is processed into a large affinity diagram that serves as a model of how customers do their work. Contextual inquiry has the advantage of capturing real work in progress, not a condensed or inaccurately remembered recounting of how the work is done. It is extremely valuable for identifying task roadblocks and usability problems.

It was also important for us to work within the framework of HP's longstanding definition of software quality. Decades ago, HP had developed a broad view of software quality, called FURPS+. FURPS+ was defined in terms of the attributes customers expected, and still expect, their software to have. It therefore has served the Patch Program well as we have worked to improve the customers' assessment of HP-UX patches. FURPS+ is an acronym for "Functionality, Usability, Reliability, Performance, Supportability, and whatever else may be important for the particular piece of software."

The direct feedback from customers and the results of the contextual inquiry helped us understand what each aspect of FURPS+ meant in terms of patch quality *over and above what it meant to product quality*. This allowed us to assess the strengths and weaknesses of HP-UX patches and patch delivery mechanisms and to undertake to address our weaknesses.

For **Functionality**, there are two important principles:
1. A patch must not change existing behavior except to fix a defect or to conform to a new industry standard.
2. Customers want to be able to select what, if any, new functionality to adopt.

The most important rules for **Usability** are:
1. Every patch must install and remove without logging errors or doing any damage to the system, such as disabling other functionality.
2. A patch must not make user-visible changes by default, except of course when the change is required to fix a defect. The customer must take some action to turn on new or changed features. This might involve setting an environment variable, or adding a customization to a configuration file, or creating a link.

The expectations for **Reliability** are:
1. A patch must not cause any regressions.
2. A patch must not expose latent defects in released software or in other patches.

Performance should be maintained or enhanced by installing a patch. Any performance degradation beyond roughly 5% is considered unacceptable.

Patch **Supportability** implies:
1. Patches can easily be identified on customers' systems.
2. There should be straightforward ways to determine what patches are needed and then to obtain and install them.
3. Patches must behave well. For example, all code changes must be cumulative, patches must not install unless prerequisites are met, and removing a patch must restore the system to its pre-patch state.
4. Patch documentation must explain the patch content clearly so that a customer can decide on the patch's relevance and must explain any manual intervention necessary to complete the installation.

For patches, the "+" in FURPS+ is **Availability**. When there is need for a new patch, customers should not need to wait unduly long for it to be developed and tested.

The Improvement Effort

In the context of our definition of patch quality, several principles have evolved for developing and delivering HP-UX patches in ways aimed at enhancing TCE. These principles and some of the changes they have inspired follow.

1. **Minimize the number of patches needed**.

   The best way to minimize the number of patches needed is to **release a defect-free product**. With this ideal in mind, the teams that created the HP-UX 11i operating system undertook an aggressive quality improvement program. Their efforts resulted in a six-fold reduction in customer-reported defects during the first year after release, as compared to the previous version of HP-UX, even though the rate at which customers upgraded to HP-UX 11i was faster than the rate at which they upgraded to its predecessor (Balza[2]). The quality of the HP-UX 11i release has dramatically reduced the number of patches required to fix defects and therefore the amount of time customers must spend on reactive system maintenance. Consequently, HP-UX 11i has been a big win for TCE. Version 2 of HP-UX 11i is currently under development, and the development teams are working hard to achieve yet another two-fold reduction in incoming defect reports. This should further reduce the need for patches that fix defects as well as the time customers spend in reactive patching.

   Another approach to minimizing patches is to **deliver multiple changes in a single patch**. This method has both advantages and disadvantages. On the plus side, the organization that produces the patches benefits from having fewer patches to package and document and to run through its test suites. The customer benefits from having fewer patches to install, which

may result in less down time. However, as the number of changes in a patch increases, the risk of delivering a defect also increases, so this approach may not be useful for code that has a poor track record for quality.

2. **Create the right patches**.

   HP has a set of guidelines that specify what should be patched on what operating system releases in order to **guarantee the availability of the patches customers will most need**. According to these guidelines, HP usually creates patches for all critical customer-reported defects and for most serious defects reported by customers while the affected release is shipping. When time allows, HP also develops patches that fix lower-weight defects that are encountered frequently.

   With the release of HP-UX 11i, we began to **proactively patch critical and serious defects found within HP**. Creating a fix as soon as the problem is discovered is beneficial both for customers and for HP. For customers, it means that a solution is often available by the time they discover the problem. For HP, it removes the tension between quality and schedule when producing a patch, which leads to fewer corners being cut in the name of schedule and therefore fewer problems being discovered in patches.

   Fixing defects may be the major reason, but certainly isn't the only reason, to create patches. It is also important to **anticipate the customer need for enhancements**. There is a steady stream of new hardware releases, both CPUs and peripherals. Patches are often necessary to allow the operating system to work with or drive the new hardware.

   Patches can also provide support for new industry standards and other enhancements to existing operating system releases. This allows customers to take advantage of new capabilities without waiting for them to be incorporated in a release of the operating system and carrying out an expensive migration. However, loading enhancement patches exposes customers to some additional risks, because our data show that patches containing enhancements are twice as likely to be defective as patches that deliver only defect fixes.

   To mitigate the risks associated with patches that contain enhancements, HP has adopted a strategy of **segregating defect fixes from enhancements** whenever the system architecture permits. When an enhancement is required, both a minimal enabling patch and a new product are created. The patch contains only the hooks necessary to allow the existing software to work with the new product, which delivers the actual enhancement code. The customer who wants the enhancement installs both the enabling patch and the new product. The customer who doesn't want the enhancement never installs the new product and may never install the enabling patch. However, even if the customer does get the enabling patch or one of its successors, there is little risk of finding a defect in the hooks.

   Knowing which patches not to create is also important. A little over a year ago, we examined the volume of patches we were releasing and determined that many unnecessary patches were being created. Often this was because the labs creating them did not change their habits to match changes in requirements due to updates in support status. For example,

once HP-UX 11.0 replaced HP-UX 10.20 as the volume shipping release, the rules for porting fixes of defects reported only on HP-UX 11.0 back to HP-UX 10.20 changed to require fewer ports. Labs were still required to patch a critical defect reported on the 10.20 version, but did not need to automatically port fixes of defects reported on later versions of the operating system back to HP-UX 10.20. Yet, labs continued to do these ports as a matter of course. We changed this behavior by publishing and publicizing color-coded tables representing the rules for defect fixing and fix porting.

Today we update our tables every time there is a change in support status of an operating system release. Labs consult these tables at the start of the investigation for each proposed patch and cancel unneeded patch projects before much effort is spent on them. Rarely does an escalation by a customer require us to reverse the decision not to create a patch. This simple change is one of the major reasons HP released 25% fewer patches in the first six months of fiscal year 2003 than in the same period a year earlier. Not only do customers have fewer patches to select from, but also HP has more time to devote to the patches that our customers really need. In addition, the patches we don't produce don't deliver new defects that require even more patches.

3. **Strive to make each patch flawless**.

Almost nothing makes customers more angry than a defective defect fix. At a minimum, defective patches cost unplanned downtime. They may also cause a variety of other problems including data loss or corruption.

One of the best ways we have found to avoid making mistakes in patches is to **understand the mistakes we have already made**. Whenever an HP-UX patch is found to contain or expose a defect, the team that created it performs a root cause analysis. The point is to determine how the defect escaped the release process undetected and then to recommend changes to the patch creation and test processes aimed at preventing future occurrences of the same kind of mistake. Not only is this common sense, we have also established a highly significant statistical correlation between following through on the recommendations and a reduced mistake rate.

Many of the changes stemming from root cause analyses apply only to particular situations within the teams that produced the defective patch. However, several recent changes are helping patch creators throughout the HP-UX labs reduce patch defects. Among these are: requiring test plan reviews in addition to design and code reviews, improving the code review process, documenting what was done during patch development, and increased management participation in patch initiation and release checkpoints.

We began to **require test plan reviews for each patch** a little over a year ago because root cause analyses revealed two common failure modes associated with testing. One was that the changed code was configuration-sensitive, but an appropriate variety of configurations had not been tested. Another was that a particular new test was needed to exercise the changed code, but had not been written. We still find situations where more or better testing would likely have found the problem in the patch, but the configurations needed are becoming more

subtle, meaning that fewer customers are likely to experience the problems. Also, the test plans themselves now need less revision after they have been reviewed. The authors are thinking about their testing challenges in advance.

Root cause analysis data shows that among the more common reasons that patches have failed is that the code reviews that were conducted did not uncover some fairly obvious defects. In response, we have made several **changes to the code review process**. First, we **perform a risk analysis and use the results to drive the formality of the code review**. Risk factors include the magnitude of the code change, the experience of the engineer(s) making the code changes, the availability of domain experts, and the complexity and quality history of the code being patched. Patches that have high risk are required to conduct formal inspections of all changed code. Patches with lower risk may choose to use either formal inspections or less rigorous (and less time-consuming) desk checks or code walkthroughs. Next, we **choose the review team carefully**. Review teams should include a senior engineer or technical consultant to provide a broad overview, a domain expert, and an internal customer of the code being changed. If the code author is inexperienced in the code or in the patch process, the author's mentor should also join the review team. We also encourage **re-review of any changes made as a result of the code review**. These simple changes have reduced the number of defects missed in code reviews by 80%.

About a year and a half ago, one of our labs was having difficulty maintaining some particularly complex code. On the hypothesis that documenting the actions taken and decisions made during each patch creation would help them to create better patches, they created a **patch quality checklist** (PQC). This document template served to guide the patch developer through the patch creation process and at the same time provided a place to record such information as whether a design review could be omitted and why, what style of code review should be conducted and why, what new tests should be created, and who approved the completion of each stage of development. Certain sections of the PQC were to be reviewed at an initiation checkpoint for the patch and the remaining sections were to be reviewed at a patch release checkpoint.

Although long and laborious to fill out, the PQC proved useful for multiple reasons:

a. It made the patch developers conscious of the decisions they were making and the tradeoffs, and made sure they had answers to important questions that might be asked at checkpoint meetings.
b. It provided better guidance for inexperienced patch developers than the existing process documents that were often out of sight and therefore out of mind.
c. It provided an easy way for the management team to review each patch and therefore got them more actively engaged in patch checkpoint meetings. Checkpoint meetings took on new meaning. In the early months, at least half of the patches were sent back with specific recommendations for more work.
d. It provided a written record that can be consulted months or even years later when the patch needs to be superseded.

The lab that created the PQC has experienced a dramatic decrease in the number of problem patches they release. Their mistake rate decreased almost instantly from greater than one per

month for patches created before the PQC to less than one per quarter for patches created using the PQC.

Since the PQC proved to be a best practice for the creating lab, several other labs have begun using it as well, and with good results. HP is currently working to get more labs to adopt the PQC and to produce a web-based version of the checklist that automates as much of the data collection as possible.

Also based on the results of root cause analyses and customer feedback, the HP-UX Patch Program has **increased the amount of centralized pre-release testing** that all patches must pass. In the past, there has been centralized testing to validate various aspects of the integrity of the patch: no checksum errors, all referenced patches actually exist, etc. The additional centralized testing guarantees that all patch documentation is evaluated according to new, higher standards. It also provides diverse configurations for installation testing and tests combinations of patches on the same hardware and software configurations used by some of our major accounts. If a patch does not pass any of these tests, it is sent back to the creating team to be reworked before being retested. Only when it finally passes the centralized testing is it actually released. Centralized testing also provides economy of scale. Individual labs would not be able to afford the staffing or the diversity of configurations provided by our centralized test facility, called the Patch Clearinghouse (PCH). There have been two notable results from the increased centralized testing. First, problems with patch installation and removal have virtually disappeared. Second, customers have reported greatly increased satisfaction with HP-UX patch documentation.

Until recently, it has been difficult to get customers to help with the HP-UX patch testing effort. This has been due partly to the awkward process involved and partly to customers' unwillingness to take a patch that had uncertain quality. Customers were asked to test an experimental version of a patch. Then, if their testing proved successful, HP created a new, official version of the same fix, whereupon the test customers were expected to remove the old test patch and install the new patch. HP has completely **revamped the acceptance testing process** to address both obstacles.

Customer acceptance testing in now deferred until the last step before general release of the patch, after the patch has passed all of HP's pre-release patch tests. Customers can therefore be assured that this is a patch that HP considers ready to release. Furthermore, the patch that is given to the customer to test is exactly bit-for-bit the patch that HP intends to release for general distribution at the end of the customer acceptance testing period. This means that when acceptance testing finds no problems, which happens at least 90% of the time, the customers doing the testing already have the final solution on their systems. These changes have made acceptance testing much more convenient for customers, so the number of patches going through acceptance testing is increasing. In the first months after the new process was put into use, acceptance testing found an average of one defective patch per month. The faulty patches were then reworked before being released, thereby minimizing customer exposure to the problems. This number of patch defects found during acceptance testing should increase as more patch development teams take advantage of the new process.

4. **Release patches in a timely fashion.**

   It takes time to create and test a high-quality patch. Yet customers expect a solution quickly, especially when a critical problem has crippled their systems. Partial solutions to this dilemma range from addressing defect reports promptly to shortening the development lifecycle by increasing parallelism to equipping centralized testing facilities so they can provide rapid turnaround.

   Managing the backlog of customer-reported critical and serious defects guarantees that customers won't need to wait indefinitely for solutions to the problems they are experiencing. We aim to have a solution coded within two weeks after a defect report arrives in the responsible lab, although it may take considerably longer before a patch containing the fix is actually available to customers.

   One effective strategy for **increasing parallelism** has been to develop patch code and patch documentation concurrently rather than to document a completed patch. Another has been to allow patch developers to get early feedback on the integrity of their patches and the suitability of their patch documentation through voluntary pre-release testing at the centralized Patch Clearinghouse. Many of the tests performed in the PCH release testing discussed in point #3 are also available as soon as a defect fix has been packaged as a patch. The PCH pre-release testing can be done concurrently with regression and other system testing. This usually clears the way for the release tests to pass on the first attempt.

   Creating patches proactively when HP discovers software defects, as discussed in point #1 above, means HP can have a solution ready before customers discover the problem rather than having customers wait for a solution. It also means that the defect fixes contained in these patches are likely to appear in an earlier Quality Pack bundle (For a discussion of Quality Pack bundles, see point #5 below).

   An alternate approach to timeliness is to **release patches only on a regular, predictable schedule**, say once a quarter. This has worked particularly well for HP-UX graphics workstation software, which has an excellent reputation for producing software with few defects. Customers have come to expect quarterly updates and are almost always willing to wait for the next update to get their fixes. Only once per year at most is there an urgent need for a fix that causes this lab to create a patch between scheduled updates. This approach is not likely to be successful for an organization that receives a large number of critical and serious defect reports.

5. **Make patch management straightforward.**

   Just creating the right patches and making them highly reliable doesn't solve the customers' system problems. HP must also make it possible for a wide variety of customers to quickly and easily obtain the patches they need, whether they are seeking a fix for a particular problem that is impeding their work or preparing for proactive system maintenance. In some cases, customers pay HP to select patches for them, but many customers do their own patch maintenance. HP's approach to simplifying patch selection involves rating patches according

to the risk of failure each one carries, separating defect fixes from enhancements whenever possible, and creating patch bundles for different needs.

Years ago, HP had the philosophy that the latest patches must be the best patches because they contained the latest fixes and enhancements. We learned over time that the latest patches are not necessarily the most appropriate for all customers. Some customers can tolerate almost no unscheduled down time. The best patches for them are likely those that have stood the test of time.

For the past three years, **every HP-UX patch has carried a patch rating**, which is an assessment of the risk that the patch contains or exposes a defect. When a patch is newly released, it is rated at level one, the lowest rating. After it has been available for a period of time and has been downloaded onto a certain number of customers' systems without any problems being reported, the patch's rating level is raised to two to indicate that the risk of adopting it has been reduced. Finally, the patch is tested by our Patch and Enterprise Solutions Test Center (PEST, formerly the Enterprise Patch Test Center), which runs a number of test suites on system software stacks representative of those used by our most risk-averse customers. If it passes the PEST testing, its risk is judged to be low, and it is given HP's highest patch rating, level three. Customers who confine their patch selection to HP's level three patches reduce their risk of installing a defective patch by at least 80%.

Customers can benefit from patch ratings during both reactive and proactive patching. HP's online IT Resource Center (ITRC, itrc.hp.com) provides a tool called Patch Database for customers and HP support engineers searching for a patch to solve a particular problem, that is, patching reactively. In response to a search query, this tool returns what HP considers to be the best patch that matches the search criteria. "Best" is defined as the newest patch with the highest rating. For proactive patching, HP creates Quality Pack patch bundles that deliver defect fix patches, usually at level three. Level two patches are included only when necessary to fulfill a dependency or when there is no level 3 patch that fixes the same defect. Rarely is a level one patch included in a Quality Pack bundle. For customers who want more customized proactive patching, the ITRC's Custom Patch Manager tool on the ITRC also recommends patches based on their ratings.

We have also learned that **it is not necessary, or even wise, to instruct customers to remove all problem patches from their systems**. Sometimes the choice to keep using a problem patch is the best alternative for the customer. Prior to October, 2001, HP issued a patch recall notice whenever a patch was found to be defective. Often, these notices directed customers to remove the faulty patches and either revert to an older patch for the same problem or install a newly produced patch. We now understand that doing so causes unplanned down time and destabilizes systems, often unnecessarily. Today, when we confirm a patch problem we issue a patch warning instead of a recall. This simple change in terminology better communicates that we have produced a defective patch that may be causing problems on the customers' systems. We now **categorize each warning as critical or non-critical**, which lets customers know how rapidly they should address the problem. Also, we rarely direct all customers to remove the patch. Instead, we give them **detailed information about symptoms of the problem, the circumstances under which it will**

occur, and the options that are available to them. Customers then determine whether they are impacted by the problem and make a conscious decision to remove it or to live with it.

The benefits of segregating defect fixes from enhancements in patches were discussed in point #2 above. HP carries out this **separation of defect fixes from enhancements at the patch bundle level** as well. The Quality Pack bundle provides defect fixes, the Software Pack provides enhancements, and the Hardware Enablement bundle provides the patches needed to support new hardware.

Good patch documentation is an important component of making patch selection straightforward. Customers may prefer to select only certain patches from a bundle when doing maintenance, or may need to select from several candidate patches for a particular problem they are experiencing. They need to understand the purpose of every patch they are considering. They also need to understand any manual customizations they will need to perform for the patches they select. As discussed above, the reviews performed by the Patch Clearinghouse have improved the content and consistency of HP-UX patch documentation.

Results, Conclusions, and Future Directions

The HP-UX Patch Program continues to solicit customer feedback in several ways:

- We visit customer sites whenever possible to observe the customers' work environment and their problems and to gather feedback on our future plans.
- Several of our managers have cultivated one-on-one relationships with particular customers and can call these customers as necessary to solicit their opinions on proposed changes.
- We conduct one or two patch-focused customer surveys per year, generally at HP World and over the Internet.
- A year ago we formed a small Customer Advisory Board, a diverse group of customers who meet several times per year by phone and once a year in person to air their concerns and help us assess the potential impact of improvement projects we are considering.

Recent survey results and anecdotal evidence indicate that HP-UX customers are considerably more satisfied with HP-UX patch quality than they were five, or even two, years ago. From the high-level view, our efforts have been successful. We have made a positive contribution to the Total Customer Experience of HP-UX customers. However when we look at the impact of individual changes we have made, the results are mixed.

Our biggest disappointment is that we have not made as much progress as we would have liked in the area of releasing flawless patches. We have made many beneficial changes over the past five years, but there have also been setbacks. Patch warnings declined by roughly 30% from 1998 to 1999 and again from 1999 to 2000. They were virtually unchanged from 2000 to 2001 and actually increased in 2002. After three quarters of 2003, we appear to be on track to return to the 1999 levels. Many of the patch problems in 2002 and 2003 are traceable to loss of key domain experts over the past two years through retirement, downsizing, and the transfer of maintenance responsibilities to offshore labs. Too much knowledge was carried only in the

experts' heads. When they were no longer available to develop or review patches, mistakes went undetected until after the patches were in use by customers. Hopefully, consistent use of the Patch Quality Checklist will not only compensate for, but actually be better than, relying on experts.

Customers used to complain that defective patches were causing them to bring down their systems too frequently outside of planned maintenance windows. They now tell us that they rarely need to incur unplanned downtime to cope with a problem patch. The improved patch selection methods that use patch ratings are keeping the most risky patches from being installed on systems that can't tolerate downtime. Also, the more comprehensive information we are providing in patch warning notices is allowing them to live with a good percentage of the problem patches on their systems, at least until their next scheduled maintenance window.

It is too soon for us to be able to assess the effects of the more recent changes targeting patch reliability. Because of the time it takes to produce and test a patch and the additional time required for customers to install and exercise it, we cannot see the results of a particular change until roughly a year after it was introduced.

Although customers are more satisfied than they were in the past, they still are not completely delighted with HP-UX patch quality. Moving forward, we will continue to work on improving patch reliability, not only to please customers, but also to reduce costly rework for ourselves. We will need to implement the reliability improvement methods with careful attention to their potential effect on the time needed to deliver patches, since customers would like solutions to be available more quickly than they often are today. We will continue to try to find more patch problems before the patches are released, and we will look for ways to find these problems earlier in the lifecycle. We are also beginning to work with customers to determine what additional improvements we might make to the information we provide them about patch problems.

It will not be easy to continue to make improvements to HP-UX patch quality despite continued management commitment to our efforts. For one thing, we have already made the most obvious changes. Also, software maintenance is likely to continue to migrate offshore, so we will face more loss of expertise. In addition, the complexity of our code will increase as we integrate Tru64 cluster technology into future releases of HP-UX, raising the risk of making mistakes. Nonetheless, we will persevere in our efforts. Improving the Total Customer Experience is a basic tenet of the HP culture, and the HP-UX Patch Program must do its part.

References

[1] Beyer, Hugh and Holtzblatt, Karen. Contextual Design. San Francisco: Morgan Kaufmann Publishers, Inc., 1998.

[2] Balza, John. "Management Commitment to Quality Requires Measures," PNSQC, 2003.

Test Estimation

Ross Collard,
Collard & Company

Question: When will the system testing be completed??? (Asked by an eager pest -- your boss -- with a tone of great anxiety.) At the time he or she asks this question, you do not know (a) the final scope of the functionality, (b) when the developers will deliver the final system for testing, and (c) what test resources you will have available. The boss wants a definitive answer and a drop-dead commitment from you in two minutes anyway. Let's face it: developing realistic and credible estimates is a critical survival skill for test professionals and managers. This session provides a practical approach to improve your estimating and negotiating skills.

Ross Collard is a software testing consultant whose clients have included Alcatel, American Express, Bank of America, Blue Cross/Blue Shield, Boeing, Cisco, Citibank, Dell, EDS, Exxon, General Electric, Goldman Sachs, the Federal Reserve Bank, Hewlett-Packard, IBM, Intel, Johnson & Johnson, JP Morgan, Merck, Microsoft, NASA, Nortel, Procter & Gamble, Prudential, Sears Roebuck, State Farm Insurance, Swiss Bank, the U.S. Air Force and Verizon. He has conducted seminars for businesses, governments and universities, including Harvard and U.C. Berkeley.

REALISTICALLY ESTIMATING SOFTWARE TESTING

Ross Collard, Collard & Company

Question: When will the system testing be completed??

(Asked by an eager pest -- your boss – with great anxiety.)

At the time this question is asked, you do not know

(a) the final scope of the functionality to be tested,

(b) when the developers will deliver the system for testing (and how testable it will be), and

(c) what test resources will be available to you.

The boss wants a drop-dead commitment from you in two minutes anyway.

Question: How do you weigh a pig?

Reasons Why Good Estimates are Important

- Testing time is often cut back with minimal notice.
- The test team is blamed for late delivery of the system.
- Deadlines are often imposed by outsiders on the testers.

Let‘s face it: developing realistic and credible estimates is a critical survival skill for testers.

- Our track record is poor... a Computerworld study says *less than 5% of all systems projects deliver on time and on budget*.

Reasons for Poor Estimates

- Imperfect requirements and functional specs.
- Scope creep and changes.
- Unknown start date: when can the testing begin?
- Readiness of the system for testing is not known.
- Vague acceptance criteria for the system after testing.
- Unknown availability of competent people for testing.
- Variations in productivity among test professionals.
- Test equipment and facilities is unknown or untried.

.... And so on.

Estimating is hard work, and requires time and good thinking.

The message: we must manage the testing uncertainties, in order to be able to estimate effectively.

Knowledge is power. Good estimates depend on good information.

Estimating Techniques (adapted from Barry Boehm)

(1) Development Ratio
(2) Prior Test History
(3) Bottom-Up
(4) Top-Down or Global
(5) Formulae and Models
(6) Parkinson's Law
(7) Pricing to Win
(8) Cost Averaging
(9) Consensus of Experts
(10) SWAG
(11) Mandated Deadlines
(12) Self-Fulfilling Prophesy
(13) Estimating Tools
(14) Re-Estimating by Phase

Overview of Techniques

- ***Development Ratio:***

- Testing is performed within a larger context, either coordinated as part of a system development project or as part of on-going system maintenance activities. The ratio approach estimates the size of the testing effort as a percentage of the overall development project or maintenance budget.

- This ratio method has the advantages of being expedient and easy to apply. It also has the important advantage of political acceptability.

- Anther way to estimate the testing is based on the staffing ratio (developers to testers).

There are three main limitations to the ratio method:

- The test estimate can be only as accurate as the development estimate, and probably is less accurate.

- The ratio of development to testing to be used in estimating may be just a guess, and a politically sensitive one.

- How large a “slice of the pie” do the testers get? Haggling over its size can lead to an “us versus them” mentality instead of cooperation.

(2) Prior Test History:

- Related prior experience is the *best predictor* of the test effort. But is the history available, applicable and accurate?

(3) Bottom-Up or Micro-estimating:

- Develop a detailed test work plan.
- Estimate each atomic task, and add the total.
- Cannot be performed until a detailed task list is developed.
- Tasks that are omitted receive an estimate of zero hours.
- Nothing internally uncovers a consistent bias.

(4) Top-Down or Global Estimating:

- Big-picture, compare the overall scope to other similar efforts.
- Provides a useful cross-validation on other techniques.

(5) Formulae and Models:

- Measure the characteristics and feed them into a formula.
- These characteristics may be difficult to measure. There is always a question of how well calibrated and

(6) Parkinson's Law:

- Estimating is not just calculation; but also human factors like negotiating skills.
- Sets the estimate at the maximum the market will bear.
- Provides generous factors of safety.
- Parkinson's Law estimates are difficult to justify.

(7) Pricing to Win:

- The opposite to Parkinson's Law; setting the estimate the tightest and lowest possible in order to win a bid.

(8) Cost Averaging:

- Varies the set of assumptions used in estimating. Estimates are be calculated under
  the most optimistic scenario,
  the most probable scenario and
  the most pessimistic one.

Uses the weighted average: [{ a + 4b + c } / 6].

(9) Consensus of Experts:

- Ask knowledgeable people to guesstimate independently. Then have them compare the independent estimates and derive a consensus.

(10) SWAG:

- Develop a quick, off-the-cuff guess.
- Give a range instead of a single number as an answer.
- With SWAGs, the uncertainty is high.
- SWAGs should be the exception, not the rule.

(11) Mandated Deadlines:

- The test team is told the deadline date and informed that they must meet it -- or else. People feel it is futile to develop estimates: they will not be heard.

But: ***never underestimate your power to negotiate!!***

(12) The Self-Fulfilling Prophesy:

- An optimistic date for test completion is established.
- The test team then makes a heroic effort to make the date.
- If they make the date, they have set expectations for the future.

(13) Estimating Tools:

- Commercially available software packages contain formulae to estimate project size and duration.
- No tool focuses specifically on estimating the test effort.
- The formulae and assumptions are proprietary and hidden.
- Tunable – which can be dangerous: “get the answer you want”.

(14) Re-Estimating by Phase:

- Estimating should not be viewed as one-time, but as a recurring activity.
- Estimates need to be re-visited during a project, as more reliable information becomes available.
- Useful points at which to estimate (and re-estimate) the test effort in a project are at the end of the major work phases – if a phased approach is being used.

Informational estimates need to be distinguished from ***commitment*** estimates.

Attainable Levels of Estimation Accuracy

(For moderate-size, moderate-risk projects.)

Milestone	Scope of Estimate	Target Accuracy (*)
Feasibility Study	Overall Test Effort	50% to 75%
System Reqts.	Overall Test Effort	40% to 50%
Document	Test Planning Effort	25%
Test Plan	Overall Test Effort	30%
	Test Execution Effort	20%
Completion of	Follow-up Debugging	15%
of First Cycle	& Fixing	
of Testing	Re-Testing after Fixes	10%

(*) In my observation

WHAT WE NEED TO KNOW IN ORDER TO ESTIMATE

- The clients' quality expectations.
- When the system will be ready to test.
- The testability, quality and completeness of the system when it is delivered to testing.
- The efficiency of the test resources.

... And there's lots more.

Factors of Safety

A “fudge factor”needs to be included in the test estimates to provide a contingency for uncertainties.

A reasonable factor of safety for a moderate-size, moderate risk testing project is 15% to 25%. For a high-risk, large-size project, the factor of safety should be much higher: perhaps 50% to 100%.

This assumes the managers and clients will allow this generous factor of safety.

GUIDELINE FOR THE ESTIMATING PROCESS

1. Define the Estimate Objectives

- What is the purpose of this estimate? What is the target accuracy? By when is it needed? What is the scope?
- What background information is already known?
- What "denominations" should the answers have?
- Who needs to be involved in developing and reviewing the estimate?

2. Plan the Estimating Effort

- Select the most appropriate two or three estimating techniques to fit this situation. Do not choose only one technique.

- Identify the input data which needs to be gathered or guesstimated, to use the selected estimating techniques.

- Document all major assumptions. (Such as: 3.1459 rocks equals one pig.)

3. Calculate the Draft Estimates

- Gather the input data needed for the selected estimating techniques. Make appropriate assumptions where hard data is not available.

- Calculate the estimates using each of the selected techniques.

- Re-check the calculations and fix any obvious errors.

- Again, document your assumptions and keep an audit trail.

4. Compare and Refine the Estimates

- Check if the spread among the estimates is acceptable.

- The target accuracy provides a stopping rule. (The spread among estimates must be less than the target accuracy.)

- If the spread is not acceptable, loop back to the earlier steps and re-compute. The wider the divergence, the further back you should go: (a) re-compute, (b) re-check input data and assumptions, or (c) question selection of techniques and target accuracy.

5. *Perform an Independent Review of the Estimate*

- Review and validate the draft estimate:
  - With knowledgeable, experienced reviewers.
  - With the people who we need to help us achieve the estimate.
  - With the people who need to approve and sign-off.

- Document the estimate in its final form. Attach the supporting paperwork (assumptions and calculations) which bolster your estimate.

6. Keep the Estimate Up-to-Date

- Plan the follow-up processes to:
  - Re-calculate and refine the estimate as conditions change and/or better information becomes available.

7. Track Progress vs. Plan

- Monitor the accuracy of the estimate vs. reality and learn from the comparison.
- Keep the original estimates and updates.
- Plan for a end-of-project review of lessons learned.

Time Needed for Developing Estimates

Good estimates require time, analysis, and contemplation. These guidelines assume that the test project is reasonably well known and the project risks are not huge.

Test Project Size (Person-days of effort)	**Time Needed for Estimating** (Hours)
10 (2 testers, 5 days)	1 hour (1.25% of total effort)
100 (5 testers, 20 days)	10 hours (1.25%)
500 (10 testers, 50 days)	40 hours (1.25%)

Common Oversights in Estimating

- Time delays waiting for bugs to be fixed. (5% to 15%) (*)
- Delays because of changes to functionality. (5% to 35%)
- Gremlins in the test environment. (5% to 15%)
- Re-testing after fixes have been applied. (10% to 20%)
- Re-running test cases which were mis-applied. (2% to 5%)
- Documentation of the test results. (3% to 10%)
- The overhead for meetings and coordination. (5% to 10%)
- The effort to track and report the testing status. (3% to 8%)

Common Oversights in Estimating (continued)

- Follow-up on test cases which failed, including problem reporting and unofficial debugging. (5% to 5%)
- Overhead for test support activities like managing the test environment. (5% to 15%)
- Overhead for test-related activities, like psychological counseling with distraught programmers. (15%)

Together, these omitted activities can easily add 50% to 150% to a "bare bones" budget.

Typical Overhead

What ratio is typical of the total test effort to the hands-on test execution? In a survey which I conducted of 145 testers, their overhead ratios, in a situation where the hands-on testing took two hours, were as follows:

Ratio of Total Test Effort to Hands-on Test Execution	Percentage of Testers
2-to-1, or less	0%
2-to-1 to 4-to-1	6%
4-to-1 to 6-to-1	21%
6-to-1 to 8-to-1	37%
8-to-1 to 12-to-1	16%
12-to-1 to 16-to-1	10%
16-to-1 to 20-to-1	5%
20-to-1 or more	5%

SCOPING THE TEST ESTIMATE

A major reason for inaccurate estimates is fuzziness about the scope of what's included in these estimates.

Does the estimated testing time include:

- Only application functionality testing, or specialized testing?
- Debugging and fixing?
- Test project management overhead?
- Re-testing after fixes have been made?
- Testing done outside the testing team, e.g., by customers?

RELATIVE PRODUCTIVITY OF TEST PROFESSIONALS

(Electronic Data Systems)

Test Situation:	Small	Moderate	Large, Complex
Test Trainee	4-8 hours	20-40 hours	□
Tester I	2.5-5	12.5-25	35-75
Tester II	1.5-3	7.5-12.5	25-50
Senior Tester	1-2	6-12	20-40
Lead Tester	1-2	4-8	15-30

Per function point (FP) or 125 LOC (lines of code).

Estimate Adjustment Factors

The Need for Calibration

The estimating rules of thumb which are introduced here need to be calibrated to your specific environment and testing culture.

Calibration is difficult.

Instead of trying at the beginning to calibrate the estimating rules of thumb, a better approach is to use the suggested rules "as is" for a few weeks and see how well they work.

Estimate Adjustment Factors

- Product complexity and size
- Product stability vs. volatility; readiness to test
- Risk inherent in the product
- Efficiency of the support infrastructure and test execution process (e.g., test automation)
- Tester skills and resource availability

… and many more.

ESTIMATING RULES OF THUMB

1. The Number of Test Cases
2. Test Planning
3. Test Execution
4. Debugging & Fixing
5. Re-Testing after Fixes
6. Test Project Management Overhead
7. The Ratio Approach
8. Test Resource Allocation
9. Specialized Types of Testing

1. Number of Test Cases

Definition of a Test Case

The first step in test project estimating is to develop an estimate of how many test cases will be needed.

One problem in estimating the number of test cases is a lack of consistency in what the term "test case" means (apples & oranges).

Basis for Estimating Test Cases

- Number of source code statements.
- Complexity (number of branches or paths).
- Size of the functionality (function points).
- Number of features to be tested.
- Number and complexity of the input transactions.
- Number, size and complexity of the data bases.
- Number, size and complexity of the user interfaces.
- Size of the executable file(s).
- Number, size and complexity of the classes or objects.

1. Number of Test Cases: Based on Features

The number of test cases needed are:

- 1 - 2 test cases per atomic feature (for simple, low risk features).
- 10 - 20 test cases per feature(for complex, high risk ones).

Based on Size in Function Points

The number of test cases needed are:

- 2 - 3 test cases per function point for high-risk software.
- 0.5 to 1 test cases per function point for moderate to low-risk software.

1. Number of Test Cases: Based on the Executable File(s)

On average, 1,000 LOC in a higher-level language (like C), compiles to form an executable file which is 10 to 12.5 kilobytes in size. This ratio depends on the compiler, operating system, and the platform.

The number of test cases needed, based on the size of the executable file, are:

- 3 to 4 test cases per 100 kilobytes of executable files (for simple, low risk functionality).
  30 to 45 test cases per 100 kilobytes of executable files (for complex, high risk functionality).

1. Number of Test Cases: Based on the Input Transactions

The number of test cases needed are:

- For each data field within each input transaction, 2 test cases (one positive and one negative).
- For each high-risk data field within each input transaction, 9 test cases (classic boundary value).
- For each interrelationship among two or more data fields, 2 test cases (one positive and one negative) for each interrelationship.

1. **Number of Test Cases: Test Cases for Modifications to Existing Code**

   For Small Modifications

   New and modified software components:

   If n% of the existing software LOC is modified, the number of test cases required just to test the modifications is n% of the original test cases.

   This rule is valid for small values of n (<= 5%).

For Small Modifications (continued)

Related but unmodified components:

Comprehensive regression test:

Same number of test cases as for new software.

Partial or light regression test:

10% of the number of test cases as for new software.

Unrelated and unmodified components

Partial regression test:

0% to 10% of number of test cases as for new software.

1. Number of Test Cases: Feature-Based Estimating

(a). New Software

Number of Features in each Category	Category	Number of Test Cases Per Feature	Total Number of Tests Cases Needed
_____	Simple, Low Risk	4	_____
_____	Simple, High Risk	8	_____
_____	Complex, Low Risk	8	_____
_____	Complex, High Risk	16	_____

(b). **Existing Software**, which has not directly been modified but is affected by the modifications.

Number of Features in each Category	Category	Number of Test Cases Per Feature	Total Number of Tests Cases Needed
_____	Simple, Low Risk	1	_____
_____	Simple, High Risk	3	_____
_____	Complex, Low Risk	3	_____
_____	Complex, High Risk	6	_____

Estimating Rules of Thumb (continued)

2. Test Planning: Productivity of Test Case Development

(a). Small, Simple Test Cases (1 to 8 atomic test steps per test case).

10 to 20 test cases per day for manual test cases.
3 to 5 test cases per day for automated test cases.

(b). Large, Complex, Multi-Step Test Cases (10 to 50 atomic test steps).

1.5 to 3.0 test cases per day for manual test cases.
0.5 to 1.0 test cases per day for automated ones.

Estimating Rules of Thumb (continued)

3. **Test Execution: Key Factors to Consider**
   - Number of test cycles or runs.
   - Elapsed wait time between test runs.
   - Regression test strategy.
   - Number of test cases executed per test batch.
   - Time and resources needed for each run of a test batch.
   - Overhead to manage and maintain the test environment.

     Number of different configurations or platforms to be tested.

3. **Test Execution** (continued)

The time per test run depends heavily on the environment..
The work effort to be estimated for test execution includes:

- Preparation (e.g., time to understand the test case).
- Set-up of the test environment.
- Execution of the test case or test suite.
- Capture and evaluation of the test responses.
- Determination of a test case's pass/fail status.

  Logging and reporting of the test results, etc.

3. Test Execution (continued)

Test Environment Set-up & Maintenance

The effort required for set-up can range from trivial to the dominant portion of the entire test execution effort.

Re-testing after Fixes

The re-testing after fixes have been applied can be substantial. To estimate the re-test effort, we must determine two factors:
(1) How many re-cycles of testing are likely?
(2) What percentage of the test cases will be re-executed in each cycle?

3. Test Execution (continued)

Defect Aging

The elapsed time to re-work and turn-around a fix, ready for re-testing, is called the defect aging. This may cause waiting time before the testing can proceed, especially in the case of blocking defects.

Defect aging is usually not under the control of the tester, who has to either (a) wait, (b) develop a work-around in order to continue testing, or (c) re-sequence the work flow.

Estimating Rules of Thumb (continued)

4. Debugging and Fixing

Defect estimates based on function points (FP):

- The number of defects which will need to be fixed is roughly equal to the FP count for the system, raised to the power of 1.25 (Capers Jones).

Defect estimates based on LOC (lines of code):

- 1.5 bugs per 100 LOC (for new code in a 3rd generation language, before integration and system test).

An Example of Estimating the Defect Count

Suppose that 10,000 LOC in the C programming language has been newly written, and has been unit tested, integrated, and integration tested by the developers. The software is then delivered for a system test. If typical, it will still contain 100 to 150 hidden defects at the time of entry into the system test.

If the system test is typical, 80% of these defects will be found, though not all need to be fixed prior to the system release: about 80 to 120 defects will be found in all. The other 20% of the defects remain within the product and ship out the door to the customer.

4. **Debugging and Fixing** (continued)

Time Needed to Fix Defects

(a). Small number of simple defects (Jack Adams, IBM)

Severity	**% of defects**	**Time per defect**
Simple	85%	1-4 hours
Moderate	10%	8-20 hours
Complex	5%	40-80 hours

Time Needed to Fix Defects (continued)

(b). Large number of defects (Jack Adams, IBM)

Severity	% of defects	Time per defect
Simple	60%	4-8 hours
Moderate	20%	12-24 hours
Complex	20%	40-80 hours

Hewlett-Packard's standard is more aggressive: 6.3 hours per defect.

4. **Debugging and Fixing** (continued)

Overhead for the Testers' Involvement in Debugging

Involvement (*)	**Ratio (**)**
Minimal	10%
Moderate	33%
High	100%

(*) Degree of the tester's involvement in debugging & fixing.
(**) Time expended by the tester, for every developer hour (in hours).

5. Re-Testing after Fixes

Factors determining the amount of re-testing include: the number of defects found in testing, the degree of regression coverage planned for successive iterations of testing, and the likely number of test-and-fix cycles needed before the testing is completed.

Type of Re-test	Percentage of Passes	Degree of Regression Test	Number of Test Cycles	Re-Testing Effort
Low	97%	None	One	3%
Moderate	85%	25%	Two	50%
High	67%	100%	Four	400%

6. Test Project Management

Project Size	System Duration	Risk	Project Style	Mgt Overhead
Small	Short	Moderate	Self-directed	3%-5%
Moderate	Moderate	Moderate	Informal	5%-10%
Large	Long	High	Formal	10%-15%
Gargantuan	Very Long	Very High	Bureaucratic	20%-25%

Estimating Rules of Thumb (continued)

7. The Ratio Approach

The ratio approach estimates the testing effort as a percentage of the total project effort.

If we are told that a project will take 100 hours to program, the question is how many hours to allocate for testing.

It is useful to look at historical patterns (on the next page) to gain a sense of what the ratio is likely to be.

	Traditional Projects	**Projects with Built-In Quality**
Project Phase	*Time Needed*	*Time Needed*
Project Initiation	25 hours	30 hours
Analysis & Design	200	230
Test Planning	--	50
Coding & Debugging	100	120
Unit Test	100	75
Integration & Sys. Test	200	100
Re-work (Coding)	25	10
Installation	25	20
Total:	675 hours	635 hours

Source: IBM

7. **The Ratio Approach** (continued)

These ratios surprise many systems professionals. When asked how many test hours are required for 100 hours of programming effort, many professionals reply: "about 20 to 30 test hours".

The numbers above indicate that the ratio of programming to testing, instead of being about 3 to 1, actually is about 1 to 3.

> ***The conventional wisdom ratio of 3 to 1 appears to be wrong.***

An Example Using the Ratio Approach:

The time expended in building Microsoft Windows 95 (measured in million of hours) was:

Specification and design	3.22
Coding and debugging	2.45
Unit testing, build process & build testing	3.65
Total effort before system testing:	9.23
System testing	2.33
Total effort including system testing:	11.57

Testing hours do not include the beta testing effort, another 2.5 million hour by users.

8. Test Resource Allocation

Activity	Percentage of Test Time	
	Manual Testing	**Automated Testing**
Test planning (*)	20% - 30%	50% - 75% (**)
Test execution	45% - 65%	10% - 20%
Test evaluation & follow-up	25% - 35%	20% - 30%

(*) Assuming that no automated test cases are already available.

9. Specialized Types of Testing

Type of Test	Importance of Test		
	Low	Mod	High
Change Test (x%)	x%	1.5x%	2x%
Cross-Platform Test	2-5%	5-10%	20-35%
Regression Test	2-5%	15-25%	100%
Usability Test	2-5%	5-10%	20-35%
Performance/Stress Test	2-5%	5-15%	25-50%
Security & Controls Test	2-5%	5-15%	25-50%
User Acceptance Test	2-5%	5-10%	20-35%

Caution: averages like these are ***dangerous***.

HOW HUMAN BEHAVIOR AFFECTS ESTIMATES

Behaviors which lead to poor estimates:

- Biasing people by telling them the desired answer up front.
- Treating every estimate as a starting point for negotiation.
- Regarding estimates, especially early ones, as commitments.
- Failing to understand the assumptions behind an estimate.

HUMAN BEHAVIOR

- Not knowing or understanding the degree of uncertainty.
- Asking for, and giving, estimates immediately and off the cuff.
- Expecting estimates to hold firm when the conditions change.
- Not providing sufficient time and information to calculate a realistic estimate.

DEMOCRATIZATION OF ESTIMATES

The people who will be doing the work should estimate its size and duration, not their managers, clients or third parties such as marketing.

The initial development of a draft set of estimates is best done by the test team leader or by a small inner core of skilled people.

Once the initial set of estimates is developed, it is important to obtain buy-in from the people who are actually going to do the work.

IN SUMMARY

Regard these rules of thumb as starting points for your own estimating efforts.

We would appreciate your input to refine and improve our estimating data and techniques.

Provide feedback on your estimating experiences to Ross Collard at rcollard@attglobal.net. Thank you.

A Survey of Quality Practices in Open Source Software and the Linux Kernel

Craig Thomas, Manager Test and Performance Group
Open Source Development Labs, Inc.

Abstract

Many IT companies are seriously looking into Open Source Software products as solutions to their business needs. Corporations are looking for lower total cost of ownership for their business applications, while at the same time maximizing their productivity. The Linux™ operating system has become one of the most recognized collection of open source software products to date and has begun to replace the Unix® operating system as the OS of choice for many IT servers. Telephone Equipment Manufacturers are looking to Linux to run their network appliances. Data centers are working to influence the Linux kernel developers to add features required of sophisticated databases. What makes the development of Linux so interesting is the fact that there is no coordinated management activity to plan and implement new features in the kernel. Linux has evolved through numerous contributions of developers working from locations all over the world and sending code improvements to a single maintainer.

Unlike closed source software where the development team consists of closely managed developers and testers, the Open Source Software community is widely distributed, with very little coordinated project management activities. With active Linux kernel developers numbering in the thousands world wide, how is anyone able to deliver a quality product with amorphous ownership, unwritten community rules and whose development community goes beyond corporate and cultural boundaries?

This paper explores some of the quality practices and tools used to develop the Linux kernel and its supporting utilities. The paper will outline some of the development practices as well as the verification, validation, and test efforts used to aid in the quality of Linux releases. Examples of some of the tools used within the Open Source Development Labs to enhance the quality of Linux will also be presented. The paper concludes with identified areas where more effort is needed to further improve the quality of Linux.

Craig Thomas is the Manager of the Test and Performance Group at Open Source Development Labs. With over 16 years testing in Unix and Linux environments, he has been involved with functional test, integration test, system test, and performance test for enterprise systems. Craig has a BS in computer science from Washington State University and has served as the co-chair of the ASQ Certified Software Quality Engineer examination review committee.

Introduction

Open Source Software (OSS) products are becoming viable alternatives to closed source software in desktops, IT servers, and even main frame computers. Familiar OSS products such as Apache, Mozilla and Samba are used throughout the computing industry. Apache, for instance, is has the greatest market share of any HTTP server across the internet[1]. Mozilla is a feasible option for a desktop web browser. Samba is a valuable application that allows machines running different operating systems to share files or print servers, for example. Now days, there is a good chance that one can find OSS products running on one system or another in a computer intensive business.

OSS projects vary in size and scope. They can range from one person developing a small application to several thousand people contributing to an operating system. Sourceforge, a common site where people manage their projects, has over 62,000 open source projects. Some projects provide extremely useful software products of high quality, while others are deemed totally useless due to lack of quality. For the successful projects, how is a high standard of quality obtained? What practices are used by these project teams?

The Linux kernel community has adopted some unique quality practices that are not employed by a conventional software development team. For example, with a widely distributed community of developers, communication between them involve public mail lists and Internet Relay Chat (IRC) channels. Development activities in areas such as configuration management, development models, verification, validation, and test have evolved to fit the particular needs of the programming community. Because there are so many elements involved with software development, this paper attempts to expose some of the more commonly used techniques, focusing on testing and defect tracking. In order for the reader to understand why this is different, it is first necessary to describe Open Source Software in general.

What is Open Source Software

Open Source Software is essentially defined by the terms of a distribution agreement that maintains the integrity of the author's program and provides the recipient the right to modify and redistribute the program, so long as the original license agreement remains in tact. Among other elements of an open source distribution agreement is that all programs must include source code or provide the source at the user's request. A full definition of OSS software can be found on the Open Source Initiative web site[2].

In simplified terms, Open Source Software is typically distributed with its source code. It may or may not be free of cost. The license of OSS allow recipients to modify and redistribute the code without further need to pay royalties or other compensation to third parties. The Open Source Initiative has several open source license examples that have small variations on the general theme[3]. One of the most commonly known licenses used by Open Source products is the GNU General Public License (GPL).

As a result of the way these licenses are structured, people from all over the world are able to take copies of GPL code, enhance it and redistribute it for others to improve upon it. Several OSS products have been improved in this way and are widely used throughout the computing industry: web server software, mail transfer agents, domain name server programs, and operating systems. Linux itself is controlled by a GPL license and has undergone continual quality improvements because of it.

Development Methods for the Linux Kernel

The Linux community is composed of students, professors, private volunteers, and people from the corporate world. Improving the Linux kernel follows a highly distributed development model where volunteers from all over the globe contribute to aspects of the operating system including process schedulers, virtual memory, file systems, networking, and device drivers. With concurrent activities going

on throughout the world, a means to track changes and manage the base source tree is imperative to stable releases of the kernel.

Source Management and Code Review

Changes to sources are released as patches. Patches are provided as a single file showing the differences between a modified file in the source tree and the original file. Linux patches are posted through email to the Linux kernel mail list where they can be reviewed, discussed, or applied to other individual source trees. The modifications are publicly reviewed and sometimes debated on the mailing list by many members. Once a patch is posted, anyone can incorporate that patch into their source tree, rebuild a kernel, and test the change. If the patch is fairly simple (a few lines of changed code) the patch can be reviewed through code inspection. For larger patches, performance tests are run against them to assure that performance has not degraded. The important aspect of submitting patches to the Linux kernel mailing list is that many people have an opportunity to review or test a patch. The patch undergoes extensive scrutiny and if problems exist in the patch, they are quickly identified and resolved through code inspection and email discussion or through IRC channels devoted to Linux kernel development. Formal code inspections are deemed inefficient because of the highly distributed nature of the development process. It is far more effective to have hundreds of reviewers commenting on patches through a mail list than to convene a group of developers to collaborate on a formal inspection.

An owner of a source tree, known as the maintainer, can choose to apply a patch by merging the diff file into the source tree or rejecting the patch. Specialized tools are used to merge patches into the maintainer's source tree. The Linux kernel is managed by a single maintainer for each stable release and for the current development release.

The current Linux kernel under development, utilizes Linus Torvald's source tree as the root of the Linux kernel. Linus is the creator of Linux. From there, other developers create and maintain their own copies of that kernel as versioned branches in order to test specialized modifications that enhance various system capabilities such as those used for embedded systems, for execution on different system architectures, or for scaling to large numbers of processors. Others make copies from the maintained version of the tree and work on different elements of the kernel, such as virtual memory, process scheduling, file systems, and I/O features. These branches of Linux in turn have their own following of contributors that will provide improvements to these source trees. As maintainer, Linus can approve certain people to push their fixes into the base kernel tree or he can pull specific changes he desires; specialized tools automatically merge and version the change sets. The underlining goal for all people working on the Linux kernel is have their patches accepted into the base kernel tree: Linus' source tree. Once merged into that tree, all future copied trees will have that patch.

Configuration items are controlled by source control software packages such as CVS or BitKeeper. Entire source trees are cloned by individual owners to carry out improvements to a particular released version of the kernel; they act as unofficial distributed branches where sections of the kernel can be worked upon by specialists without affecting the official kernel. It is not uncommon for an individual maintainer to have many copies of the source tree, each tree containing different code modifications so the maintainer can see the effect of a single patch or a specified set of patches.

Traceability of source versions are maintained by the use of source control log files. Log files of check-ins are maintained and publicly available to anyone in the world. For example, the log files for the Linux 2.5 development tree can be found at http://linus.bkbits.net:8080/linux-2.5. Documentation of specifications are sporadic and are not controlled by software configuration management tools with the source code. Instead, they may be found in mail archives of the Linux kernel mail list, or in the bugzilla database created to track version 2.5 defects found in the kernel[4]. User documentation is created by a collaboration of yet another open source community. The project is distinct from kernel development and, like other open projects, does not necessarily have its project synchronized with the kernel development releases.

Testing

Small OSS projects tend to have virtually no testing performed by anyone other than the author of the program. Larger projects depend upon a community of users to find and fix problems through user feedback rather than disciplined test methods. The Linux kernel is becoming an exception. In this case an increasing number of professional software test engineers are developing tests and test environments specifically for the purpose of testing the Linux kernel.

A few individuals in the Linux community maintain micro benchmarks, run the tests against a version of the kernel, and repeat the runs with subsequent versions. Results are then posted to the kernel mail lists. The testers are frequently asked to run the tests again with different patches supplied by the Linux developers. The key to this form of test and check method is that the feedback is rapid; usually a turnaround time for new results is within hours. This is a different approach from commercial operations because there, test cycles are usually planned and scheduled. The test department must complete the other scheduled tests before attempting to rerun a test against a patch. In the Linux community, testers typically know how to apply the developer patch and build the kernel themselves; the feedback of the test result for a new patch is swift. In the Linux community, rapid feedback is a primary objective of testing.

Other testers are expending their efforts to build comprehensive test suites that provide decent coverage of kernel operations. Over the last couple of years, projects that develop test suites for the Linux kernel have begun to appear on Sourceforge. The Open POSIX Test Suite was created to allow professional testers to develop test cases that validate a correct implementation of the POSIX standards[5]. The Linux Test Project was created to provide a series of functional tests that exercise a large portion of the kernel's system calls[6]. The Scalable Test Platform, developed at OSDL, provides a suite of stress and performance tests to exercise kernel releases[7].

Defect detection and correction

When defects are found, they are logged in two manners. The most common method is to submit the problem to the Linux kernel mail list. Many readers see the logged defect and it is not uncommon to receive a patch in a matter of hours for simple problems. Complex defects are usually posted and logged in the open source bug tracking software package, bugzilla. This bug tracking system has a web based interface and is integrated with an SQL database. The list of defects for the 2.5 kernel release can be obtained at the following URL: http://bugme.osdl.org.

As the 2.5 development kernel progressed toward the promotion of the 2.6 stable kernel, the designated maintainer of the stable release developed a detailed list of items that remained to be fixed before the 2.5 versions could be promoted to the 2.6 version[8]. The list complemented the defects already logged in the bug tracking system and covered general features that were integrated into earlier versions of the kernel but were not yet completed. This was a measure of tracking the completion of the remaining features for the 2.6 release. The items in the list covered large issues where fundamental features were not working such as:

- All new Linux enhancements to work on RAID0 devices
- Complete new network driver enhancements for different architectures
- Complete file system enhancements for the various file systems

This list was followed up with two distinct IRC chats, where over 130 people participated. The chat sessions were used to clarify the issues, perform a triage of the problems on the list, determine the current status of the issues, and to assign owners to resolve the issues[9].

Community Groups Working to Stabilize Linux

In addition to Sourceforge test projects, professional testers have formed a new community devoted to testing the kernel. IBM's Linux Technology Center, for example, is composed of many professional testers whose primary job responsibility is the creation of test cases for the Linux kernel. They provided the initial work to the Linux Test Project. The Linux Test Project is an open source project that provides a test suite of functional tests, stress tests, and other workloads to exercise the kernel. The center's test suite has received contributions from testers in other organizations as well, showing the growing community of test developers at work.

The Linux kernel needs the addition of compilers, utilities, management tools, and other applications to make a complete operating system. Distributors package these components with the kernel and sell the bundled software to customers. Distributors such as Red Hat SuSE, and MontaVista manage their own software quality departments; each provides additional tests beyond those exercised in the Linux community. The distributors consider their packaging and testing functions as a differentiation between each other.

The Linux Standards Base is a group formed from the Free Standards Group to create a certification program and test suite that will validate that the Linux kernel and OS utilities adhere to a unified Unix specification[10]. This group's function is to prevent the fragmentation of Linux variants that is prevalent in Unix distributions. By providing a certification test suite that adheres to a specified Linux standard, it is hoped that an application developer can be assured that his application will be portable across different Linux distributions.

Another site devoted to the tracking and removal of defects in the Linux kernel is the kernel janitors discussion group. This group is dedicated to providing information and tracking activities of kernel clean up tasks by way of its mail group and web site. These items generally do not fall within the scopes of the kernel defect tracking system or the 2.5 must fix list; the activities are typically carried out by people wanting to get involved with the Linux community, but don't have the expertise of those that directly contribute to the development of the kernel. The location of the web site containing the current list of clean up activities is found at http://kernel-janitor.sourceforge.net/.

Open Source Development Labs (OSDL) is a vendor-neutral non-profit organization dedicated to enabling and guiding Linux development. It provides data center class resources free of charge to enable developers working on open source projects to test data center and carrier grade enhancements for Linux on large, complex systems. Within its organization, a group of dedicated test and performance engineers are chartered to create system level tests that can be used to provide reliability and performance data to kernel developers. Additional OSDL personnel provide bug fixes and feature enhancements to the kernel and perform tests on kernel releases.

OSDL Projects and Test Tools

OSDL has been involved with several projects and initiatives that are aimed to improve the quality of the Linux kernel. Test workloads and test tools have been developed as open source to be used by developers as they make modifications to the kernel. These tools and tests have also found their way to other Open Source projects. Some of OSDL's projects and tools are noted below.

Database Workloads

Database workloads that simulate real world database activities were developed at OSDL and provided to anyone as open source tests. Three different workloads are available. One simulates the activities of web based users browsing and buying items from an on-line book store. The second provides a transaction intensive environment that mimics a business model of a wholesale parts supplier. The third simulates a decision support workload consisting of a database of business data from many suppliers. Together, these three workloads place considerable stress on the entire Linux kernel as well as specific I/O, virtual

memory, and CPU functions. The workloads are used within OSDL to collect performance data for comparisons between a current release of the Linux kernel and a previous release. They are also used to test new feature improvements to determine if significant performance gains can be shown while utilizing the new feature.

Others within the open source community have used these workloads to test their implementation of open source database products in order to improve the operational capabilities of those products. The workloads are also used to characterize and improve the performance of these databases. The activities performed by these workloads are characteristic of the operations databases must perform in the data center so the open source database communities are grateful to receive a set of open source tools that can be customized to exercise their specific databases.

An overview of the database workload characteristics and how they exercise the kernel is shown in table 1. These represent OSDL's implementation of the workloads. Other implementations have different characteristics (database size, amount of I/O and CPU utilization, etc.).

Workload Name	Application Environment	Tiers	Variants Tested	Database Size	Disk Read Activity	Disk Write Activity	Memory Activity	CPU Activity
DBT-1	E-Commerce	1,2,3	10K users 1-tier	5 GB	Light	Heavy at intervals for data disks; Medium for single log device	Intense 1GB; no paging or swapping	Near 100%
			100K users 3-tier	60 GB	Light	Heavy at intervals for data disks; Medium for single log device	Intense 2GB; no paging or swapping	Near 100%
DBT-2	Transaction Processing	1	Cached 1-tier	11 GB	Light	Light except for single log device which is very heavy	Intense 4GB; no paging or swapping	Med. – High based on I/O bandwidth
			Non-Cached 1-tier	11 GB	Heavy	Heavy at intervals for data disks; heavy for log device	Intense 4GB; no paging or swapping	Low – High based on I/O bandwidth
DBT-3	Decision Support	1	Scale Factor 10	16 GB	Light – Heavy	Light to none	Intense 4GB; no paging or swapping	Varies throughout run

Table 1: Database Characteristics for OSDL Database Workloads

In an effort to improve the stability and performance of the Linux kernel, OSDL developed a build and test management framework to facilitate easy, intuitive kernel testing. The framework is composed of two integral components: the Patch Lifecycle Manager and the Scalable Test Platform. These tools relieve the burden of maintaining stable configurations for repeatable testing and make it easy to build a custom kernel

and run various tests against a range of hardware. As open source projects, these tools are available to anyone, free of charge.

Patch Lifecycle Manager

The Patch Lifecycle Manager (PLM) is a software product developed and maintained by OSDL that allows a developer validate that a patch compiles on several configurations. A developer can apply a single patch to a previous kernel build whether it be a familiar baseline kernel with no previous patches or a kernel with many experimental patches already applied. The PLM provides a configuration management system that maintains a unique identifier for each kernel configuration built. All patches created through the PLM are archived in a database. A patch can be applied either through an email gateway or via a web browser. PLM client systems are available to perform the patch applications and the build of kernel configurations. An overview of the PLM infrastructure is shown in figure 1.

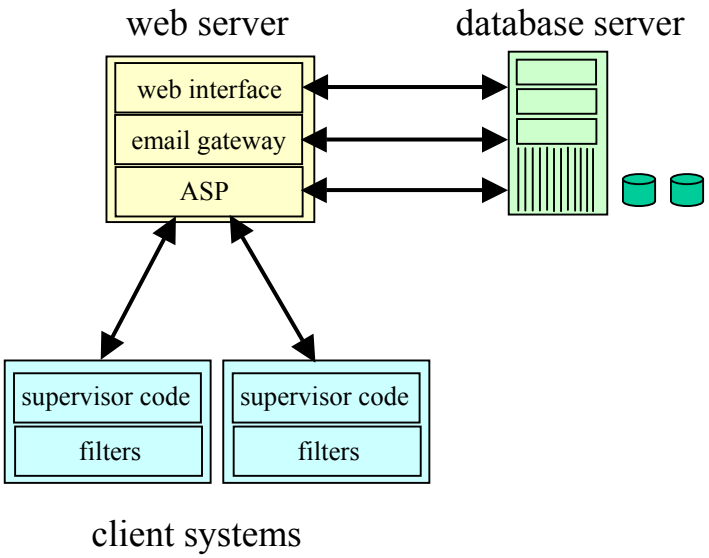


Figure 1: PLM infrastructure

The web server controls three key components of the Patch Lifecycle Manager: a web interface module, an email gateway, and an application service provider interface. The web interface allows a user to apply a patch to a specified PLM kernel by using a web browser. Also provided through the web interface are search capabilities to examine previous patches. The email gateway is provided to users who either do not have web browser capability or do not desire to use a web browser. This interface allows command line scripts for easy access to the PLM through mail. The ASP is the basic connection between the client systems, where the work is actually performed, and to the database where all results are archived. The client systems at OSDL are a series of servers that can scale horizontally to handle several requests. Each client system contains a supervisor code module that controls the execution of the code under test and applies the filters. A filter is a program that is executed against a set of source files. They can be created to handle any task that does not require a reboot of the client system. Examples include static analysis tools such as lint, branch coverage identification, code complexity analysis, static profilers, etc. The filters used within OSDL at the moment are patch application checkers and compilers with varying kernel

configurations. The supervisor is also responsible for sending information about the outcome of applied filters to the ASP in order to be archived in the database.

Patches can be linked to other existing patches in the PLM. This allows a chaining effect of source dependencies that are tracked by the patch system. A developer can then select the top-most dependent patch and the patch manager automatically builds all of the dependencies in the proper order. Patches may also be downloaded or viewed from the PLM web interface.

The PLM is currently used against Linux kernel versions. Future plans will allow other source projects to be managed such as database applications, operating system utilities, run time libraries, and cluster software. The use of the PLM is strengthened when used in conjunction with OSDL's test framework, the Scalable Test Platform.

Scalable Test Platform

OSDL's Scalable Test Platform provides a test framework in which to exercise a Linux kernel build. The framework includes a variety of test systems ranging from 1 to 8 processors and a set of over 20 system level tests that can be run against a specified kernel. STP is provided to anyone in the Linux community that wants to test changes they may have made to an existing version of the kernel. The framework is also freely available for download if there is a desire to integrate this platform into another test environment.

A web interface allows a tester to select the kernel tagged by the PLM identifier to be tested, the test case to run against the particular kernel build, and a system to run on (1-8 processor systems). Results of the test run are archived in a database so that all historical test information is maintained. A web interface allows search functions to obtain the desired historical test run information. A user's page shows the most recent tests executed by the tester. Upon a test completion, mail is sent to the tester with an overview of the test results.

OSDL has been creating new tests and integrating them into the framework in an effort to exercise the kernel in new ways. With each subsequent release of the kernel, OSDL and others in the Linux community have been running tests on the Scalable Test Platform to gauge the quality of each Linux kernel release. The management of kernel patches and tests against those patches are simplified due to the close connection between the Patch Lifecycle Manager and the Scalable Test Platform.

Test requests are queued in priority by a request broker. Once the test is at the front of the queue, the request broker finds the next available system that meets the tester's hardware configuration requirements. Before loading the kernel patch, the STP engine first cleans the system of all code. STP ensures that a new installation of a system occurs for every test run. With a clean system in place, the STP engine loads a distribution from scratch, loads the kernel patch, unpacks the selected test on the system and executes that test. Test results are collected and a notification is sent to the tester in an email form. A diagram of the flow of execution is shown in figure 2 below.

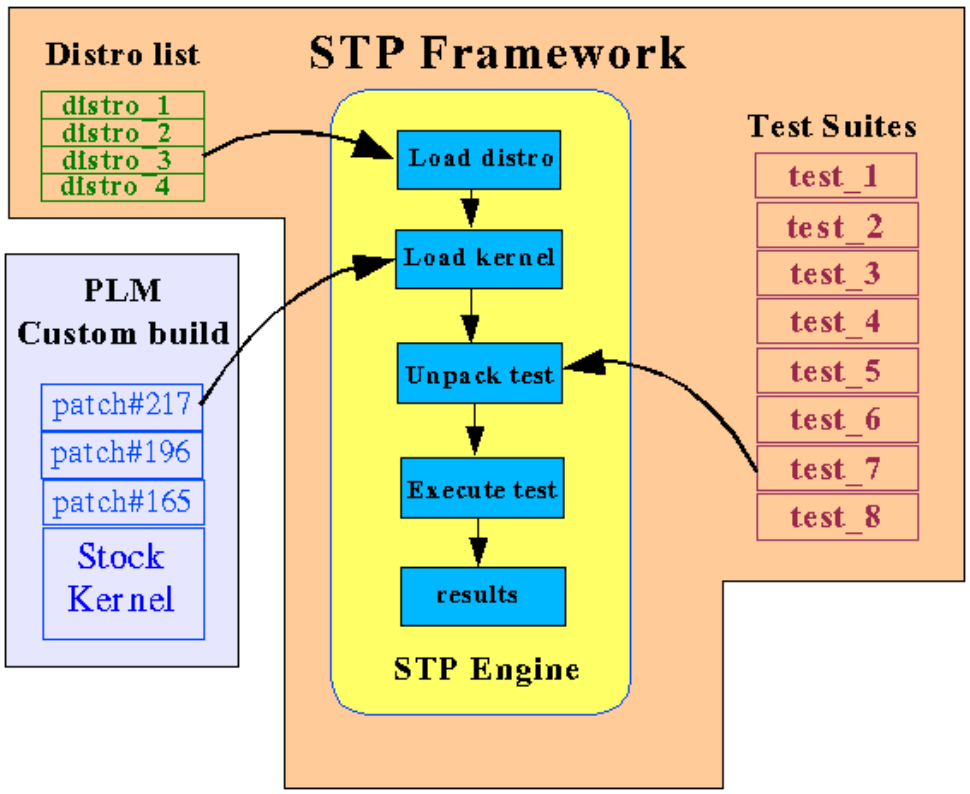


Figure 2: STP Framework

Future enhancements for this framework include the additions of new tests, the ability to run tests against other patched source trees other than the kernel, and better tools to allow a tester to compare current test results with any set of test results from a previous version of a kernel or other application. These improvements are for the benefit of all users of Open Source Software; as more tests are developed and executed against an OSS package, there is a greater potential for developers to find and fix defects.

The Linux Stabilization Project

In addition to the test tools contributed to the community to help improve the test process of Open Source Software, OSDL has initiated a significant project to help stabilize the Linux kernel: the Linux Stabilization Project. Kicked off in December 2002, the goal of this project is to provide tests and test results to the Linux community in an effort to promote faster defect fixes and to accelerate the acceptance of Linux version 2.6[11]. The project is still underway and will continue with other minor releases of the kernel.

OSDL is working with a collaboration of other volunteers to execute tests and collect test results performed at different organizations. All of the test results are collected and retrievable form a single web site. This provides kernel developers with access to these results from a single URL.

The plan is based upon the execution of correctness tests, stability tests, and performance tests. Correctness tests are basically functional tests that assure the system calls and commands operate as specified. Stability tests incorporate stress and reliability tests to determine how well the kernel sustains loads. Performance tests measure the performance characteristics based upon a previous, stable release (e.g., 2.4.21). The tests will be executed on as many different system configurations as possible, thus exposing the kernel to various devices and architectures.

The project utilizes the 2.5 kernel bug tracking database to log defects when found. In addition to the tests outlined above, basic compilation tests are performed on the kernel source using various configuration files. Errors and warnings are posted to the Linux mail group to identify the problems with the current release.

As part of the OSDL charter, lab equipment is offered to contributors who want to conduct activities that will aid in the stabilization of Linux. These activities could range from optimizing kernel components to running additional tests.

The Linux stabilization effort utilizes several available open source test suites and develops new ones as appropriate. Some of the tests available for use are listed below.

- *The Linux Test Project*: a joint project with a good variety of functional tests to validate system calls. http://ltp.sourceforge.net/
- *LSB certification suite*: a functional test suite to show compliance with the Linux Standard Base Specification. http://sourceforge.net/projects/lsb/
- *Open POSIX test suite*: functional test suite to validate POSIX 2001 standards. http://sourceforge.net/projects/posixtest/
- *TAHI Project Test Suites*: a joint project formed with the objective of developing and providing the verification technology for IPv6. http://www.tahi.org/
- *Scalable Test Platform*: a test framework to assist in the testing of a kernel build; it contains a fairly good suite of stress and performance tests. http://www.osdl.org/stp/
- *OSDL Database Test Suite:* A suite of database tests that simulate real-world database workloads: http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/

All of the available tests in the STP are utilized with each new kernel release; tests are run automatically when a new kernel becomes available. The web page for the results are automatically regenerated every hour, so that new test results can be displayed in a timely fashion.

The results from the execution of the above tests are maintained on an OSDL web server that displays the list of current stabilization activities. The page serves a simple anchor point with links to other pages within OSDL or to other sites entirely. Anyone can join and contribute information to the web page by following a simple process. The method used allows contributions by numerous volunteers and still provides quick updates.

The web content files are saved under CVS so that archives of the files can be easily retrieved in the event of an update failure. The script automatically checks out the latest web file templates and builds a web page.

Test results are analyzed and if any defects or anomalies are found, the kernel community is informed through a mail list used by most kernel developers. The most productive postings seem to come from those who provide a comparison to an earlier set of kernel releases and offer an analysis of the discrepancies between test runs. This seems to be highly desired by the kernel developers. Our goal is to provide the same fundamental information. If a problem appears, a developer can use PLM to apply a patch to a specific kernel version and rerun the test.

Challenges Facing Open Source Software

Although good development and test practices evolving, Linux still suffers from other inadequacies. There is a great need for improvement of documentation. When new features are implemented, good technical documentation is rare and is often not created at all. There have been individuals that do indeed document features in the kernel, but they have occurred well after the initial implementation. Usually this is caused by bandwidth issue; the developer of the kernel feature often does not have sufficient cycles to devote to documentation.

In an effort to combat this situation, the Linux Documentation Project was formed[12]. The project accepts contributions from the community and its web site serves as a repository for Linux documentation. Also on the web site are links to lists of items to do, and information about how to contribute to the project. Additional documentation of the Linux kernel and utilities come from the distributions in the form of man pages. The distributors provide their own documentation of the commands and OS interfaces in an effort to distinguish themselves from other distributions.

Another challenge being met by the Linux community is the desire to create a broader spectrum of test cases to exercise the kernel and integrate them into automated test suites. The Linux Test Project contains a framework to run a wide range of functional tests to exercise system calls. The project is beginning to venture into stress tests as well. The Scalable Test Platform contains a set of performance and stress tests, and there is a desire to expand the set of tests run in the framework. Even though test suites are improving, there is still plenty of opportunity for improvement. Many test projects are looking for volunteers to contribute their time to help create new tests or modify existing ones.

For customers trying to use Open Source Software in their IT environments, a problem with Linux is that there are too many releases within a short period of time. Distributors are working to make their releases more acceptable to customer's delivery needs. Longer times between major versions of their releases are helping overcome this problem. Modifications are still supplied in smaller doses via patch mechanisms. This allows customers to perform full upgrades less often, allowing their systems to remain up longer.

Concluding Remarks

The quality practices used on OSS projects depend upon the experience of the people involved and their criteria of "good enough". In the Linux community, several aspects of project development and test methods are employed; the quality practices are directly proportion to the experience of the people that contribute to the project.

Throughout the Linux community, many people and groups have formed to provide greater quality to the Linux kernel. Whether through development efforts, test efforts, or establishment of standard requirements, all of them strive to reach the same goal: to provide the best product possible.

Although this paper focused on the Linux kernel, other widely used OSS products do employ quality practices, but in differing ways. Just as different corporations practice different methods for software development, Open Source projects vary in their software engineering practices. It is difficult to state that processes employed by the Linux developers or other OSS projects are superior to conventional software developments methods. It is safe, however, to conclude that the practices followed by the Linux community work for *that* project. Perhaps a software engineer can study the methods used by the Linux community and determine if their own development processes can be improved by utilizing some of these practices.

References

[1] Netcraft Web Server Survey, May 2003:
http://news.netcraft.com/archives/2003/05/05/may_2003_web_server_survey.html

[2] The Open Source Initiative definition of Open Source Software:
http://www.opensource.org/docs/definition.php

[3] The Open Source Initiative collection of approved open source licenses are found at:
http://www.opensource.org/licenses/index.php

[4] Bugzilla web site, http://www.bugzilla.org

[5] The Open POSIX Test Suite Project, http://sourceforge.net/projects/posixtest/

[6] The Linux Test Project, http://ltp.sourceforge.net/

[7] OSDL Scalable Test Platform, http://www.osdl.org/stp/

[8] Andrew Morton. must-fix, version 6. email sent to Linux kernel community
http://marc.theaimsgroup.com/?l=linux-kernel&m=105433828721449&w=2}

[9] IRC chat sessions are archived at http://lwn.net/Articles/32156/ (May 14, 2003) and
http://lwn.net/Articles/33220/ (May 21, 2003)

[10] The Free Standards Group, http://www.freestandards.org/

[11] OSDL Linux Stabilization Project, http://www.osdl.org/projects/linstab

[12] The Linux Documentation Project web site, http://www.tldp.org/

A Survey of Quality Practices in the Linux Kernel

Craig Thomas, OSDL

21st PNSQC

October, 2003

Objectives (what can I take back?)

- Learn about some quality practices used in the development of the Linux kernel
  - Can you apply some of these to your projects?
- Get a list of FREE tests suites and test tools that may be applicable in your work
- Find out how to get involved in Open Source projects

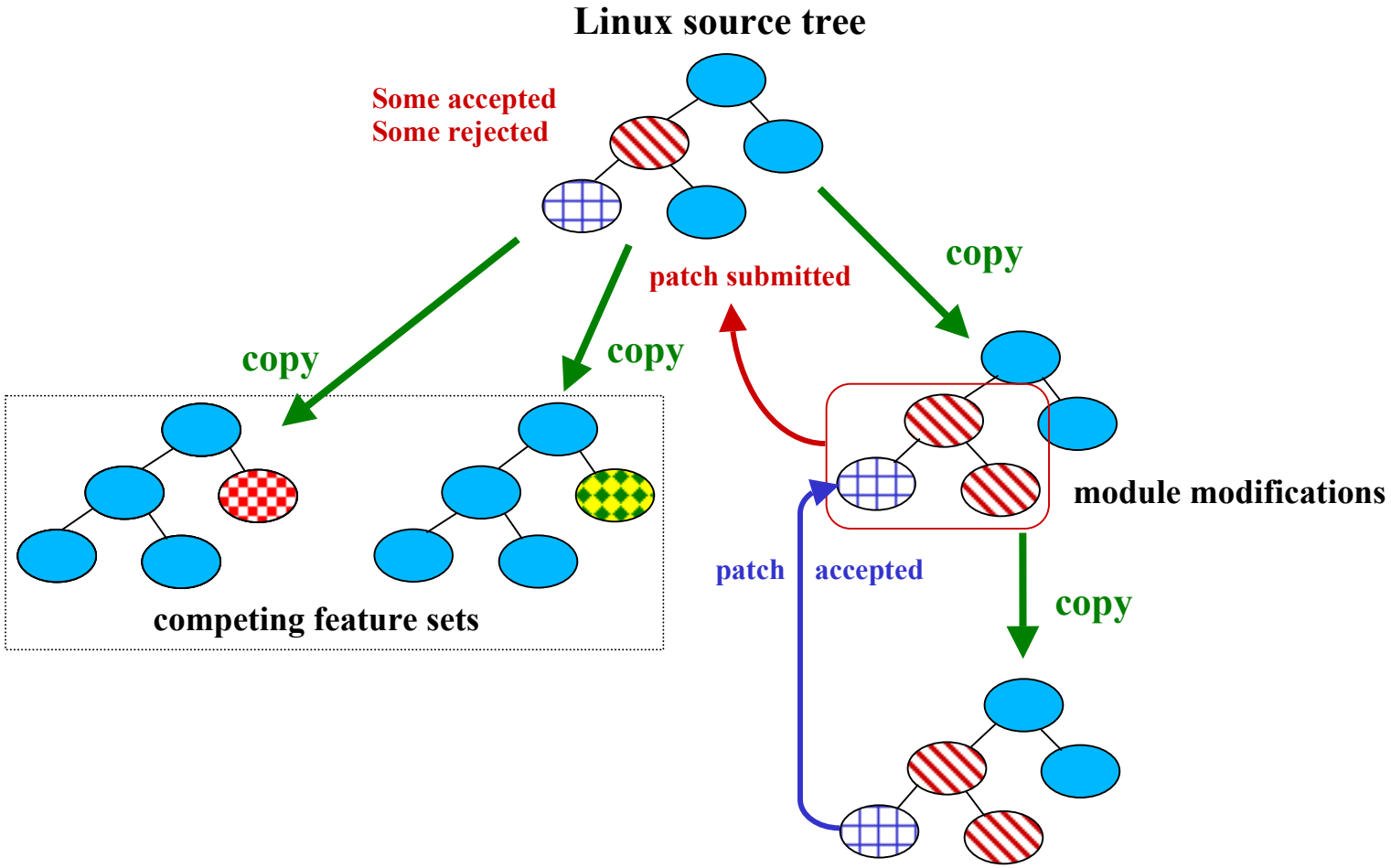
Quality Practices Summary

- Division of labor distributed throughout the world by motivated developers
- Source control is used extensively to create and manage distributed branches of source trees
- One owner of a source tree
  - the source tree is available publicly for download or copy
  - owner decides what changes are incorporated in the source tree
- The code change process allows for “survival of the fittest” patch
- Professional testers forming communities and developing tests and tracking defects

Linux Kernel Development Process

- Development model is highly distributed with no central management of the project
- A single person, known as a maintainer, has write access to a copy of a source tree
  - Linus Torvalds is the maintainer of the officially recognized Linux source tree
  - others can make copies of a tree and then maintain their own tree
- Changes can be proposed by anyone in the world
  - modifications are sent as patches to a kernel discussion mail list and are reviewed by the members of the mail list
- Maintainer accepts or rejects a code modification submitted for their tree

Linux Hierarchy of Source Trees



Linux Kernel Development Process

- Maintainer releases a new version of the source tree with a collection of accepted patches
  - an email notice is sent to the kernel mail list announcing the new release and describing the patches applied since the last release

Linux Stabilization Project

- Project started to test and monitor the progress of quality issues relating to the Linux kernel
- Several contributors from different companies involved with project
- Tests automatically executed and posted to a web page when a new kernel is released
- Links to other test results performed by community testers

- http://www.osdl.org/projects/26lnxstblztn/results/

Linux Stabilization Page

Linux Stability

Goal

The Linux Stabilization project was establilshed to help in the effort of stabilizing linux for a 2.6 release as early as possible. The goal of this effort is to support the stabilizaion of the 2.5 kernel by executing various tests and providing the results at this location. The desire is to make this a collaborative effort within the Open Source community and to evolve this site to support other stabilization results. By providing the results of test activities, it is hoped that they will provide additional confirmation of the kernel's quality state.

STP Test Results of the Latest 2.6 Kernels

Hardware Configuration Information

Patch Name	PLM ID	Applies	ia32 Warnings	ia32 Errors	ia64 Warnings	ia64 Errors	LTP	contest	reaim	lmbench3_long	bash-memory	memtest	iozone	tiobench	dbt1-1tier	dbt2-1tier	dbt3	DOTS
2.6.0-test3-mm2-filemap-1	2058	PASS	1041	85	1988	215	1-way 2-way	1-way 2-way	1-way 2-way 4-way	1-way 2-way	1-way	TBR	2-way	1-way 2-way 4-way	TBR	TBR	TBR	1-way
linux-2.6.0-test3	2049	PASS	997	94	1992	224	1-way 2-way	1-way 2-way	1-way 2-way 4-way 8-way	1-way 2-way	1-way 2-way	1-way 2-way	1-way 2-way	1-way 2-way 4-way	2-way 8-way	4-way	TBR	1-way 2-way 8-way
osdl-2.6.0-test2-3	2028	PASS	0	0	0	0	1-way 2-way	1-way 2-way 4-way	1-way 2-way 4-way 8-way	1-way 2-way	1-way 2-way	1-way 2-way	1-way 2-way 4-way	1-way 2-way 4-way	2-way 4-way 8-way	4-way	4-way	1-way 2-way 4-way 8-way

TBR -- test results not complete and are scheduled to be run
++ A special patch was applied to the kernel for vm_regress test runs to obtain report data

OSDL Production Systems Running 2.6 Kernels

Server Function:	www.osdl.org	Internal Network Server Master	Internal Network Server Slave	PLM Compile Machines
Linux Kernel:	linux-2.5.66 (PLM Id)	linux-2.5.66 (PLM Id)	linux-2.5.66 (PLM Id)	linux-2.5.66 (PLM Id)
Applications:	Apache 2 Bind Sendmail SquirrelMail Mailman Bugzilla	Bind DHCP LPRng	Bind DHCP LPRng	
System Uptime:	86 days - 19:34	20 days - 4:01	83 days - 13:35	50 days - 16:54
Load Average:	0.19, 0.27, 0.20	0.05, 0.06, 0.01	0.00, 0.00, 0.00	0.01, 0.01, 0.00

Tables last updated Fri Aug 15 17:10:15 2003

Document: Done (0.247 secs)

Test Suites

- Linux Test Project
  - functional tests to validate system calls
  - contains simple harness to accept new tests
- Linux Standard Base Certification Suite
  - functional tests to validate LSB specification
  - runs on TET framework
- Database Test Suite
  - fair use implementations of TPC benchmarks
- Scalable Test Platform
  - stress and performance tests
  - web interface allows point and click operations to select test to run

Patch Lifecycle Manager

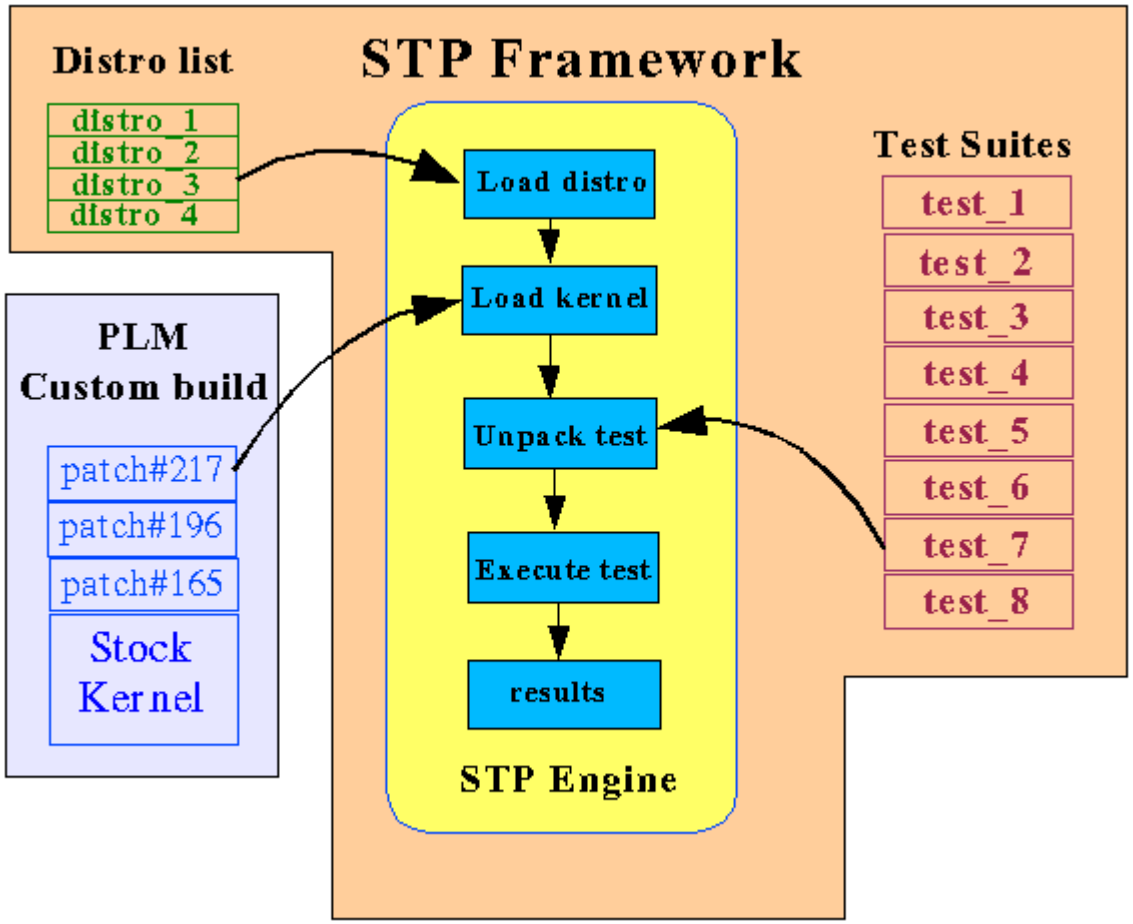
- Storage, test compilation, and management of kernel patches used by the Scalable Test Platform (STP)
- Patches can be applied to any stock kernel release or to kernels containing other patches
- Provides a unique identifier for each kernel configuration (kernel plus applied patches)
- Identifiers and source code to build versions are managed by an internal database

- http://www.osdl.org/cgi-bin/plm/

Scalable Test Platform

- Automatically builds a system from scratch, installs a custom kernel, runs tests against it, and reports the results
- Runs against any Linux kernel configuration managed by PLM
- Run any test from a selected list
  - Database workloads
  - I/O stress
  - Memory stress
  - standard regression tests
- archives test results against a specific kernel
- http://www.osdl.org/stp/

STP and PLM Process



Future work – STP

- Adding more tests to exercise the kernel
- Creating tools to extract performance data from historical runs and compare results
- Improve the framework to allow it to be installed at other sites easily
- Provide analysis tools to map test regressions with change logs of a kernel version to pinpoint the patch that caused the regression
- allow testing of other code sources beyond the kernel (libraries, applications, etc.)

Future work – PLM

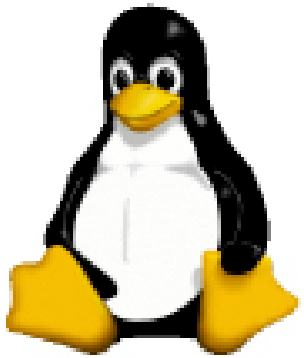
- Provide static analysis tools as filters
- cross compilers
- capability to handle other source trees beyond the kernel

Linux Development Challenges

- Lack of abundant documentation
  - documentation for new capabilities typically lacking software progress
- Broad spectrum of tests are lacking
  - not all system calls are exercised
  - not all code covered
  - new features recently implemented don't have good performance tests
- Releases are too frequent for business users
  - distributors release too frequently

Parting Shots...

- The development process for Linux kernel development works for *this* community
  - other OSS projects have different processes that effect their quality
- The software development practices are constantly evolving to provide better quality
- More test professionals are needed to stabilize Linux
- Get Involved!



OSDL

Roadmap to Successful Outsourcing

Wolfgang B. Strigel
Software Productivity Center Inc.
strigel@spc.ca

Abstract

Software outsourcing is becoming a concept that soon will be integral part of most software development organizations. Understanding the dynamics, challenges and risks is becoming an important complement to running internal projects. This paper discusses these challenges and related success criteria with particular focus on adjusting the development process.

The paper is based on the author's exposure to the international software engineering community as a member of the IEEE Software editorial board and from frequent experience exchanges with leading experts in software engineering. It also draws on practical experience at the Software Productivity Center Inc. (SPC) in outsourcing product development to India, managing outsourced projects and developing process material and courses on the topic. Although many of the lessons learned relate to offshore outsourcing, most concepts are also applicable for subcontracting development to local suppliers.

About the Author

Mr. Strigel has over 25 years of experience in software development. He is the founder and President of the Software Productivity Centre (www.spc.ca) and of QA Labs (www.qalabs.com). Mr. Strigel received a B.Sc. and M.Sc. in Computer Science and an MBA. He is an Associated Editor and member of the IEEE Editorial Board for the Software Magazine.

ROADMAP TO SUCCESSFUL OUTSOURCING

Wolfgang B. Strigel

INTRODUCTION

Outsourcing parts of software development and maintenance activities is becoming a competitive imperative. According to research by the Gartner Group, 75% of all Information Technology (IT) companies will outsource parts of their IT efforts by 2003[1]. It is becoming difficult for companies to ignore this trend and stay competitive. In total, 36% of all US companies will have a "virtual team" business model and there will be 140 million virtual team members. India, the leading outsourcing country for software, has now 42 companies assessed at SEI CMM level 5 and serves over 50% of the Fortune 500 companies. Since 1999 its software exports grew annually by 46% and in 2008 there will be 17 million people available to serve the IT industry[2].

Reasons for these trends include:

- Outsourcing to offshore suppliers offers a 40% - 60% cost savings. Companies need to take advantage of global resources to cut costs.
- Companies are increasingly able to use their core staff for strategic work while contracting out other activities. This improves company focus and increases margins.
- Shortening delivery cycles and quick reaction to demand fluctuations require greater flexibility to ramp up or down.
- Increasing complexity and specialization of new technologies make it difficult for companies to have in-house expertise in all areas.

The outsourcing model is not new. General service and manufacturing industries that need software components, but don't produce software as their core business, have done it for many years. Companies that develop software products as their principal business used to see their permanent development staff as a critical company asset. But this perception is shifting and the asset is becoming limited to fewer key individuals who are the visionaries, architects, and domain knowledge experts. Activities further down the development life cycle become commoditised.

What is the relevance of these trends to the community of software engineering professionals? Over the coming decade we will see a shift in demand for skill sets in the North American software industry. The availability of jobs will shift up the food chain from developers to designers and project managers. North American IT staff and development departments need to be prepared for this shift and acquire the necessary skills. Companies will derive their competitive capability from a successful use of the outsourcing model.

This paper describes those key practices that are of particular importance for outsourced projects. The target audience includes students, interested to focus their skills on areas of future demand, software engineering professionals and managers.

BACKGROUND

This paper is based on the author's exposure to the international software engineering community as a member of the IEEE Software editorial board and from frequent experience exchanges with leading experts in software engineering. It also draws on practical experience at the Software Productivity Center Inc. (SPC) in outsourcing product development to India, managing outsourced projects and developing process material and courses on the topic. Although many of the lessons learned relate to offshore outsourcing, most concepts are also applicable for subcontracting development to local suppliers. The paper also draws on material developed by Karl Wiegers for a subcontract management process.

INGREDIENTS FOR SUCCESS

With the pressure to convert in-house development departments into smaller units to leverage on-demand use of global, virtual teams, a new set of business practices is needed to cope. Some of the traditional business practices won't work with this new paradigm. New development processes must be adopted or the transition to a distributed model will fail. Just like any other paradigm shift, this transition is fraught with risks. Companies are in a dilemma: the decision not to become an early adopter will put them at a disadvantage but jumping on the bandwagon of virtual teams may temporarily increase their project risk.

Outsourcing is not a silver bullet. In fact, it can backfire if it is not well prepared and carefully executed. Any software project, even when executed in-house, is a challenge and these challenges are compounded when external contractors are used. The application of industry best practices as embodied in process frameworks like the Capability Maturity Model (CMM) is generally a good start to mitigate against outsourcing risks. For example, CMM level 2 includes a key practice area for subcontract management. The following sections describe additional considerations for outsourcing projects.

Choosing the Right Partner

Choosing an outsource contractor is the equivalent of hiring a whole new department of developers. But it goes beyond just finding the right skills and knowledge.

Contractual terms must ensure the use of a dedicated team. The client needs to prevent the supplier from switching resources to react to their own internal needs. In many offshore situations the supplier can offer a "shadow team" at no additional cost. This will offer added protection against staff fluctuations or the inevitable turnover of key resources.

Apart from technical qualifications, the supplier should have a well-defined development process. With the proliferation of outsource suppliers, many companies have sprung up to ride the opportunity wave without having the infrastructure and processes for predictable quality work. For this reason, and to avoid the "low cost equals low quality" perception, many offshore companies, especially in India, have become obsessed with quality certifications such as ISO or SEI-CMM[3] accreditation. In fact, India has now more CMM level 5 certified companies (the highest level on the CMM maturity scale) than all other countries combined.

Understanding cultural differences between customer and offshore supplier is essential to avoid surprises. For cultural reasons, contractors in some parts of the world may be too agreeable and eager to please rather than asking for help or communicating uncertainty. This desire to "save face" can lead to an accumulation of unresolved issues or the implementation of misinterpreted specifications unless the client is adequately prepared to deal with this cultural difference.

To evaluate proposals, it is advisable to rate bids with a proposal evaluation matrix. Depending on the client's needs and project priorities, each evaluation criterion can be assigned a relative weight. Some useful evaluation criteria include:

- Software development process
- Quality and CM processes
- Quality on previous projects
- History of meeting commitments
- Previous customer satisfaction
- Staffing and skills
- Application domain experience
- Technology experience
- Facilities and resources
- Development cost
- Delivery schedule.

Choosing the Right Project Type

Experience has shown that the risk of outsourced projects increases with the degree of innovation required. Projects for the development of brand new products often require frequent changes along the way. The specification is typically not very stable and as the application is built, new insights can result in frequent changes. Outsource suppliers don't focus on creativity or innovation as their core competency. They are more like software factories that can produce quality products based on detailed blueprints. They would rather not deviate from the specification and focus on diligent implementation of detailed directions in a cost-effective and high quality execution. The innovative capability should remain the core asset of the client's own development team.

The less interaction required, the higher the chances of a successfully outsourced project. User Interface design (or screen design) is a highly iterative process with many prototypes before getting it right. This is not the kind of activity a client company wants to conduct across the ocean. In this particular case the client should also know that esthetics and cognitive responses differ greatly between cultures. A screen designed in Asia may not be appealing to a western audience.

An evaluation matrix to decide on the suitability of a given project for outsourcing should weigh considerations, including:

- Need for innovation is limited
- Close collaboration is not essential

- The product does not involve critical or strategic code
- The need for specialized domain knowledge is limited
- The project has minimal dependencies on other projects
- The underlying hardware and software platforms are likely to remain stable
- Requirements, performance goals and acceptance criteria can be clearly defined
- Internal management and domain expertise are available to support the supplier.

If a project fails to meet one or several of the above considerations, special care should be taken and the risks should be documented and carefully managed.

Choosing the Right Project Phase

As a general guideline, the earlier phases of projects are less suited to outsourcing than the later ones. This is related to the previous point on innovation. Most projects start with a definition phase during which requirements and other critical specifications are defined. This requires domain expertise and creativity and is best done by the client's staff. As the project moves into design it is feasible to consider outsourcing. The situation depends on the criticality of the design. If the application is to integrate tightly with other systems or if the design is under other critical constraints, it is better to keep that in house. If it does not matter as much how the system is designed, as long as it meets the specification criteria, design can be outsourced. Later phases are generally good outsourcing candidates. This includes coding, testing and maintenance. By fortunate coincidence, these latter phases are also the most labor intensive and that is where most of the savings can materialize.

OUTSOURCING DEVELOPMENT PROCESS

It makes sense to choose a supplier with a well-defined and proven development process. As mentioned earlier, industry certifications such as ISO or CMM give some assurance that a defined process exists. It is less obvious that project success also depends, to a large extent, on the maturity of the development process at the client organization. As long as the client company is involved in managing and tracking the outsourced project, its success potential is similar to the success potential of in-house projects. However, the impact of the in-house process becomes amplified in an outsource project. While clients can compensate for loosely defined processes internally by increased interaction between all project participants, this is not possible when part of the project is run on the other side of the globe. It certainly is a fallacy to think that "throwing a project over the fence" and letting the outsourcer compensate for the client's lack of defined procedures will be the solution.

The degree to which processes need to be defined depends on the size of the development team and the geographic separation of teams. It also depends on how critical the application is to the client company. The need for a defined process can be shown on a continuum as illustrated in figure 1.

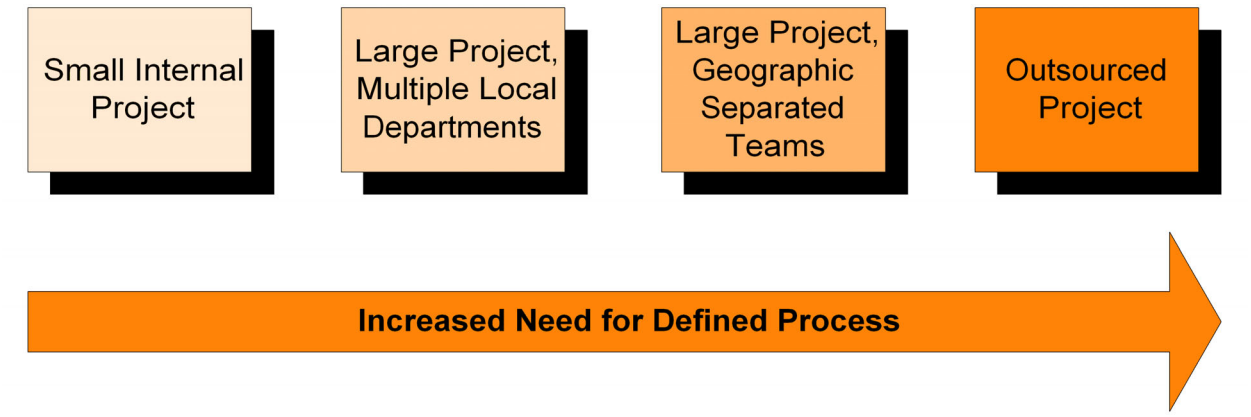


Figure 1

A well-defined process for planning, synchronizing, and monitoring activities of globally distributed teams is essential for success. Incremental lifecycles are useful for reducing the time between milestones. The desire for frequent checkpoints needs to be balanced against the overhead involved for generating and testing each build. The required process infrastructure generally consists of a set of methods and supporting tools that are described in this section.

Contractual Considerations

The first step for a new outsourcing situation is to find the right supplier. We are all bombarded with email messages from offshore suppliers offering their development services. Asking other companies in a similar application domain about their recommendations for offshore suppliers is a good way to get started. Not all outsourcing needs to be offshore. US based companies may contract Canadian companies and get some economies from the currency differential between the two countries. The proximity, close time zone, cultural and legal similarities also reduce the risk of this option. When going offshore, considerations including price, cultural differences, legal system and reputation for protection of intellectual property come into play. Suppliers with formal accreditation such as ISO or CMM may be preferred, but it should be noted that even a CMM level 5 does not guarantee that all projects will be executed according to these practices. The chosen supplier must also be familiar with the tools that are used to communicate between the two companies. These include tools for requirements management, change management, project management, defect tracking etc. Some of the less obvious considerations should include a guarantee of staff continuity, proficiency in the English language, availability of high speed internet connection and single or multiple power backup systems, security and political stability.

The key document for a contractual agreement will be the statement of work (SOW) that defines commercial terms and usually refers to other documents such as the requirements specification, project plan, acceptance criteria, etc. It is very important to define in detail a communication protocol. This includes frequency and format of status reports, means of information exchange and conflict management and resolution mechanisms. Scope control and procedures to deal with change orders need to be carefully defined. The more of these critical issues that can be defined and mutually agreed up-front, the lower the chances for unpleasant surprises.

A frequently used risk mitigation technique is to start the contractual relationship with a short pilot project that is representative of future work but does not create major problems if it gets

delayed or even if it fails. There is always pressure to proceed with the most important projects at hand, but if those were to fail the implications could be irreparable.

Define mutually accepted performance metrics. This may be problematic if the client company is not used to tracking its own performance and has no historical data. The definition of performance metrics such as lines of code per person day, number of defects per 1,000 lines of code, code complexity (which has an impact on maintainability), are preferred over negotiating a fixed-price contract. Except for very simple situations, it is difficult for both parties to agree on a fixed price that does not penalize either party. Many outsource projects have failed when prices have been fixed at unrealistically low levels since the contractor has been forced into cutting corners.

Communications Management

Maintaining effective communication is probably the most critical aspect for the success of an outsourced project. References to communication techniques are therefore included throughout this paper. Software projects are people intensive and their success depends on the effective interaction of team members. Many tools have been developed to support the management and communication of information. Outsourcing projects spread the spectrum from contracting the whole project and all of its activities to subcontracting a small part of a larger project. The criticality of communication between off-shore supplier and client depends on where the particular project is located on this spectrum. In the former case, it may only require the project managers or contract managers on each end to communicate. But this is a rare situation. More typically, team members on both sides collaborate on the same project. Portions of a project may be parceled out with a clean definition of interface points. In these cases, multiple team members on each side need to communicate. This requires a detailed definition of roles and responsibilities, means and frequency of communication and means to resolve issues. The physical separation may not appear much more challenging than in the case of a multi-site development team across North America, but given the added challenge of time zone separation, cultural differences and differences in corporate practices, the situation can be complex.

Specifications

Specifications are the essential reference document for an outsourcing contract. Depending on what phase of the project is outsourced this may be a requirements specification or a design specification. In-house projects have some degree of tolerance for incomplete specifications. Questions can be quite easily answered by seeing somebody “down the hall” or calling a quick meeting to resolve the issue. Furthermore, misinterpretation of a specification can be caught more easily by convening periodic reviews.

There are several reasons why specifications need to be considerably more detailed, especially in a new outsourcing situation. Unfamiliarity with the application means that details that may appear to be obvious to internal staff need to be spelled out. Differences in language and culture can lead to unexpected interpretation. In our experience, clarifications by phone or email are sometimes difficult and may require several cycles before a single issue is resolved. This can be very time consuming and can even stall a project if too many issues are pending.

A side effect from the need of clarifying a large amount of specification issues is that the contractor may argue he is losing planned project time, leading to schedule slips and cost overruns. Certain clarifications may also be interpreted as scope changes with the resulting cost and schedule impact. When estimating an outsourced project, the effort allocated to specifications should be significantly higher than for in-house projects.

In many cases it is advisable to divide the project into several contractually defined phases. For example, the contractor may be assigned to develop design specifications from a requirements specification and the first project phase ends at the completion of an in-depth review of the design specification. It is recommended to include the definition and agreement on acceptance tests and acceptance criteria in this phase. This will form a solid foundation for the subsequent project phases. It will also give both parties a more realistic understanding of the remaining effort.

Given the importance of requirements specifications and the need for an unambiguous understanding, the use of a requirements management tool is advisable. Such a tool acts as a facilitator for long distance, asynchronous communication on complex issues. It ensures that both parties always refer to the same document, and that changes are tracked and linked to acceptance test cases.

Project Management

Good project tracking and oversight is essential for the success of an outsourced project. Some contractors may try to resist frequent and detailed status reporting. But the client project manager will not have the opportunity to feel the pulse of the project by talking to team members on a frequent basis. He will be isolated from the team and will only see progress through his communication with team leads or project managers at the supplier end. This lack of human contact must be compensated by detailed and frequent status reports. However, even these reports are not sufficient to really understand what is happening. In addition, these reports must be backed up by incremental deliverables that support the status reports.

Meeting commitments such as milestones is equally important for the supplier as it is for the client. The client's project manager must make sure that his team lives up to agreed expectations in response time. For example, if the client is late in providing feedback at review milestones, the supplier will rightly ask for schedule relief. In our experience this can stress the relationship and careful attention to realistic planning of turnaround times must be given in the project plan.

Project management also relies on the effectiveness of human interaction. The best communication tools cannot replace the face to face contact. It is advisable to plan several trips to the supplier site. This tends to be more informative than visits from key staff from the supplier. These trips can be costly but certainly pay off. We were involved in projects that ran purely on the basis of long distance communication and others where periodic trips to the supplier were scheduled. The latter proved to be very successful, not only to ensure that the project is on track but also to establish a personal relationship that makes subsequent long distance communication more effective.

An important part of project management, especially for outsourcing situations, is managing the risk. Risk management should be seen as an early warning system to catch problems early on before they become major issues that could derail the project. Project risks need to be identified

and monitored to ensure that both the supplier's and the client's risk mitigation actions are performed. Staff responsible to manage risk at both sites needs to be identified. A list of the top ten risks should be created at the start of the project and periodically reviewed and updated as necessary.

Outsourcing related risks may include:

- Change of project staff
- Delays in delivery schedule
- Poor quality of deliverables
- Missed status reports or slow response to queries
- Scope creep
- Finger pointing
- Supplier does not follow agreed upon process.

Lower level risks are similar to typical risks for any project and include items like inability to meet certain specifications, increasing defect density, poor coding practices, lack of documentation, etc.

There are many tools that can support the job of a project manager for a long distance relationship. The first one is a web enabled project management tool. Whether the project uses simple Gantt chart tracking or earned value techniques, the tool should allow shared access by both parties. Next on the list is an issue tracking tool. In contractual relationships, the management of issues can become an important device to avoid or to resolve conflict. Tracking issues on the basis of email exchange or swapping spreadsheets is normally insufficient.

At the end of any outsourced project, whether it succeeded or failed, a project review should be conducted. Lessons learned should be exchanged and suggestions for improvements should be documented for the next project. If you plan to use this supplier again, key staff from the contractor should be included in this project retrospective.

Change Management

Almost all projects undergo frequent change. Here again, the physical separation of client and supplier makes a tool based solution the preferred approach. The ideal tool is a configuration management (CM) tool that has change tracking capability. If all aspects of the project are outsourced, the synchronization of artifacts is less critical. But if parts of the project are handled on both ends or if for example testing is performed by a third party, the use of an industrial strength CM tool is required. A formal change control board (CCB) should be established to deal with changes in the specifications. Members of the CCB must be identified at both ends.

It is good project management practice to allocate a contingency fund to accommodate changes. This budget has to be determined depending on the complexity of the project and the familiarity with the supplier. The contingency budget can be used to accommodate a certain amount of changes without revisiting the contract. The supplier is also likely to have a contingency allowance built into the contract price.

Acceptance

As previously mentioned, an iterative project life cycle allows for incremental acceptance of deliverables, which can offer early insight into a number of potential outsourcing related problems. It is a good idea to partition the project into components and complete a full development cycle on the first component as early as possible. This will show how the contractor approaches the full development lifecycle and the quality of each lifecycle deliverable. It also serves as a test run for both parties and an opportunity to adjust processes on both ends for the remainder of the project. In many cases, the first contract is a small pilot project used to evaluate the relationship.

An acceptance process should be defined right at the beginning of the project and is typically included in the statement of work. The process should be followed for all deliverables. It should define roles and responsibilities and a time limit for each review cycle. The time limit is important to set expectations on both sides. It also allows the contractor to make realistic schedules and avoid disputes over project delays caused by acceptance review cycles.

Acceptance criteria must be defined in detail so that both parties know what is expected. Defects should be categorized (e.g. high, medium, low) and each category must be defined. Acceptable and unacceptable levels of defects can then be mutually agreed. The acceptance test plan should be based on the requirements specification. A requirements traceability matrix offers useful guidance to define the acceptance test plan.

In some of our outsourced projects we used an independent third party testing company to perform quality assurance on all interim and final deliverables. This approach offers an objective perspective on the quality of deliverables and reduces the potential for conflict between the client and the contractor.

CONCLUSION

Outsourcing is becoming a concept that soon will be an integral part of most software development organizations. Understanding the dynamics, challenges and risks is becoming an important complement to running internal projects. The recommended process for conducting outsourced projects is not much different from generally accepted best practices for software development. But certain parts of the process need much more attention than with traditional in-house projects.

Because of the economic advantages that can be gained from outsourcing, the first outsourced project is probably not the last. It should be used to establish a good working relationship and to fine tune the processes for a lasting long term relationship. Project plans should include an allocation for conducting a project retrospective to review lessons learned and to get prepared for the next project.

---

[1] Gartner Group Reports (2001)

[2] Information gathered from NASSCOM, www.nasscom.org

[3] The Capability Maturity Model (CMM) is a quality framework developed by the Software Engineering Institute (SEI). Its 5 levels specify the degree of process maturity associated with a software organization.

ABSTRACT

Eight years ago, Flight Dynamics (FD) developed software without Software Configuration Management (SCM), Software Quality Assurance, (SQA), or repeatable processes. The company's continued existence was heavily dependent on extremely talented engineers who consistently rescued the company from missed delivery dates.Today FD is the world leader in Head-Up Displays (HUDs) and continues to defeat the competition because of its edge in superior software development and product quality.

This presentation will cover:

- What was the stimulus for change?
- How did the change occur?
- What were the first steps?
- What were the barriers to improvement?
- What were the gateways for improvement?
- How did management react?
- What benefits have we seen?

About the authors

Kathleen Vasconcellos joined Flight Dynamics in 1998. She was hired as a Software Quality Assurance Engineer, but quickly grew into other areas including AS9100 Lead auditor, part-time Software Development Engineer and LEAN champion.

Bryan Stickel started at Flight dynamics as a Software Engineer in 1995. He worked on several programs before moving into a Software Quality Assurance position. Since then he has: initiated the Engineering Process Improvement Council (EPIC), created and continues to manage a metrics program, and is a company Designated Engineering Representative with the FAA.

Devin Stinger began his career with Flight Dynamics in 1995 as the first Software Configuration Manager. In his early years he created a System Tracking Report database which the company continues to use. Lately, Devin has become the company expert in database development.

Loring Mercil joined the team in 1999 as a second Software Configuration Manager. For the last few years he has been responsible for SCM and improving the System Tracking Report database.

PerlF, Regr, and other Cheap Testing Tools

Richard Vireday, Richard.Vireday@intel.com
Neel Patel, neel.a.patel@intel.com
Arunarasu Somasundaram, arunarasu.somasundaram@intel.com
Ajay Dave, adave@mit.edu

Abstract

Reduced budgets for tools and equipment often settle the buy-vs.-build decision for us by default. However, the products we make and test require extensive system-level testing. This includes serious regression testing of multiple tools and configurations, in addition to custom testing of unique configurations of hardware and software.

To work around these problems we have done what many others have in the past: produced our own tools. Our "cheap" tools have surprised us sometimes with results beyond their original expectations. We have group knowledge of the best breeds of tools, and try to apply that knowledge to produce our own solutions.

This paper also follows up on a Year 2000 PNSQC paper, where one of us (Vireday) presented the Regr module. Now we are able to make it fully available under Open Source licensing.

Author Backgrounds

The authors are members of an Intel® development team, provide customized workstations and servers based on pre-production silicon and software.

Richard Vireday is a Sr. Software Engineer and Project Manager for the Intel Development Platform Services Engineering (DPS-Engineering). A member of ACM and IEEE, he is a 1986 M.S. graduate of the Oregon Graduate Institute in Computer Science and Engineering.

Neel Patel is a Software Engineer in DPS-Engineering working on development, testing and integration of software and hardware platforms. He is a 2002 graduate of the University of Virginia in Electrical Engineering.

Arunarasu (Arun) Somasundaram is a Systems Administrator and Project Lead. He is a postgraduate from India. He works on testing and integration of software and hardware platforms, manages the project lab servers, has built many computer centers, and acts as a jack-of-all-trades for the team.

Ajay Dave served as a 2003 summer intern on the team. He is a full-time student at MIT, studying for the Bachelor of Science in Electrical Engineering and Computer Science, and expects to graduate in May 2006.

PerlF, Regr, and other Cheap Testing Tools

Richard Vireday, Richard.Vireday@intel.com
Neel Patel, neel.a.patel@intel.com
Arunarasu Somasundaram, arunarasu.somasundaram@intel.com
Ajay Dave, adave@mit.edu

1 The Problem

We integrate and test customized workstations of pre-release hardware and software. This often requires system testing that is fairly complex in nature. As an example, pre-release versions of the Intel® Threading Tools required system tests using multiple configurations of computer systems, peripherals, compilers, and different beta versions of the Intel VTune™ Performance Analyzer.

System testing technology is often not much different from the regression testing phases of any software product. Because of the number of different software packages we are testing, however, we often do not do more than basic functionality (smoke) and only a few in-depth tests. There are unit tests for the pieces of software we develop, but the overwhelming majority of our tests are functional, or system integration tests.

For the automated portions of our testing, we rely heavily on system imaging to quickly reload and test various combinations of software packages, operating systems, and hardware peripheral configurations. Much of the actual automated testing is performed via scripting. Rather than using commercial scripting tools that we know and have used before, we rely on command scripts and Perl, but with a twist.

One critical requirement includes doing the system testing in circumstances as similar to those of the user experience as possible. This means we must take great care to not unduly disturb the configurations of the system under test.

Primarily for this reason we developed PerlF, a pre-compiled version of Perl that can be injected into a test system with **no setup**. It contains the necessary Perl modules for our test use in one single executable. Developed with the ActiveState*[1] PerlApp program, this represents an interesting twist on regression testing tools, and we are finding other possibilities beyond just testing.

To capture the results of the automated tests, we have used various methods, with increasingly better results over time. We have been migrating to using XML output built with the Regr Interface. Regr is a Perl module that was presented several years ago in a PNSQC session. Here we will show the evolution of the Regr module, and how it is used in conjunction with PerlF. We have plans to enhance and expand Regr, especially in putting the results of multiple tests into common databases for easy display of status and comparison of results.

Finally, the configuration from a regression test run is currently specified in Excel*, but is output into text files for the test system to use. We will show how the output data is merged back from XML to Excel* for analysis.

As a side note, we also extensively use such third party tools as Ghost*, partimage* and, most recently, VMWare*. We can spend money judiciously on certain tools necessary for our business, and find that the technologies these tools represent are fantastic bargains at their current prices.

[1] *Other names and brands may be claimed as the property of others.

1.1 Summary of Home-built Test Tools

PerlF executable	Test driver, complete Perl sub-system	
Tk::	Used to provide user interfaces	*built-in to PerlF*
Win32::GuiTest	Used to drive keystrokes	*built-in to PerlF*
XML::Parser	Available, used in xml2Excel* converter	*built-in to PerlF*
Other Perl Modules		*built-in to PerlF*
Regr.pm (perl module)	Used to record test results to XML file.	*In test suite tree*
QA.pm (perl module)	Common test suite functions for specific tests. Relies on Regr and Perl being there. Template APIs used by Excel* generated scripts.	*In test suite tree*
regrdiff executable	Standardized "diff" executable	*In test suite tree*
Excel* Spreadsheets	Holds descriptions of tests used to generate	*In test suite tree*
Xml2Excel* converter	Takes information from XML output and merges in back into original Excel* spreadsheet	*In test suite tree*

1.2 Other Testing Goals

Beyond just getting PASS/FAIL test results, we have some specific goals that we established and use as criteria to judge our testing and development efforts.

- Increased automation of regression tests
- Automated runs for overnight and long-stride testing
- Low disruption of the system under test
- Quick prototyping and development
- Minimal overhead, so less fear of throwing away and rebuilding tests

1.3 A Test Cycle Overview

The first prerequisite of a selected test system is that the OS needs to be up and running, ideally with networking configured and connected to the test network. If the system is isolated, then we must burn a CD or somehow get the test suite copied onto the system.

In some cases, the Test System will be available as a Ghost* Image. In that case, restoring the image will make the system ready for testing very quickly.

In most cases, the Intel® Tools we wish to test, such as Intel® MKL, Intel® IPP, Intel® VTune™ Performance Analyzer & Intel® Threading Tools, were preinstalled for those systems. Then, either by automated or manual installation, other tools and software may need to be installed in a specific order. Naturally, the order can change depending on different problems and interactions of software and hardware.

1.4 A test cycle walkthrough

1. Connect the test system to the Lab Test Network.

2. Log in to the Source Control Server using a Web browser with a Test User and get the latest version of the PDEQA directory to the root of the C: drive.

3. Make sure the whole tree of C:\PDEQA is correct and the latest version by checking version numbers. This is a quick manual test.

4. Write down the name of the test system, processor, mother board, and any other information that has an impact on testing. Use a single word to reference the test system with all the noted values. For example, if the processor is XYZ, the mother board is ABC, and the build image is PQR, then create the test build name as XYZ_ABC_PQR.

5. Make sure all software tools are installed correctly and have valid licenses to run.

6. Clean the results-Excel* file by clearing all PASS/FAIL results.

7. Run the script **Start_Tests.bat.** This actually starts as batch script, but re-invokes itself as a PerlF script.

8. Leave the system for about 2 hours by not disturbing the keyboard or mouse. The test results will be updated to the regr.xml file in the C:\PDEQA directory.

9. Run the xml2xsl script to convert those results and update the Excel* file.

10. Run all manual tests and update the Excel* sheet with results (PASS/FAIL). Updating has to be done on a machine with Excel* installed.

11. Rename the Excel* file with the name of the test build and copy that file to the results directory on the web server.

2 Specific Test Tools

*2.1 Excel**

A large number of test cases are specified in Excel* and can be turned on or off easily. The Perl script that reads the spreadsheet checks whether the first column of a test case contains a number. If it does, Perl performs that test.

Others have used spreadsheets to control test suites, but often those require that the spreadsheet software is installed on the test machine. We do not have that requirement. Once specified and generated, the tests can run on virtually any machine.

2.1.1 Using VBA within Excel*

Recorded macros within Excel* use some simple VBA scripting. It then requires only a few clicks to load the test cases from the Excel* document into the text files that Perl parses to determine which tests to run. By adding this feature we greatly reduce the amount of time needed to update and load our test cases.

At the end, results are reported back from the test cases into the Excel* spreadsheet.

2.1.2 How Test Cases are Written in Excel*

The tests are written in Excel* using a homegrown nomenclature. This allows the Excel*-driven test cases to parse through and perform the tests easily. The Excel* test cases are each described by a handle format, such as the following:

```
Tool_sourcefile_xtoy_IDE_compiler_Debug/Release
```

where each part (except for source file) must be one of the predefined parameters. For example, **IDE** must be either cmd, make, vs6 or vs7. Also, **compiler** must be either Microsoft or Intel.

Each handle then has a description. With the handle and the description, the test is fully described. For example, the following notation describes a test which is used to compile a project or solution file in Visual Studio 7*.

```
Handle
       Tool_sourcefile_xtox_IDE_compiler_Debug/Release

              Tool:         CPP, Fortran, IPP, MKL , VTune
              Sourcefile:   name of source
              Xtox:         32to32, 32to64, 64to64
              IDE:          cmd, make, vs7
              Compiler:     ICC, IFC, MS
              Debug:        Debug or Release
```

```
Compile
       Compile [handle] [source] [output location] (parameters)

Example
Compile CPP_Hello.c_32to32_cmd_ICC_Debug C:\Progra~1\Intel\EDB70\Sampia32\Hello.c
c:\output

       Notes:
              Places handle.exe into [output location]
              Test passes if exe created.
```

```
Execute
       Execute [handle] [exe location]

Example
Execute CPP_Hello.c_32to32_cmd_ICC_Debug c:\output

       Notes:
              Test passes if output of file == gold.
```

```
makeCompile
       Compile [handle] [makefile location] [success file] (parameters)

Example
makeCompile MKL_blas_32to32_make_IFC_Debug C:\Progra~1\Intel\MKL\examples\blas
C:\Progra~1\Intel\MKL\examples\blas\_results\def_intel\zherkx.exe
```

```
Execute
       Execute [handle] [exe location]

Example
Execute MKL_blas_32to32_make_IFC_Debug
C:\Progra~1\Intel\MKL\examples\blas\_results\def_intel\zherkx.exe

Notes:
       Problems verifying Execute success.  Only works with text output.
```

```
vs7
Compile
       Compile [handle] [project/solution file] [success file] (parameters)
```

```
Execute
       Execute [handle] [exe location]

Notes:
       Verifying Execute success only works when executable produces text output.
```

2.2 PerlF

PerlF.exe is a single executable[2]. It contains a version of Perl, the core Perl modules, and additional Perl modules that we have developed. At any time during the test setup it can be injected into a test system with no complicated setup. There is nothing to register, no DLLs to include.

The heart of PerlF is simply the eval() function. External scripts are read in, then dynamically operated on by eval().

Developed with the ActiveState* PerlApp program, PerlF represents an interesting twist on regression testing tools, and we are finding other possibilities beyond just testing. With PerlApp providing Linux, Solaris and HP-UX versions, we can have a common tool that runs on virtually all of our target platforms.

[2] Note: We are not releasing any executables at this time. However, the source code is available and can be used to generate your own versions. See the download section in the Bibliography.

```
#############################################################
#
# PerlF - Perl Full, as a single executable.
#
# (C) Copyright 2002-2003 Intel Corporation.
#
# This software program is licensed subject to the BSD License,
# available at http://www.opensource.org/licenses/bsd-license.html
#
#
# All modules listed here will be included in the final executable

… module definitions

… command line stuff

#
# Process command line, and read each file specified
#
foreach $_toparg (@FARGV)
{
      # print "File: $_toparg\n"; if $opt_verbose;
      if ($_toparg eq '-sargs') {
            last;  # skip rest of arguments
      }
      if (-e $_toparg)
      {
            $oldIRS = $/;
            undef $/;               # Set slurp mode
            $_ = <TO>;
            $/ = $oldIRS;           # Unset slurp, for scripts read in
            print "#[[$_toparg]]\n", $_, "#[[end $_toparg]]\n" if $opt_verbose;

            eval;                   # user script executed

            warn "WARNING perlf SCRIPT ERROR: ".$@ if $@;
      }
      else {
            print "Warning: Unable to execute: $_toparg\n";
      }
}
```

2.2.1 Legal Review

Getting legal approval to release PerlF in some of our products proved to be a very difficult process. The confusion and difficulty came with the use of the GNU Public License (GPL) and Perl Artistic license. There was concern that the GPL would infect the entirety of any distributions in which it might be used.

That fear is not unreasonable, and we took great care to ensure that we were in total compliance. The newness of this situation was of great concern to our legal review teams. The fact is that most of the common Perl modules have no license comments within their source code or documentation. They say that “Either the GPL or Perl Artistic License” could be used. All this can drive lawyers nuts! (And nearly did!)

The use of PerlApp to build the executable also created some of the confusion. However, we were able to show three things:

- First, we learned through the PerlApp documentation that the Perl code we are distributing is not statically linked, but is instead dynamically invoked. See Figure 1 for details. This is a major reason why we do not violate the GPL.

- A second point is that the Perl Artistic License allows the redistribution of all the code.
- Third, we were in no way or fashion modifying **any** of the Perl modules being redistributed. We were just redistributing already available code.

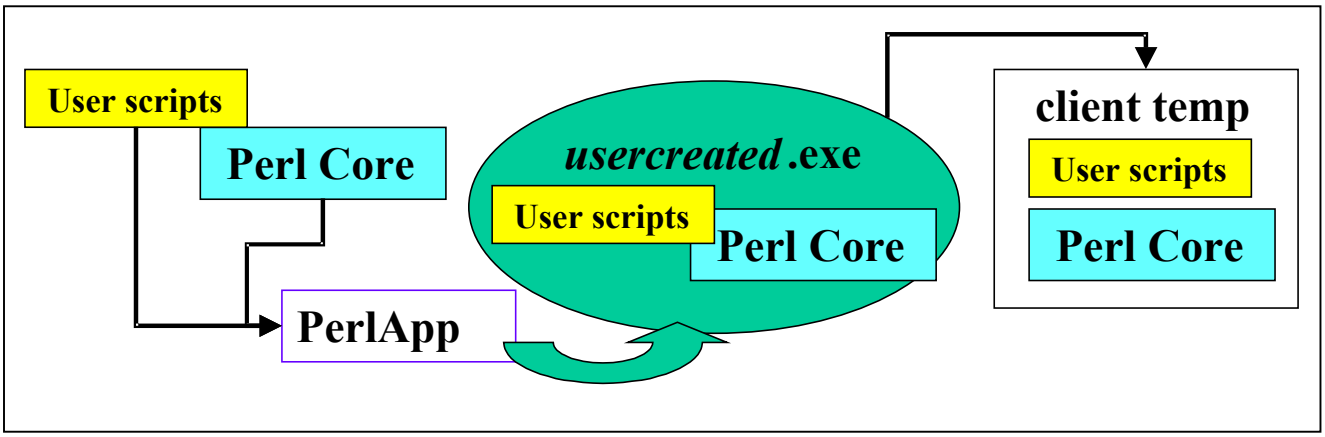


Figure 1: How PerlApp creates a self-contained executable

In the end, we were able to get approval from our legal team to ship PerlF. However, we are mindful of effort it took, and caution anyone using PerlApp that they might face a similar situation.

2.2.2 PerlF did not handle redirect

This script below documents one shortcoming we had with PerlF: using the redirect operator (<<) in the scripts.

```
# perlf_testredirect.pl
# Test that shows the failure of PerlF to work with redirect.

# works: perl  perlf_testredirect.pl
# fails: perlf perlf_testredirect.pl

print <<ENDOFPRINT;

-- A side effect using eval() is that the << redirect
-- does not seem to work with PerlF.

ENDOFPRINT
```

The fix was quite simple. Where we used to just read the script into an @array, now we read the file in with slurp mode set. Unfortunately, this one bug kept us from using a really useful scripting feature of Perl.

2.3 PerlF Modules

The following modules are explicitly used in PerlF. Explicitly specified modules are included in the final executable by PerlApp. Some of the implicitly included modules are included by default, but most of the others below have to be explicitly included.

```
use Win32::GuiTest;         # Controls
use Win32::OLE;             # Go thru OLE interfaces to Windows applications
use Win32::Process;         # start sub-processes
use Win32::TieRegistry;     # read/write the Windows Registry

use XML::DOM;               # Level 1 DOM.
use XML::Parser;            # used in xml2Excel* converter
```

```
use FileHandle;
use Cwd;
use Tk;                    # other Tk modules
```

Note that there is a fine balance in deciding what modules to include, as the more included the larger the end executable. Our current PerlF.exe on Windows is over 2 Megabytes.

The following subsections contain some specific notes on the important modules.

2.3.1 use Win32::GuiTest;

The Win32::GuiTest module can control a Windows* application's mouse and keyboard interfaces. We use it as a cheap replacement for GUI automation tools, especially the SendKeys() method.

From the Readme for Win32::GuiTest, this code snippet gives a good overview.

```
  use Win32::GuiTest qw(FindWindowLike GetWindowText SetForegroundWindow SendKeys);

  $Win32::GuiTest::debug = 0; # Set to "1" to enable verbose mode

  my @windows = FindWindowLike(0, "^Microsoft Excel", "^XLMAIN\$");
  for (@windows) {
      print "$_>\t'", GetWindowText($_), "'\n";
      SetForegroundWindow($_);
      SendKeys("%fn~a{TAB}b{TAB}{BS}{DOWN}");
  }
```

If you have an older version of Win32::GuiTest, be sure to add the following useful bits of code to the end, for MoveWindow() and SetActiveWindow() functions. Other Windows* functions can be added in the same fashion.

```
BOOL
MoveWindow(hWnd, X, Y, nWidth, nHeight, bRepaint)
    HWND hWnd;
    int  X;
    int  Y;
    int  nWidth;
    int  nHeight;
    BOOL bRepaint;

    CODE:
        RETVAL = MoveWindow(hWnd, X, Y, nWidth, nHeight, bRepaint);
    OUTPUT:
        RETVAL

HWND
SetActiveWindow(hWnd)
    HWND hWnd;

    CODE:
        RETVAL = SetActiveWindow(hWnd);
    OUTPUT:
        RETVAL
```

2.3.2 use Win32::OLE;

We use the Win32::OLE function extensively to read/write to Excel* spreadsheets. Often, the only problem with OLE is finding the exact methods and variables of the application being used. Rather than discussing that issue here, we refer the reader to the various examples in the standard Win32::OLE documentation.

2.4 Regr.pm – Regression Test Harness

Capturing the test output results is the job of the Regr Perl Module.

Regr is a Perl-based test harness that can be run from under PerlF or other Perls. Regr is a simple and sturdy regression test system that supports high and low-level regression testing, from System, GUI, User and Application levels, down to sub-components and drivers.

The intent is an easy-to-use system for code developers and testers, which can also be applied at all stages of testing and validation of a product.

The basic results of the system are delivered to a local XML file. By default this is Regr.xml in the current directory. This XML file can then be compared to other databases (remote, local) and can automatically merge the test results with separate utilities. The test results can also be uploaded to the golden comparison database as fresh results are added to the system.

2.4.1 Usage

In a Perl script the following is the basic Regr use. The Regr module must be included. The primary function is then invoked (&Regr::Regr), then a Tests() function that gets called back.

```
use Regr;
&Regr::Regr;
sub Tests {
    my (@args) = @_;    # just ARGV
    ... test scripts and scenarios ...
            // never exit(), always return!
            return $Regr::PASS;
}
```

Regr::Regr

1) Parses the @ARGV command line value.
2) Preps a new value that it passes to the Tests() routine.
3) Reads the ./Regr.cfg and any other appropriate Regr.cfg script (passed as a –db argument) for local and test suite settings.
4) Preps the appropriate Regr.xml for either overwriting or appending.
5) Sets up other variables needed by the Regr module.
6) Calls the Tests() routine, and transfers control back to the user script.
7) The Tests() routine should never exit, but should always return() back to the calling program. This preserves the Regr control functions.

Once within the Tests() routine, the user script has complete control and you can write any tests required in standard Perl code.

2.4.2 Regr.cfg

A special file in the test suite area helps set the defaults options for Regr, pointing to the Perl to use, the text diff program, and the copy command that can be used to update golden tests. Note that this is the Win32 version, because it changes the path separator.

```
$Regr::Path =~ s#/#\\#;
$Regr::perlexe = $Regr::Path.'\perlf.exe';
$Regr::subargs = '-sargs';
$Regr::TxtDiff = $Regr::Path.'\regrdiff.exe';
$Regr::copycommand = $Regr::Path.'\cp.exe';
```

The Regr.cfg file is just Perl code itself, run through eval() by the Regr module.

2.4.3 Linkages to sub-scripts – Regr::Script

When Regr invokes a sub-script via the Regr::Script(), that process is pointed to use the Regr.cfg file of the parent. In this fashion, the global test suite arguments are propagated down to sub-scripts.

Rather than using environment variables, Regr::Script() invokes the sub-scripts with command line arguments. This is more effective and portable, since the Environment Variable propagation on Win32 is not reliable.

2.4.4 Regr Methods and Values

These are the current interface methods for Regr.

$pid = Regr::Script – invokes other Regr scripts. Passes linkage arguments, and return the process return value. Also starts a <Script> sub-structure in the XML file. Closes with a </Script> on return from the invoked script.

Regr::TestResult – puts one <TestResult> structure into the output. <TestResult> can have multiple sub-

Regr::System – calls the Perl system() function, but logs the calls. Can be set to skip execution if $Regr::opt_donothing = 1;

$x = Regr::BackTicks – calls an executable with backticks (``) and returns the output string.

$passfail = Regr::GoldDiff – Does a textual diff of output files. Designed for easy use, it also has many override options on the default behaviors.

Values returned by Regr and available for scripts. These values should be used in the scripts to indicate conditions and status.

$Regr::PASS, $Regr::FAIL, $Regr::ERROR

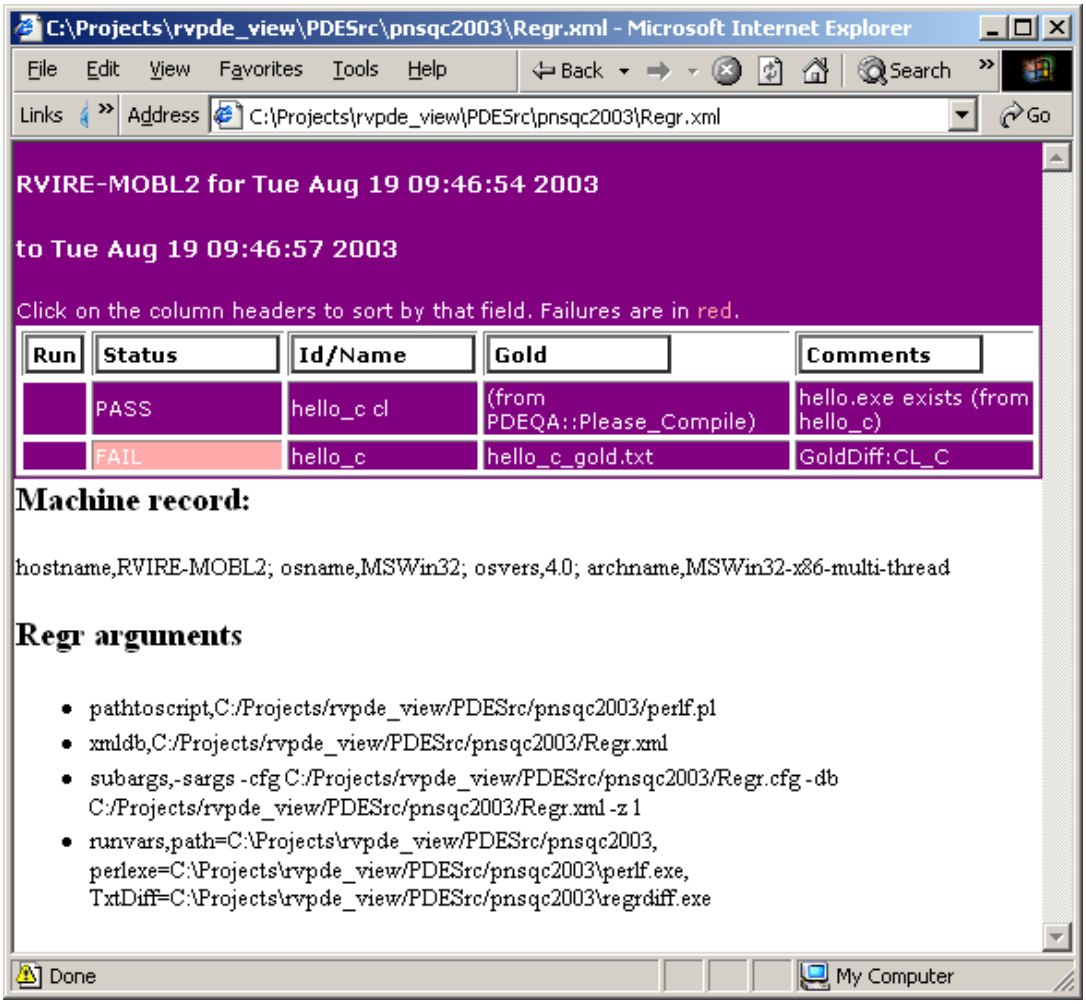
2.4.5 A Short Example

This is a short example using the “Hello, world!” program. There are two steps to the test: recompiling the program and running the newly-built executable.

```
use lib "../..";  # so we can find Regr.pm and QA.pm, which are located up from here
use Regr;
use QA;

&Regr::Regr;            # For section 2.4.5 A Short Example

sub Tests {
   my (@cmdargs) = @_;
   my $status, $teststatus;
   my $comment;

   $teststatus = $Regr::PASS;

   unlink("hello.exe") if -e 'hello.exe';   # in case of errors
   unlink ('hello.obj');

#-------------------------------------------------------------------------
# MSVC
#-------------------------------------------------------------------------
   $status = &QA::Please_Compile (
                  'handle'   =>  'hello_c', 'source'   =>  'hello.c',
                  'exe'      =>  'hello.exe', 'compiler' =>  'cl',
   );
   $teststatus = $Regr::FAIL if $status eq $Regr::FAIL;

   # Run the generated executable.  If not found, error output will occur.
   Regr::System ("cmd /c hello.exe > hello_c_out.txt");

   $status = &Regr::GoldDiff(
                  'handle' => 'hello_c',
                  'comments' => "CL_C"
   );
   $teststatus = $Regr::FAIL if $status ne 0;

   return $teststatus;
}
```

The Please_Compile() function puts one result into the Regr.xml file (using Regr::TestResult). The GoldDiff() function compares the expected output to a known golden output.

The output of the test run is shown in the following illustration.



2.4.6 Regr::GoldDiff()

GoldDiff() is designed to have default behaviors that make it easy to put it quickly anywhere in a test script. Its primary purpose is to compare a Golden output file with a new one, put the results into a new output file, and record the results to the XML output with a <TestResult> field.

The default difference engine is ‘regrdiff’, which is a text diff with simple modifications on the return values. This code is available for download. Rather than a text diff, another diff program or filter program can be specified with the ‘diffprogram’=>$string option shown below.

The default behavior for GoldDiff is controlled by the handle.

```
'handle' => "$hostname/x/y",     # basename to test results
```

If only handle is specified, then the following default values are assumed.

```
'filegold' => '$hostname/x/y_gold.txt'
'filediff' => '$hostname/x/y_diff.txt'
'fileout'  => '$hostname/x/y_out.txt'
```

You can overwrite any of the 'file...'=> parameter defaults with your own. This is useful for legacy tests and customized tests reusing files and values.

2.5 'value' Arguments, instead of file differences

If no file 'fileout' exists, then one type of value **MUST** be specified. This value is compared against the contents of the ‘filegold’ argument. This technique is useful to cut down on the number of test files, if the output can be captured in simple values.

```
'value'   => \@list              # list to compare against
'value'   => \%table             # table to compare against
'value'   => \$string            # string to compare against
'value'   => $SCALAR             # SCALAR to compare against
```

2.6 Optional Arguments

These other named arguments can be used to modify the default behaviors of GoldDiff(). They are not required.

```
'id'  => "this is weird test",   # passed to TestResult
'label'  => "this is weird test",# use label instead of 'handle'

# comments are appended to any GoldDiff comments in XML output
'comments'  => "Appended",
'leaveoutput'     => 0,        # set to 1 to leave 'fileout' file
'leavediff'       => 0,        # set to 1 to leave 'filediff' file
'nothing'         => 0,        # do nothing, just see what would be done
'donotcreategold' => 0,        # set to 1 to not create 'filegold'
                                 #   if it does not exist
# default is regrdiff.exe, specified in $Regr::TxtDiff
'diffprogram'    => "someotherdiff.bat"

'updategold'     =>  0,       # set to 1 to copy fileout to filegold
                              # only if filegold does not exist
                              # VERY DANGEROUS! USE SPARINGLY!
```

2.6.1 Regr Shortcomings

The biggest shortcoming with Regr is the XML output file. We have found that XML is a marvelous transmission format, but is a lousy database. One of the reasons Regr writes only XML was the initial requirement that it could eventually be used on lower-end devices that may not have a full XML parser available. Also, early Perl systems did not have stable XML parsers that we could rely on to read and write the files.

There are other shortcomings we plan to address over time. These are discussed in the “Future Plans” section near the end of this paper.

2.7 QA.pm

QA.pm is where we put functions and definitions that a test suite needs, but with which we do not want to pollute the Regr module. It may have specific tool functions that can be re-used by many test scripts. QA.pm is basically the toolbox for the testing suite.

A method in this module may serve in an auxiliary or a terminating role to the script that calls it. If it serves an auxiliary function, the method is useful as an intermediate step in a test case. The following is an example of this method.

```
$y = TellMeDestinationMachine($handle);
```

This function simply parses the handle(described in section 2.1.1) and returns **$y,** the type of machine meant to be the destination. The test case can then continue to do its test using this information. Once the test case is completed, the test script must report the results using one of the methods from Regr.pm

A method that serves as a terminating role finishes the test and then itself reports the results back to Regr. A complete test case is conducted and then reported. The following is an example of this method.

```
PleaseCompileVS7($handle @description);
```

This function takes the test case completely described by the handle and compiles it using Visual Studio 7. The function then does a test to see whether the test passed or failed. The result is then reported through one of the methods in Regr.pm.

We are not releasing the code in our QA.pm at this time, because we have some proprietary testing information in there. The following are some of the methods in the module. Most are used to provide the functionality for the Excel* spreadsheets.

- `sub Please_Compile`
- `sub Please_Compile_cmd`
- `sub Please_Compile_make`
- `sub Please_Compile_vs7`
- `sub Please_Execute_cmd`
- `sub Please_Execute_vs`
- `sub Please_Execute_make`
- `sub PickCompiler`
- `sub Select_Options`

2.7.1 Xml2Excel*.pl

This Perl script enables us to take the XML output that Regr.pm generates and incorporate it into the Excel* spreadsheets which contain the name of the tests. It uses Perl modules XML::Parser, Win32::OLE (to access Excel*), FileHandle, and Cwd. The system on which the script is operating must have Excel* installed in order for the script to function.

This script is available with the other download code.

3 PerlF-based Tools

The following are short descriptions of other specific tools that we have created to run with PerlF.

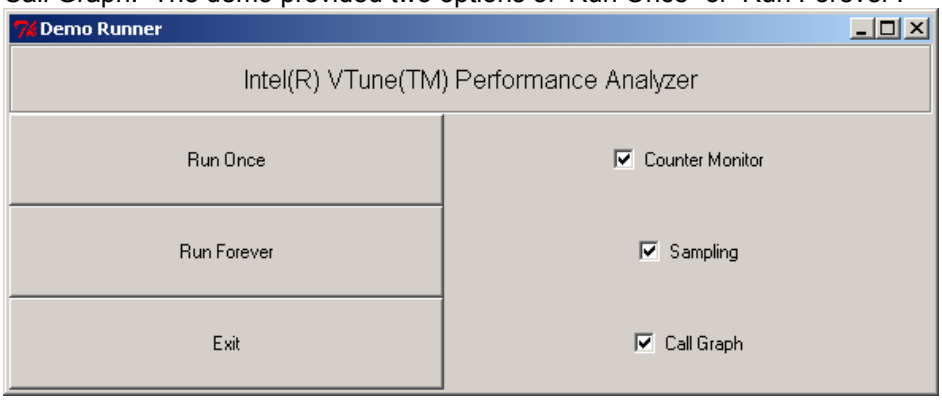
3.1.1 Thread Checker Smoke Test

A system going to a remote customer, we were required to provide some mechanism

3.1.3 A Trade Show Demo

Using PerlF, we created a small script to run a trade show demo that used the VTune™ Performance Analyzer. This script uses the Perl/Tk and Win32::GuiTest modules to drive the application.

The VTune™ application has three features we were using in the demo: Counter Monitor, Sampling, and Call Graph. The demo provided two options of 'Run Once" or 'Run Forever'.



3.1.2 Custom Client Installer

Since they install a client application for many hundreds of users, the deploying business unit had a problem with the installation process. A fully automatic installation was not possible. The installation also presented to the installing users options that they did not have available to deploy, but there was no way to disable this dialog. Also, if PowerPoint* was not installed, the dialog button would be grayed out.

```
#####################################################################
# INSTALLER WINDOW LIST (in order)
#
# This is the list of window titles that the Installer expects for
# this version of the D client.  If the windows change or order changes,
please update.
#
my @Window_List = (
        'Setup',
        'Welcome',
        "Select Options",
        "Choose Destination Location",
        "Choose Destination Location",
        "Setup Complete",
);
```

For this internal application, two scripts had to be developed. The first is the command script that runs the whole client. It checks that the windows occur in the right order, scans the pop-up dialog checkboxes and sends appropriate keystrokes and monitors the system.

The second was OkCancel.pl, whose sole job is to pop up a dialog so the user can select **OK** or **Cancel**. We discovered that if you call the Tk subsystem to display a dialog, it disables the other functionality of the command script which controls the windows and sends keystrokes. In the future, threading may allow us to keep all of this in one script.

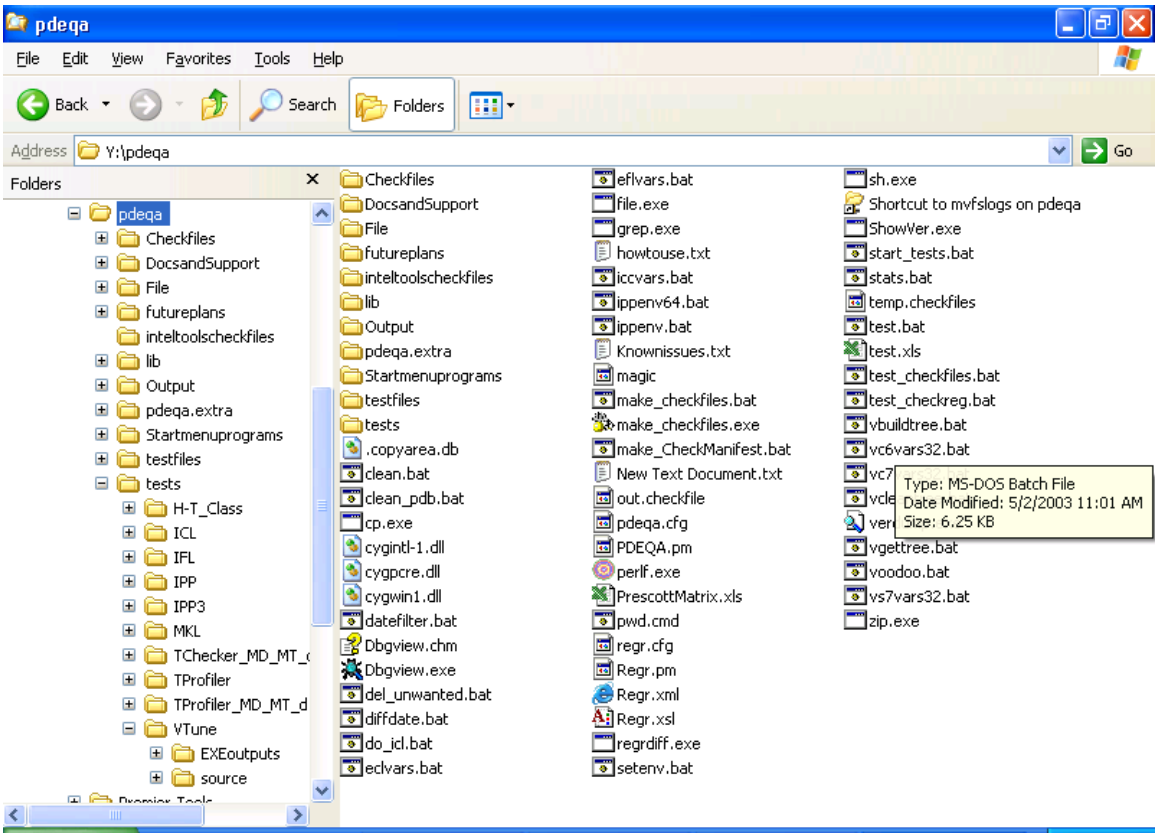
4 Test Suites

Here we try to pull all these tools together into a working Test Suite.

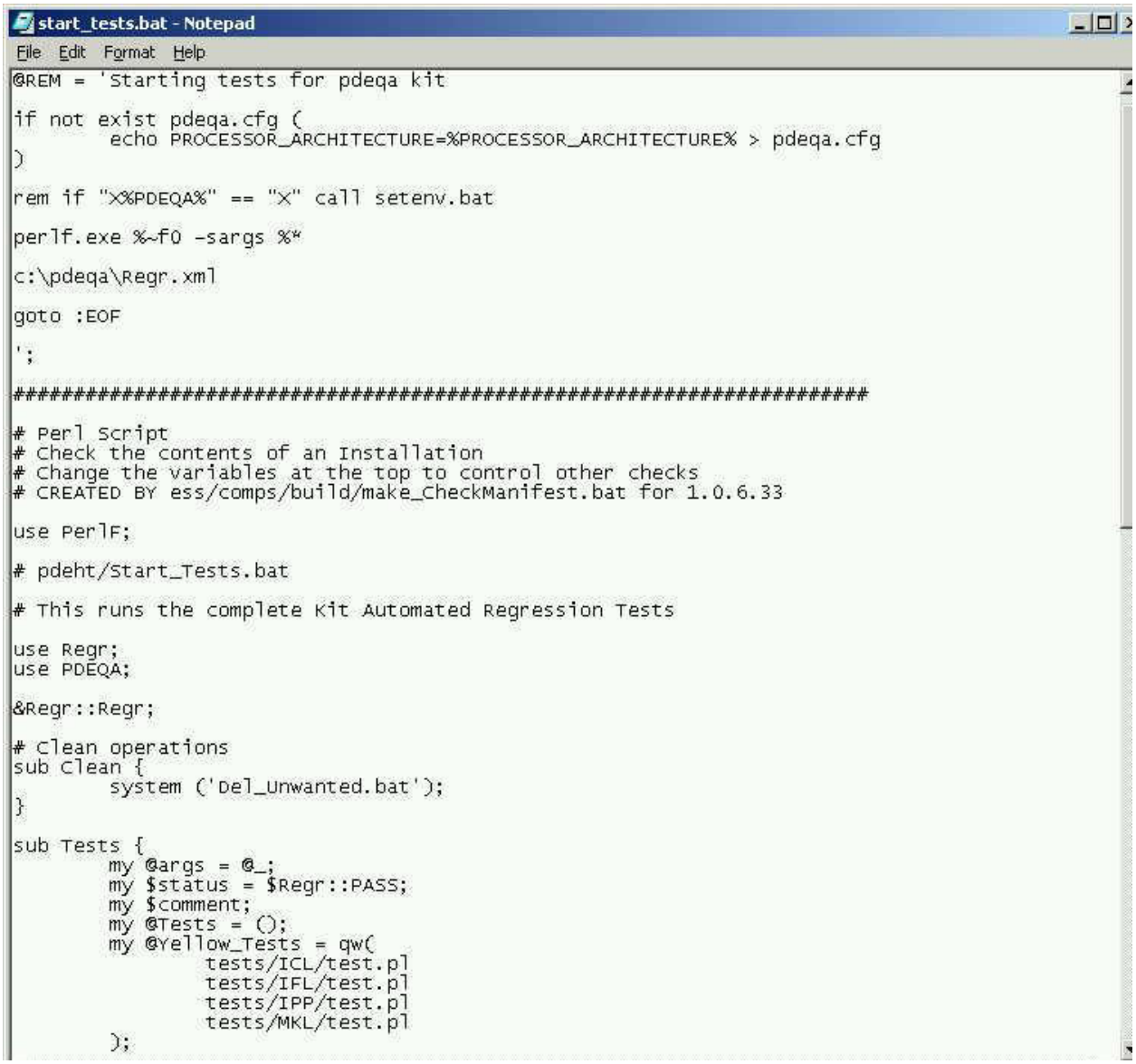
There are two test script styles. From a Spreadsheet, tests come as Macro calls. The second style is a handwritten test.

The actual test suite is one large tree that can be transferred easily between different systems. The test suite contains many files in the root directory, including the spreadsheet from which the macro test cases are copied out to various **cases.txt** files. By using some VBA scripts pre-programmed into the

spreadsheet, the test cases in the Excel* spreadsheet are transferred to the appropriate location of the cases.txt files. The Perl script reads the test cases from these files



The Perl script knows how to read these test cases because this information is specified in the start_tests.bat file. What looks like a batch file is actually transformed into a Perl program by the first few lines of code.



Before running the test suite you must open the start_tests.bat file and specify which tests need to be run. This is accomplished by calling separate Perl modules custom-designed to deal with particular types of Intel® tools. For example, the tests/MKL/test.pl script is designed to deal with the Intel® Math Kernel Library tests.

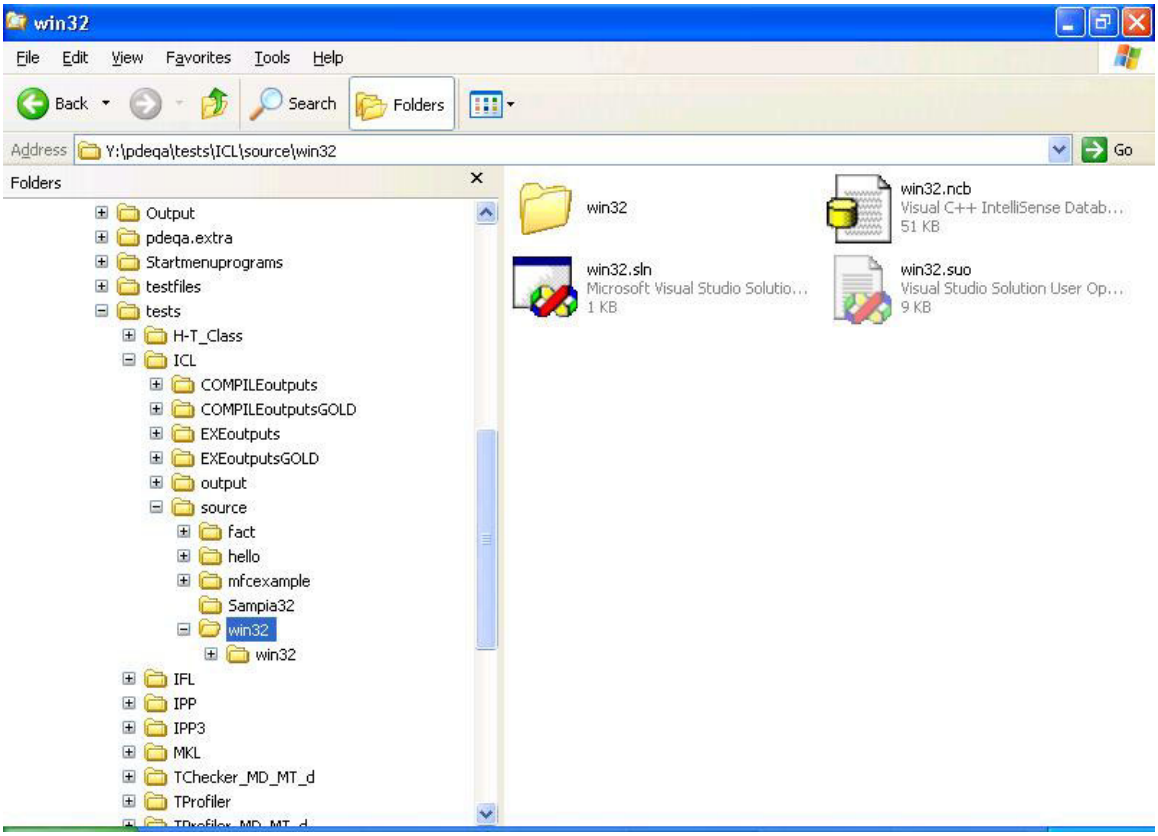
One can also categorize their tests with the test.bat file. Within this file, one can choose whether to run up to three levels of testing (yellow, green, red). In the start_tests.bat file the array shown above runs the yellow tests.

To start the test suite, run the test.bat file. This causes the tests to begin loading in a command line window. When running the VTune™ tests and other tests that use the perl module Win32::GUItest, one should take extra precaution not to alter the test system while the tests are running. Additionally, any screen savers or monitor-power saving functions should be shut off so that Win32::GUItest can run properly.

Most of the examples that are actually compiled and executed do so from within the test suite folder. Within the test folder are subfolders for the various tools (e.g., ICL). Within each of these folders there is a

source folder that contains projects or source code to test on the current system. As newer versions of tools arrive, the test suite and sometimes this source code must be modified.

For programs that produce text output, the output is often saved into text files within the EXEoutputs folder. This output file is then compared with a gold version within the EXEoutputsgold folder. This tests whether the executable has been built properly.



At the end of the test cycle the results of the test have been recorded into the Regr.xml file which pops onto the screen for easy viewing. The Regr.xml file contains the name of the test (see information above describing how tests are named) and the PASS/FAIL results. By running the xml2excel converter one can easily convert this xml output back into the Excel* spreadsheet (or matrix).

Finally, one can run any number of manual tests and record the PASS/FAIL results into the Excel* spreadsheet. The Excel* spreadsheet uses the printarea function to calculate the total number of tests passed, failed, and “to do”. It also displays a pie chart for quick viewing of the overall status of the tests.

5 Work Environment

Although the primary focus of this paper is on the software tools we use, another aspect of testing that we do not ignore is the physical work environment. Simple, often free, solutions can present themselves. Anything that facilitates communication and team effectiveness should be considered.

5.1 Quick Meetings

Lab conference area. We were fortunate as a team to inherit a large lab space. We converted part of the space into a general meeting area. This lets us have unscheduled and quick meetings by anyone on the team. We also use the area for visitor demonstrations and filming training videos.

Lab conference area, with large monitor. For a few weeks during one project, we were able to borrow a large 48" plasma screen monitor. This proved invaluable during brainstorming sessions, bug reviews, and project demonstrations. Everyone was able see and talk to the same topics. We are attempting to obtain an LCD projector in the future for this area, given that recent prices are very reasonable.

5.2 Daily Project Meetings

Half-hour daily project meetings are invaluable to uncover problems quickly, cross-pollinate information, and keep a finger on the pulse of the overall project. We have tried since our inception to have daily (or two or three per week) specific project meetings when a project is close to release. However, getting information from five or six people and passing down all that information in 30 minutes is a challenge.

5.3 Coding Guidelines

Having a team agree on standard Coding Guidelines can sometimes be an exercise that can result in near-religious wars. Nevertheless, such guidelines should be viewed instead as helping to build the culture of the team. In the end, they help with testing.

5.4 Design and Code and Re-Design Reviews

We think the literature is quite extensive on the value of design and code reviews. There is no need to rehash those issues here. Our experiences are that they can be costly in terms of time, but in the end lead to much better code.

As we progress through our code base and review each program, it also is a training opportunity for all the team members. Seeing different ways of programming and understanding different problem domain.

5.5 Training

Sharing scripts and tests, code and design reviews, Brown Bag sessions. All of these contribute to the overall environment. We are a small enough team that we try not to make much of a distinction between different roles people play. Cross-Training is very important to our efforts.

5.6 Agile/XP

Most of these methods discussed will seem very familiar to Agile/XP developers. We have been using many of them in some form or fashion. In terms of the physical environment, we are attempting several upgrades in line with Agile and XP philosophies.

One rather out-of-budget effort is we are trying to upgrade developer systems to have dual-monitor systems. This not only gives a 20-30% productivity boost, but also facilitates pair-programming.

In addition to pair-programming, we continue to do frequent builds, continuous customer engagement, unit testing and incremental development with short delivery times of less than 2 weeks. We also ***try*** to budget 10-15% of any project to process improvements (tools, process, techniques, training).

6 Future Projects

These projects represent our current TODO lists. We have no idea if or when we will be able to tackle them, but we keep this list rolling forward in the spirit of making progress.

6.1 Regr

- Upgrade Regr to smaller test output and XML Parser.
- More Regr methods, as need arises.
- Hooks for JUnit and other test hooks.
- Add in ACE/TAO Toolkit sub-process control Perl modules to Regr.pm library The process_unix.pm and process_win32.pm functions should allow us greater control of sub-processes and tests, even (possibly) parallel test suites in some cases.
    - sub Spawn ()
    - sub WaitKill ($)
    - sub SpawnWaitKill ($)
    - sub Kill ()
    - sub TerminateWaitKill ($)
    - sub Wait ()
    - sub TimedWait ($)

6.2 PerlF

- A Linux equivalent of Win32::GuiTest!
- Update PerlF to newer XML Parsers. Get rid of XML::DOM. Updating PerlF to 5.8, and eventually 6.0.
- Using threads in Perl, to invoke Tk dialogs and other sub-systems.

6.3 Test Suites

- Control panel, to give easier tester configuration for a test run.
- A Test Suite Manager. Do something other than just a command line, by using Perl/Tk to give a nice portable UI, with checkboxes and select functions and such.
- Expand beyond Excel* for macro test specifications. Again, looking at Perl/Tk for something more portable.
- Parallel Test suites, using Perl threading.

7 Bibliography

PerlApp Manual. ActiveState Perl Development Kit.

Testware for Free. by Danny Faught, http://www.stickyminds.com/r.asp?F=W6454

“Using XML as a QA Foundation Technology.” Vireday, Richard and Steven B. Augustine, PNSQC 2000 Proceedings. *First Discussion of Regr.pm module.*

Regr man page – PLD Business Unit, 1991. Internal document.

“Learning Perl Objects, References and Modules”, Randall L. Schwartz with Tom Phoenix, O’Reilly Press, 2003.

7.1 Downloads

You may download the source code for PerlF and the other scripts mentioned here either from the www.pnsqc.org proceedings area, or from www.vireday.com/perlf . The code should be available just before the PNSQC 2003 conference.

8 Thanks

Thanks to Warren Nickerson, Intel® Legal, for the first review of PerlF code.

Thanks also to Miriam Ezriam, Intel® Legal, for PerlF legal review, and the Linux teams at Intel® for the Open Source Review and Approval.

Special Thanks for the Win32::GuiTest author, Ernesto Guisado, erngui@acm.org, for providing such a nice and useful Perl module.

Good Thanks to Devinder Ahluwalia, who let us take the risks to use and develop the tools when he was manager of the DPS Engineering group.

Finally, Thank you to Arne Bowman and John Gardner for the first half-dozen versions of the Excel* spreadsheets that have evolved to our current macro test specifications.

PerlF, Regr, and Other Cheap Testing Tools

for PNSQC 2003

Richard Vireday
Neel Patel
Arunarasu Somasundaram
Ajay Dave

October 2003

development platforms & services

software engineering

intel

Agenda

- *We do software right*
- Problems in testing
- Tools Solutions
- Future work

development platforms & services

software engineering

intel

Code Summary

- Released under the BSD License
- Specific Source Code
  - **PerlF.pl** – used to create PerlF.exe, a full Perl distribution as a single executable.
  - **Regr.pm** – Perl Module, Regression Test harness.
  - **RegrXml2Excel.pl** – Converts Regr XML output to Excel Spreadsheets.
  - **Regrdiff.exe** – C "diff" program
  - ***Misc. Scripts*** – demonstrating PerlF.exe and Regr.pm usages

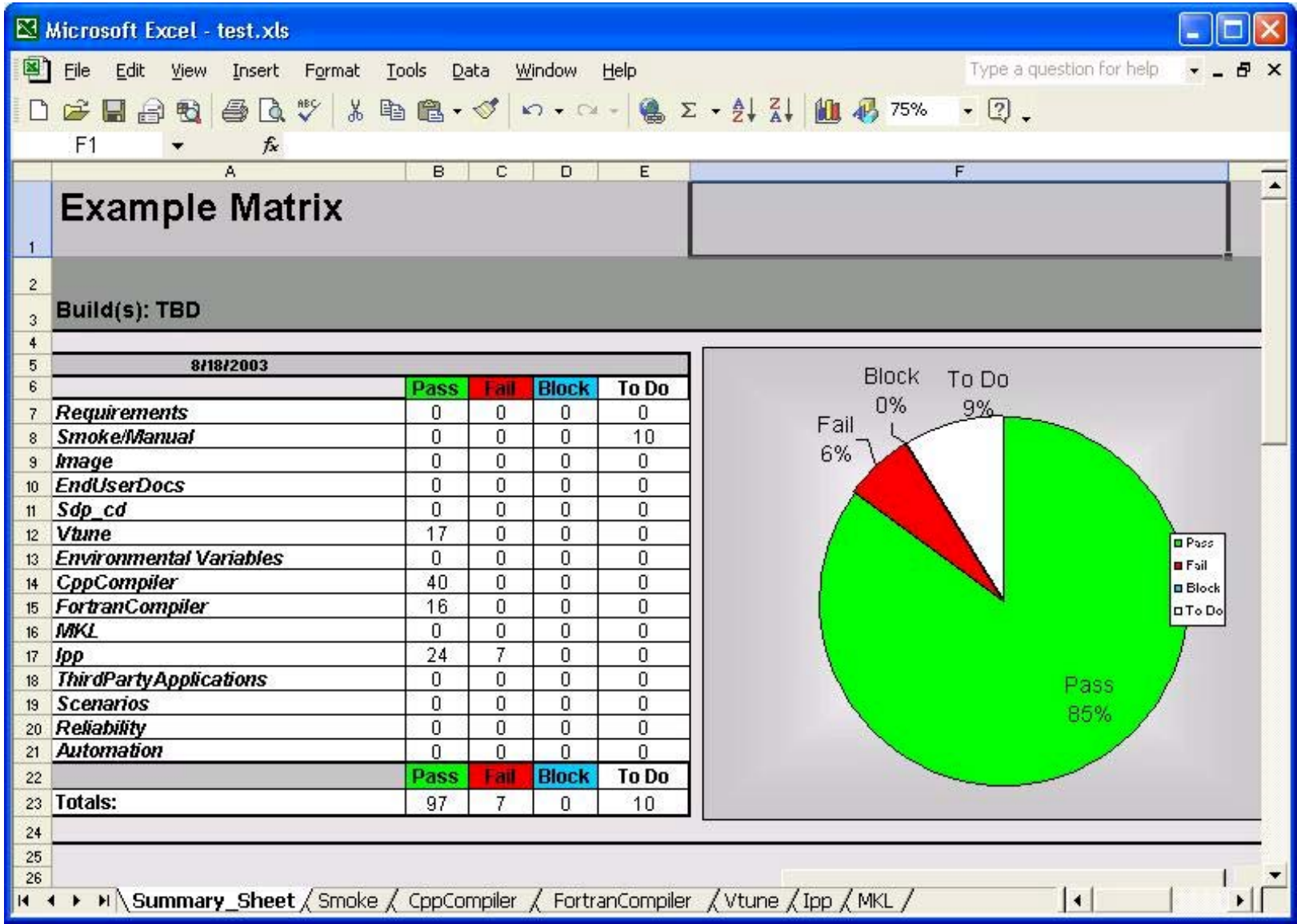
The Problems

- Injectable Test Systems
  - $3-5K for each client
  - Visual Test had a runtime model
  - Segue Silk, QA Partner, WinRunner
- Automated On-Demand Builds
  - On-Demand, Nightly Regression Tests
- Other Chores with SW Development

Excel Summary Sheet here

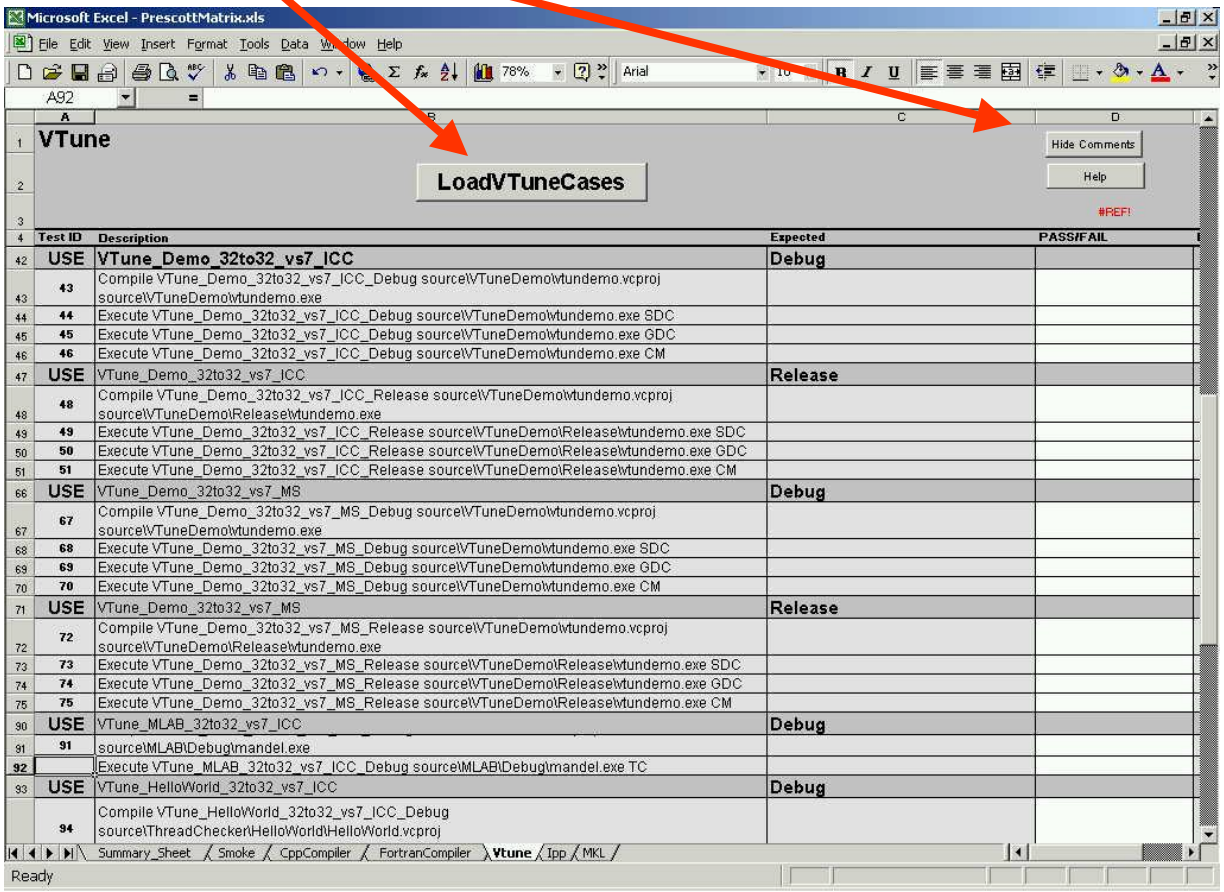
development platforms & services

software engineering



Macros

- Macros automate exporting of data from matrix into respective cases.txt files and other chores

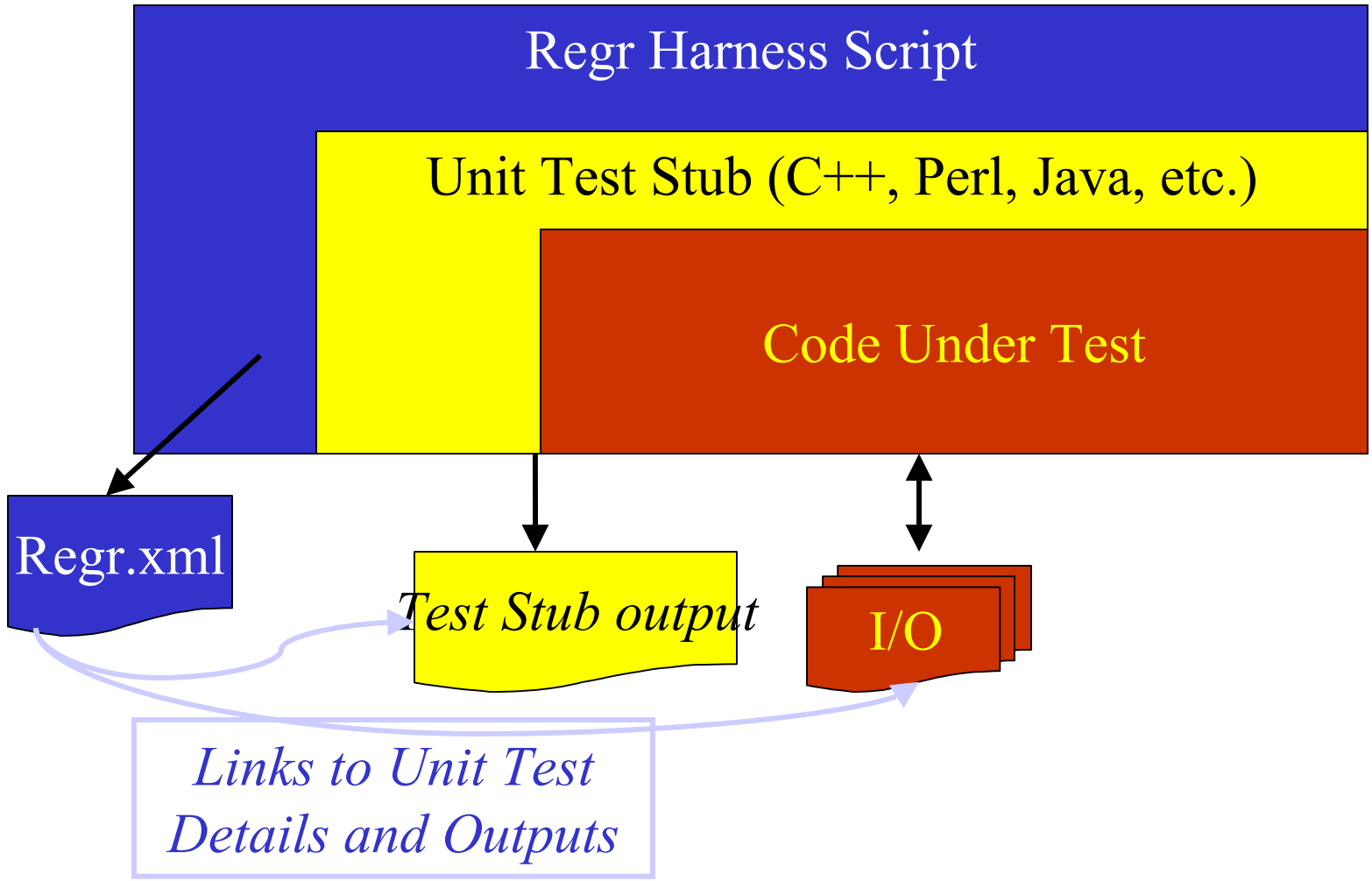


development platforms & services

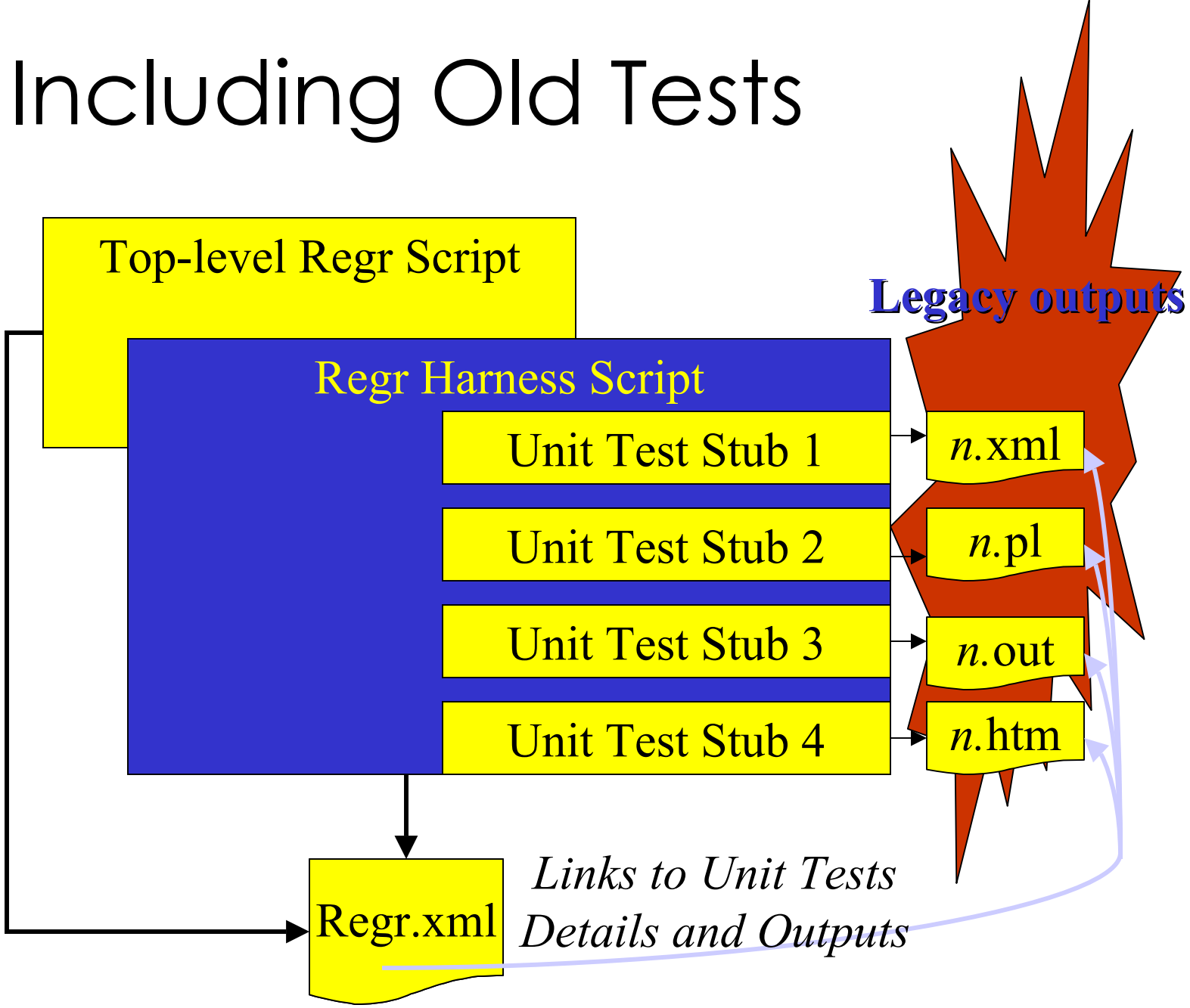
software engineering

intel

Regression Tests



Including Old Tests



Regr – Integration Front

- Regr is a Perl wrapper and API to drive different Unit Tests
  - Simple makefiles are good, but not sufficient
  - An assistant to take care of the paperwork

```
# template.pl - Template Example of using Regr Module to create a Test Suite
   use Regr;
   use Win32::GuiTest;    # optional

   &Regr::Regr;    # Invoke Regr Module system.
         # returns control to local Tests() function below
   sub Tests {
   # Demonstrate test scripts and scenarios ...
      my (@cmdline_arguments, @attributes) = @_;
      $ret = Regr::TestExe("test1.exe", "-i test1.in –o test1.out",
   …
   }
```

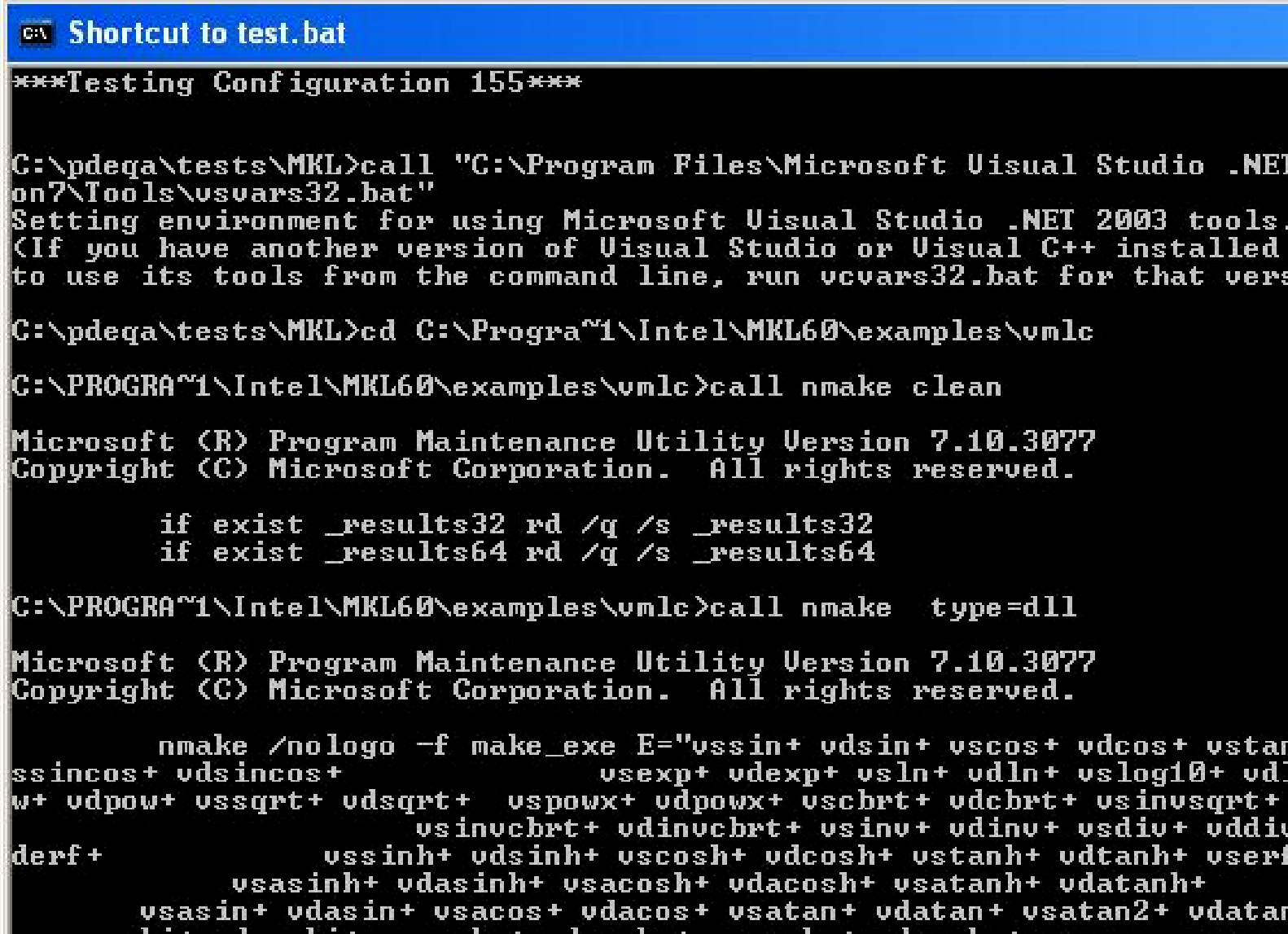
Regr Description

- Primary purpose to capture test suite results
- Also facilitates automatic testing.
  - GoldDiff – uses regrdiff external
- *History of Regr*

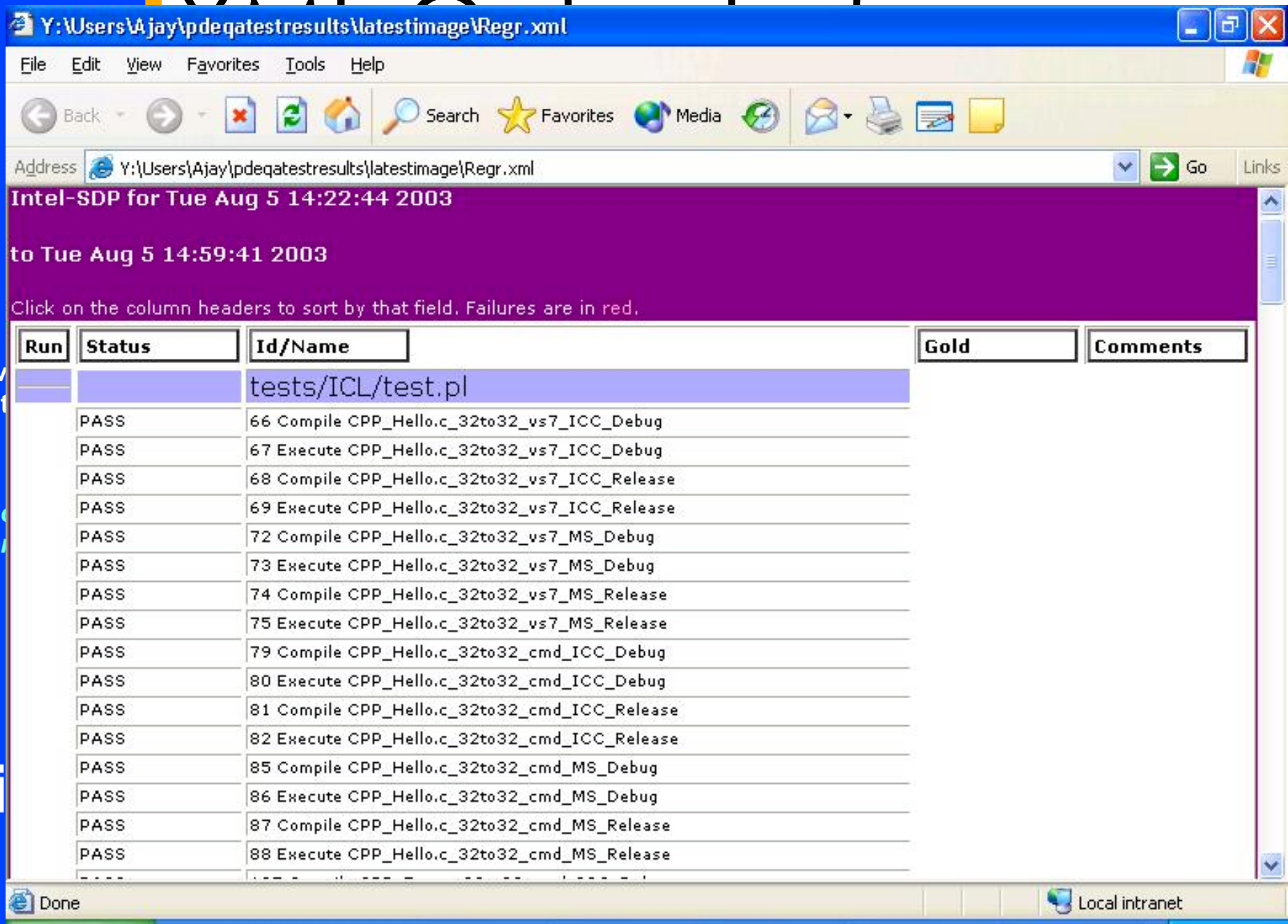
Test Case Run

development platforms & services

software engineering



XML Output of



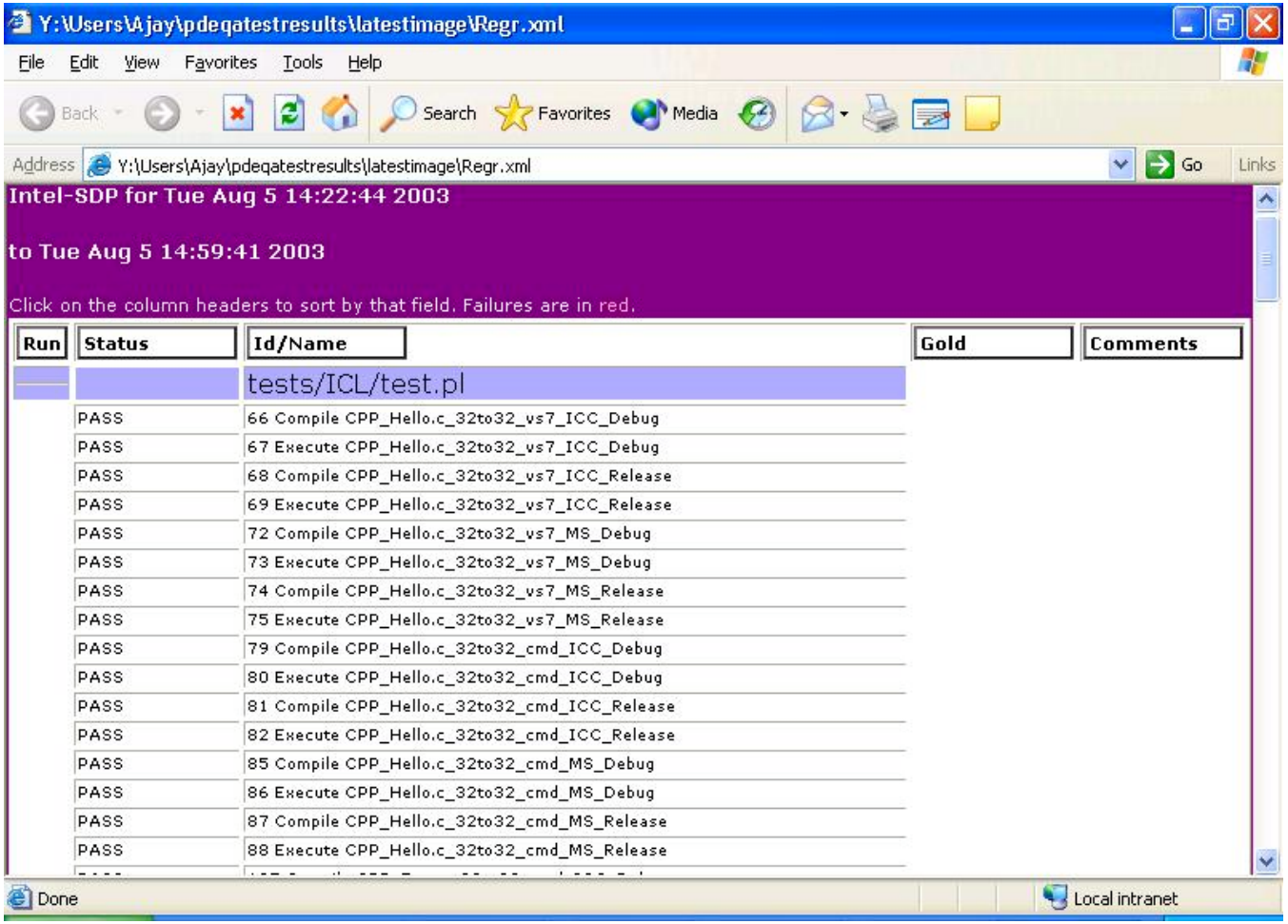
dev
plat

sc
e

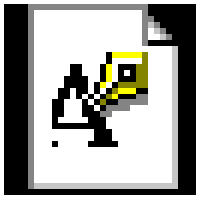
i

XML2Excel Converter

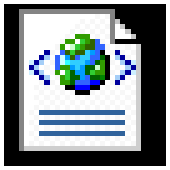
- Converts “Regr.xml” data back to Excel spreadsheets



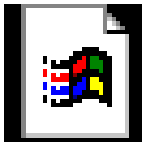
The Regr Code



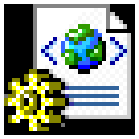
Regr.pm



Regr.xml



Regr.cfg



Regr.xsl

Regr.htm

- Regression Test Harness Module.
- *Needs more cleanup of documentation*

The Regrdiff Code

- Code is from 1992, old Intel group
- **Not required** to run Regr.pm or PerlF, but is very handy!
- *Needs to be stripped down for portability. New copyright added.*

REGRDIFF RELEASE [DBG Eval tools] SID [1.1 October 1996]

(C) Copyright 1988-1996 Intel Corporation

usage: regrdiff file1 file2 [outfile] [statusfile]

Status File return status 0 - no diffs

status 1 - files differ

status 2 - diffs or errors

Test Drivers

- Injectable Test Systems
  - No cost to use on test client
  - Low overhead to install and configure
  - Reliable across multiple hardware, OS, network, and peripheral configurations
  - Similar functionality across multiple hardware, OS, network, and peripheral configurations

PerlF solution

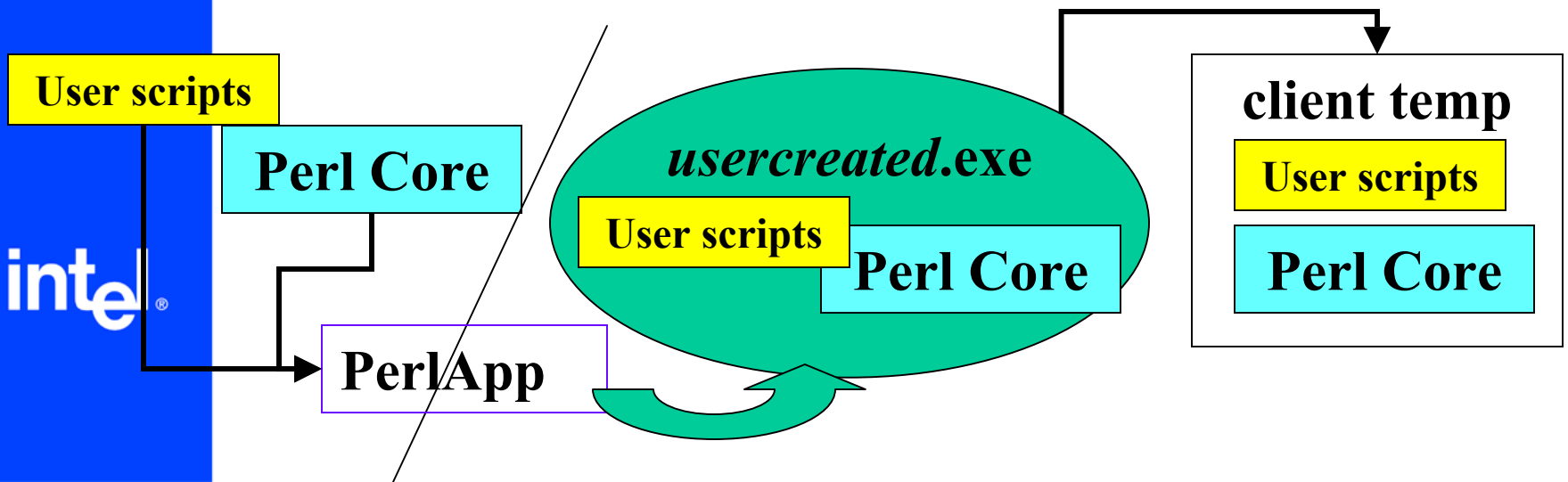
- Single executable containing Perl system
- Runs on Windows, Linux, Solaris, etc.
- Taps into Perl Modules
  - Win32::OLE, XML::, LWP::, etc.

PerlF other attributes

- Win32 Perl can Drive COM/COM+ objects
- Interfaces to HTTP, TCP/IP and other protocols
- Perl is the Duct Tape of the Internet Age
  - Lots of modules and support

How does PerlApp work?

- Can Package code into single exe
  - Linux, Solaris and other versions available
- Unpacks to temp directory, and runs Perl Interpreter on code there
- Does not violate GPL, because all Perl code is dynamically linked.
  - Do have to be careful that modules included have distribution rights!



What is PerlApp?

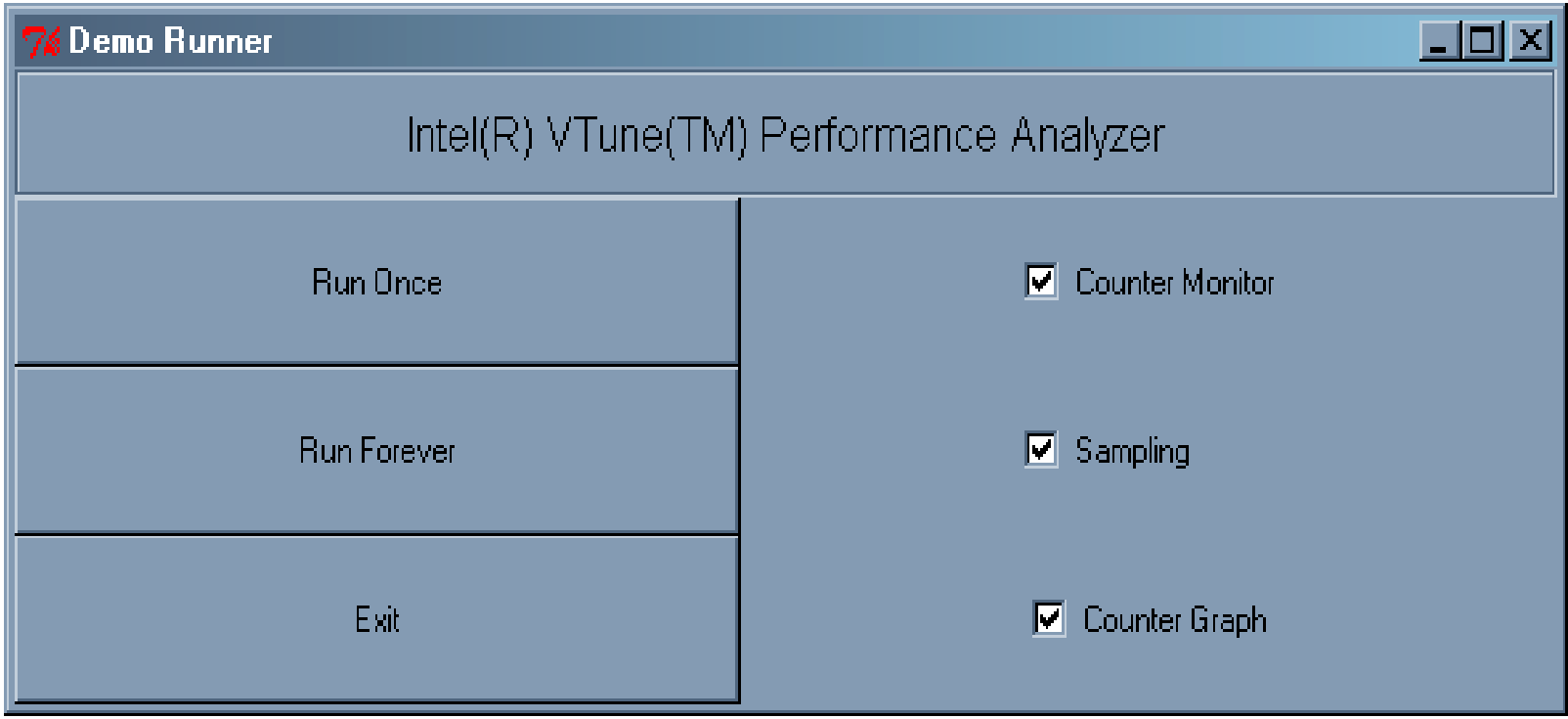
(From the manual)

The PerlApp utility converts a Perl program into an executable file. The executable can be freestanding, for use on systems where Perl is not installed, or dependent, for use on systems where Perl is installed. Freestanding programs built with PerlApp will be larger in size than dependent applications, as required Perl modules are included in the executable.

PerlApp is not a compiler. That is, the Perl source code and the contents of embedded modules must be parsed and compiled on the fly when the executable is invoked. However, PerlApp makes it easier to distribute Perl scripts, as Perl (or a specific version and combination of Perl modules) does not need to be resident on the target system. PerlApp applications will not run any faster than the source Perl script.

The Perl Dev Kit includes both Windows and Linux / Solaris / HP-UX versions of PerlApp. The command-line options for the two platforms differ; consult Linux / Solaris / HP-UX Options or Windows Options depending on the target platform.

Other PerlF Examples



development platforms & services

software engineering

intel

PerlFng – a variation

- PerlF, with No Graphics.
- Used for an install tool, where we don't want command windows to appear

```
perlapp -force -freestanding -gui
  -exe=perlfng.exe @perlf.perlapp
  perlf.pl
```

Win32::GuiTest Perl Module

- Need A Keyboard/mouse tool?

```
use Win32::GuiTest qw(FindWindowLike GetWindowText
  SetForegroundWindow SendKeys);

$Win32::GuiTest::debug = 0; # Set to "1" to enable verbose mode

my @windows = FindWindowLike (0, "^Microsoft Excel",
   "^XLMAIN\$");
for (@windows) {
    print "$_>\t'", GetWindowText($_), "'\n";
    SetForegroundWindow($_);
    SendKeys ("%fn~a{TAB}b{TAB}{BS}{DOWN}");
}
```

Thanks to…

- … **Warren Nickerson**, Intel® Legal, for the first review of PerlF code.
- … **Miriam Ezriam**, Intel® Legal, for PerlF legal review, and the Linux teams at Intel® for the Open Source Review and Approval.
- Special Thanks for the Win32::GuiTest author, **Ernesto Guisado**, erngui@acm.org, for providing such a nice and useful Perl module.
- … **Devinder Ahluwalia**, who let us take the risk to use and develop the tools
- Finally, **Arne Bowman** and **John Gardner** for the first half-dozen versions of the Excel* spreadsheets that have evolved to our current macro test specifications.

Questions?

- Get a copy of perlf.exe – sorry. You have to build yourself.
- ?

Home Brew Test Automation: Learning from XP

Bret Pettichord
Pettichord Consulting

Extreme Programming (XP) teams often reject commercial GUI test tools, finding it more useful to build their own support for automated testing. This presentation explains some of the strategies they use, which can cross over to any team where developers agree to help build automation support. The XP community has already made a major impact on tools and practices for unit testing in the wider development community. Learn how customer-perspective testing also benefits from this attention.

- View demos of open source testing tools
- Discover when it's better to build your own tools
- Examine how to make software more testable

Bret Pettichord is an independent consultant specializing in software testing and test automation. He is co-author of *Lessons Learned in Software Testing: A Context-Driven Approach* and a regular columnist for StickyMinds.com. He also publishes TestingHotlist.com. He has developed test strategies, and automated testing support for various domains, including publishing, taxes and accounting, sales tracking, internet access, education, benefits administration, and systems, application and database management. He is currently researching practices for agile testing. He founded Pettichord Consulting in 2000 in order to devote his time to consulting, research, writing and teaching. Before that he worked as an internal test automation consultant at IBM/Tivoli and BMC Software and an external consultant at SegueSoftware. He has a bachelor's degree in philosophy and mathematics from New College of Florida. He is a member of the IEEE Computer Society and the ACM.

Home-Brew Test Automation

bret@pettichord.com
www.pettichord.com

October 2003, PNSQC, Portland, Oregon

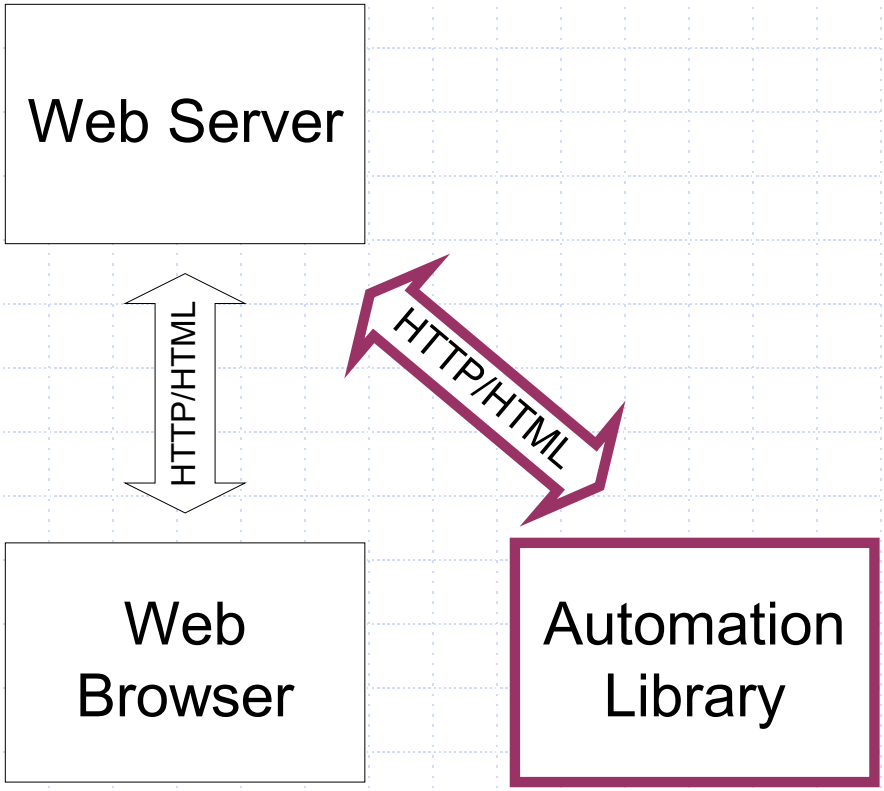
XP and Automated Testing

- Programmers write automated unit tests.
- Acceptance tests must also be automated.
- Programmers and testers *work together* on acceptance tests.

XP Teams Rarely Use Commercial GUI Test Tools

- Objections to Commercial Tools
  - Price
    - Everyone on the team needs to be able to run the tests.
    - Can't afford to give copies to everyone
  - Tool Languages
    - Understood by few
    - Often weak and limited: "Heinous"
- Often prefer building their own testing frameworks

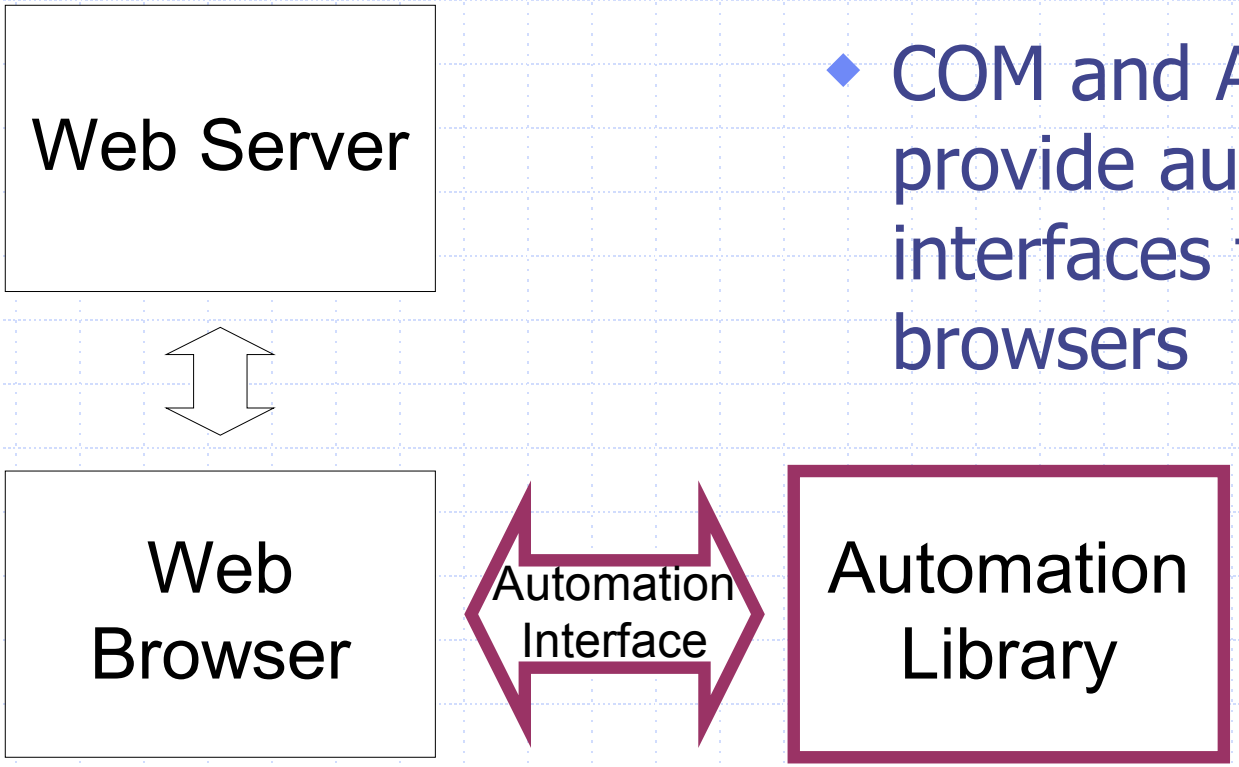
Example: Browser Simulation



- HTTP-Unit
  - Java
  - Perl
  - Ruby
  - Python

Tests execute directly against the web server

Example: Browser Automation



- COM and Applescript provide automation interfaces to browsers

Tests execute against a browser

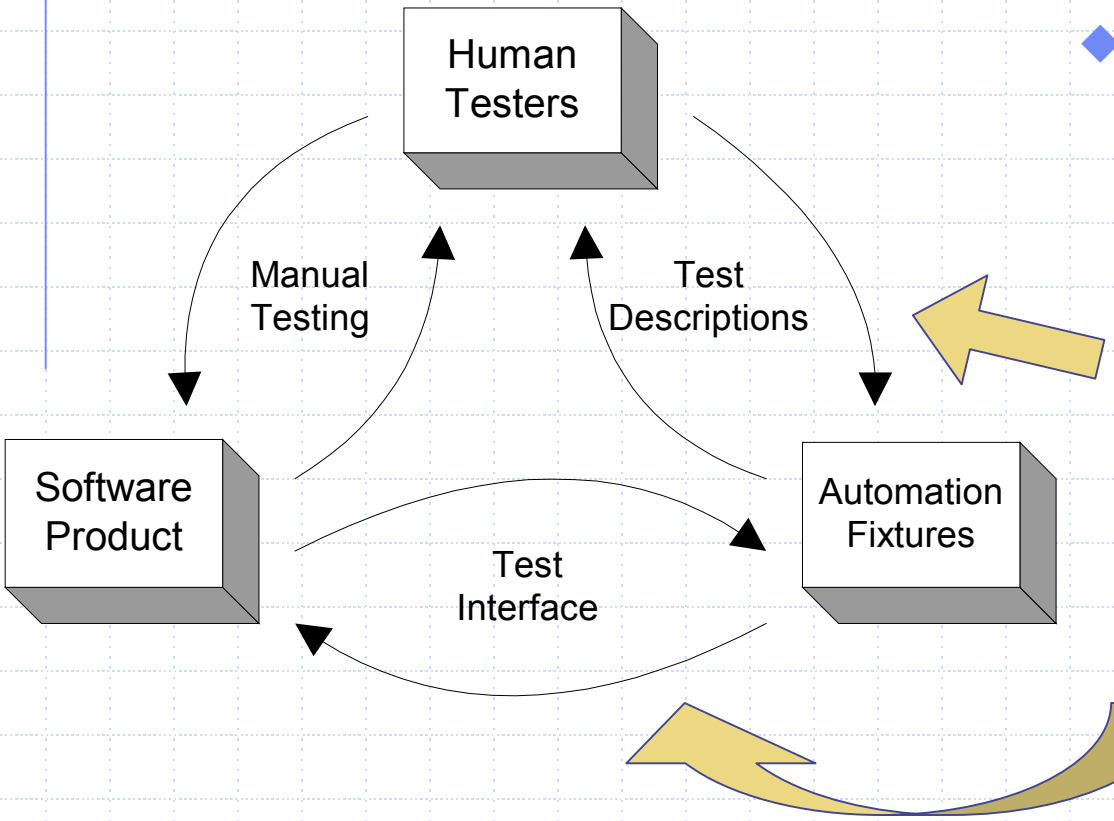
Home-Brew Strategies

- Extending Unit Testing
- Adapting the Product
  - Thin GUI
  - Test Interfaces
- Building Your Own Tool

These strategies are:

- *Used by XP teams*
- *Available to you*
- *More effective when combined*

Test Interaction Model



- Customizing Testing Frameworks
  - For tester usability
  - For product compatibility

Agenda

- Beyond Unit Testing
- Scripting Languages
- Interface Drivers
- Building Your Own Tool
- Test Description Languages

Beyond Unit Testing

- XP has made programmers *love* unit testing
  - JUnit has been:
    - ported to dozens of languages
    - extended for dozens of frameworks
    - incorporated in dozens of IDEs
  - Developers all over are now writing unit tests
- XP leaders are now building tools for acceptance testing…

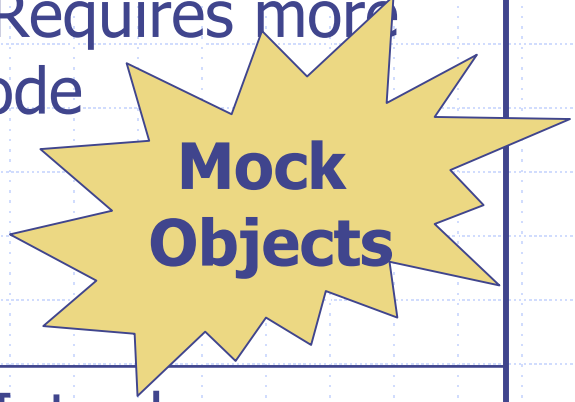
What is Unit Testing?

- Units are functions, methods or small bits of code, usually written by a single programmer.
- Unit tests are written in the same language as the code being tested.
- Unit tests are written by the programmers who wrote the the code being tested.
- A test harness or framework collects tests into suites and allows them to be run as a batch.

Unit Integration Testing

- How to test units that depend on other units?

Unit isolation testing *Test each unit in isolation*	*Create stubs and drivers objects for external units*	◆Requires more code
Unit integration testing *Test units in context*	*Call external units*	◆Introduces dependencies. ◆Test suites take longer to run



Test-Driven Development

- Developers write unit tests *before* coding.
  - Motivates coding
  - Improves design
    - reducing coupling
    - improving cohesion
  - Provides regression tests
- An approach to design
  - More than just as test strategy
  - Specification by Example
  - Refactoring

```
public void testMultiplication() {
  Dollar five = Money.dollar(5);
  assertEqual(new Dollar(10), five.times(2));
  assertEqual(new Dollar(15), five.times(3));
}
```

Home-Brew Ingredients

- Test Harness
  - Collecting tests so they can be executed together
- Language
  - Creating the automation fixtures
  - Providing a language for describing tests

  (These may be the same or may differ)
- Product Interface Driver
  - Giving access to the software product

These are the ingredients of any *automated testing system.*
This talk will discuss languages and drivers.

Three Kinds of Languages

- System Programming Languages
  - Optimized for *performance*.
  - What your programmers are probably using.
  - **C, C++, Java, C#**
- Scripting Languages
  - Optimized for *ease of use* and *high productivity*.
  - Command interpreters facilitate learning and exploration.
  - **Perl, Tcl, Python, Ruby, VBScript, JavaScript, Rexx, Lua**
- Data Presentation Languages
  - Optimized for *readability* and *structure*.
  - No logic
  - **HTML, XML, CSV, Excel**

Scripting Languages for Testing

- **Perl**
  - Well-established
  - Vast **libraries**
- **Tcl**
  - Well-established
  - Compact
  - Popular with **embedded** systems
- **Python**
  - Concise support for object-oriented programming
  - Integrates well with **Java** (Jython)

- Everything's an object
- Principle of **least surprise**

- **Visual Basic & VB Script**
  - Popular
  - Integrate well with **Microsoft** technologies

- Tcl, Python and Ruby have *command interpreters*
- All but VB are *open source*
- All are well supported

The best language is the one your team knows.

Language Choices

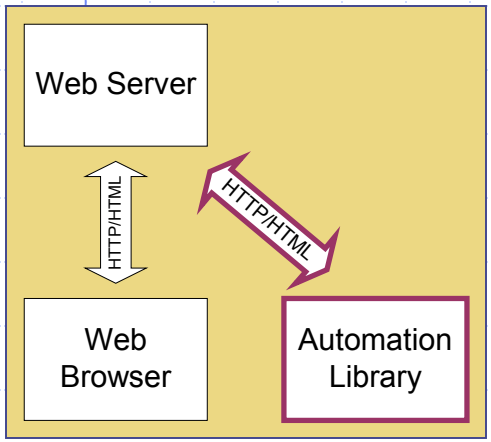
What should you write tests in?

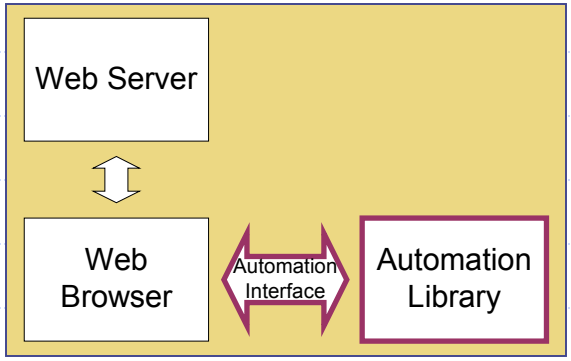
- **System programming language**
  - Reuse unit test harness
  - May result in lower productivity
- **Scripting language**
  - Requires interface to product
  - Allows most kinds of tests
- **Data presentation language**
  - Requires support code in a scripting or system language
  - May improve understandability

- Options
  - 1 only
  - 2 only
  - 1 and 3
  - 2 and 3
  - All three?

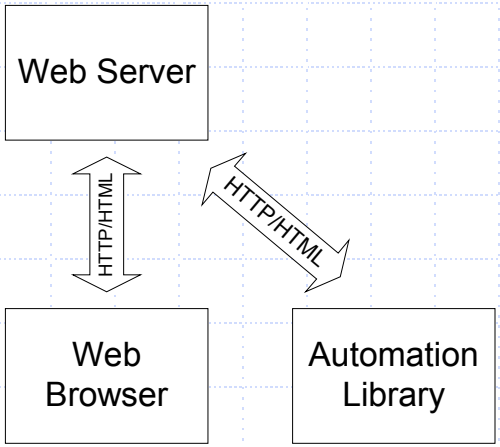
Interface Drivers

- How do your tests access the product?
  - Simulation
    - Access the server in the same way as the client or browser.
    - Use Thin GUI to minimize the untested code.
  - Automation
    - Automate the client or browser using automation interfaces.





Open Source Browser Simulation



- **HttpUnit**, Russell Gold
  - Browser simulation in Java. Popular & well-extended.
  - http://www.httpunit.org/
- **jWebUnit**, Thoughtworks
  - A refinement on HttpUnit and FIT. Java-based.
  - http://jwebunit.sourceforge.net
- **Canoo WebTest**, Canoo Engineering AG
  - Java-based browser simulation with XML.
  - http://webtest.canoo.com
- **TestMaker**, Frank Cohen
  - Python test scripts, Java-based tool. Also simulates non-browser clients.
  - http://pushtotest.com
- **HTTP::WebTest**, Richard Anderson and Ilya Martynov
  - Browser simulation in Perl.
  - http://search.cpan.org/author/ILYAM/HTTP-WebTest-2.00/
- **WebUnit**, Yuichi Takahashi
  - Browser simulation in Ruby.
  - http://www.xpenguin.biz/download/webunit/index-en.html
- **Puffin**, Keyton Weissinger
  - Browser simulation in Python and XML.
  - http://www.puffinhome.org/

More Tools

- http://www.junit.org/news/extension/web/index.htm

HttpUnit Example

```
public void testLoginSuccess() throws Exception {
    WebConversation conversation = new WebConversation();
    String url = "http://localhost:8080/shopping/shop“;
    WebResponse response = conversation.getResponse(url);
    assertEquals(“Login”, response.getTitle());

    WebForm form = response.getFormWithName("loginForm");
    WebRequest loginRequest = form.getRequest();
    loginRequest.setParameter("user", “mike”);
    loginRequest.setParameter("pass", “abracadabra”);
    response = conversation.getResponse(loginRequest);
    assertEquals(“Product Catalog”, response.getTitle());
}
```

Example Courtesy of Mike Clark

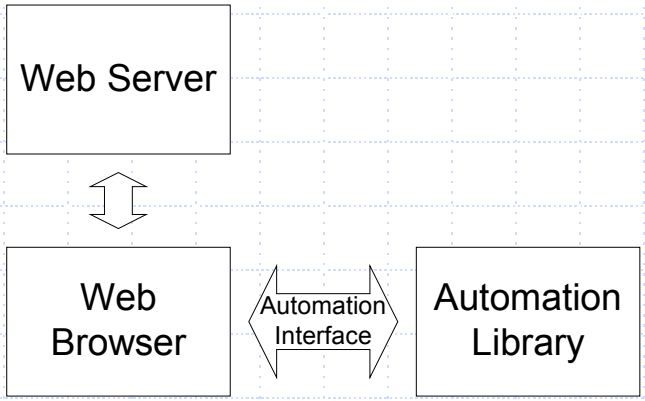
Canoo WebTest Example

```
<project name="ShoppingCartTests" default="main">
    <target name="main">
        <testSpec name="loginSuccessTest">
            <config host="localhost" port="8080"
                protocol="http" basepath="shopping" />
            <steps>
                <invoke url="shop" />
                <verifytitle text="Login" />
                <setinputfield name="user" value="mike" />
                <setinputfield name="pass" value="abracadabra" />
                <clickbutton label="Login" />
                <verifytitle text="Product Catalog" />
            </steps>
        </testSpec>
    </target>
</project>
```

Example Courtesy of Mike Clark

Open Source Browser Automation

- **Cliecontroller**, Chris Morris
  - IE and .Net Windows forms automation in Ruby
  - http://www.rubygarden.org/ruby?IeController
- **Samie**, Henry Wasserman
  - IE Browser automation in Perl
  - http://samie.sourceforge.net/



Open Source Java GUI Drivers

- **Marathon**, Jeremy Stell-Smith et al, Thoughtworks
  - Java Swing GUI driver using Python scripts. Includes a recorder.
  - http://marathonman.sourceforge.net/
- **Abbot**, Timothy Wall
  - Java GUI driver and recorder using XML scripts.
  - http://abbot.sourceforge.net/
- **Pounder**, Matthew Pekar
  - Java GUI driver and recorder.
  - http://pounder.sourceforge.net/
- **Jemmy**
  - Java GUI driver. Integrated with NetBeans.
  - http://jemmy.netbeans.org/
- *More Tools*
  - http://www.junit.org/news/extension/gui/index.htm
  - http://www.superlinksoftware.com/cgi-bin/jugwiki.pl?TestingGUIs

Free Windows GUI Drivers

- **Win32-GuiTest**, Ernesto Guisado
  - Perl Library. Popular. Strong support for various controls.
  - Open Source
  - http://search.cpan.org/author/ERNGUI/Win32-GuiTest-1.3/
- **Win32-CtrlGUI**, Toby Everett
  - Perl library. Strong support for window identification.
  - Open source
  - http://search.cpan.org/author/TEVERETT/Win32-CtrlGUI-0.30/
- **Novell AppTester**
  - API typically called from C++.
  - Distributed as binary as part of Novell's system testing tools.
  - http://developer.novell.com/ndk/softtestv3.htm
- **Bugslayer Tester**, John Robbins
  - Recorder and library written in VB and C++. Well-documented.
  - Provides COM interface supporting VBScript & Jscript.
  - Source and executables published on MSDN.
  - http://msdn.microsoft.com/msdnmag/issues/02/03/bugslayer/default.aspx
- **AutoIt**
  - Recorder
  - Library delivered as ActiveX component (VBScript)
  - Free download, freely distributable. Closed source.
  - http://www.hiddensoft.com/AutoIt/

Other Open Source Test Libraries

- **Expect**, Don Libes
  - Command line driver in TCL. Long-established.
  - http://expect.nist.gov/
- **Android**, Larry Smith
  - Unix/Linux GUI driver in Tcl.
  - http://www.wildopensource.com/larry-projects/android.html
- **Framework for Integrated Test** (FIT), Ward Cunningham
  - Parses tests in HTML. Supports multiple languages.
  - http://fit.c2.com
- **OpenSTA**, Cyrano
  - Web performance testing driver and recorder. Uses a "proprietary" scripting language.
  - http://opensta.org/
- **Avignon**, NOLA Comp Services
  - XML integration with JUnit.
  - http://www.nolacom.com/avignon/
- **STAF**, Charles Rankin, IBM
  - Distributed multi-platform testing framework
  - http://staf.sourceforge.net/index.php

Building Your Own Tool

- It's now easier than it used to be!
  - Use standardized automation interfaces
  - Only support one interface technology
  - Reuse test harnesses and languages
  - Use and extend open source interface drivers

Home-Brew Ingredients

- Test Harness
- Language
- Product Interface Driver

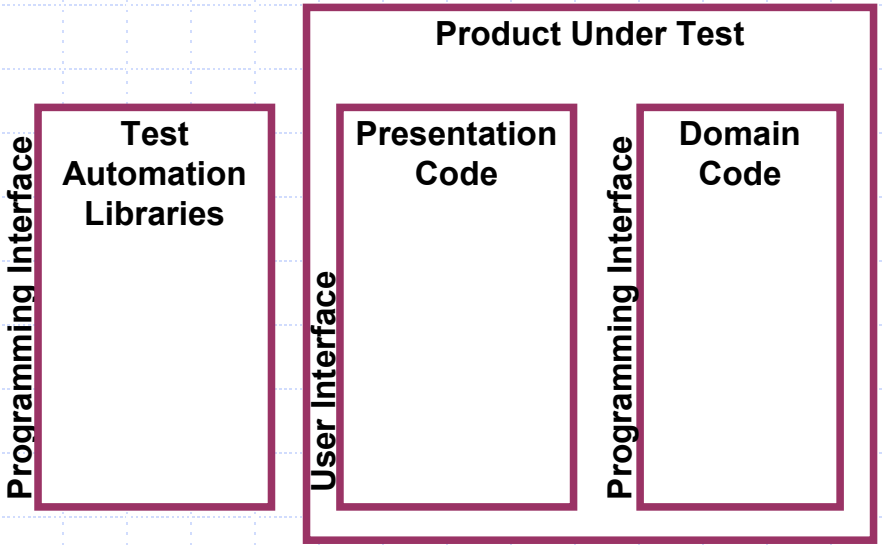
Approaches

- Scripting *Easiest*
- Data-driven
- Capture/Replay *Hardest*

Build a Tool or Adapt the Product?

- Choosing the interfaces will you use for testing is a key strategic decision.
  - **Build a Tool.** Add fixtures and tools to access existing interfaces; or
  - **Adapt the Product.** Expose or create interfaces for direct testing of the product.
  - *This difference is actually rather moot with test-driven development!*
- A sound approach requires close cooperation and trust between testers and developers.

Adapting Your Product



Use Existing Interfaces

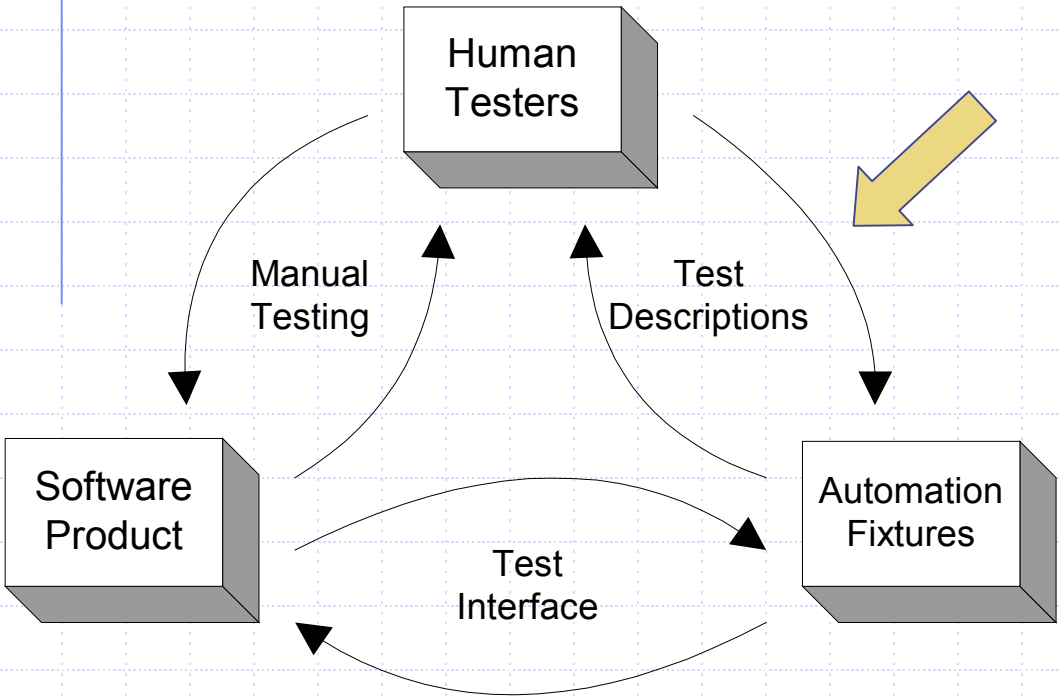
- Products using existing APIs for testing
  - InstallShield
  - Autocad
  - Interleaf
  - Tivoli
- Web Services Interfaces are ideal!

Create New Interfaces

- Products exposing interfaces specifically for testing
  - Excel
  - Xconq (a game!)

Test interfaces provide control and visibility

Test Description Languages



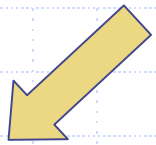
Providing an effective means for describing tests which…

- Testers can create.
- Fixtures can execute automatically.
- Anyone can understand.

Test Description Languages

What is the best test description language for expressing tests?

- Tables
  - Often readable to more people
  - Require fixtures & parsers
- Scripts
  - Better support for variables and looping
  - Require less fixturing



```
Timeclock> start 'misc'
Timeclock> pause
Timeclock> start 'stqe'
Timeclock> jobs
misc, started 02002/08/30 4:32 PM, is paused.
stqe, started 02002/08/30 4:33 PM, is recording time
```

Example courtesy Brian Marick

FIT tests

- Scrape tests from HTML docs
- Keep requirements & tests together
- Check automatically
- Browse results online
- Understandable by everyone

Division shall work with positive and negative numbers.

eg.Division		
numerator	denominator	quotient()
1000	10	100.0000
-1000	10	-100.0000
1000	7	142.8571 *expected* 142.85715 *actual*
1000	.001	1000000 *expected* 999999.94 *actual*
4195835	3145729	1.3338196

Coaching Tests

- Use tests to drive development
- Tests provide:
  - Goals and guidance
  - Instant feedback
  - Progress measurement
  - Health check of the project
- Tests are specified in a format:
  - Clear – so any one can understand
  - Specific – so it can be executed

eg.ArithmeticFixture					
x	y	x + y	x - y	x * y	x / y
200	300	500	-100	60000	0
400	20	420	380	8000	20

Test Automation in the Silo

- Traditionally test automators have worked in a separate space from developers
  - The code is separate
  - The teams are separate
  - *Ex post facto* GUI automation
- Reasons for change
  - Tool/application compatibility (testability)
  - Maintenance when GUI changes
  - Testing needs to be everyone's concern

You can change whether you are are using XP or not!

Automating the Hunt for a Software Regression

Janis Johnson
IBM Linux Technology Center
15450 S.W. Koll Parkway, DES2-02
Beaverton, Oregon 97006
janis187@us.ibm.com

Abstract

A software regression is a bug that exists in some version of software and did not exist in a previous version of that software. Regression testing helps to identify bugs which creep into software with new changes. In most projects, however, test suites don't catch all regressions, and some are discovered by users upon upgrading to a new release. A new bug must have been introduced or exposed by a specific set of changes, and knowing which change introduced a regression can be valuable information for the developer who is fixing that bug.

This paper describes general strategies for identifying the cause of a regression and when it might be practical to automate the search for a particular bug in a particular project. It then describes the use of automated regression hunts for the GNU Compiler Collection, including background information about the project and its development process. The techniques and tools described here can, in theory, be used by a variety of projects.

Biographical sketch

Janis Johnson is a contributor to the GNU Compiler Collection through her job with the IBM Linux Technology Center. She has over 20 years of experience developing and testing compilers and other software development tools while working for a variety of computer manufacturers, including Cray Research and Sequent Computer Systems. Janis has an MS in Computer and Information Science from the University of Oregon.

Automating the Hunt for a Software Regression

Janis Johnson
IBM Linux Technology Center
janis187@us.ibm.com

1 Introduction

A software regression is a bug that exists in some version of software and did not exist in a previous version of that software. Regression testing helps to detect bugs which creep into software with new changes. In most projects, however, test suites don't catch all regressions, and some are discovered by users upon upgrading to a new release. Knowing which change introduced a regression can be valuable information for the developer who is fixing the bug. For many projects it's possible to automate searches for the causes of regressions, allowing searches to be done with less effort and higher reliability. Experiences of automating regression hunts for the GCC project are described, along with an introduction to the project and reasons why it's important in a distributed Free Software project to make it easier for experienced developers to fix bugs.

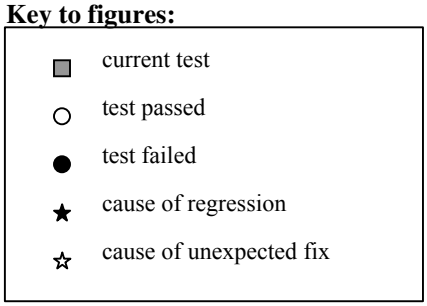
2 General strategy

The general strategy for identifying the cause of a regression is the same regardless of the type of software or the source control system used for it. A regression hunt uses a binary search by patch set identifier, if the source control system supports that, or by date. The first step is to identify the range of dates to search, from the latest date at which the test case is known to pass to the earliest date at which it is known to fail. The next step is to verify that the test case gets the expected result for both of those dates when the product is built and run using the procedures which will be used for the hunt. Starting with the original range of dates or patches, the mid-point of the range is tested; depending on the result of the test, that date becomes one of the new end-points of the range. This process is repeated until a single patch or a small time interval is identified. Each check of a date or patch updates the source code, builds the software, and runs the test case.

Folk wisdom says that when asked to identify the most powerful force in the universe, Albert Einstein replied "compound interest." Binary searches are similarly powerful. To verify that a test result changes in one five-minute period requires a check at each of the two end-points of the range. Finding the five-minute period within a ten-minute range requires one additional check at the mid-point of the range. When the initial range is 24 hours, eleven checks are needed including two for the end-points. For an initial period of a year the total number of checks is 19, fewer than twice as many as for a single day.

The example in Figure 1 shows the progress of the binary search for a regression. The first two tests are for the end-points of the range; each further test is for the mid-point of the current range, with the version just tested becoming either the low or high point of the next range.

Key to figures:



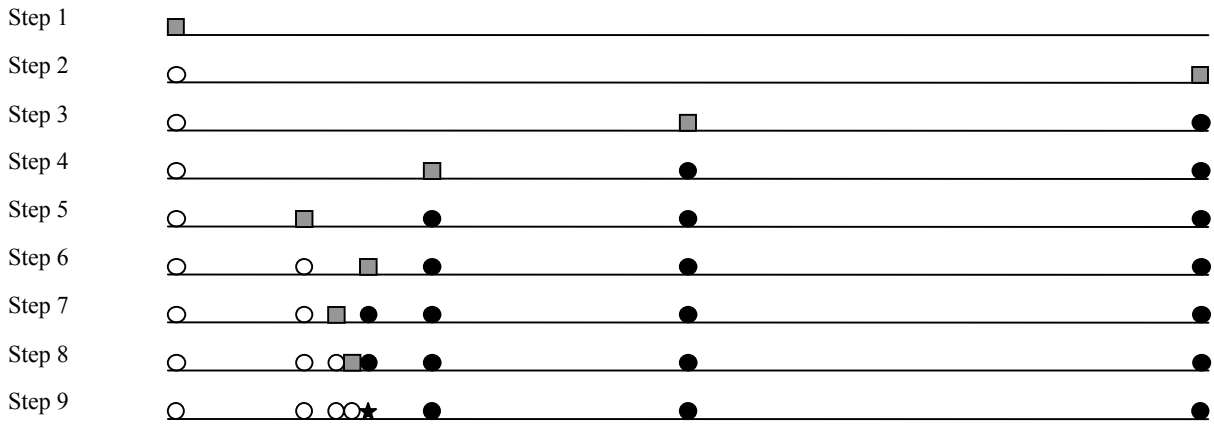


Figure 1

Checking each patch is not practical, but for illustration purposes if each patch were tested to find the regression whose search is shown above, the results would look like Figure 2:

Figure 2

The same strategy can identify the fix for a bug that unexpectedly went away. If the code that was modified is similar in both versions of the software and the identified patch doesn't merely paper over the problem, then it can be ported to the branch on which the bug still exists.

The decision about which range to test is more difficult when a project uses branches, and a bug exists on one branch and does not exist on another. In Figure 3, the bug exists on a release branch but not on the mainline, and there is a version with known good test results on the mainline before the branchpoint:

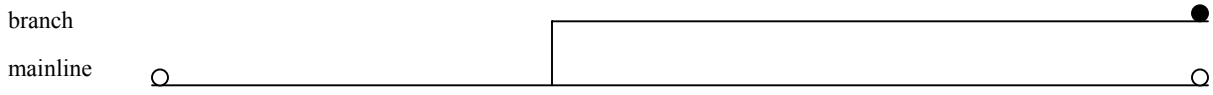


Figure 3

Testing all patches for both branches might show that the regression was introduced only on the branch, as shown in Fugure 4:

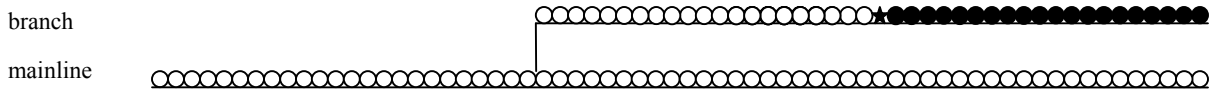


Figure 4

For another bug whose initial information was identical, testing all patches for both branches might show that the regression was introduced on the mainline before the branchpoint, and the bug was then fixed on the mainline but not on the release branch, as shown in Figure 5:

Figure 5

The strategy in this case, then, is to test the mainline just before the branchpoint to determine whether to perform a full search on the branch for a regression, or on the mainline for both a regression (between the lowest date for which the test is known to pass and the branchpoint) and an unexpected fix (between the branchpoint and a date known to have the fix).

Development branches are split off from the mainline sources and are later merged back in. A regression hunt on the mainline that identifies the merge as the cause of the regression can be followed by a new search on the development branch to determine which patch to that branch caused the regression, as in Figure 6:

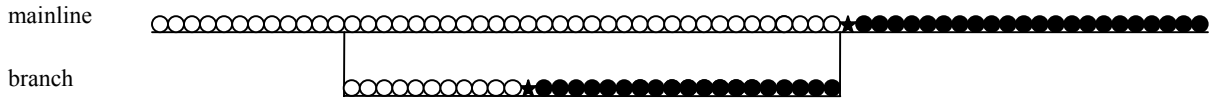


Figure 6

3 Automating regression hunts

3.1 Requirements for automating a regression hunt

Basic requirements for automating a regression search involve the source control system, the product build, and the test case.

The source control tools must allow accessing all source files as they were on a particular date or when a particular patch was added. If the project uses multiple branches of the sources, then searching for a regression on a branch requires accessing the sources on that branch as of a particular date or patch addition. This is easier if the source control system operates on patches rather than changes to individual files, but it's sufficient to be able to determine which files were modified for the same change even if that information is not built into the tools. The search for the cause of a regression involves accessing many sets of sources, so it must be possible to obtain each version without adversely affecting the productivity of other people who use the

same source repository.

Automated regression hunts are possible only if the product build is automated and can be performed in a reasonable amount of time. For a particular regression, a hunt is possible only if there is a test case that fails when the bug is present and passes when it is not and if that test case is automated, self-contained, reasonably quick, and safe to run, e.g., it won't corrupt the test system. A regression hunt also requires access to a test system on which the failure can be reproduced.

3.2 Short cuts

Short cuts are often available to reduce the time needed for a regression hunt. When the remaining interval is small it's often possible to look through descriptions of the patches that were made during that time and identify the one most likely to have caused the regression, and then test the software just before and just after that patch was added. If the guess was wrong, the hunt can continue with that date or patch as one of the end-points.

A hunt based on dates can recognize when there have been no changes to the sources between the new date and one of the current end-points of the range, and then skip the test for that date and simply make it one of the end-points.

For some source control systems it's possible to make a local copy of the repository to avoid or lessen network overhead and to leave out portions that are not needed for the hunt, further speeding up the process of obtaining a particular version of the sources.

The build of the product need not be a complete build; it can produce only the components that are needed to run the test. Depending on the build process it might be possible to use incremental builds rather than starting from scratch each time.

Some short cuts are successful most of the time but fail occasionally, such as incremental builds if not all dependencies are identified correctly. A hunt can try a short cut and then fall back to the normal procedure if the short cut fails.

Guessing to start with a range that is as small as possible is not much of a short cut, since an incorrect guess requires additional work by the regression hunter, while doubling the initial range requires only one additional check.

3.3 Benefits of automation

Once the steps of obtaining a specific set of source code and building a subset of the product are automated, those tools can be used in a fully automated regression hunt. The first few hunts generally require adjustments to those tools, but once they are stable the incremental time required to set up additional hunts is fairly low. The actual work of an automated regression hunt consists of setting up the test case and a script to build the right subset of the project, selecting an initial range of dates or patches, verifying that the hunt reports the expected results for those initial dates, monitoring results occasionally to detect problems that require manual

intervention, and then examining and reporting the results at the end.

Performing a regression hunt manually is error-prone. If there is a time lag between steps, the person performing the hunt can become distracted by other tasks and get confused about just what is being tested in each step. An automated hunt whose tools create log files recording what is done at each step makes it easier to check back on what was done and recognize mistakes, as well as reducing opportunities for careless human error during the hunt itself.

Personal experience has shown that incorrectly identifying the cause of a regression can be embarrassing, and might even lead to bad haiku:

> Where's the regression?
> I thought it was located;
> alas, I was wrong.

3.4 Manual intervention

Several problems that require manual intervention can arise during an automated regression hunt. Mistakes in setting up configuration files or test scripts can cause test results to be reported incorrectly, which makes it doubly important to begin the hunt by verifying the results for the original end-points of the range, including examination of test output to verify that a test failed for the expected reason and not because the test was set up incorrectly. There might be dates for which the product doesn't build; this is frequently the case in the GCC project, which supports a wide variety of configurations, not all of which are tested frequently. Sometimes the test output changes due to modifications that have nothing to do with the bug, such as changes to messages that the test script examines. The build process might have changed over time, requiring different scripts to perform the build for different versions of the source.

3.5 Verification of results and intermittent failures

If an automated regression hunt uses any kind of short cuts or makes assumptions that are not always true then it's a good idea to verify that the bug does not yet exist just before the identified patch and that it does exist after that patch. It's important to examine the identified patch to see if it makes sense for it to have caused the bug; if not, the bug might be intermittent. A binary search for 1 or 0 will always converge, whether the results make sense or not. Figure 7 shows what it would look like to test all patches in a range which contains a normal regression, followed by what it might look like to test all patches in a similar range which contains an intermittent bug, and the checks and results for a binary search which could match either of those patterns (this is the same binary search which was illustrated earlier).

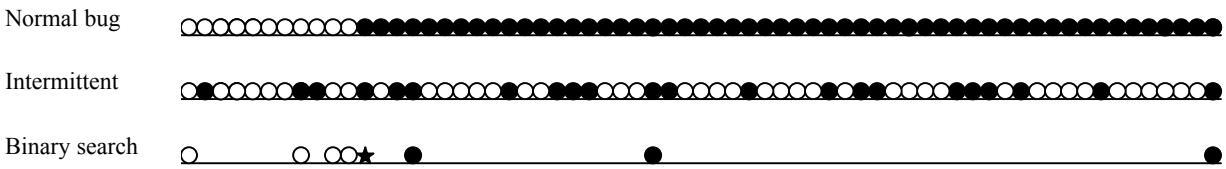


Figure 7

Testing at regular intervals can show whether a failure is intermittent, depending on how often it fails and whether the periodic testing is lucky and shows the irregularity of the results. In Figure 8, the first set of periodic checks uses a large interval that happens to show the same pattern as would be seen for a regression, while the two sets of checks with smaller intervals demonstrate the intermittent failures:

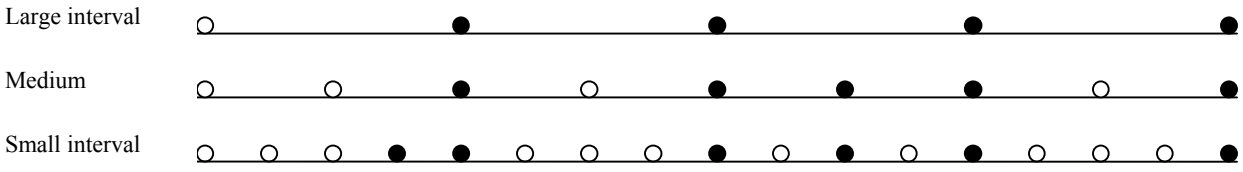


Figure 8

4 Hunting regressions in GCC

4.1 The GCC project

The work described here was performed within the GCC (GNU Compiler Collection) project. GCC is part of the GNU Project, which is Free Software covered by the GNU General Public License, with copyrights held by the Free Software Foundation [1]. GCC includes compilers for C, C++, Fortran 77, Java(TM), Objective-C, and Ada, and supports a wide variety of processors and several operating systems, including embedded systems. GCC is the primary compiler used for GNU/Linux and other Free Software and open source operating systems and is the system compiler for Mac OS X. As an example of the variety of targets GCC supports, in the first two weeks after the release of GCC 3.3 there were reports of successful native builds for 27 unique targets, with a combination of 16 processors and 10 operating systems [2].

Unlike proprietary or in-house software projects, GCC is developed and maintained by a large number of contributors who are either volunteers performing the work on their own time or people whose work on GCC is funded by their employers or through contracts. Twelve global maintainers can make or approve changes to any part of the project, a few dozen additional maintainers can make or approve changes to specific parts of the project, and another hundred or so people have write access to the sources, allowing them to check in their changes after approval by a maintainer. Many other people contribute patches but are not able to check them in themselves. This development model can be very surprising to people who have not been exposed to it, but it has been successful for many Free Software and open source projects [3].

GCC contributors tend to focus primarily on current development; volunteers work in areas that are of personal interest to them, and funded contributors do work that is a priority to their employers. Stabilizing an upcoming release receives adequate attention, but fixing regressions for bug-fix releases is not a particularly popular activity. GCC's Release Manager can encourage contributors to fix bugs but has no real authority over them; from his perspective they are all volunteers.

GCC is a large, complex project [4], and many GCC bugs can only be fixed by an expert who has a deep understanding of the interactions between the various parts of the compiler, which is

usually gained through of years of experience. Other contributors who want to help get bugs fixed can often provide the most support by providing information and analysis that will make it easier for the experts to fix bugs.

4.2 GCC processes

Source control

The GCC project uses CVS [5] for source control. CVS handles changes to individual files rather than patches that modify a set of files. A CVS commit of several files is not atomic, although it does support providing a list of files that are usually checked in within a very short period of time. By convention, each GCC patch is described in a ChangeLog entry that lists all of the files that are modified for that patch; the same information is included in the CVS log of each of those files. These conventions allow a human being to recognize patches even though automated tools only know about changes to individual files.

CVS supports multiple branches in a single source tree, and the GCC source tree includes several branches. The mainline development branch is used for new functionality that is ready for use by the GCC development community. A major release gets a branch when all new functionality for that release is in the mainline and is relatively stable; that branch continues to be used for later bug-fix releases. Experimental work is often performed on a development branch that is merged back into the mainline when it is stable; CVS tools support automated merges of mainline changes into a branch, and GCC development branches generally get regular merges from the mainline.

Figure 9 shows the mainline, release branches, and representative development branches of the FSF GCC CVS tree between January 2001 and June 2003. The branches shown above mainline are release branches, with a star showing a release. The branch for 3.2 releases is unusual in that it is actually a continuation of the 3.1 branch, but an ABI change that broke binary compatibility for C++ required the use of a version number showing that there was a major change between 3.1.1 and 3.2. Branches shown below mainline are development branches that were merged back into mainline. During this time there were several additional development branches that are not shown here.

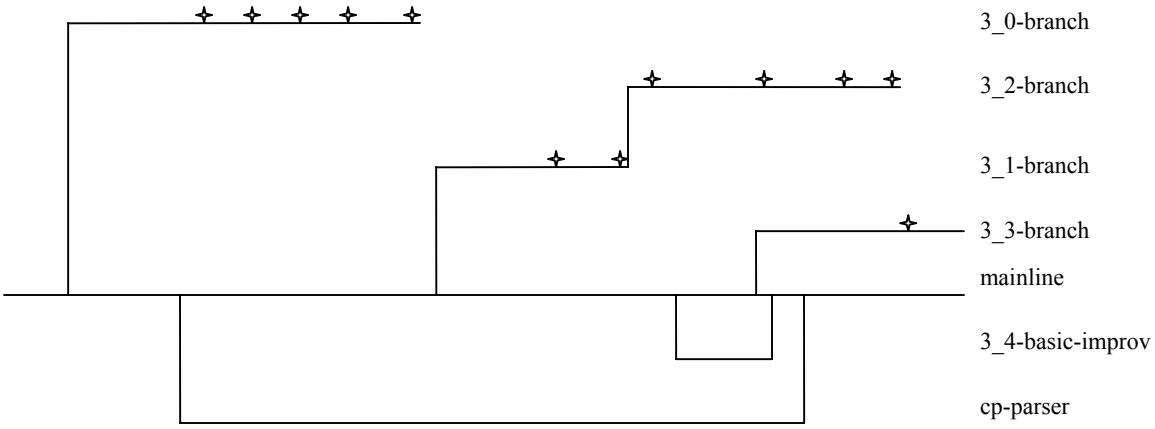


Figure 9

Development process and regressions

The GCC development process [6] specifies that bug-fix releases only fix regressions. This makes it important to identify which bugs are regressions so they will be fixed in a bug-fix release.

When a patch to the mainline causes a significant regression, such as a build failure on a major platform, that patch must be reverted if the regression is not fixed within 48 hours. Often the cause of a new failure is not obvious and must be hunted down.

Bug tracking

GCC's bug tracking system is available for anyone to view via the GCC home page, and anyone can submit a problem report against the FSF GCC sources. The instructions for reporting a bug [7] specify that a problem report must be accompanied by a self-contained test case and enough information about the software environment to allow GCC contributors to reproduce the problem.

Problem reports are analyzed by volunteers. Currently there is a dedicated group of volunteer "bug masters" who do this very quickly. A volunteer determines whether a bug report is valid (is not based on a misinterpretation of the rules of the language or some other user error). He or she determines whether the bug still exists on active branches and whether it is a regression from a released version of GCC. Some of these volunteers provide minimized versions of the test cases that are submitted with the problem reports; it's not unusual for a talented bug minimizer to cut down a 30,000-line test case to 10 or fewer lines of code that trigger the same compiler bug. In addition to being extremely valuable to people who fix bugs, the minimized test cases can be used in regression hunts.

4.3 Types of compiler bugs

Compiler bugs come in a wide assortment of flavors: incorrect code generation causing the compiled test case to abort or to produce incorrect output; internal compiler errors for valid or

invalid source code input; an undetected error or missing warning, or an incorrect error or warning message. Each of these requires a source file to compile, perhaps a specific set of compiler options, and a test script which recognizes whether or not the test case was compiled correctly.

4.4 Tools for hunting regressions

GCC contributors have been identifying the causes of regressions for a long time, but until recently the methodology was not described and there was uncertainty about whether it was even possible to obtain the sources from a CVS branch for a particular date. In December 2002 there was a big push among the "bug masters" to minimize test cases, determine which bugs were regressions, and in general provide as much useful information as possible to the people with enough experience to actually fix the bugs. There was a concerted effort at this time to identify the causes of regressions, and the people involved agreed to share tips and to document them [8]. During this effort it became clear that it is possible to automate regression hunts.

A shell script called `reg_search` was developed at this time to provide the framework for automated regression hunts. Most GCC developers prefer to use their own scripts to obtain sources and to build GCC, so `reg_search` invokes separate tools to perform those functions. The various kinds of compiler bugs require different kinds of tests, so the script that runs a particular test merely returns a value of 1 if the search should continue with later dates, or a value of 0 to search earlier dates. A configuration file specifies the locations of these scripts plus the low and high dates of the range to search and the desired length of the final period to report (5 minutes by default).

GCC runs on systems where other GNU tools are either available by default or are easy to build and install, so `reg_search` takes advantage of a nonstandard extension to GNU `date` to format dates as the number of seconds since 1970-01-01 00:00:00 UTC [9]. `reg_search` converts all dates to this format, making binary searches by date straightforward.

`reg_search` supports continuing a search after manual intervention has been required. When a source update or build fails, `reg_search` reports the low and high dates of the range with which the search should continue. If GCC fails to build for the tested date, the configuration file can specify an initial mid-point to test rather than using the actual mid-point of the current date range. By default `reg_search` begins by verifying the initial low and high dates, but these checks can be skipped if they've already been verified by hand or in an earlier stage of the search.

A related shell script, `reg_periodic`, builds and tests at regular intervals in a specified range of dates. This is useful when the regression hunter suspects that a bug might be intermittent, to find out whether the test result varies over time. Running a test at regular intervals is also useful when hunting for a performance regression for which there might be several small performance degradations rather than a single large one.

`reg_search` and `reg_periodic` are available in the GCC source tree in a directory of contributed tools [10]. They are copyrighted by the FSF and licensed under the GNU General

Public License. Besides depending on an extension to GNU `date`, they use features of `bash` (GNU Bourne-Again SHell) [11] that are not available in other shells. These scripts contain no assumptions that they are being used for the GCC project or for a compiler; they can be used for any project which requires searches by date rather than by patch set identifier.

4.5 Benefits to GCC

Between December 2002 and April 2003 there were 32 regressions which affected the then-upcoming GCC 3.3 release whose causes were identified and the bugs fixed. During the same period, 85 other regressions were fixed for the GCC 3.3 branch. Most of these were regressions which were introduced during the development of that branch, but some also affected released versions of GCC. One might have expected that the regressions whose causes were identified were fixed sooner than the others, but that didn't turn out to be the case. The people who fixed those bugs, however, agreed that in general knowing the cause of the regression was valuable, although having a minimized test case is even more helpful than knowing the cause of the regression. None of these people had fixed enough regressions with identified causes to be able to quantify the time savings.

The value of knowing which patch introduced a regression varies widely depending on several factors, including the nature of the bug, the size of the patch, the familiarity of the bug fixer with the modified code, and whether the patch introduced a new bug or merely exposed an existing latent bug. In some cases, a close examination of the identified patch reveals a mistake or omission in logic allowing the bug to be fixed easily. If the identified patch was a large change for new functionality or a rewrite of a portion of the compiler, at least the fixer knows in which part of the compiler a bug might lie. Identification of a patch that exposed a latent bug allows examining debugging dumps of compilers with and without the change or setting up simultaneous debugging sesssions of those two versions of the compiler to determine at which point the behavior changed. Zachary Weinberg, a GCC global maintainer, describes his bug-fixing approach and how it is affected by knowing the cause of a regression:

> The procedure I follow when chasing a bug (regression or otherwise) is to start with the test case, identify what has gone wrong - in most cases this is obvious from assembly or RTL dumps - and trace a causal chain backward through the compiler, by running it repeatedly under the debugger. Once I find the ultimate cause, then I try to figure out the intent of the author of that piece of code. There is often an explanation in the gcc-patches message, if available, so knowing that is helpful. However, the patch which *exposed* the bug (which is what will show up in the regression hunt) is often not the patch which *introduced* the bug. Therefore, even if I know the former, I still have to do the causal-chain trace to find the real culprit; but it can make the process shorter [12].

Another benefit of identifying the cause of a regression in a volunteer project like GCC is that regular contributors are more likely to look at problem reports for bugs which they have introduced. In some cases that's when an expert notices that the code in the test case is ill-formed, which is particularly common for inline assembly code and for C++ code which the compiler has rejected. GCC's Release Manager is able to use the information about which patch introduced a regression to ask the author of that patch to investigate and fix the problem.

5 Conclusion

Automated regression hunts are, in theory, possible for many projects, and their results are less prone to error than are the results of manual regression hunts. The value of identifying the patch which introduced a regression varies widely, but for the GCC project there have been enough cases where the information is useful to make regression hunts worthwhile.

6 Acknowledgements

Most of this paper is based on the author's experiences hunting for the causes of regressions in GCC and developing `reg_search` and `reg_periodic`. Craig Rodrigues spearheaded the initial push to track down causes of regressions in GCC. Wolfgang Bangerth and Volker Reichelt provided many ideas for the tools and suggested regressions to hunt; their minimized test cases were invaluable. Richard Henderson, Jason Merrill, Mark Mitchell, Nathan Sidwell, and Zachary Weinberg provided insights about the value to a bug fixer of knowing the cause of a regression. Dorothy Bayern provided valuable review comments.

Legal Statement

References

[1] Richard Stallman, "The GNU Project", http://www.gnu.org/gnu/thegnuproject.html.

[2] Build status for GCC 3.3, http://gcc.gnu.org/gcc-3.3/buildstat.html.

[3] Eric S. Raymond, The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary, O'Reilly, 1999.

[4] Zachary Weinberg, "A Maintenance Programmer's View of GCC", Proceedings of the GCC Developers' Summit, 2003, http://wwwlinux.org.uk/~ajh/gcc/gccsummit-2003-proceedings.pdf.

[5] Per Cederqvist et al, Version Management with CVS, http://www.cvshome.org/docs/manual/index.html.

[6] GCC Development Plan, http://gcc.gnu.org/develop.html.

[7] GCC Bugs: Reporting Bugs, http://gcc.gnu.org/bugs.html.

[8] How to Locate GCC Regressions, http://gcc.gnu.org/bugs/reghunt.html.

[9] GNU Core Utilities, http://www.gnu.org/software/coreutils/.

[10] http://savannah.gnu.org/cgi-bin/viewcvs/gcc/gcc/contrib/reghunt/#dirlist.

[11] Free Software Foundation, Bash Reference Manual, http://www.gnu.org/manual/bash/index.html.

[12] Personal communication with Zachary Weinberg.

XML/STREAM-BASED AUTOMATION

James T. Heuring

Micro Encoder, Inc.
11533 NE 118th Street
Building M, Suite #200
Kirkland, WA 98034

JTHDEV@AOL.COM

Abstract

This paper describes a system using a lightweight interpreter and XML to perform automation tasks. Simple object-oriented instructions are fed into the interpreter to perform the actual automation. XML can be used to store the instructions with an XML schema (XSD) to validate the instructions and provide a structure for a tester to create the automation.

On top of the interpreter is a user interface to manipulate the XML form of the automation. The XSD defines the syntax of the language and is used to provide a way for the user to discover the objects used by the system with their properties and methods. In addition, the program's resource files are parsed and presented by the user interface so that the user can construct automation and instructions that can run on the system being tested.

This system provides a new approach to automation in general.

Keywords

XML, XSD, Automation, Stream-based, Localization testing, Object-Oriented Interpreter, C++.

A glossary is included at the end of the paper.

About the Company

Micro Encoder Inc., a subsidiary of Mitutoyo Corporation of Japan, is a state of the art research and development facility for computer aided measurement technologies. The software developed here, QVPAK®, uses the advanced sensor hardware developed here to perform measurements used to improve manufacturing processes.

About the Author

Jim Heuring has worked as a software engineer since 1985. This experience includes the development and testing of software for medical devices, avionics and software tools. Other experience includes over 3 years as a tester at Microsoft on Exchange (the mail software) and 9 months at Microsoft as a developer on the data access team where he wrote the data access install verification tool called "Component Checker". Jim is currently a Senior Software Test Engineer at Micro Encoder.

Business needs that initially drove this effort

The QVPAK® has over 300 user interface components including dialog boxes, menus, etc. The product is localized into 10 different languages. As a result we have over 3000 user interface components to support and test. For each user interface component we need to perform the following operations:

1. Provide bitmap images for the localizers to confirm

2. Provide bitmap images for the testers to validate user interface layout and consistency

3. Provide the technical writing team with a user interface product map

4. Validate that the automation can access all the product functionality utilizing the user interface.

5. Provide example automation for training

1. A new approach to automation and test system development

Automation the old way. As automation engineers we have what might be considered two sets of tools. The first set of tools are the code-based ones like PERL, SED, and AWK. These tools are powerful; but have steep learning curves and are frequently hard to read and maintain.

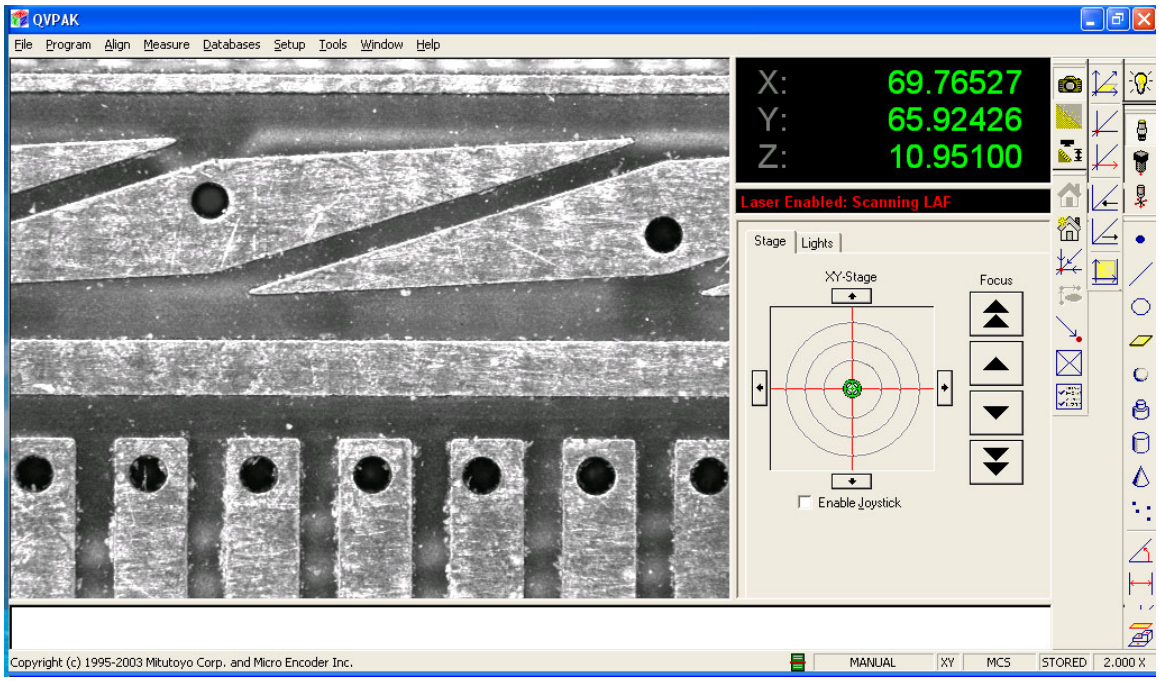
The second set of tools are the GUI based tools such as Visual Test, Silk Test, etc. These tools lock you into their framework because each has their own language elements and structure that, realistically, make it impossible to change to a different system.

Much of the automation developed at Micro Encoder before the tool described in this article was written in Visual Test. However, when our product, QVPAK, was ported to Win 2000; the Visual Test scripts failed to run properly. The owners of Visual Test, Rational, suggested that we move to their beefier test system. However, this would have required a lot of work and was judged to be too expensive.

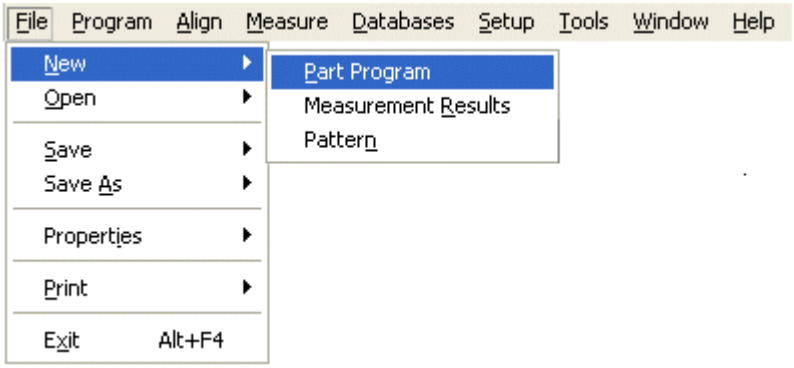
A new approach. This forced Micro Encoder to develop a new approach to testing, outlined below, that can help you to develop automation that is not made obsolete by changing technologies, nor tied into specific test tool languages and implementations.

2. Walkthrough of a simple scenario

Objective: Capture bitmap images of all the user interface items including dialog boxes and menus. A QVPAK image is shown below:



QVPAK can be used to measure many things, for example, auto parts, circuit boards, etc. Most often the user wants to create a program that can be executed repeatedly to measure many parts. This program can be called a “Part Program.” To perform this operation we simply choose the following menu items: “File.New.Part Program” where the “.” indicates that you select the sub-menu. Below you can see the menu items in series:



A stream is a set of instructions that are used to perform a given operation, in this case, the stream would be:

Menu.EnglishString=&**File**
Menu.Select
Menu.EnglishString=&**New**
Menu.Select
Menu.EnglishString=&**Part Program**

Menu.Select

This stream is a text file that contains object-oriented instructions. An object is a piece of code that can perform a set of operations based on its properties. In this case we are using the “Menu” object. The Menu object has a property “EnglishStrings” and an operation “Select”.

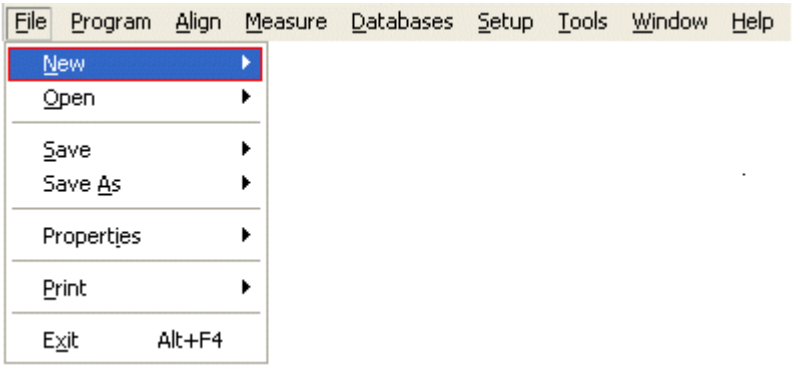
Benefits. By keeping the instruction syntax simple, the design, implementation, and maintenance of the parser is kept to a minimum. The instructions are text that can be easily generated.

An interpreter reads the object stream one line at a time. The interpreter performs the operations based on the properties of the object as show in the following example:

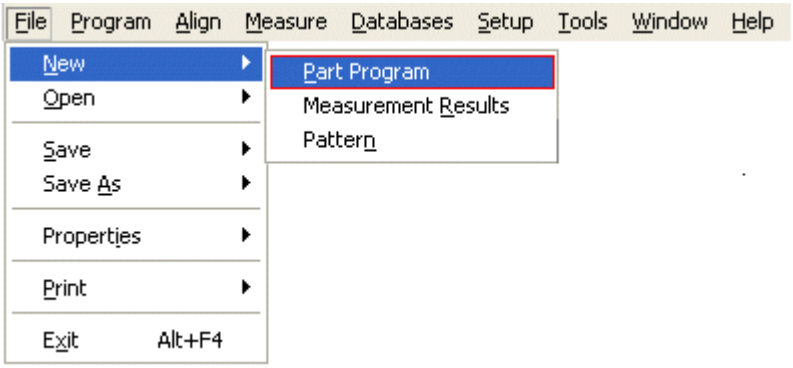
Note: a box is put around the selected menu item.

Menu.EnglishString=&**File**
Menu.Select

Menu.EnglishString=&**New**
Menu.Select



Menu.EnglishString=&**Part Program**
Menu.Select



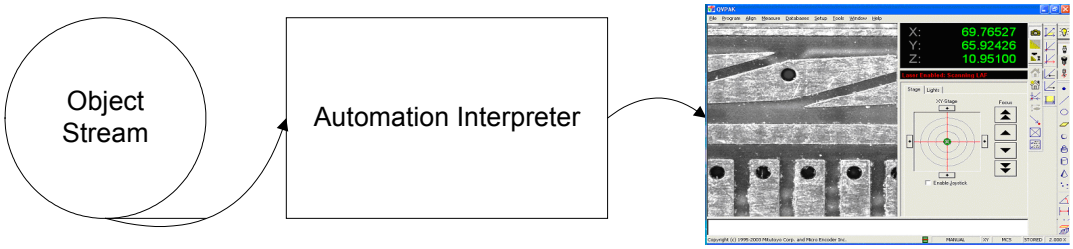
Note: the test tool has a “ScreenShot” object that captures bitmap images of the menus that we have shown above. The ScreenShot operations are performed against each new menu item. No example is given in this paper for brevity sake.

Benefits. Every time the interpreter reads an instruction, we have an opportunity to perform additional operations. This provides a fine level of granularity in which to inject additional processing. For example, we can have the interpreter capture the screen image whenever a

menu operation is performed. This can be further extended to allow the capture of any user interface gesture in sequence. These bitmap images can in turn be used to create a running visual display of operations. You can review what happened just up to the point where an error occurred. This can be useful in debugging the target system, as well as the automation. Other kinds of processes that you might want to insert are memory utilization monitoring, checks for heap corruption messages, etc.

3. More details: stream-based part of the automation

The Automation Interpreter waits for instructions to perform. When new instructions are detected Automation Interpreter reads the stream and performs the requested operations against the system being tested. As diagramed below:



Benefits. The interpreter is written in C++ and utilizes a minimum number of framework components. Framework components, such as the MFC require support DLLs that need to be installed and there are sometimes problems with certain versions or mixtures of DLLs that can have problems. In addition, frameworks have a habit of being abandoned, obsoleted, or changed.

Benefits. The interpreter can be directed to read from any source that supplies text files. This includes: files on a machine, a network server, even TCP/IP that allows the interpreter to be run remotely. The interpreter can be fed instructions that are simply text files or from other programs that generate the instructions. A program that permutes the instructions in a certain way can be written to feed these instructions to the interpreter. Example: there of several ways to access a given user interface item. You can access a menu item by directly selecting it using the mouse or by using accelerators, “&F” in “&File”. A program could be written that runs a given set of automation with one preferred access method, say, MOUSE, then changes the preferred access method to say, ACCELORATOR.

3.1 Store the instructions in a database

Benefits. High-level logic is less likely change as there are no dependencies on other "high-level" automation products. It is easier to maintain a database in the case of searching, making global changes, etc.

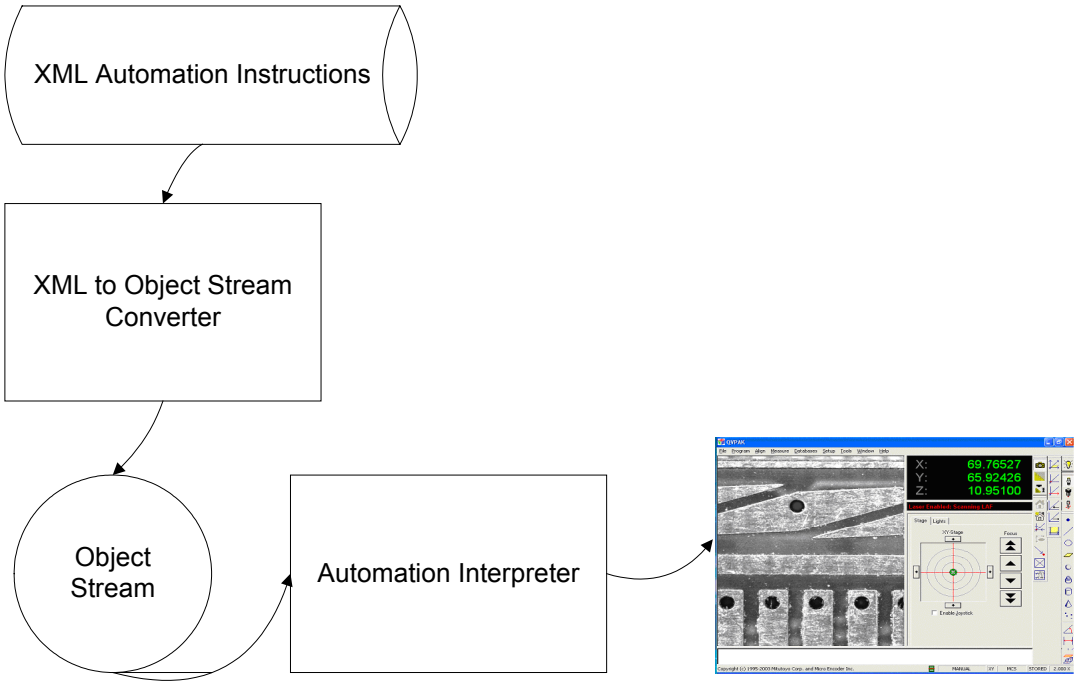
3.2 Use XML as your database

Benefits. By using XML to store the automation, you can avoid being locked into a proprietary database format that might not be properly supported now or in the future. There are many vendors of XML support software to choose from. You can even write your own.

Pitfalls. Because of the many vendors of XML support software, the decision can be quite difficult. I have been using the Microsoft XML DOM version 4 and have encountered many problems documented in section 5, “Lessons Learned”. In general, the technology seems a bit

immature as evidenced by the bugs in the support software and disagreements of the vendors over the XML specification.

The automation instructions can be generated from XML that contains the logic to perform the operations. The XML is read by an XML reader and is written out into the stream that is read by the Automation Interpreter. This process is diagrammed below:



Use the database schema for object construction and validation.

Some database languages have a mechanism for defining the structure of the data you are storing. This structure is often called the database “schema”.

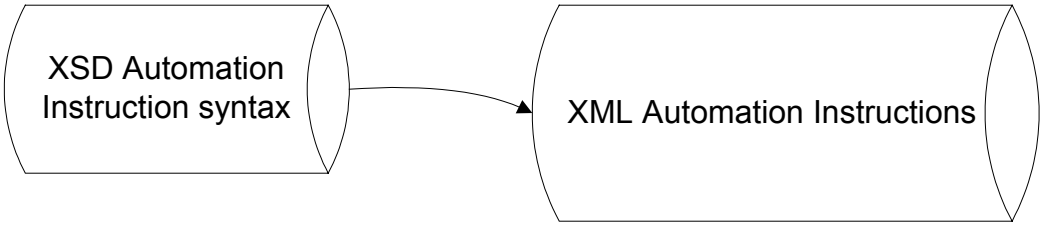
Benefits. Use the database schema (structure definition) to validate your instructions and to provide a structured way to create automation.

3.3 Use XML schema (XSD) as your schema

Benefits. The schema defines the objects and their properties. This information is used for populating the user interface. See benefits for XML to store the database.

Pitfalls. One might consider that the features of the schema are more advanced than those of the XML itself. As a result you are more likely to find bugs in the access software when using the more advance features. Also, see pitfalls for XML to store the database.

The XML Schema contains information about the syntax of your objects that allow testers to easily write valid tests. We choose to use the XSD format to store the schema. Currently, there are more than 20 objects that you can use in our system. These include objects to manipulate the Windows user interface and programming objects such as timers and objects to construct more advanced sets of instructions. To discover the object's properties and operations, we store their structural definition in an XSD file. The following diagram shows the relationship between the XSD and XML.



In essence, the XML is an instance of the XSD data/class. The XSD file is used to validate the instructions being added to the XML file. The XSD file is also used by the automation editor to assist the automation author using a graphical tool.

Below we have a snippet of an XSD file that defines a portion of the menu object.

XSD example:

```
<xs:element name="Menu">
      <xs:complexType>
            <xs:attribute name="EnglishString" type="xs:string" use="optional"/>
            <xs:attribute name="Operation">
              <xs:simpleType>
                        <xs:restriction base="xs:string">
                                  <xs:enumeration value="Select"/>
                                  <xs:enumeration value=""/>
                        </xs:restriction>
              </xs:simpleType>
            </xs:attribute>
      </xs:complexType>
</xs:element>
```

Above we can see the HTML-like format of the text. The highlighted items are strings that are eventually used to create the XML and ultimately the Object Stream, which in turn is read by the interpreter.

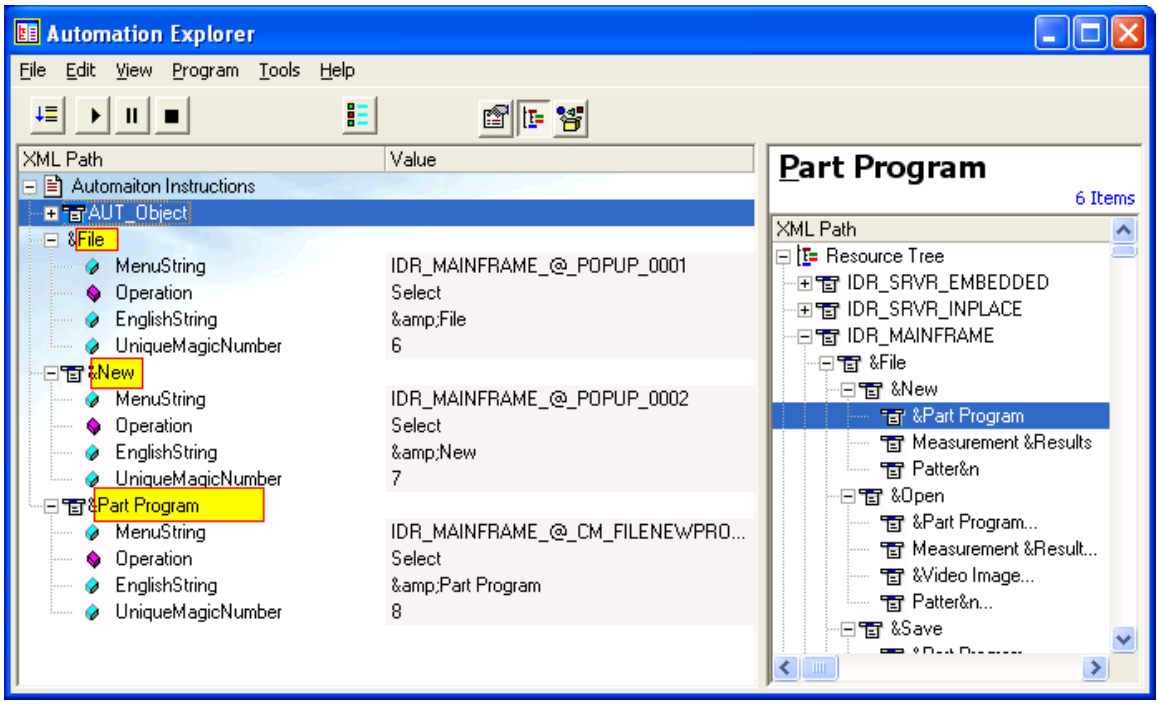
XML example:

Below we have the XML that selects the menu items “File.New.Part Program”.

```
<Menu  Operation="Select" EnglishString="&amp;File" />
<Menu  Operation="Select" EnglishString="&amp;New"/>
<Menu  Operation="Select" EnglishString="&amp;Part Program" />
```

4. Automation Editor

To simplify the task of creating these scripts an editor was created. A picture of the editor is included below.



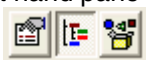
Above we can see the menu objects used that perform the "**File.New.Part Program**" menu selections are highlighted. Adjacent to the strings you can see the menu icon as shown below.

This indicates that the object being worked against is a menu item with the properties below each object.

4.1 Anatomy of the Automation Explorer

In the right pane there is a tree view that contains the objects and their properties. Each object is adjacent to a +/- sign. The contents of the object are listed when you expand the object view.

4.1.1 View Buttons

The right hand pane contains one of 3 different views. The views are selected via the radio buttons:  . Each button's functionality is described below:

Icon	Description
	View the properties of the object with the current focus. Assuming that only one item is selected. **Properties View**
	View the resource tree of the object in graphical form. More detail on this will follow. **Resource View**
	Browse all the objects and their properties/operations. **Browser View**

The views are populated as follows:

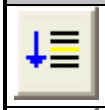
Properties View is populated using the XML for the current properties and the XSD to provide further options for editing the object.

Resource View is populated using the XML generated from the resource file.

Browser View is populated using the XSD.

4.1.2 Program Control Buttons

The program can be run, paused and stopped via the radio buttons:  . Each button's functionality is described below

Icon	Description
	Run the automation one line at a time showing the currently executed item as selected in the treeview.
	Run the automation.
	Pause
	Stop.

Benefits. Because the interpreter reads one instruction at a time, we can easily implement the different program control functionality. In the case of running one line at a time, the editor passes a single instruction at a time to the interpreter. To run an entire program, the editor passes a text file with all the instructions to the interpreter. The pause and stop operations are supported by checking the user requests in between instructions.

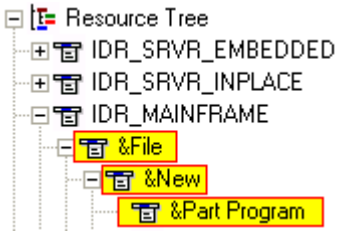
5. Localization using the resource files

The resource files used to create the target program, QVPAK, are parsed into an XML file that represents the hierarchy of containment of all the resource items. The hierarchy is displayed graphically to allow the automation developer to design programs that access the items represented in the resource file. In addition, programs created using the resource tree can run on any language that the target system supports. As you construct a program, token constants are added to the objects being created. The token constants represent the specific resource item independent of the language.

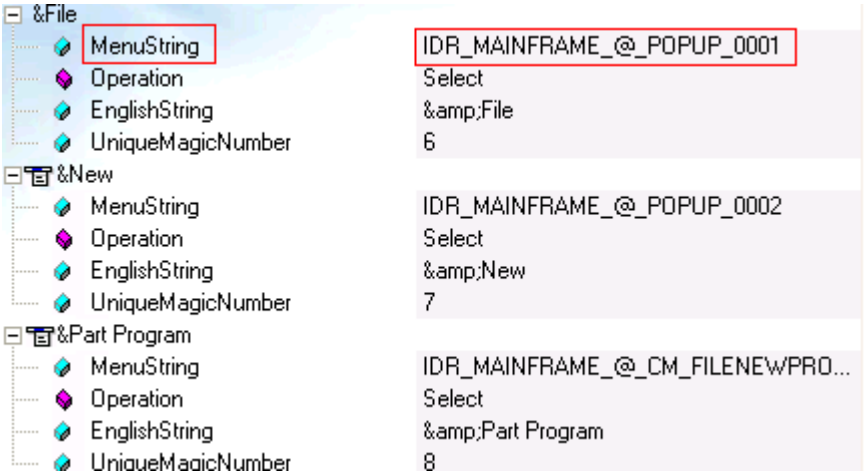
5.1 Example: construct a program using the resource view

In the example we have been discussing we wish to construct a program that accesses the following menu items "**File.New.Part Program.**"

The user constructs the program by simply double-clicking the items "File", "New", "Part Program" in the resource treeview as shown below with the selected items highlighted.



In the program treeview pane you can see the resultant objects created in the image below:



Above you may notice that there is an object property "MenuString". This is the token constant for the specific item you have selected. The value is used to look up the string based on the language.

Benefits. The graphical display of the instructions in the automation provide a quick visually clue to determine what the automation is doing.

- A user can create automation instructions by selecting items in a hierarchy that avoids choosing the wrong localized constant. A typical approach to localization support is that the strings in the resource files are mapped into constants. These pairs of strings and

constants are often put in a file that the automation author has to hunt through to find the correct constant. Often times the file is large and it is difficult to determine which constant is the right one because there are often duplicate strings in different dialog boxes, etc. When the wrong constant is chosen sometimes the automation fails on only one or two languages (usually the hardest ones to read). This failure might be detected after hours of running and as a result be very inconvenient to run again after correcting the problem. Because the user chooses the string via exploring the hierarchy of the user interface, there is very little chance of making this kind of mistake because it is clear where the string is obtained.

It is downright fun to create automation that has this “what you see is what you get” interface.

Pitfalls. To support localization the software needs to uniquely identify the string in the resource file by assigning them some kind of token. Most of the items in the resource file can be referenced uniquely via a resource ID; however, the POPUP menu items are not unique and can only be referenced by their actual value. A solution to this problem is to provide literals as a set of ordered pairs that are preprocessed depending on the version of the product.

6. Lessons Learned

6.1 Pass simple “pre-digested” XML to consumers of the XML

Consider the case where we wish to render some XML in a tree view. We could pass over the XML and the XSD and use the XSD to fill in things such as common attributes. The problem here is that the tree view handling code then needs to know the intricacies of the schema. The tree view control can be made much simpler if it is assumed that it just renders XML with no schema. In this case you have to write a program to convert the XML and XSD into a simple XML file.

6.2 Choose your XML support code carefully

A lot of time was spent working through bugs in the MSXML4 DOM. In retrospect, it might have been better to go with something like the Apache project's Xerces XML parser.

Problems with the MSXML DOM version 4:

It is very difficult to explore the contents of the XML Element pointer in C++. Because the Microsoft DOM is implemented through a COM interface you cannot simply open the contents of a structure. You must unwrap the object through a series of calls. Also, the COM learning curve is very steep and it is advantageous to have some prior experience with it or know someone who can help. The VB implementation of the object to access the DOM allows you to successively expand and view an object’s contents in the debugger. When I ran into problems in C++, I would explore the DOM using the access object in VB. Then I switched back over to the C++ code to make any necessary changes.

The code examples in the SDK and other sources were often in VB and other languages. As a result, the examples were often not useful. This meant that I had to consider examples in many different languages and somehow come up with a solution in C++.

Code examples did not cover modifying the XML using the DOM very thoroughly. I found lots of examples of displaying data, but very few covered modifying data.

Benefits of MSXML DOM version 4:

Because the Microsoft XML DOM is based on COM, you can uniquely identify a given node or element pointer. In COM this is accomplished by doing a Query Interface for the IID IUnknown pointer. The value returned is unique for the current session. XML itself is weak with regard to determining the identity of a given node or element. In fact, there is no mechanism for doing this other than obtaining the fully qualified path and using it to determine the identity. Being able to do this is particularly important when trying to debug software that caches node/element pointers. When you are developing a tree view control that binds to some XML, you need to associate the tree branches and leaves with the XML. A cache is used to associate the tree elements with the XML.

7. Concluding remarks

There are 4 powerful items that this technology is based on:

Stream-based automation interpreter

A hierarchal user interface is used to create localized automation

XML schema (XSD) is used to explore the instruction syntax and validate automation instructions

XML is used to store the program logic

The big surprise from these technologies was that the hierarchal user interface could be used to create localized automation. This feature was not part of the initial objectives of the architecture for this system. However, this technology has been used time and time again to create automation quickly that is intuitive and localized properly. This has made it the most valuable feature of the system.

8. Glossary of Terms

Object Stream – a set of object oriented instructions that are performed one at a time. Example:

```
File.FileName=test.HTM
File.ShellOpen
```

Schema – structure used to define the layout of a database

Stream – a set of instructions that are performed one at a time.

XML – (Extensible Markup Language) - a text-based format for storing data. The syntax is similar to HTML.

XSD - XML Schema Definition, structure definition for a given instance of XML data

Management Commitment to Quality Requires Measures

John Balza
Hewlett-Packard Company
john.balza@hp.com

Abstract:
In 1998, Hewlett-Packard began a program for its UNIX software to reduce the number of customer-found defects by a factor of 10 over 5 years. With the release of HP-UX 11i, we saw a 6X reduction in defects (placing us among the top 1% of software vendors) and are now working to achieve a mainframe level of quality in our next release.

True to the literature, we found that strong management commitment to quality was key to any improvement. Engineers and managers, however, were only being measured on delivering functionality on schedule. In order to change the culture, we had to demonstrate to our software development teams that management valued quality over functionality while maintaining schedule. A key enabler was providing management with ways to measure quality as easily as schedule.

Biographical Sketch:
John Balza has been the Quality Manager for HP-UX since 1994, leading the 10X quality improvement program. In his 30 year career, he has managed over 50 software projects for Hewlett-Packard at various management levels. He has been exploring software quality and processes since his first failed project, which was a year late and had to be completely rewritten.

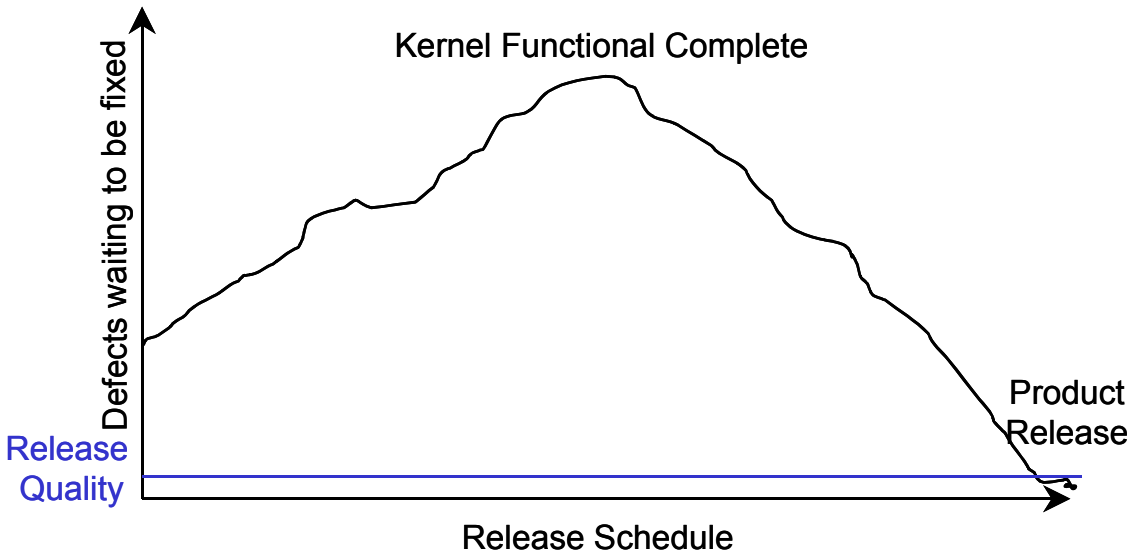
Background:
The literature is full of lessons on how important it is to have management commitment in order to drive change. In our case, we wanted to improve our quality. Management quickly declared this to be the number one objective. But in order to make an objective truly number one, it needs to be measurable. How can you provide management real-time measures of quality, so that it can be measured as easily as schedule? This paper discusses our real world experience in making quality number one, measuring that quality, and in the end seeing an 83% reduction in customer-reported defects.

The HP-UX product is Hewlett-Packard's version of the Unix operating system. It consists of approximately 18 million lines of code produced in a distributed organization of 1500 people in 7 worldwide locations. These 1500 people are organized into 13 different R&D labs reporting into several division managers. The HP-UX 11.00 version of our product was released in 1998 with significant functionality additions to the previous release; in particular it was our first 64-bit operating system. Many customers began to use the operating system in mission critical applications where availability and reliability were very important. In these new environments, customers were demanding higher levels of quality than in our previous releases. In fact, in 1999, in a survey conducted by Interex, the HP User Group, developing higher quality software, was the 2nd highest issue in the poll.

Typical Tradeoffs
Quality has always been a top priority within Hewlett-Packard, in fact it was usually described as "Priority 0" among the traditional tradeoffs of functionality, resources, and schedule.

We had always had rigorous quality standards, but most of these standards could only be measured at the end of a release. Resources, functionality, and schedule are much easier to measure throughout the lifecycle. The most typical measure of functionality and schedule was "did the functionality get checked-in to the release on the scheduled date". Managers did detailed plans for meeting the functionality complete milestone for each feature. They were measured primarily on meeting these check-in targets. Without an adequate measure of quality at check-in, however, the quality of the code varied widely. While they may have met their targeted date, they may actually have caused the entire products shipment to slip because of checking-in low quality code.

One way of thinking is that each new check-in introduced new defects, so as we progressed the number of defects to be found was constantly increasing until we reached functional complete. We then spent the testing cycle of the release, removing all these introduced defects. Typically, it would take us 9 months from the last kernel check-in until the release met our release criteria. 2 months were spent stabilizing the kernel, 2 more months adding the higher-level layers of functionality that depended upon this kernel functionality, and 5 months to stabilize the entire system.



This behavior was further aggravated by the marketing and management behavior of selecting a target date for a release and asking for as much functionality that they felt could be reasonably accomplished by that date. The expectation was unless you could 'prove' that you couldn't make the date with the functionality, you weren't allowed to remove it from the requirements. This naturally led to slips in schedule. Every time the schedule slipped marketing would insist upon more functionality to meet the 'competitive environment'.

Management Commitment

Given the new markets we were entering and the customer concern about our software quality, management knew that we needed to improve our software quality. The general managers agreed that for our next release quality was going to be the number one objective. But how was this any different than quality being priority 0?

First, we decided to set a tough goal. The key metric we chose to measure our quality was the number of customer-reported defects in the product's first year of shipments. This metric was chosen because it was easy to measure, it reflected customer experience, we had historical data from previous releases, and HP had used a similar measure in the late 1980s in a corporate program to improve quality. We initially proposed a 2X reduction in our next release. Based on our reliability models, a 2X reduction was what the software needed to contribute to meet an important customer availability goal that our systems would be available 99.999% of the time. But after further thought, we decided that quality required a longer-term commitment, one that would change the culture and the behaviors in our organization. We decided upon a 10X improvement over a 5-year time frame compared to our 11.00 release. That is in 2004 (5 years), customers would report only 10% as many defects on that versions of HP-UX as they reported on the 11.00 version in 1999.

Next, we realized that if we set quality as the number one goal, we needed to allow the project teams flexibility in some other dimension of the resources, schedule, and functionality tradeoff. Management decided functionality would be the primary trade-off.

But this would require a cultural shift, a change in everyone's behaviors. In order to make these real, top managers had to start demonstrating a change in their behavior before any of the lower level managers or the engineers would begin to change their behaviors.

The first management change was to significantly reduce the ‘must’ functionality for the next release to just a handful of key features. Historically, each marketing member responsible for a key functionality area would label quite a few features, as ‘must’ priorities for the next release. The program team would typically be dealing with hundred's of ‘must’ features. Instead, we decided upon a couple key business needs that the next release should satisfy. What would be the marketing message for this release? Looked at this way, the ‘must’ features were those that supported the marketing message. Instead of hundreds of features that were musts, we could reduce it to 10s.

The second change was a consistent message sent in all employee communications about quality being our number one objective, and functionality being the lowest objective. This was reinforced during checkpoint meetings whenever a lab said they couldn't meet the schedule. The first question asked was whether some functionality could be cut to hold the schedule. Budgets were examined for significant increases in quality resources - such things as training on quality and project management methods, new tools and process improvements; and investments in tests and the machine resources to run those tests. Schedules were examined to ensure that they included time for peer reviews and development of new tests. By subordinating these other dimensions to quality in several ways, people began to understand that upper management really was calling for a change to make "quality the number one objective".

The third key change was that management established what became known as the "hour of power", an hour each week where the program management team came into the division manager's staff and they examined the progress on key milestones and the quality of the release. This meeting tended to be very action oriented - the division manager's direct staff would be given action items or asked to become involved in a tough problem area (usually leading daily meetings). Since action items flowed down the management chain rather quickly, the whole organization soon found out when their manager had an action item.

Of course, this required that we be able to measure our quality at least weekly, so that managers had action items related to improving quality as often as they had action items related to milestones.

Measuring Quality Real-Time

The first quality metric we put in place was a defect backlog metric. From our defect tracking system, we could graph for the system as a whole, for every lab, and every project within a lab, the number of incoming defects, the number of defects fixed and the current number of open defects. We set a goal that every lab would have at most a two-week backlog of defects. This allowed us to set a consistent goal across the entire organization; no matter what the size of their code base or how many defects were being found.

For example, if for subsystem x, they've had 60 incoming defects in the last 6 weeks, that averages out to 10 defects/week. Their two-week backlog goal would be 20 defects. Their team was asked never to have more than 20 open defects outstanding. Another team getting only 12 defects every 6 weeks would be asked to keep the number of outstanding defects below 4.

Why two weeks? That was our practical experience of how long it typically took to get a defect fixed. We would spend approximately a calendar week to root cause the defect and another week to get the repair in the source management system.

Everyday we published the defect backlog for every project in every lab. We now had our first metric where we could clearly decide quality versus functionality. In the Hour of Power, the division manager asked his staff to stop submitting functionality to the release, if they did not have their defects under control. This wasn't cut and dry, occasionally exceptions were made when critical functionality was needed to meet other dependencies, but usually staffing was moved to fix defects whenever a lab was out of limits on this metric.

If a lab's defect backlog grew too big, the lab manager would often send voicemails or emails to the offending lower level managers asking what they were doing to get this under control, whether they had all the resources they needed, and if he or she could help in any way. At certain points of the release, engineers were lent to other organizations to help triage and fix defects, because everyone knew it would increase the overall productivity of the release, if these defects were fixed. This began to strongly reinforce the 'Quality is the number one objective' because it is what your boss talked about as often as they asked about whether you were on schedule.

The Critical Resource

The second quality measure that was put on place was the result of an analysis of our development process to find the critical resource. Eliyahu Goldratt in his book, Critical Chain[1], discusses the concept of the critical resource. This is the resource that controls the throughput of the entire process. While you could make process improvements in several areas of the process, they may have no effect on the process throughput. Only by improving the throughput of the critical resource, can you improve the overall throughput of the entire process.

As we examined our process, we identified that the critical resource in our process was the top-of-branch of our software configuration management system. This is the point where an engineer publishes his latest software changes and makes them available to the rest of the development community. We realized that if we kept the quality of this point at high quality, we would improve everybody's productivity. This top of branch, became known as the "mainline". Productivity was improved because we recognized that engineers stumble over each other's defects all the time. Sometimes they ended up tracing a defect in their code to find that it actually was a defect in someone else's code, but it wasn't known yet. You're unable to make progress on your project until this defect is fixed. Other times, while you're triaging your defects, others are continuing to submit new code, and may mask the defect that you've been trying to work. So it was very easy to be in a situation where the quality of 'mainline' continued to deteriorate as more functionality was added.

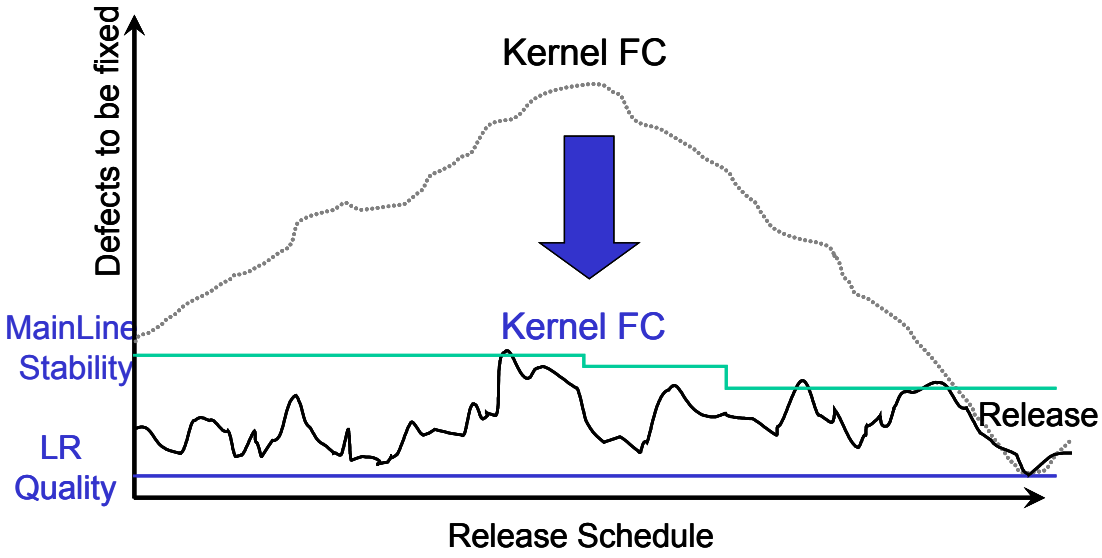
We put in place what we called the 'mainline stability test', which we ran everyday. This suite of tests consisted of our critical functionality and reliability tests. We required that the reliability tests run 48 continuous hours of operation and that there were no regressions in the pass rate on the functionality tests. (Note: we actually have 3 test set-ups and start a new one every day.) These tests were run in a wide variety of machine and networking configurations and represented a significant step toward the quality requirements to release the product. Anytime, the tests failed, we would stop any new check-ins and triage the problem until we found the guilty check-in. Either we removed that check-in or we put in a quick fix to remove the problem, and then re-ran the mainline stability test. Here was another real-time measure of reliability, everyday we could report that our mainline was at a known good quality level.

This mainline stability test complimented our backlog metric. By running these rigorous tests we increased the number of defects found early in the product lifecycle. These defects were added to the backlog, so that they were quickly fixed before the next submittal to the mainline.

We now had a second measure that was brought into the Hour of Power. The number of days the mainline was 'open' versus 'closed'. If mainline had been closed for a while, a lab manager was assigned to ensure adequate resources were applied to solve the problem. The division manager also held his lab managers accountable for the number of times, their submission shut down the line. Each defect was root caused, and if the engineering team determined that the problem should have been found earlier, it was credited to that lab. It became well known which lab had caused the line the most and affected that lab manager's pay.

Now we could really measure 'quality submissions on time'. This became the new key metric for all of our managers. It was no longer acceptable to just get the functionality checked in, it also had to be at this know good quality rate. By changing our process to shut down the line whenever the release wasn't stable, we also changed the behavior of the entire organization because of peer pressure. When the line was shut down, no

engineers could submit their functionality, no one wanted to cause this shut-down. Thus not only was this a measure for upper management, but project managers and engineers starting being driven by this key measurement point. As shown below, now instead of the defects increasing through functional complete we keep our defects under control throughout the release process. Since the release must always be at least at the mainline stability criteria.



Mainline stability test introduced a new problem - it often found problems that the engineers hadn't been able to find themselves, because they only had 1 or 2 systems to test their code on. The engineers demanded more test resources at their disposal, particularly more varieties of machine and networking configurations, in order to have a higher probability of finding the defect before mainline testing. This, of course, would be to everyone's benefit because the defect would be found before it was published on the mainline.

We couldn't afford each engineer to have a rigorous test environment, so to satisfy this need we set up a distributed test environment available to everyone. Each geographic location had one or more test centers with a variety of machine and network configurations usually a set of our lower cost machines and storage devices. For the large systems and more complex configurations, we centralized those resources in one location that we shared across the entire organization. From a web interface at the engineer's desk, they could schedule tests to be run in their local test center, as well as the complex tests and configurations in the central test center. Now engineers routinely run their new code through a series of 'pre-submittal' testing which runs the critical functional test and at least 10 hours of reliability in a small configuration. Today, we run over 100,000 tests a day in this distributed environment.

Testing

Both of our real-time quality metrics depend heavily upon our ability to test correctly. So a key part of this improvement program also required us to improve our testing capability. Three separate efforts were undertaken to improve our testing. All 3 are driven by metrics, but at a lower level of the organization.

First, we found that many of our functionality tests were ‘flaky’, that is, they would pass in some circumstances and fail in others. So we began what we called a project to remove these flaky tests. Any time a test failed if we determined that the test was the problem, not the product, we removed the test from our regular test runs, calling it a ‘known test failure’. We set a goal to have our known test failures track close to zero. This encouraged folks to either permanently remove the test or fix the test. Of course, the short-term expedient might be to remove the test, but that was precluded by our second metric: code coverage.

Measuring the percentage of functions called and branches taken during our functional test runs was the second test metric. For purposes of setting goals for this metric, we divided our code into “old” code, which had existed on previous versions of the release and was being modified for this release; and “new” code, modules either being added or significantly changed as part of this release. For “new” code, we asked teams to reach 100% functional coverage (all functions called) and 85% branch coverage (85% of all branches taken). In our environment, the development team was responsible for both the new product code and new test code. By having this requirement defined early in the release, teams actually designed their product code to both be functional and testable. Many of our teams actually achieved over 95% branch coverage because of how the code was designed. “Old code” was more problematic. Here we asked teams to use the code coverage tool to identify what they saw as high risk areas in their product code that were not being covered by their tests. For the release, we set a want requirement to increase branch code coverage by 15 percentage points over the previous release or to reach 85% coverage. The must goal was that code coverage not decrease over the base measurement made at the end of the last release. All teams met the must goal and over 35% met the want goal.

The third metric we employed was the defect escape rate. This measured the percent of defects that were found outside of the development team – by other development teams, by system test, and, of course, by customers. Our long term goal is that 90% of all defects are found by the development team through peer review, and testing. This metric was accompanied by a regular review of the defects escaping the team. By examining these defects and performing a root cause on the most common types, the team learned what particular improvements they needed to make in their development process. Often, this led to bringing in additional tests from teams that used their modules or directed the team to hot modules that might need to be redesigned or where the code coverage needed to be improved. Just as often, however, it led to other improvements – improvements in the peer review checklists, training in areas like HP-UX internals or locking mechanisms, or to do joint design reviews with other teams.

Results

Our test metrics encouraged significant improvements in out tests. By incorporating these improved tests into our mainline testing, we improved our two key real-time quality metrics: defect backlog and mainline stability. We have added other real-time quality metrics over time, but these two were the key measures that allowed us to measure quality as easily as schedule and resources. They caused a significant change in the culture that truly made 'quality the number one priority'. In particular changing the key measure of results to 'quality submissions on time' resulted in a significant change in behavior.

The 11i release of HP-UX surpassed our initial 2x improvement in quality goals. This release had 83% fewer defects found by customers in the first year of life compared to any of our previous releases, even though we sold it to more customers. Another measure of our success is defect density. The defect density in delivered defects/function point show that this product in the top 1% of products as tracked by Capers-Jones [2]. With a further 2X improvement, we should be at the quality level of the IBM mainframe.

The quality improvements were sufficient that we significantly improved our productivity. D. H. Brown in 2002 rated our version of UNIX #1 in all 5 categories of functionality. To quote their report [3]:

> "Clearly reflecting HP's increased investment in its Unix product line, HP-UX moves to the head of the class for UNIX operating systems functions. HP-UX occupies the top spot in every studied category, with a particularly strong lead in internet and web application services, and an impressive surge forward in the intensely competitive RAS (reliability, availability, supportability) category"

Just as important are the cultural changes that have occurred which will serve us well in the future as the organization changes. Trading off functionality to achieve quality and schedule are now the norms. The defect backlog metric and mainline stability metric continues to reinforce that we submit quality functionality on time. We're continuing in our efforts to improve our quality and productivity for our current release under development.

[1] The Critical Chain, Eliyahu M. Goldratt
[2] Applied Software Measurement, 2nd Edition, Capers Jones
[3] 2002 Unix Function Review, D.H. Brown Associates Inc.

Pacific Northwest Software Quality Conference 2003 White Paper

FINAL submitted for review August 18, 2003

Business Process Mapping for IT Professionals

Written by Victoria Hawley
President
Process Flow Specialists, Inc.
50960 Dike Road #5
Scappoose, OR 97056
Phone: 503.240.4954
E-mail: VHawley@pfs-usa.com

Abstract:
Studies show that a leading cause of information technology (IT) project failures is lack of business unit involvement and failure of business units to understand how to use information systems to improve the way they conduct business. Data flow maps mean little to the average business customer and often deepen the perception that IT professionals do not understand the day-to-day issues of getting data into and out of complex information systems. When the effort to understand business flow does occur, it is often after investing in technology that locks business processes into a solution that may not address the root cause of performance issues. This paper addresses the issue of how approach IT projects from a business process perspective that takes into account both IT and user requirements.

Biography:
Victoria Hawley is president of Process Flow Specialists, Inc. (PFS), a consulting, training, and documentation firm specializing in mapping, measuring, and improving business processes. She developed the Enterprise Map method of identifying and linking all of an organization's business processes to each other and to IT systems. She also developed a process measurement system to flow down performance measures from business plans to business processes and jobs, ensuring all levels of the organization are working towards common goals. Clients include NEC America, ODOT, PacifiCorp, Parker-Hannifin, Cypress Semiconductor, and ESI. Prior to forming PFS in 1993, Victoria was the process improvement coordinator for Pacific Telecom, where she led cross-functional process improvement projects and was involved in the implementation of a $40 million enterprise billing and collection system.

Victoria also facilitates a Boeing Supplier AS9100 users group, is Past Chair Portland Section of American Society for Quality (ASQ), an ASQ Certified Quality Manager, past board member of Oregon Partnership for Excellence (OPE), two-time Sr. Examiner for the Oregon Quality Award, and six year member of the Portland ISO 9000 Steering Team.

"When a bridge falls down, it is investigated and a report is written on the cause of the failure. This is not so in the computer industry where failures are covered up, ignored, and/or rationalized. As a result, we keep making the same mistakes over and over again."
The Standish Group Report, 1995

Local IT Project Failures:

Due to a faulty computerized billing system, bills aren't going out to water customers in Portland, Oregon. The city water bureau estimates it is losing a little over $13,000 a day as they struggle to fix a computer system that has been broken since it was turned on in February 2000. [1]

Two years into a planned 5 year / $50 million project to automate the Oregon DMV, the project was re-estimated at 8 years and $123 million. Even worse, after the rollout of a pilot, lines of local DMV offices backed up around the block. In response to public outcry, state officials killed the project. [2]

When I was at Pacific Telecom in the mid 1980's, the organization committed $4 million and 2 years to design and implement a new billing and collection system; six years and $40 million later, it had a system that increased the time to take an order from 5 minutes to 15 minutes.

Causes of IT Project Failure:

According to The Standish Group, failed IT projects cost US companies an estimated $145 billion per year. A nationwide survey[3] revealed the following as the top ten reasons for IT project failure:

Lack of User Involvement
Incomplete Requirements
Unrealistic Expectations
Changing Requirements and Specifications
Lack of, or poor planning
Lack of Executive Support
Lack of Resources
Unclear Objectives
Unrealistic Timeframes

According to a 2002 Robbins-Gioia, LLC, enterprise resource planning (ERP) system implementation survey, 46 percent of the participants noted that while their organization had an ERP system in place, or was implementing a system, they did not feel their organization understood how to use the system to improve the way they conduct business.[4]

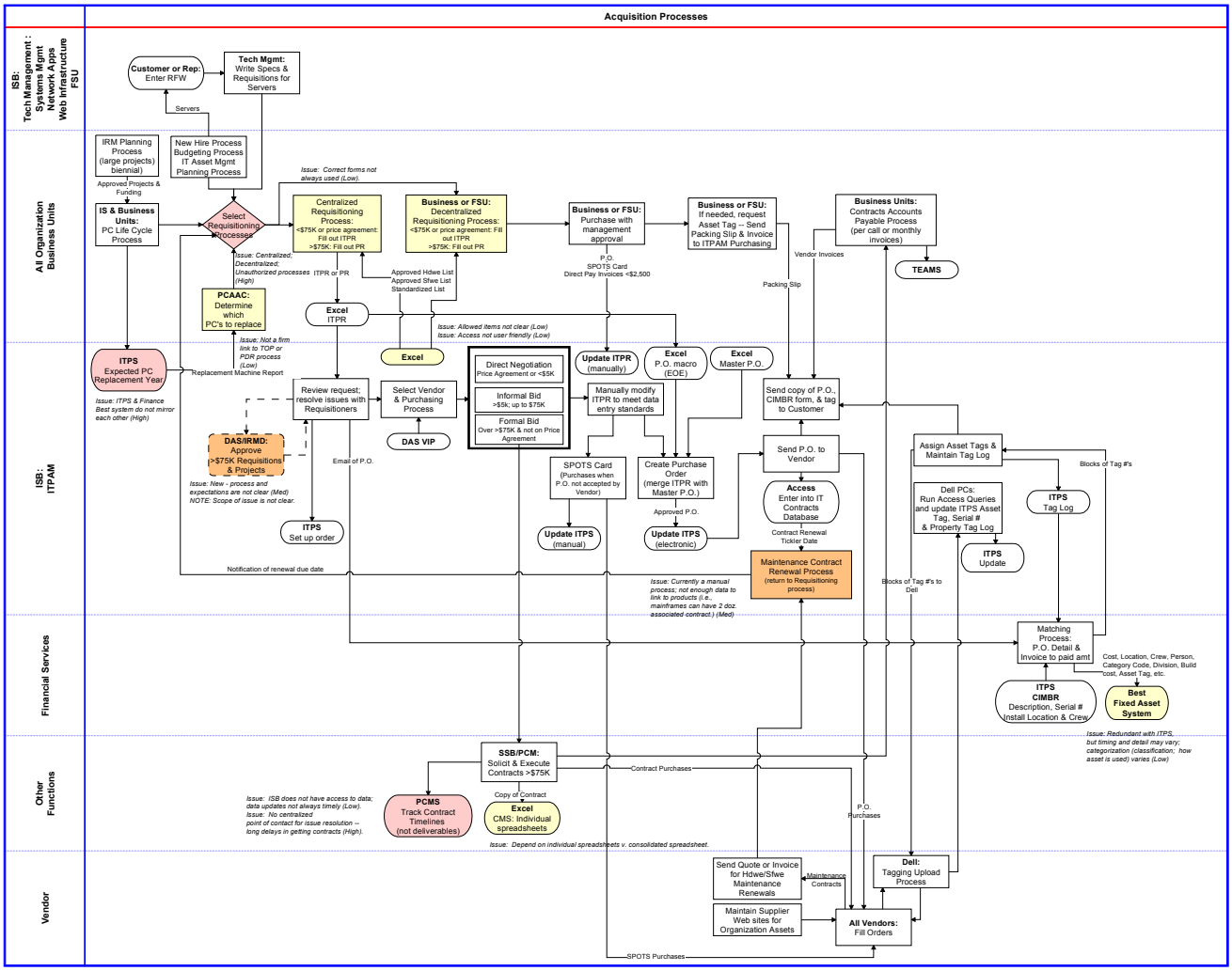
Users who do not want to change how things get done can make a good system fail. By treating IT implementations as a result of – not a trigger for – business process improvement, organizations can save millions in implementation costs and post implementation nightmares and get upfront user buy in to the project.

In my experience, a common cause of IT failure is to 'pave the cow path,' in which an obsolete manual process is automated with the expectation of improving efficiency. Automation will not make a bad process better; it will only allow the organization to perform non-value-added functions faster. More and more organizations are making business process analysis part of the IT implementation strategy.

In a recent project to upgrade the IT asset management system for a large organization, we were hired to map the As-Is business process and to facilitate a Should-Be process design session with the business units to match stakeholder expectations to the IT project phase 1 features.

We interviewed business units and information systems (IS) to document the activities and issues associated with the IT Asset Management process, which we identified as having four phases: Acquisition, Installation, Management and Disposition. The diagram in Example 1 links business processes and activities to information systems. Notes (*in italics*) identify issues with the current process. Color is used to indicate the severity of the issue (Low, Med, High).

Example 1: As-Is IT Asset Management Acquisition Phase process flow & issues



Overall, the project was quick (5 days) and gave the IS project team a useful tool for rolling out the project. The business units were delighted to be involved, the project team learned how the new system affects the current process (both positively and negatively), and both sides learned more about each others expectations and constraints.

The issues with this project are common to many IT projects:

- Project delays resulted in management setting an artificial deadline. To meet the deadline, the project team scaled back the features to be released in Phase 1. The partial release means formerly automated processes will have to be done manually until the phase 2 roll out.
- Process roles and responsibilities are not clearly defined. In any process, there should be no question of who is responsible for each task – exceptions should be well documented. For example, the install procedure could say *"The Field Services Unit is responsible for the PC installation process from receiving through data entry, including training and follow up on installs performed by business units in remote locations."*
- Lack of performance measures. This contributes to the roles and responsibilities issue and can keep even good technology from working well from a business point of view. A real risk is suboptimizing the organization by moving work from one area to another or investing in technology without measuring the overall impact on the organization.
- Timing of process mapping and analysis: Better late than never. For best results, complete an As-Is analysis when performance issues can no longer be resolved with minor fixes. Redesign the process to remove disconnects caused by hand-off issues, training issues, conflicting performance measures, and process flow. THEN determine if and where information system technology can be used to improve performance.

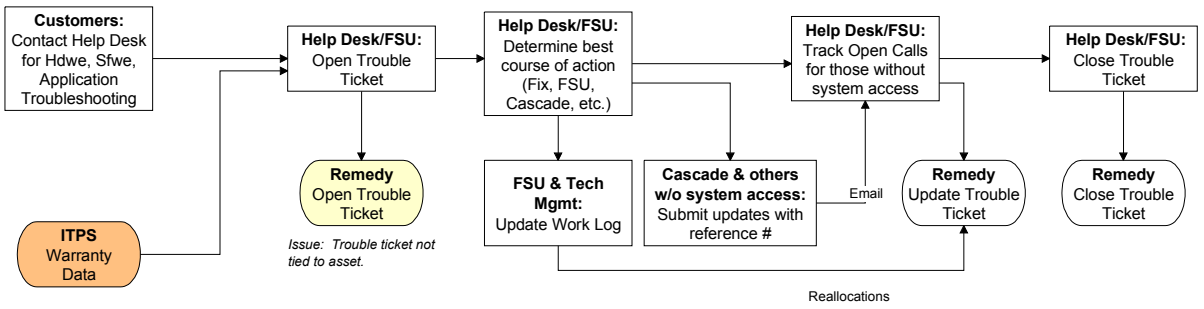
Recommended – Apply good process management practices:

1. Identify a process owner with cross-functional responsibility and authority to manage each key business process. A process owner is responsible for ensuring critical business processes are mapped, measured, monitored and improved.
2. Develop and flow down performance metrics from the organization level to processes and tasks. Effective performance metrics are measurable, relevant, clearly understood, attainable, monitored and actionable.
3. Analyze the As-Is process to determine the root cause of performance issues.
4. Involve process owners and stakeholders in redesigning the process to close the gaps.
5. If needed to eliminate a root cause, invest in technology to improve the process (Note: The decision to invest in technology occurs in step 5; not step 1…)
6. Document the new process; train to the procedures and measures.
7. Update performance measures to support the new process.
8. Monitor performance results; perform root cause analysis on significant issues and trends.
9. Have an objective and trained process auditor review critical processes at least once a year to evaluate compliance with policies and procedures and to evaluate the overall effectiveness of the process and related information systems.
10. Improve the process as needed to meet needs and expectations.

Process maps help the IT professional graphically link information systems – mainframe, network, and 'shadow systems' – to business processes.

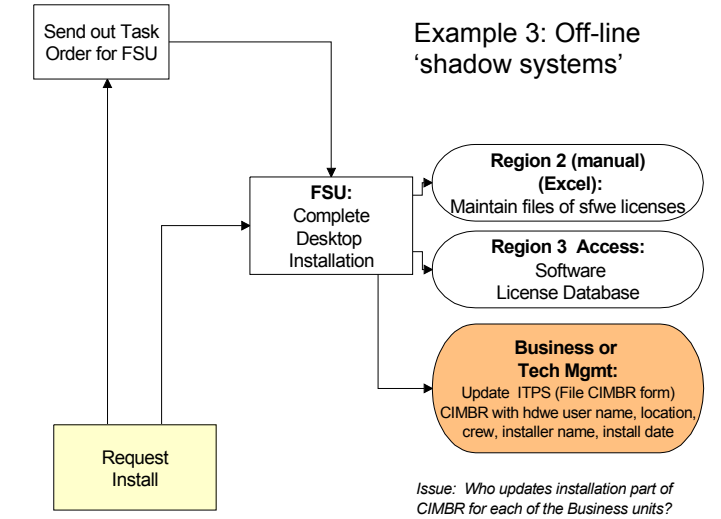
A large drain on IT resources is managing a large number of independent systems, many which evolved over time, are not linked to related systems or processes and/or do not meet the current needs of the organization, its workers, or it customers. In Example 2, warranty data is not tied to the trouble ticket system, not all repair techs have update access to the trouble ticket system, and trouble tickets are not tied to the asset management system meaning business unit managers cannot determine when it is more cost effective to replace than to continue to repair a faulty piece of equipment.

Example 2: Independent Systems



Managing disparate systems takes time and attention away from system enhancements and design. Feeling a lack of support from IT, process owners often develop off-line systems, spreadsheets and databases to meet their needs. At best, these 'shadow systems' contain information that is not available to the rest of the organization; at worst, decisions are made based on information that is out of sync with other areas, looks the same but is actually different, or is just plain wrong. Process maps identify these various systems and their relationship to the rest of the organization. They allow a common basis for understanding of the complexities of managing these systems and the impact on productivity and quality of service throughout the organization.

Example 3: Off-line 'shadow systems'



Process maps clearly highlight information flow disconnects, such as dead ends, redundant systems, and bottlenecks.

A process map is simply a schematic of all or part of an organization. There are two time references for process maps: How things really work today (the 'As Is') and a futuristic, desired or idealistic state (the 'To Be,' 'Should Be' or 'Could Be'). Even when developing a brand new process, take a careful look to see if there is an As-Is process operating in the background. It will save time in the long run to complete an As-Is analysis up front than it is to have one or two team members trying to rehash how things work today while the rest of the team is trying to design the new process. Use As-Is maps to identify disconnects in the process (see issues in *italics* in Examples 2-3).

Process maps help IT and process owners come to agreement on where to spend IT resources to effectively meet the organization's information and data needs.

'It is not that we know so little, it is simply that so much of what we know is not true.' (Author unknown.) Truth is a matter of perspective based on experience and beliefs. The 'truth' about how to make a process more effective varies depending on where a person is on the organization chart, technical expertise, and history with similar issues.

Most IT views of a process show how data flows through systems independent of the work (business processes) it takes to get data into and out of systems. When process owners feel that IT does not understand or care about the work it takes to turn data into useful information, they do not support IT initiatives. A process map that focuses on a business process perspective (workflow) with IT systems (dataflow) in a support role goes a long way in forming an effective work alliance between IT and process owners based on mutual understanding of how things work and how they can be improved.

--END

---

[1] December 16, 2001 by John D. Biggs

[2] http://www.cpm-solutions.com/whenprojectsfail.html

[3] http://www.cpm-solutions.com/toptenreasonsforprojectfailure.html

[4] Robbins-Gioia, LLC, Press Release: January 28, 2002, Alexandria, Virginia

Shifting Paradigm to Agile Methods - Embrace the Change

Bhushan Gupta
Hewlett-Packard Company
bhushan_gupta@hp.com
18110 SE 34th St.
Vancouver, WA 98683
(360) 212-8186

Binnur Al-Kazily
Hewlett-Packard Company
binnur_al-kazily@hp.com
11311 Chinden Blvd
Boise, ID 83669
(208)396-6372

Biography:

Bhushan Gupta is a program manager at the Hewlett-Packard Company and is responsible for developing software processes and their continuous improvement. In his current, role he is involved with software processes, software lifecycles, data driven development, and project retrospectives. In his six-year tenure at Hewlett Packard he has developed product release criteria, testing strategies, test plans, and quality reporting processes. He has lead the XP feasibility and adoption process for the DPS laboratory. Bhushan has a total of 18 years experience in software engineering, 10 of which were in academia developing and teaching software engineering courses.

Now a section manager, Binnur Al-Kazily has taken on a variety of roles, a software engineer, technical lead, and technical support manager, during her 11 years of tenure at Hewlett-Packard. She now leads the XP team across multiple geographies and oversees the transition to XP methodology.

Abstract

This paper describes the approach we have used to shift our development methodology from a semi-iterative to a more agile eXtreme Programming (XP). The motivation for the shift was to achieve higher level of customer involvement for requirements definition as well as expose the developers directly to the customer requirements and needs, provide mechanism to enable requirements prioritization on demand, and establish and track to a more predictable schedule. Our group was faced with the challenges of geographically dispersed teams in different time zones, varying cultures, and different development environments. The XP Feasibility team was established to understand the XP principles, evaluate our existing ecosystem in the light of XP concepts, and determine our key enablers and inhibitors for success. The team laid out the success conditions and shared its findings with the upper management. The paper also highlights the early disappointments and successes the group has achieved in our continuing journey to XP.

Shifting Paradigm to Agile Methods – Embracing the Change

Bhushan Gupta and Binnur Al-Kazily
Hewlett-Packard Company

Abstract

Agile software development methods are attractive especially when the requirements are fluid, product market is not well defined, and there is a time-to-market pressure to capture market share. However, to be successful, the agile methods require an ecosystem that facilitates their adoption and minimizes the probability of failure.

This paper describes the approach we have used to shift our development methodology from a semi-iterative to a more agile eXtreme Programming (XP). The motivation for the shift was to achieve higher level of customer involvement for requirements definition as well as expose the developers directly to the customer requirements and needs, provide mechanism to enable requirements prioritization on demand, and establish and track to a more predictable schedule. Our group was faced with the challenges of geographically dispersed teams in different time zones, varying cultures, and different development environments. The XP Feasibility team was established to understand the XP principles, evaluate our existing ecosystem in the light of XP concepts, and determine our key enablers and inhibitors for success. The team laid out the success conditions and shared its findings with the upper management. The paper also highlights the early disappointments and successes the group has achieved in our continuing journey to XP.

Background

"But we have always done it this way" is a typical human reaction to change. Shifting a paradigm is difficult, as it often requires a change in our work habits. A controlled amount of change introduced into the system in a timely fashion often has a higher rate of success as it causes less chaos. In the software development environment where success is measured by "on time and within budget", adopting a new paradigm always hampers initial success due to unpredictable and unstable environment.

While developing a web utility, a group at Hewlett-Packard (HP) was faced with the dilemma of ever changing requirements. The group was following Evolutionary Lifecycle [1] (a form of iterative development) for this project. The lack of well-defined customer as well as the continuous changes in the supported hardware and the solution deployment infrastructure resulted in non-stop changes in the initially identified requirements which eventually caused schedule slip. A retrospective of the project made the engineering community realize the need for better customer understanding. It was concluded that the subsequent version of the application should be developed with a higher level of customer participation. The business needs such as on schedule and high product quality still had to be met. This resulted in the team to investigate XP as a potential development methodology.

The management expressed a strong desire to change the software development methodology. Since XP promises a high level of customer engagement and high product quality with optimal customer value, the management showed considerable interest in adopting XP. The management also realized that the introduction of new methodology would require ramp up time for the development team; and some important business needs, such as meeting tight schedules might have to be compromised before realizing any benefits of the change. With that understanding the group started evaluating a shift to XP. The rest of the paper describes the steps we took to bring in the new paradigm, the choices we made, and the progress we have seen so far on the project that is currently using XP.

Gathering Success Stories and Self Education

To understand the XP principles and methodology at a high level, the management invited an upper level manager from a company that was successfully using XP. This was an open question and answer session attended by a cross-functional group at multiple sites and provided a good exposure to XP, especially its pros and cons. The discussion firmed up our belief in XP. This introduction perked up our interest and individuals in the group started to look for other external organizations that were successfully using the XP practice. With encouragement from another source there was a strong motivation to further evaluate the practice.

Since Hewlett-Packard is a large organization, there was a natural instinct to gather some testimony from within the company. Another group at a different site had successfully completed two projects using XP and had realized its benefits. We had frequent informal conversations with this group. Our enquiry mostly pertained to the level of success the group had achieved and the amount of time it took the group to successfully adopt the XP practice. The group strongly emphasized its success with meeting project schedule and producing a better quality product with XP. This enforced our confidence in XP but we wanted to evaluate our ecosystem before adopting the practice.

With greater community interest in XP, the next step was to build a core knowledge base on XP principles and traits. Two individuals collected literature [2, 3, 4] on principles and created educational material to raise the awareness of the entire group by making organized presentations. The group reached the point of reasonable understanding of the practice and was capable of making a diligent decision to shift to XP.

Studying the Ecosystem and Determining the Delta – the amount of change

At this juncture a “feasibility team” consisting of technical leads and architects started to evaluate our existing development practice with reference to XP. This evaluation concluded that we were already following a few XP traits. In particular, our methodology used iterations, release planning, frequent releases, continuous integration, and retrospectives. However, there were areas of significant gap such as pair programming, test-driven development, and story telling, that needed a substantial amount of change. We also faced a situation where the development teams were geographically dispersed and instant hallway communication and the daily huddles were not practical. This was substantially different than the XP principle where the development team is small (10-12 people) located at the same site.

This lead to the following Delta that we would have to bridge between our existing methodology and XP:

- Refactoring legacy code
- Pair programming
- Single customer voice
- Test driven development
- Large geographically dispersed team

These were all significant challenges and required multiple discussions and evaluations of trade offs as described in the following section.

Minimizing the Delta and Identifying other Enablers – what can get us there?

The feasibility team started to explore the ways to minimize the gap. The team felt that in order to be successful, we would need a strong management commitment. Fortunately, the upper management was determined to improve the development environment and not experience the requirements turmoil and schedule slip they have had experienced before. So the management empowered the engineering community to do what was necessary to ensure the success of the shift.

From our conversations with the groups that were successfully using XP, it was concluded that we did not have to introduce all traits of XP at the same time to achieve modest success. For example, we were convinced that the customer must provide the set of stories for each iteration (an iteration is a few weeks long cycle which delivers incremental functionality to the customer) that delivered the most value but the use of test driven development was not deemed absolutely necessary. The other HP group that had achieved a very high level of success with XP was not practicing pair programming.

Refactoring Legacy Code

We had a reasonably large code base (over a million lines) that we had to leverage for our upcoming version of the product. Refactoring the entire code base would be time consuming and delay our product development. The team decided to refactor only the code segments that would interact with the new code. This was feasible as we could introduce only the necessary amount of refactoring with each iteration and focus on building the test driven development aspects of this functionality as part of the starting framework.

Pair Programming Environment

A typical software development shop layout is rows of cubicles. HP is no different. This layout is not conducive for pair programming. A better more conducive environment would be an open area where two programmers can sit together without much discomfort. We really had to work hard at this problem. The site facility-code and cost of shop remodeling prevented us from making the programming area more conducive to pair programming. Finally our solution was to designate an area with a few workstations where engineers can pair program as needed.

Pair programming is not possible unless the engineers believe in objective critiquing of each other's work. Our engineers, although they did not oppose the pair programming practice, did not warmly support it either. Therefore the feasibility team did not make a final decision about adopting pair programming. Alternatives such as "substitute code review for pair programming" and "review only the code if deemed necessary by the engineer developing the code" were proposed. The choice of pair programming was left up to individual engineers. Management also perceived pair programming as a low-productivity activity as two programmers were developing the same code segment. The industry perspective of pair programming is that it is a resource sink and the project managers have not yet internalized the gains in product quality. Lack of a concrete decision by the feasibility team created confusion about adopting pair programming. We recommend that when formalizing a shift, the decisions are clear and adequately communicated to the team.

Test Driven Development

The feasibility team understood the value of test driven development. The engineering community had not practiced the technique in the past and could not diligently support or down play the test driven development. The team anticipated that the majority of engineers would not be involved in the test driven development as it required simultaneous development of the test module and code. The team did not formally adopt the test-driven development but decided to provide training to the engineering community so that the interested engineers were capable of pursuing test-driven development. There has been strong emphasis on unit testing and merging code only if it has passed the smoke test.

Single Customer Voice – Customer Commitment
"Future product marketing team" which acts as a surrogate customer represents our customers. The success of XP depends heavily on customer involvement. We met with the marketing management and set fourth our expectations about their role and participation in the new paradigm. The future product marketing was delighted to find out that they would have a pivotal role in the product development and promised to work very closely with the development team. They also committed to go through XP training. However, we still don't have a single-point of customer voice. The team receives requirements from at least four different sources and organizationally consolidating those to one voice has not been possible. As such, the development team has designated the program manager as the representative of the voice of the customer, which prioritize and clarifies the requirement after communicating with these different sources.

Globally Dispersed Teams – Commitment form remote teams
As already mentioned, we are a geographically dispersed division with varying culture and software development practices. It is critical that we all believed in one practice and adopted it effectively. The feasibility team perceived it an absolute must that the remote teams understood our motivation to change and support it even if they wanted to continue with their own existing methodologies. It was also necessary that these teams delivered their functionality in a timely manner to maintain our release plan. We also emphasized the concept of sub-teams so that practices such as standup meetings, could take place at each location and decisions could be made at a higher level in a joint meeting of team managers.

Today, our stand-ups have become virtual meetings. This is a challenge for the teams to adjust to, however with the support from each project manager the transition has been positive and the team has been adjusting to the new development paradigm.

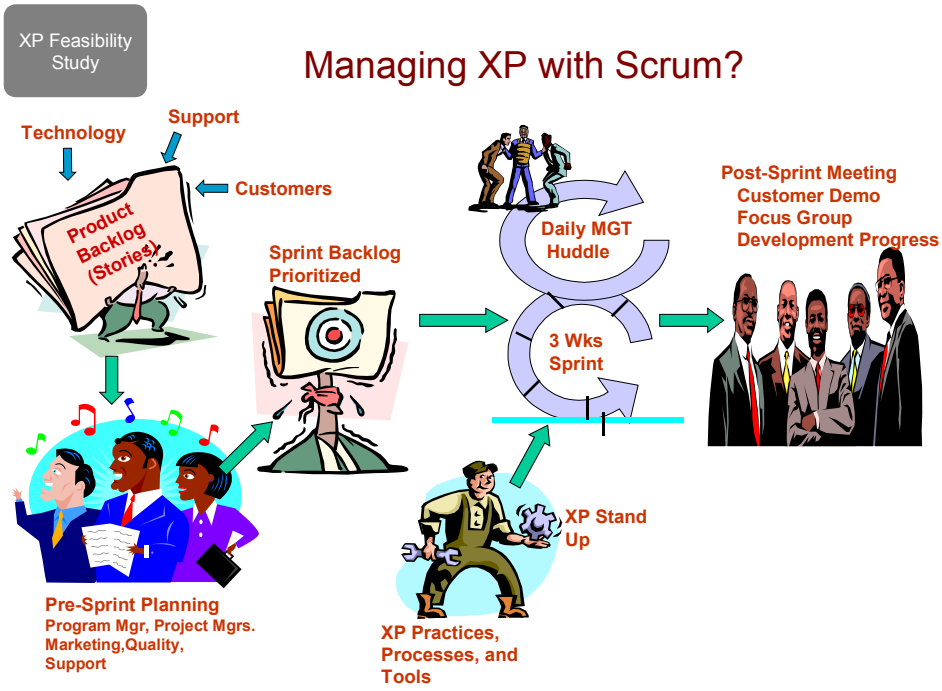
Test Automation
XP practice requires nightly builds and integration testing of all new code that is developed each day. Test automation provides an efficient means to achieve that objective. We only had a small amount of automation in place and needed much more. The quality manager took this requirement very seriously and started efforts to automate our testing as much as possible. We now have an adequate automation and are capable of testing nightly builds.

Creation of a Roll Out Plan
Our initial thought was to have a pilot team further explore XP and establish practices that will be effective in our environment. Once that stage was achieved we would be ready to spread it over a larger development community. However, we saw the momentum, and a desire to jump in with both feet and completely immerse ourselves in XP. The feasibility team developed a roll out plan to facilitate the adoption of XP through-out the division. The rollout plan included the following elements:

Training
We arranged for two levels of training, management and engineering. The management training included story telling, release planning, iteration planning, standup meetings, and conducting retrospectives. This management training, although added value, did not seem to be an absolute must. The engineering training was mostly focused on test-driven development, pair programming and refactoring and was well received. A few trainees then took upon the trainer role and conducted instruction for the entire engineering community.

XP Champions
We appointed XP champions on each site and an overall champion to facilitate the adoption of XP practice. They were responsible in motivating people to understand the XP traits and adopt them for software development. They were also responsible for creating necessary processes and documenting them for consistency and continuous improvement. The champions also assisted the program managers with creation of release and iteration plans.

Execution and management
The XP methodology manages development issues with standup meetings. Our environment was relatively more complex since it consisted of both hardware and software. We decided to manage the program issues using *Scrum* methodology [5]. Scrum defines a project management framework that includes a fixed development period called *sprint* to develop a subset of functionality. It has daily management meetings (huddles) to manage the program issues. We mapped each sprint to an iteration period. The results of this combination are represented in the illustration below. Currently the huddle takes place every week. As the development activities intensify, these huddles will be more frequent.



Progress Measurements
To evaluate our progress and implement continuous improvement we chose to measure our velocity (number of stories completed per iteration) and our efficiency, the ratio of actually implemented stories to planned stories. Since not all stories are created equal we established a technique similar to function point to determine the story complexity that provided a basis to normalize our stories. In essence, we assigned an index to a story based upon the number of classes (or methods), number of inputs and outputs and the number of database interactions. Each of these parameters was assigned a weight and the overall story index was the sum of the indices. For example, each database interaction had a weight of 2, so if there were 2 database interactions in a story, that contributed a weight of 4 to the complexity index. Although, this provides a sound basis for measuring the velocity and efficiency, it also has an overhead of determining a story's complexity index.

Execution – smooth start and a bumpy ride

The Partner Pleasure
We started the execution right after the training. Before really immersing ourselves into XP on a full-scale project, we chose to try out the main principles on a current product enhancement. This enhancement was intended to address some functionality issues that customers were facing. Since the customer support team was intimately involved with the customers it acted as the

surrogate customer. Going into this process, the support team's initial reaction was negative but, after realizing the power of the voice of the customer and experiencing a very high level of cooperation from engineers, the support team was ecstatic. We categorized the enhancements into themes and used each enhancement as an XP story. Each story was prioritized to generate a release plan. We did not use any other XP traits for this development as the project management felt that it would enable a quick turn around to make limited product enhancements.

Our intent was to use XP for an upcoming project. The marketing team had already collected the stories under several themes and defined the minimal product requirements. Both the marketing and development team then prioritized the stories. However, the development efforts needed to deliver these stories far exceeded the available resources. The marketing team had its mind set on these stories as "the minimal product requirement" while the engineering community created their own list of functional requirements and we violated the important XP principle of total customer involvement to provide the maximum value to the customer. The two artifacts, stories generated by the marketing team and functional requirements developed by the engineers, were merged over a period of time but that was an additional step in the process. This experience highlights the importance of understanding story creation, prioritization and ability to determine whether the schedule with required functionality can be met. The engineering team tried to utilize the functional requirements process to scope down the high-level stories defined by the customer, but the process was slow, difficult and caused confusion among the functional groups. We considered this as a lesson learned and hope not to repeat it in the future.

We also did not adhere to the true definition of a customer story. A true story must provide value to the customer. We created stories that were related to building the infrastructure. Although it helped us understand our immediate needs and help us get ready for the actual product development, there was an overhead incurred in writing and managing these stories. The engineers were not very enthusiastic about this activity.

We were using an existing requirements management system that was rather complex. When we started using it for our stories the system became unmanageable due a large number of stories organized into themes and required a new schema. We chose to use a simple spreadsheet until the system was improved. The principle of simplicity as emphasized by the XP practitioners must be adopted wherever possible. We are now using a web-based system in public domain to manage our stories and track progress.

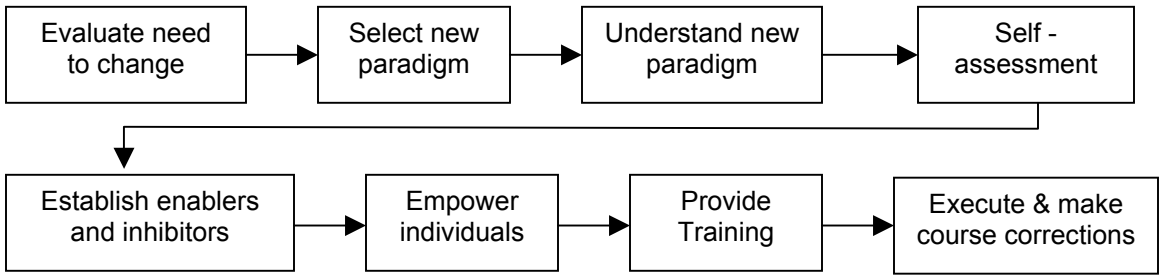
Where are we – did we make it?

It took us over four months to overcome our initial hurdles. We have now created an infrastructure, understood the important XP processes including empowerment of engineers to make important decisions. We have also made an effort to document our processes for possible future enhancement of our practices.

We have had reasonable success with release planning, iterations, refactoring, and test automation. We are not actively using pair programming and test-driven development, the two important XP traits to enhance the code quality. We do feel that we have better control on the requirements and releases and have successfully met the first hardware delivery checkpoint. We are tracking our progress by measuring our efficiency, the ratio of the actual number of stories completed to the stories planned. These are not normalized stories as we had intended. Since the project is in its early stages there is not enough evidence of any gains in product quality. Two out of five teams have adopted the XP practice.

Conclusion

It is our experience that the following pre-cursor helps in understanding the requirements of a paradigm shift and evaluates the readiness for such a shift thereby increasing the probability of success.



Driving for in-depth paradigm shift changes can be draining for the development team. In our case, the team kept internalizing the changes as “yet another process change” and only after awhile sensitivity to the phrase “process” was developed. However, continuous process improvement is necessary where the out of box solution is not available for something like XP.

We would like to summarize by saying “a paradigm shift should be well managed”. It is always tempting to try new methodologies without understanding the efforts and the level of commitment it will take to make the shift successful. Although we feel good about where we are today, the full potential of the change is yet be realized. There is a sense of victory from the fact that we now have our requirements under control.

References

1. Gupta, Bhushan B. and Rhodes, Steve; Adopting a Lifecycle for Developing Web Based Applications, Software Quality Week, 2003
2. Beck Kent, eXtreme Programming eXplained, Embrace Change, Addison-Wesley, 2000
3. Beck Kent; Fowler, Martin; Planning Extreme Programming, Addison-Wesley, 2001
4. Web Link - www.extremeprogramming.org
5. James Highsmith, Agile Software Development Ecosystems, Addison-Wesley, 2002

Personal Conversations

Cindy Schilling, Hewlett Packard Co
Debra Bacon, Former Hewlett-Packard employee

Case Studies in Adopting Software Quality Release Criteria

Web Program Title

Three case studies taken from actual Intel projects (Small, Medium, and Large) are discussed including the tailoring process, usage, and particular issues encountered by each.

Michael J. Anderson, Intel Corporation
JF1-46
2111 NE 25th Ave.
Hillsboro, OR 97214-5961
mailto:mike.j.anderson@intel.com

Version 3.0, 08/15/03

Mike Anderson has over 20 years of experience in the software industry including work as an application programmer and in process improvement. He has been with Intel Corporation for more than eight years and is currently the program manager for the Software Quality Release Criteria project in Intel's Corporate Quality Network organization. He holds a Bachelors degree in applied mathematics from California Polytechnic State University, San Luis Obispo and a Masters degree in applied mathematics from Oregon State University.

Abstract

Tracking a product's quality as it moves through its development milestones requires monitoring a variety of attributes. Implementing a methodology to do this that is both consistent and scalable is a challenge for most organizations. Intel has developed a Software Quality Release Criteria (SW QRC) model that provides product development teams of all sizes with a common framework for measuring and reporting project data up through final release. This paper describes approaches to implementing the SW QRC based on the experiences of three representative project teams at Intel.

Introduction

Software quality release criteria serve three functions in a project: Quality Planning, Quality Monitoring and Quality Reporting. During Quality Planning the quality release criteria helps a project team establish expectations for quality when they use it to set quality targets. In the Quality Monitoring role, the quality release criteria provide specific attributes of software quality to monitor and data to measure against previously established targets. The quality release criteria also enable effective Quality Reporting by supplying the information needed to report on software quality/risk and providing data to support a milestone release decision.

The key benefits of using the quality release criteria include: setting expectations for quality by establishing what will be measured, and when it will be measured, preventing a last-minute "scramble for quality". Usage also reduces risk to product quality by providing software quality data on a regular basis prior to the ship decision.

The Intel Software Quality Release Criteria model

The Intel Software Quality Release Criteria (SW QRC) model has been described in both Gatlin00 and Simmons01, a summary of the model and its usage follows. The current Intel SW QRC uses 16 software quality categories (see Table 1), called Assessments, based on actual usage by successful Intel product development teams.

Assessment	Description
Functional Testing	Evaluation of the testing of features and functions to determine whether the testing is adequate to meet the planned quality level.
Defect Data	Evaluation of product defect information.
Build and Archive	Evaluation of the ability to build the product on demand and accurately archive all software necessary to produce it.
Install Testing	Evaluation of install/uninstall testing to verify correct install and uninstall of product.
Customer Documentation	Evaluation of product user documentation for accuracy and consistency with product operation.
Compatibility Testing	Evaluation of the product's ability to perform its required functions while sharing its environment with one or more systems or components.
Legal Compliance	Evaluation of product compliance with legal requirements.
Continuous Operation	Evaluation of time-dependent product defects such as memory leaks and race conditions as indicators of product reliability.
Requirements Change Management	Evaluation of changes to product requirements.
Planning and Configuration	Evaluation of project scheduling and monitoring of items placed under configuration management along with changes to those items.
Regression Testing	Evaluation of the testing of the product to determine that changes have not caused unintended effects and the product still complies with requirements.
Customer Acceptance Verification	Evaluation of the testing performed to determine whether the product meets its customer specified acceptance criteria.
Performance Measurement	Evaluation of the performance attributes of the product to determine whether planned performance goals are met.
Standards Compliance	Evaluation of product compliance with product standards including Intel engineering or other standards.
Support Readiness	Evaluation of the product's readiness for services provided by the designated product support organization.
Localization	Evaluation of product compliance with specified language translations and linguistic testing.

Table 1. SW QRC Assessments

The basic structure of the SW QRC ACT model is shown in Figure 1 below. It includes Assessments, Criteria, and Targets. Each Assessment (software quality category) includes one or more Criteria, defined as specific measures of software quality. All Criteria have Targets, which are specific quality goals defined for each applicable product release Milestone. The Intel Product Life Cycle defines Alpha, Beta, and Gold Milestones. An example is given in Figure 2 below.

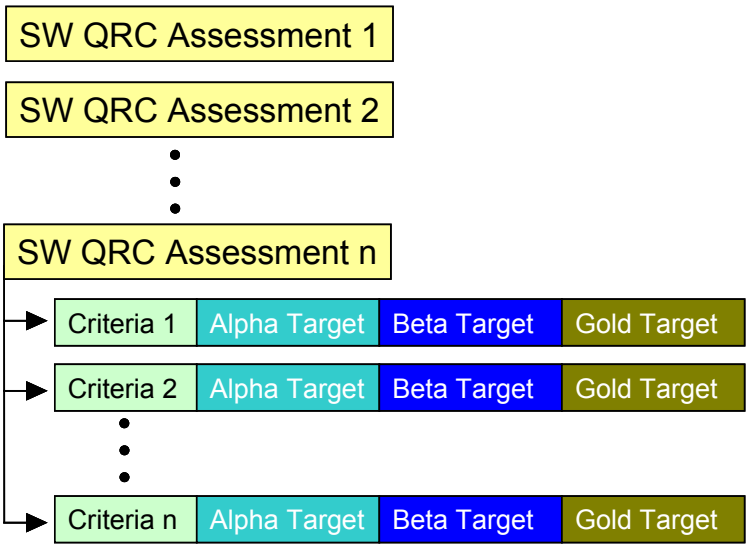


Figure 1. Intel SW QRC ACT model

Build and Archive

SWBA1	Alpha Target	Beta Target	Gold Target
The release candidate can be rebuilt on demand.	Alpha Release can be built on server hosting PDT's build environment. Plan in place to conduct "clean box" build at Gold Release.	Beta Release can be built on server hosting PDT's build environment. Dry run of "clean box" build performed.	Gold Release can be built on server hosting PDT's build environment. Gold Release is reproduced and verified on a "clean box".

Figure 2. Example Assessment Criterion and Targets

The SW QRC model has been implemented using Excel worksheets; see Figure 3 below for an example.

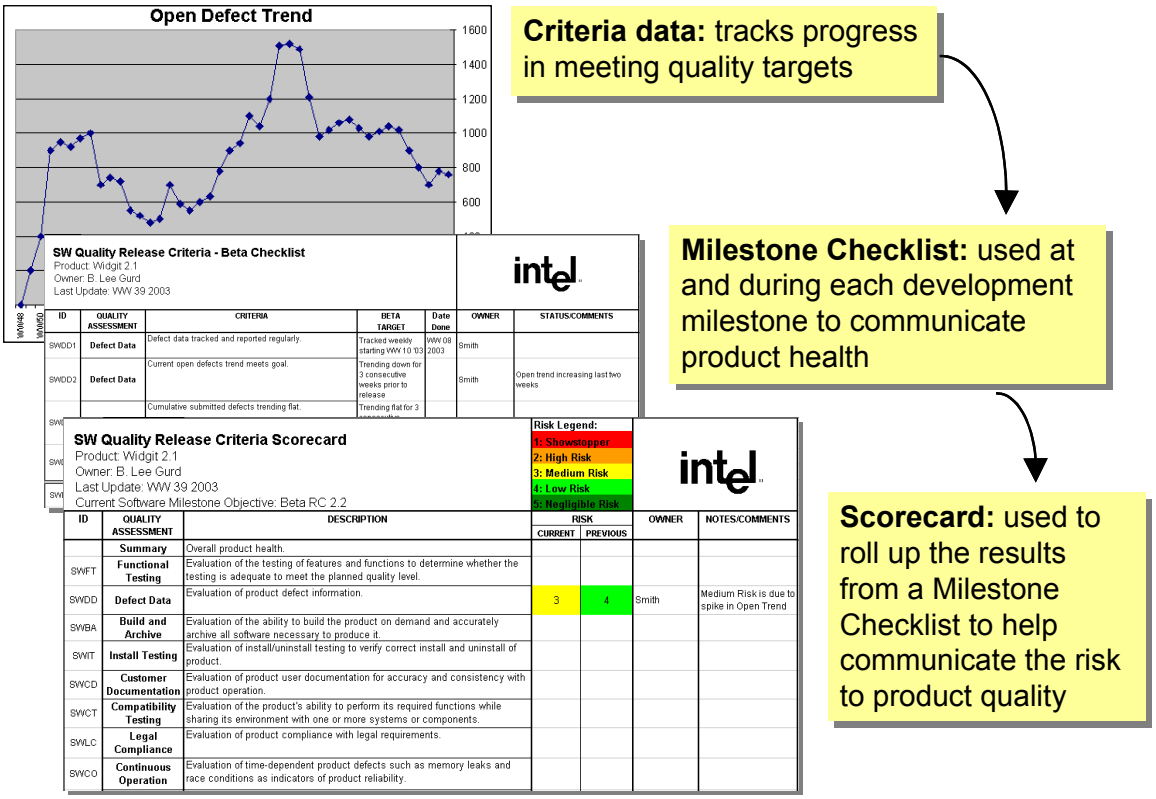


Figure 3. Example SW QRC Usage

Applying the SW QRC

Some Assessments only apply to certain situations or categories of products. For this reason, the Excel worksheets containing the Assessments and Criteria are intended as project templates. Each project team must customize and adapt them to fit their needs.

Case Studies

Three case studies taken from more than 30 Intel projects using the SW QRC and completed over the last few years are given below. They represent a range of adoption types: Small (< 10 person), Medium (> 10 and < 50 person), and Large (> 50 person). The tailoring process, usage, and particular issues each encountered will be discussed. In addition some general lessons learned about SW QRC adoption will be noted.

Small Project (7 – 8 person team)

Project description: This team was developing the hardware and software for an Ethernet adapter driver in collaboration with a third party OEM. The team used the SW QRC to provide the Gold release criteria. The project was already using a small set of release criteria, but recognized that the criteria being used provided inadequate coverage in many areas.

Tailoring process: The project manager (PM) worked with other team members to convert their existing release criteria to the SW QRC framework. This involved the following specific steps:

1. The PM downloaded the SW QRC sample materials from the internal Corporate Quality website.
2. The PM then met with the Development and SQA leads to define the Assessments to use.
3. A business unit software quality process engineer provided tailoring assistance.

The whole process took an afternoon.

Usage: The PM used the Assessments as an informal, personal checklist. Grading was based on bug scrub information, status reports, and 'asking around'. The PM rolled up the SW QRC data by simply attaching the scorecard to his status reports.

Key Findings: This informal approach worked reasonably well, except for the following:

1. Using the scorecard as a personal checklist meant that the data wasn't available to the SQA leads.

2. The team had difficulty determining when testing would be complete - the '95% done' problem.
   (For reference, the '95% done' problem refers to a task that is reported as 'nearly complete' or 'almost finished' for a long period of time - sometimes for 95% or more of the duration of the project. One way to specifically address this is to break the task down into a series of sub-tasks and then closely monitor progress against schedule for each of them. This will help identify blocking issues early and allow the team to direct resources appropriately to get back on track.)

3. The team discovered that it needed to devise a better way of measuring OEM-related schedule risks.

Medium Project (20 - 25 person team)
Project description: This team was developing software for a dual processor server including adapting software provided by a very large third party vendor. The team was also already in the process of developing a metrics program when the SW QRC became available. They used the SW QRC as a problem detection tool as much as a set of release criteria.

Tailoring process: The Quality and Reliability Engineer (QRE) took the following steps:

1. Networked to identify stakeholders.
2. Solicited participation and lessons-learned data from stakeholders.
3. Assembled documents and checklists used on previous projects.
4. Downloaded SW QRC sample materials from the internal Corporate Quality website.
5. Along with a small group of stakeholders, drafted proposals for Alpha, Beta, and Gold milestones.
6. Circulated several drafts until a consensus version emerged.
7. Held a series of one on one meetings with line managers to get their feedback and buy-in on the consensus version.

This tailoring process took several months to complete. Why did it take so long? The primary reasons were due to the fact that the product consisted of a mixture of new and reused code. For reused code, data from old Software Quality Plans made it easy to arrive at a consensus. For new code, however, consensus was much more difficult and involved much "dart boarding" about the number of defects that would be 'acceptable' for each set of milestones. Criteria also had to be established for what constituted 'new' or 'reused' code. The negotiations to come up with the criteria took time.

Usage: The Scorecard and Milestone Checklists were used to track weekly progress. Grading was based on data received from criteria owners. The information was rolled up and presented at weekly project staff meetings.

Key Findings: Use of the SW QRC uncovered three unexpected problems:

1. Ambiguous terminology, e.g. 'Alpha' – Alpha candidate or Alpha milestone?

2. Dealing with code branches - multiple branches of the code base mean multiple branches of the Scorecard.

3. Metrics infrastructure, e.g. who will maintain metrics tools developed for the project?

The confusion over terminology, such as what a specific milestone (Alpha, Beta, etc.) 'means' is quite common. To developers, 'Alpha' (for example) often means a candidate release. To the test team, 'Alpha' often means a discrete event that occurs when all the Alpha release criteria are met. This means you can get very different answers from different team members to the question, 'Are you at Alpha yet?' To developers the answer may be 'Yes,' while to test the answer may be 'No.' In order to address this issue it is important for product development teams to create clear definitions for each project milestone and make sure all team members understand them.

Note that outside dependencies on third party components are a typical cause for a project to split into multiple code branches. Typically, milestone definitions assume there will be only a single branch of the code. If a project plans on using third party software it is important to take this into account.

Automated metrics tools are a good way to generate the data necessary to determine if you've met your SW QRC targets. However, once the tools have been developed they will require at least a part-time resource to maintain and adapt them to the inevitable changes that a software project undergoes. Problems like ambiguous terminology and dealing with code branches often have been lying around a long time, waiting to be discovered. The SW QRC is extremely helpful in raising the visibility of these problems.

Large Project (80 people organized into 5 sub-teams at 3 geographically distributed sites)
Project description: This project was developing software for a TV set-top box Internet appliance and server. It was composed of an end-to-end integration team, a client software team, a server software team, a software tools team, and a hardware team. The SW QRC was seen as a way to bring commonality to SQA efforts across geographically dispersed sites

Tailoring process: Each of the five project sub-teams tailored their own set of Assessments. There was no attempt to coordinate the SW QRC tailoring, but that was consistent with the decentralized planning throughout the project. Each one did the following:

1. Downloaded the SW QRC sample materials from the internal Corporate Quality website.
2. Formally reviewed the materials with the Development and SQA leads from each team. This took about 4 hours.

Usage: Each team separately tracked their progress in their respective Scorecards and Milestone Checklists.

Key Findings: Five sets of data meant:

1. That the SW QRC didn't represent a holistic view of the product.
2. Information was not being shared between sub-teams.
3. Each team defined terms and evaluation procedures in ways that were hard to reconcile.
4. It was no easy task to summarize the data and pass it on to management.

The teams didn't realize that the SW QRC also needed a support infrastructure, with definitions for: roles and responsibilities, reporting mechanisms, grading criteria, and how to interpret and act on data.

Specific questions that came up when trying to come up with a common way to report data in each separate Scorecard included the following:

- Should the Scorecard Summary risk value always default to that of the current highest risk level reported by any Assessment?
- Has management been briefed on how to interpret the Scorecard?
- How will you grade Assessment criteria that aren't due yet?

Lessons Learned

1. There is more to tailoring than selecting which quality criteria to use on your project. It is important to also define the supporting process for each criteria as well.

2. Using software quality release criteria will uncover latent problems in the software development process. These problems are more likely to involve the software development process rather than the software quality release criteria themselves. It is important to continuously monitor and manage these problems as they are detected.
3. Management needs to be actively involved in software quality release criteria tailoring and use.
4. Setting criteria targets is not an exact science. For example, among the projects cited in this paper, more than one team reported problems trying to calibrate how many defects of a given priority were 'acceptable'. Finding appropriate industry benchmark data can be helpful in determining appropriate target ranges.
5. It is important to keep the overall objective in mind – releasing a successful product. This means that tailoring issues have to be treated as the business decisions that they are. There is no single 'right' technical answer. The goal of the SW QRC is to support informed decision making by management, not dictate decision making in a bureaucratic sense.

Reporting Issues

Determining how to report the on-going health of a project based on the current status of the individual release criteria can be a challenge. As mentioned earlier, grading criteria that aren't due yet is not a straightforward process. For example, consider the case of a software driver that requires an external certification by a standards body before it can be released. Clearly, if the certificate has not been granted it is a showstopper issue. However, early in the project it isn't expected to be obtained for quite some time. How should this criteria be graded in the meantime and how should that affect the overall risk to the project? One approach is to default the project to the highest risk of any of its criteria. In the example of obtaining the external certificate, if it is graded as a showstopper risk, the overall product summary is also at showstopper risk. It is up to the project manager or project leads to determine if this approach will convey an accurate picture of actual product risk to their management and others outside the team.

Applying 'worst-case' assumptions can reduce the usefulness of the SW QRC. If the summary is red (showstopper risk) all the time, management is not receiving useful data that can be used to make informed decisions. A more sophisticated approach is needed if the SW QRC is to be used as part of an ongoing health check.

This philosophical issue has surfaced many times during the development of the Intel SW QRC. To summarize the approaches:

1. In 'go/no-go' mode, the SW QRC is used simply as an exit criteria to determine whether or not a milestone has been achieved. This is the most literal interpretation of 'quality release criteria'.

2. In 'ongoing health check' mode, project data is monitored continuously and compared to the SW QRC assessments to determine whether the product is proceeding satisfactorily.

When using the ongoing health-check mode, a more sophisticated rollup comes in because the assessments now have to be measured over time or against progress to schedule. Note that either approach works, but everyone who reviews the data must share the same understanding and expectations of what is being reported.

Adoption Recommendations

For Small Projects:
These recommendations are based on a straightforward approach to adoption that was generally successful for most small projects.
1. PM and Test lead tailor the SW QRC.
2. PM tracks the status of each criteria.
3. Results are shared with rest of team on regular basis.

For Medium Projects:
These recommendations are based on refinements the medium sized project team introduced during follow-on adoptions of the SW QRC.

1. Tailor the SW QRC with input from the project leads.
2. Use the Goal-Question-Metric paradigm (see Basili92) to design the metrics needed to support the SW QRC.
3. Design a metrics repository database schema.
4. Set up automated collection mechanisms.
5. Administer and collect the metrics data.
6. Make collective decisions based on the metrics data.
7. Review and redesign the metrics as appropriate.
8. Repeat steps 3 through 7 as necessary.

For Large Projects:
These recommendations are the result of a post project review held by the large project team that identified ways they could avoid repeating their biggest adoption problems in the future

1. Work through a Joint Engineering Team made up of project leads and key stakeholders.
2. Develop a SW QRC Assessment roadmap (i.e. a high level schedule of Assessments planned for adoption) for the product as a whole.
3. Work out grading, reporting, and rollup procedures in advance.
4. Work with management to assign roles and responsibilities.
5. Once the product-level roadmap is ready, each team develops its own set of supporting Assessments
6. Each team expands the product-level roadmap from its perspective.
7. Craft team Assessments in support of the product-level Assessments.
8. Determine who will collect, report, and summarize SW QRC data, and agree on how they will do it.
9. When complete, document the process to make it reorganization-proof.

The bottom line for large projects is that a coordinated roadmap is essential!

Conclusion

The Intel SW QRC has provided a flexible and systematic method for product development teams of all sizes to successfully develop and release software that meets clearly defined quality criteria. The more than 30 Intel projects that have successfully adopted it represent a wide range of product types. In addition, several Intel divisions now use the SW QRC as a standard part of their software development process. It provides a way to 'bootstrap' more systematic software development without requiring burdensome process overhead to get started.

The case studies described in this paper are intended to indicate that teams are likely to encounter different sets of SW QRC implementation problems depending on a number of factors including: size and organization of the project, geographic dispersion of the project, availability of historical data, existence of prior release practices, and existence of metrics tools. Some of the specific adoption issues that teams have encountered were discussed as well as approaches to avoiding them in future projects. It should be noted, however, that the SW QRC is not a 'silver bullet' that guarantees success to every project that uses it. Like any tool, it is dependent on the skill of the individual or team using it.

References

Basili92 — Basili, Victor R., *Software modeling and measurement: The Goal/Question/Metric paradigm*. Technical Report, CS-TR-2956, Department of Computer Science, University of Maryland, College Park, MD 20742, September 1992.

Gatlin00 — Gatlin, Manny, and Hannon, Gregory, *Managing Software Product Quality*, presented at the Pacific Northwest Software Quality Conference, 2000

Simmons01 — Simmons, Erik, *From Requirements to Release Criteria: Specifying, Demonstrating, and Monitoring Product Quality*, presented at Pacific Northwest Software Quality Conference, 2001

The Evolution of Model-based Process Improvement: Challenges and Opportunities for Software Developers

By Barbara Hilden, CEO
Process Improvement for the 21st Century, Inc.

8601 Beech Hollow Lane
Springfield, VA 22153
Email: BHilden@PI-21.com

Voice: (703) 447-2001
Fax: (703) 866-5556
www.PI-21.com

Abstract

Since 1991 there have been several models against which a large organization may implement process improvement. These include SEI's Capability Maturity Models for software, software acquisition, systems engineering, and people, as well as other models such as EIA/IS 731.1 (SECM). These models have been valuable to many organizations; however the use of multiple models has been complicated and expensive. Applying multiple models that are not integrated is costly in terms of training, assessments, and improvement activities.

The Software Engineering Institute's (SEI) Capability Maturity Model Integration (CMMI[SM]) product suite includes the framework for generating reference models and associated training and appraisal materials. These reference models reflect content from the various disciplines and are intended to help organizations establish and implement process improvement. The CMMI framework is the current state in the evolution of model-based process improvement reflecting best practices in software and system development. Use of reference models to benchmark organizational capability and prioritize improvement is a proven practice.

Biography of the Author

Barbara Hilden, CEO, has over eighteen years of experience in the computer industry. She is an authorized Lead Assessor for SCAMPI and CBA IPI Assessments, Lead Evaluator (SCE V3.0) and internal auditor for ISO 9000. She has implemented process improvement initiatives, including CMMI, SW CMM, SA CMM, EIA/IS 731 SE CM, and ISO 9000, at various corporations. Ms. Hilden has led or participated in over 30 formal appraisals and multiple audits and mini-appraisal for the SW-CMM, the FAA-iCMM, ISO 9000, EIA/IS 731 SE CM, and the CMMI in a wide variety of organizations from very small to global organizations.

Acting as a consultant, Ms. Hilden has assisted organizations in determining and subsequently implementing effective strategies for process improvement efforts at a wide variety of organizations. She taught and developed a wide variety of computer science, software engineering, systems engineering and process improvement courses. Ms. Hilden has been responsible for managing government systems development programs as a government contractor and has led numerous research and development programs in the field of signal processing while working as an instructor in the university environment. Ms. Hilden has published and presented numerous papers and books on management, software engineering methodologies, and process improvement.

The Evolution of Model-based Process Improvement: Challenges and Opportunities for Software Developers

Since 1991 there have been several models against which a large organization may implement process improvement. These include SEI's Capability Maturity Models for software, software acquisition, systems engineering, and people, as well as other models such as EIA/IS 731.1 (SECM). These models have been valuable to many organizations; however the use of multiple models has been complicated and expensive. Applying multiple models that are not integrated is costly in terms of training, assessments, and improvement activities.

Major systems development today often requires integrated systems engineering and software engineering activities and components. That is, the technical disciplines and processes are tightly coupled to facilitate rapid and efficient development of complex products. It is difficult to get an accurate picture of just the software processes or just the systems engineering processes since they are so tightly coupled. This is especially true for organizations that are structured around Integrated Product Teams (IPTs) and customer-centric processes.

Most of the maturity models address many common areas such as training, planning, process infrastructure, defect prevention, configuration management and subcontract management, however there are differences in model architecture, content, and approach even within those common areas. These differences have limited the benefits of process improvement programs by forcing organizations to delute their processes with redundant and sometimes conflicting activities in order to comply the multiple models.

The Software Engineering Institute's (SEI) Capability Maturity Model Integration (CMMI) product suite includes the framework for generating reference models[1] and associated training and appraisal materials. These reference models reflect content from the various disciplines and are intended to help organizations establish and implement process improvement. The CMMI framework is the current state in the evolution of model-based process improvement reflecting best practices in software and system development. Use of reference models to benchmark organizational capability and prioritize improvement is a proven practice.

But is adopting or transitioning to CMMI going to improve your development time to market, cost or product quality? What are the differences – positive or negative – that development managers and practitioners face in transitioning to CMMI? Driving the need for answers and decisions is the fact that the legacy models and appraisal methods will soon no longer be supported. This article outlines fundamental differences between CMMI and the SW CMM® and the reasons behind the SEI's creation of the CMMI product suite. We then use this background to present actions software development organizations can take to answer the fundamental question: is CMMI right for us? If so, which representation, and which disciplines should we consider?

---

[1] CMMI for Systems Engineering/Software Engineering/Integrated Product and Process Development/Supplier Sourcing, Version 1.1, Continuous Representation (CMMI-SE/SW/IPPD/SS, V1.1, Continuous); CMU/SEI-2002-TR-011. CMMI for Systems Engineering/Software Engineering/Integrated Product and Process Development/Supplier Sourcing, Version 1.1, Staged Representation (CMMI-SE/SW/IPPD/SS, V1.1, Staged); CMU/SEI-2002-TR-012. Both are available for download from www.sei.cmu.edu/CMMI free of charge.

Background

The use of reference models for process improvement is based on the premise that the quality of the product being developed is highly dependent upon the quality of the process used to develop it. As used in this context, the term process is meant to include the methodology, the technology, and the skills of the individuals or groups employing them, to develop products or to manage or support that development. Hence, there are processes for development, development management, and development support. Each of these processes, in turn, can be broken down into sub-processes, demonstrated by the examples in the following table:

Table 1: Example Arrangement of Processes and Sub-Processes

Processes	**Sub-Processes**
Development	Requirements, Design, Construction, Test
Management	Project Planning, Cost Management, Schedule Management, Resource Management, Risk Management
Support	Configuration Management, Quality Assurance and Quality Control, Training

Processes and sub-processes are intended to be implemented and institutionalized, i.e., they document the way business is done within the organization. To achieve that goal, processes need to be well-defined and effective. This means that the processes must meet defined criteria regarding their stability, usability and maintainability.

The two primary reference models used by software and systems engineering developers to document and improve processes prior to CMMI are the SEI's CMM for Software, v1.1, and the EIA/IS 731.1, Systems Engineering Capability Model (SECM)[2]. The balance of this paper focuses on the CMM and CMMI differences.

The CMM was published by the SEI in 1993 (CMU/SEI-93-TR-024 and CMU/SEI-93-TR-025) as the culmination of nearly a decade of effort to provide a framework of best practices for software development. The CMM is considered to be a staged model; i.e., maturity levels provide a recommended order for approaching process improvement across the organization. The five Maturity Levels (stages) are:

1. Initial
2. Repeatable
3. Defined
4. Managed
5. Optimized.

A Maturity Level is defined as an evolutionary plateau toward achieving a mature process and reflects a range of expected project performance (capability). Increasing organizational maturity is correlated with better expected process performance (capability). In other words, projects within higher maturity organizations are believed more likely to deliver on-budget, on-schedule with increasing productivity.

---

[2] For clarity and completeness, it must be noted that SEI commenced work on a version 2.0 of the CMM in the mid-1990s. Drafts up through version "c" were issued for comment. However, the results of this modernization were subsumed into CMMI. For practical purposes reflecting actual community usage, this paper therefore focuses on CMM v1.1, not CMM v2.0c, and CMMI v1.1.

Each Maturity Level contains Key Process Areas (KPAs), analogous to the sub-processes discussed above. Each KPA is a cluster of related activities that, when performed collectively, achieve a set of goals considered important for establishing process capability at that Maturity Level.

Each KPA contains 5 sets of Key Practices divided into Common Features:

1. *Commitment* to perform
2. *Ability* to perform
3. *Activities* performed
4. *Measurement and Analysis* of performance
5. *Verification* of performance

The Common Features further break down the Goals of each KPA in the institutionalization and implementation Key Practices. The Activities Key Practice reflects implementation and the other four represent institutionalization of the activities. An organization using development, management and support processes that implement the key practices of a KPA have implemented and institutionalized (achieved) the KPA goals. An organization appraised as having satisfied all the goals of all KPAs at the target maturity level (and all levels below it) is deemed to have satisfied that maturity level.

An Integrated Approach

Large scale developers of mission- and/or safety-critical products often found themselves working to contracts requiring compliance with multiple maturity/capability models. This typically resulted in multiple preparatory and baseline appraisals annually. The associated cost of training, process development and maintenance, and the appraisals to support separate models was high. Worse, separate appraisals performed at different times, on segregated organizational units, used different teams. It was not unusual for the different appraisals to result in differing or even contradictory findings and recommendations. Another major problem was the need to synchronize domestic and international standards. The CMMI model successfully combines multiple disciplines, such as software and system engineering to support integrated training and appraisals.

Several premises were fundamental to this approach. First, that process capability is directly proportional to institutionalization. Increasing process sophistication and/or institutionalization, as already discussed, is believed to correspond with increased capability and effectiveness in process implementation. Secondly, additional flexibility in model implementation and interpretation, calling for additional and more fundamental judgments, reduces prescriptiveness and allows for broader organizational application. Finally, typical development organizations need to implement process improvement to meet the needs of their business environment while complying with the model.

The CMMI product suite is intended for organizations providing services, or acquiring or producing systems and/or software. SEI therefore included both a staged and continuous representation of the reference model to facilitate transition from the staged CMM or the continuous SECM architectures, respectively. Nevertheless, use of either representation is not restricted; software or systems developers can and may use either representation. In addition, organizations developing only software or only systems need to ensure that they implement using the appropriate CMMI discipline (CMMI for SW or CMMI for SE/SW). Integrated Product and Process Development (IPPD) and Supplier Sourcing (use of outsourcing and subcontractors) are additional disciplines available within the CMMI product suite. Future discipline integrations into the CMMI product suite are expected.

CMMI - Staged

The CMMI models include a staged representation analogous to the staged architecture of the CMM. The staged representation includes five familiar Maturity Levels with groupings of process areas

very similar to the SW CMM. Similarities between the CMMI Process Areas (PA) and CMM KPAs are apparent in their titles and purpose. However, these similarities tend to mask very significant differences between the models. Not only is the CMMI larger than CMM (more PAs than KPAs), it represents a fundamental change in model architecture, approach, content and application.

Architecture. As with the CMM, CMMI PAs include goals which are achieved by implementing and institutionalizing practices. However, the CMMI goals are further separated into Specific and Generic Goals. Specific Goals, analogous to the CMM KPA goals, are achieved by performance (implementation) of the Specific Practices. Generic Goals, however, are achieved by implementing Generic Practices that support institutionalization. Generic Practices are grouped into Common Features which are similar to the SW CMM's Institutionalization Common Features (Commitment, Ability, Measurement & Analysis, and Verification). These Common Features are:

- Commitment to Perform (CO),
- Ability to Perform (AB),
- Directing Implementation (DI) and
- Verifying Implementation (VE)

These Common Features are used to organize the generic practices of each process area. Common Features are not rated in any way, but provide a way to present the generic practices so that they are easier to understand and remember. The table below shows the Staged Goals and Practices and their associated Common Feature.

Table 2: CMMI – Staged: Generic Goals and Practices

Generic Goal	Generic Practices
GG2: Institutionalize a Managed Process	GP2.1 Establish an Organizational Policy (CO) GP2.2 Plan the Process (AB) GP2.3 Provide Resources (AB) GP2.4 Assign Responsibility (AB) GP2.5 Train People (AB) GP2.6 Manage Configurations (DI) GP2.7 Identify and Involve Relevant Stakeholders (DI) GP2.8 Monitor and Control the Process (DI) GP2.9 Objectively Evaluate Adherence (VE) GP2.10 Review Status with Higher Level Management (VE)
GG3: Institutionalize a Defined Process	GP3.1 Establish a Defined Process (AB) GP3.2 Collect Improvement Information (DI)

In the staged representation, each process area has only one generic goal. The generic goal describes what must be achieved to ensure process implementation is a part of the way the organization does its everyday business. The generic goal that a process area contains depends on the maturity level of the process area. All of the level 2 processes contain the following generic goal: GG2 Institutionalize a Managed Process. All process areas at level 3, or higher, contain generic goal 3: GG3 Institutionalize a Defined Process. Each level builds on the last so that in order to achieve maturity level 3 the organization must satisfy maturity level 2 and 3 process areas, and apply generic goal 3 to all these process areas.

One might think of the CMM KPAs as sets of best practices, surrounded by sufficient infrastructure to ensure that the activities are implemented and goals are achieved in the course of business. With CMMI, however, the level of infrastructure is extended. For example, to satisfy CMMI PA goals, an organization's processes must not only implement the Specific Practices, but must also institutionalize them by incorporating the Generic Practices into their implementation. Under the legacy

model it was necessary to have a process policy, assign process responsibilities, implement process training, define and use measures associated with use of the process, and verify process implementation. In addition to these elements, each process under CMMI must be planned and process performance tracked against that plan. Configuration management of process work products and stakeholder involvement in the process are also addressed.

In general, process areas that address similar concepts to those included in the GPs are critical aspects of institutionalization. It is important to be aware of the extent to which the generic practice may reinforce expectations set elsewhere in the model or set new expectations that should be addressed. Some of these similarities are shown in Table 3.

Table 3: Generic Practices and Affiliated Process Areas - Examples

Generic Practices	Affiliated Process Areas (not all inclusive)
GP2.2 Plan the Process	Project Planning
GP2.5 Train People (AB)	Organizational Training
GP2.6 Manage Configurations (DI)	Configuration Management
GP2.8 Monitor and Control the Process	Project Monitoring & Control and Measurement & Analysis
GP2.9 Objectively Evaluate Adherence	Process & Product Quality Assurance

Consider this example: GP 2.2 "Plan the Process" when applied to the Project Monitoring and Control process area is usually covered by the achieving the specific goals of the Project Planning Process Area. On the other hand, GP 2.2 "Plan the Process," when applied to the Project Planning process areas is elaborated as "Establish and maintain the plan for performing the project planning process," therefore it is expected that you also plan the approach that is to be taken to create the plan for the project, including ensuring that you have processes for creating the elements of the project plan such as estimates, schedules, role definition, and obtaining commitment to the plan. This would also include establishing estimates and assigning resources for actually creating the project planning work products. In other words, in this case, this GP addresses planning the planning process.

Generic practice 2.5 "Train People" is related to the Organizational Training process area. In this case the elaboration of the GP often gives examples of the training that people using or supporting the process need. Likewise, the elaboration of GP 2.6 "Manage Configuration" which is related to the PA Configuration Management often includes work products to control for the subject process area.

At first glance, it may seem that CMMI is introducing added scope and complexity, but when applied with the organizational structure and business objectives in mind the CMMI fits modern development domains, and assists the organizations in optimizing the efficiency and effectiveness of their processes through the Generic Practices.

Approach and Application. While the CMM KPAs typically contained sufficient content (practices and requirements for documented procedures) from which organizations could create their processes directly from the model, provided that they did not already exist. Many organizations using CMM drafted their policy, process, procedures and standards (e.g., format and content documents) for each individual KPA. Minimum levels of implementation (e.g., number of audits, number of peer reviews, etc.) and institutionalization (e.g., 3 months, 6 months, or 2-3 per cycle use) were targeted to demonstrate compliance. The litmus test during an appraisal was simply to determine whether or not the associated model practices were implemented in the organization and/or project processes. CMMI, however, calls for a more business centric approach.

CMMI encourages an organization to plan realistic processes rather than creating a process that fit the model rather than the organization. Generic Practice 2.2 requires that an organization establish and maintain the plan for performing the processes which correspond to each PA. The project or organization will therefore need to establish a realistic process within their organizational structure for implementing each PA in their real-world environment. The model provides guidance to assist organizations in constructing realistic plans that include the necessary detail for ensuring processes can be accomplished without over allocating project resources.

As part of this institutionalization, each project or organization wishing to establish a process must describe both the process and the objectives that process is intended to achieve. The project or organization must determine its own needed level of supporting procedures and standards. In other words, minimums for implementation and institutionalization will be set by the project or organization, not by the model, as it should be. This approach is less prescriptive, and it guides organizations to the establishment of processes which fit their organization and comply with the model rather than processes based on a model which may or may not fit the organization.

The PA reflects those Specific Practices which are foundational to Specific Goal achievement while the Generic Practices compliment the institutionalization of the specific practices. The culmination of the specific practices with the generic practices forces the organization to focus on their business needs rather than model requirements. Although this was the intended case with the CMM, the larger breadth and scope of CMMI makes it imperative that organizations take a more realistic approach to their implementation.

These differences in approach demonstrate the fact that CMMI represents an increase in the height of the bar in measuring organizational maturity, particularly at the lower end of the scale. They are now required to establish, early on, the foundation needed to continue to mature as an organization. In summary, applying CMMI will require an understanding of their business environment and their process domain, in order to make sound judgments which will fill in the process blanks, establish business process interfaces and create a complete organization or enterprise-level infrastructure.

Content. Table 4, below, shows the CMM Key Process Areas and the CMMI Process Areas, by maturity level. CMMI includes 75 Specific Goals with 458 practices compared with the CMM at 52 goals and 316 Key Practices[3]. A quick review of the table indicates a larger and broader model in the following:

- Level 2:
  - One additional PA in CMMI Level 2 versus CMM Level 2;
  - Expansion of the CMM Measurement & Analysis Common Feature into a CMMI Level 2 PA;
- Level 3:
  - 7 additional PAs in CMMI Level 3 versus CMM Level 3;
  - Expansion of the Integrated Software Management CMM KPA into 2 CMMI PAs (Integrated Project Management and Risk Management);
  - Inclusion of 2 CMMI PAs at CMMI Level 3 to address IPPD: 2 Specific Goals in Integrated Project Management and Integrated Teaming;
  - Expansion of Software Product Engineering into 4 CMMI PAs (Requirements Development, Technical Solution, Product Integration and Validation);
  - Expansion of testing concepts in Software Product Engineering into validation concepts in CMMI;
  - Expansion of inspection/review concepts in Peer Review into verification concepts in CMMI;

[3] Phillips, M., "CMMI Version 1.1: What Has Changed?", www.stsc.hill.af.mil/crosstalk/2002/02/phillips.html, February, 2002.

    - Expansion of the concepts in Intergroup Coordination into the integration concepts in CMMI;
    - Addition of an advanced PA to augment supplier management in Integrated Supplier Management.
    - Addition of formal decision methodology in Decision Analysis & Resolution.
- Level 4
    - 2 KPAs versus 2 PAs
    - Process and product performance measures, separated in CMM, are integrated into process and project performance management in CMMI.
- Level 5
    - Focus from prevention to root cause analysis in CMMI.
    - Integration of process and technology change management under Organizational Innovation & Deployment.

Table 4: CMM KPAs and CMMI PAs

Maturity	CMM KPA	CMMI PA
1	None	None
2	Requirements Management Software Project Planning Project Tracking & Oversight Software Subcontract Management Software Configuration Management Software Quality Assurance	Requirements Management Project Planning Project Monitoring & Control Supplier Agreement Management Configuration Management Process & Product Quality Assurance Measurement & Analysis
3	Organization Process Focus Organization Process Definition Training Program Integrated Software Management Software Product Engineering Peer Reviews Intergroup Coordination	Organization Process Focus Organization Process Definition Organizational Training Integrated Project Management for IPPD Risk Management Integrated Teaming Integrated Supplier Management Requirements Development Technical Solution Product Integration Verification Validation Organizational Environment for Integration Decision Analysis & Resolution
4	Quantitative Process Management Software Quality Management	Organizational Process Performance Quantitative Project Management
5	Defect Prevention Technology Change Management Process Change Management	Causal Analysis & Resolution Organizational Innovation & Deployment

The literature now includes a mapping[4] of the CMM KPAs to the CMMI PAs. This resource indicates a strong correlation in content between the models and can provide the starting point for facilitating transition between models. Mapping of individual organization's processes to the CMMI PA Specific Practices, a first organizational step toward CMMI use, will no doubt uncover other content differences (positive or negative gaps) between CMM and CMMI.

[4] USAF Software Technology Support Center (STSC): CMMI-SE/SW v1.1 to SW-CMM v1.1 Mapping.

CMMI - Continuous

This CMMI representation includes essentially the same content as the staged representation (same PAs, Specific Goals and Specific Practices[5]) in a continuous model format; i.e., capability levels provide a recommended order for approaching process improvement within each process area. Instead of grouping by Maturity Level, the PAs are grouped into

- Project Management
- Process Management
- Engineering
- Support areas.

For each process area, a capability level consists of related specific and generic practices that, when performed, achieve a set of goals that lead to improved process performance. When rating a specific goal relative to a target capability level, all specific practices through the target capability level associated with the specific goal must be investigated.

The continuous representation of the model has six capability levels, however these capability levels are assigned to individual process areas not to groups of process areas as is done in the staged representation of the model-- In other words, an organization may pursue higher capability levels on selected high pay-back processes or categories of processes. The capability levels are numbered 0 through 5. In this representation each process area has five generic goals; one for each capability level except zero. Capability levels are achieved through the application of Generic Practices (GPs) or suitable alternatives; however there are subtle differences between applying capability level 1 and 2 GPs and applying capability level 3, 4, and 5 GPs that will be clearer as we describe each capability level.

- **Capability Level 0: Incomplete**. One or more of the specific goals of the process area are not satisfied. There are no Generic Goals or Generic Practices at this capability level.
- **Capability Level 1: Performed**. Reaching capability level 1 for a process area is to satisfy the specific goals of the process area. This will satisfy Generic Goal 1- Achieve Specific Goals. This generic goal is supported by generic practice 1.1 - Perform Base Practices. (Note: The term base practice refers to the capability level 1 specific practices of the process areas in the continuous representation of the model.)
- **Capability level 2: Managed**. A managed process is a performed process that satisfies Generic Goal 2, "Institutionalize a Managed Process." In order to achieve this goal, it is expected that the following Generic Practices or alternatives will be implemented: GP 2.1 Establish an Organizational Policy, GP 2.2 Plan the Process, GP 2.3 Provide Resources, GP 2.4 Assign Responsibility, GP 2.5 Train People, GP 2.6 Manage Configurations, GP 2.7 Identify and Involve Relevant Stakeholders, GP 2.8 Monitor and Control the Process, GP 2.9 Objectively Evaluate Adherence, and GP 2.10 Review Status with Higher-Level Management.
- **Capability Level 3: Defined**. A defined process is a managed process that is tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the PAL. Generic practices or alternatives to satisfy Generic Goal 3, Institutionalize a Defined Process are GP 3.1 Establish a defined process, and GP 3.2 Collect Improvement Information.

[5] The Engineering PAs in the continuous representation include base and advanced practices. Advanced practices subsume the base practices in the staged representation.

- **Capability Level 4: Quantitatively Managed**. This is a defined process that is statistically controlled. Generic Goal 4, Institutionalize a Quantitatively Managed Process is implemented through GP 4.1 Establish Quantitative Objectives for the Process, and GP 4.2 Stabilize Subprocess Performance or their alternatives.
- **Capability Level 5: Optimizing**. This is a quantitatively managed process that is changed and adapted to meet relevant current and projected business objectives. Generic Goal 5- Institutionalize an Optimizing Process is implemented by performing GP 5.1 Ensure Continuous Process Improvement, and GP 5.2 Correct Root Causes of Problems.

In both the staged and the continuous representations of the CMMI, generic goals and generic practices are applied to the process areas in a similar manner to serve as reminders to do things right, however there are significant differences between the two representations. The following are some of the differences in approach to using generic goals and practices in the two representations. Also included are some other differences in the construct of the two representations.

In both representations of the CMMI, higher levels must build on the lower levels. More specifically, in the Staged Representation, an organization cannot satisfy GG 3 (Institutionalize a Defined Process) and achieve Maturity Level 3 (Defined Level) unless it has also satisfied GG 2 (Institutionalize a Managed Process) and satisfies the Specific Goals of all applicable Process Areas assigned to Maturity Levels 2 and 3. In the Continuous Representation of the model, to achieve Capability Level 3 in a particular process area, an organization must satisfy generic goals 1, 2 and 3 and satisfy all the specific goals of the process area.

An organization may work toward improving process performance by achieving GG2, Institutionalizing a Managed Process which entails planning the approach, monitoring against the plan and taking corrective actions as needed, then to achieve process capability level 3, the organization must develop a set of standards that may be tailored for project use using tailoring guidelines. To achieve capability Level 4, the organization must statistically control the defined process, and at capability level 5, the organization must adapt or change its quantitatively managed process based on business objectives.

Table 5: CMMI – Continuous: Additional Generic Goals and Practices

Generic Goal	Generic Practices
GG1: Achieve Specific Goals	GP1.1 Perform Base Practices
GG2 – see Table 2	See Table 2
GG3 – see Table 2	See Table 2
GG4: Institutionalize a Quantitatively Managed Process	GP4.1 Establish Quantitative Objectives for the Process GP4.2 Stabilize Sub-process Performance
GG5: Institutionalize an Optimizing Process	GG5.1 Ensure Continuous Process Improvement GG5.2 Correct Root Causes of Problems

As would be expected, there is strong architectural, approach, and application similarities between CMMI and SECM. In contrast, there are radical differences between CMM and the continuous representation. However, software developers may choose this representation to improve their capability in specific process groups (e.g., Engineering) or in specific PAs within groups, without having to focus on an entire Maturity Level at once. Within each representation, SEI has included guidance for choosing between representations.

Deciding on CMMI Adoption or Transition

Adopters would include organizations new to process improvement that may be considering adopting CMMI. Organizations already using the legacy models might be looking to transition to CMMI. Ultimately any decision boils down to a determination of whether CMMI is right for you[6]. Ironically, but perhaps not unexpectedly, the CMMI includes a PA called Decision Analysis & Resolution (DAR) whose purpose is to evaluate decision alternatives. The following paragraphs briefly illustrate DAR activities (sans infrastructure) you can use to determine the right improvement option for your organization.

- **Step 1:** Establish evaluation criteria. Criteria are intended to provide the basis for evaluating alternatives. Evaluation criteria might include customer requirements, business constraints, and market conditions. Criteria might address pros, cons, risks, benefits, and costs and should factor in ranking and prioritization since not all criteria are equal. It is not unusual for customer requirements, business constraints and market conditions to outweigh other selection criteria, with the possible exception of cost. Benefits may include both the non-tangible and the tangible.
- **Step 2:** Identify alternative solutions. Alternatives may come from stakeholders and establishment of evaluation criteria. In this case, however, we identify the following 3 options: 1) do nothing; 2) adopt or transition to a framework other than CMMI (e.g., IEEE, ISO, ITIL, Six Sigma, or an internal model) or 3) adopt or transition to CMMI.
- **Step 3:** Select evaluation method. Example evaluation methods may include qualitative or quantitative methods, simulations, scenario analysis, probabilistic modeling or decision theory. Method selection should consider the decision purpose and available information. Unfortunately, given the current state-of-the-practice, there is limited data for quantitative cost/benefit analysis. Most organizations, those having a Maturity Level below 4, typically do not have the metrics in place to quantify system-level changes on project, process, and product performance. Hence, a scenario-based evaluation method may work best.
- **Step 4:** Evaluate alternatives using the methods and criteria. Assumptions, uncertainties and rationale for each evaluation method should be included. In this case, a simple factor comparison is used. This activity may initially produce results similar to Table 6, below, but will likely become iterative.

Table 6: Example Analysis of Process Improvement Options

Result/Option	Option 1	Option 2	Option 3
Option	Do nothing	Do Something	Do CMMI
Pros	- no changes - no new costs	- pre-existing (proven) - match customer or market needs	- designed for modern development domains - evolved from legacy model - lays the foundation and metrics for higher maturity levels
Cons	- no improvement - status-quo	- miss-matches between internal and external need - lack of defined process upon which to implement	- new, large model - expert judgment needed - high bar for showing maturity
Risks	- increasing obsolescence	- perceived improvement overshadows reality	- learning curve not taken into consideration

[6] Shrum, S., "Choosing a CMMI Model Representation", www.stsc.hill.af.mil/crosstalk/2000/07/shrum.html, July, 2000.

	- decreasing productivity	- market or customer changes	- using CMMI as a requirements document

- **Step 5:** Select option. Selection and implementation of one option, and the resulting consequences, or the availability of new and better information, may later cause the decision to be revisited. It is therefore important to document selection rationale, risks and assumptions. One software developer may choose to do nothing, preferring to risk obsolescence against market and/or customer changes, avoid internal changes, save money, and offset decreased productivity with new talent and tools. Another organization may elect to use another framework such as Six Sigma because of the anticipated cost savings or an internally-developed model that they believe better reflects their use, product, customer and culture. A third organization may choose CMMI because their customer contractually required use of one of the legacy models that will not be supported in the future. The point is this: your decision is the right decision for you provided that it's selected in an appropriate manner, based on necessary and sufficient considerations, for the right reasons and leads your organization towards achieving its business goals.

Summary

Implementation of the CMMI can not be accomplished as some organizations did with the SW CMM, by using it as a requirements document. Implementation of the CMMI will take a solid understanding of the organizational business environment and objectives in order to realistically implement the broad scope of the model. This also requires a clearer understanding of the model itself. There are additional decisions based on the organization's product base which must also be taken into consideration. However, the CMMI allows for greater possibilities in achieving process efficiencies and effectiveness. The CMMI also encourages an organization to continue to mature by requiring a solid foundation to be established at the lower maturity levels. It should be anticipated that implementation based on the CMMI will help organizations realize practical process improvements.

Quality of High-Tech Software Get a Boost with SSQA Assessments

Giora Ben-Yaacov, Pramod Suratkar, Marsha Holliday and Karen Bartleson
Synopsys, Inc. Mountain View, California, USA
(information: giora@synopsys.com)

Our paper describes successful deployment of software quality assessment program at a leading high-tech market-driven Silicon Valley software company. The assessment program was initiated under "quality partnerships" with several of the company largest customers. These "quality partnerships" involve conducting detailed assessments of the software engineering maturity for the company key software products, and periodic mini-assessments that stimulate continuous quality improvement deployments.

The assessment model selected for our software development culture was the Standardized Software Quality Assessment model (SSQA) that was developed by Motorola and adopted to the semi-conductor industry by the Semi organization. The paper also describes why we selected the SSQA model instead of the more popular SEI-CMM or the ISO-9000 models. Fundamentally, SSQA is a methodology for disciplined quality improvements on a product-by-product basis that is the most suitable for the high-tech software development culture.

Giora Ben-Yaacov – Giora is a Quality Architect at Synopsys. He holds responsibility for quality leadership and he is working closely with the various software development communities at Synopsys striving to achieve performance excellence. Giora has B.Sc and M.Sc degrees in Electrical and Nuclear Engineering from the Technion in Haifa Israel, and a Ph.D degree in Engineering from the University of Cape Town in South Africa. He has authored more than 40 technical papers and made presentations at various conferences and industry meetings.

Pramod Suratkar -- Pramod is a Staff Quality Engineer at Synopsys. He has a M.Tech degree from Indian Institute of Technology, Bombay and M.S. and Ph.D degrees in engineering from University Of Wisconsin in Madison. Pramod has 30 years of extensive marketing research, business process reengineering and software quality improvement program management experience. Before Synopsys, he spent 12 years in quality management in telecom, database and energy industries.

Marsha Holliday -- Marsha is a Staff Quality Engineer at Synopsys. She has an extensive software quality background, including software quality improvement program management, product testing, product development and manufacturing process improvement. Before joining Synopsys, Ms. Holliday was the Software Quality Assurance Director at Intermedics, Inc. At that position, she spent 13 years directing software quality assurance for pacemaker design and manufacturing.

Karen Bartleson -- Karen is director, Quality and Interoperability at Synopsys. With more than 20 years experience in EDA, she is currently responsible for initiatives and programs that further Synopsys' product quality and EDA interoperability. Before joining Synopsys, Ms. Bartleson was CAD manager at United Technologies Microelectronics Center and manager of Logic Analysis at Texas Instruments. Ms. Bartleson is a board member of Accellera and an active member of the EDA Consortium interoperability and quality committees. She holds a BSEE from California Polytechnic University, San Luis Obispo, California.

Quality of High-Tech Software Get a Boost with SSQA Assessments

Giora Ben-Yaacov, Pramod Suratkar, Marsha Holliday and Karen Bartleson
Synopsys, Inc. Mountain View, California, USA
(information: giora@synopsys.com)

ABSTRACT

Our paper describes successful deployment of software quality assessment program at a leading high-tech market-driven Silicon Valley software company. The assessment program was initiated under "quality partnerships" with several of the company largest customers. These "quality partnerships" involve conducting detailed assessments of the software engineering maturity for the company key software products, and periodic mini-assessments that stimulate continuous quality improvement deployments.

The assessment model selected for our software development culture was the Standardized Software Quality Assessment model (SSQA) that was developed by Motorola and adopted to the semi-conductor industry by the Semi organization. In addition, we extended the standard SSQA methodology to include our own methodology for periodic mini-assessments that are very important in driving continuous quality improvements by the different product teams. The paper also describes why we selected the SSQA model instead of the more popular SEI-CMM or the ISO-9000 models. Fundamentally, SSQA is a methodology for disciplined quality improvements on a product-by-product basis that is the most suitable for the high-tech software development culture.

The paper demonstrated that the base-line software development maturity levels at different product groups vary significantly, and that significant continuous improvements in software development maturity were measured for the software product teams during the periodic mini-assessment reviews.

In our opinion, the assessment deployment has succeeded in increasing quality awareness across the organization and is continuously guiding management and employees to include quality as an important element in the decision-making process.

INTRODUCTION

In this paper we share our real life experience in deployment of a very successful software quality assessment system at Synopsys Inc., a leading market-driven high tech software company at the heart of the Silicon Valley.

Synopsys, Inc., headquartered in Mountain View, California, is a large software company with an annual revenue of over $1 billion. Synopsys creates leading electronic design automation (EDA) software tools for the global electronics market. These EDA software tools are used by electronic

engineers to design complex integrated circuits, electronic systems, and systems on a chip. The company has more than 4000 employees world-wide, including more than 1500 software development engineers. There are over 50 product teams. Many of the teams joined the company as part of acquisitions. Each of these acquired teams had different level of software engineering maturity and had their own quality processes and measurements.

The SSQA quality assessment program, the subject of this paper, is a very important component of the company Quality Management System (QMS). QMS is defined in ISO 9001 as "the organizational structure, responsibilities, procedures, processes and resources for implementing quality management necessary to achieve the quality objectives" (see reference #1 for more information on QMS). In our company, quality management has shifted emphasis from merely the reduction of things gone wrong to emphasis on the increase in things done right for the customer. This new emphasis on quality management has fostered an environment of productivity improvement in processes as well as product and service offerings. Equally important, it emphasizes communication, teamwork and employee satisfaction. Feedback from our customers was that they appreciate our effort and that the adoption of the QMS had improved the communication, the quality and the overall customer satisfaction. The Quality Management System (QMS) that was implemented at our company is illustrated in Figure 1.

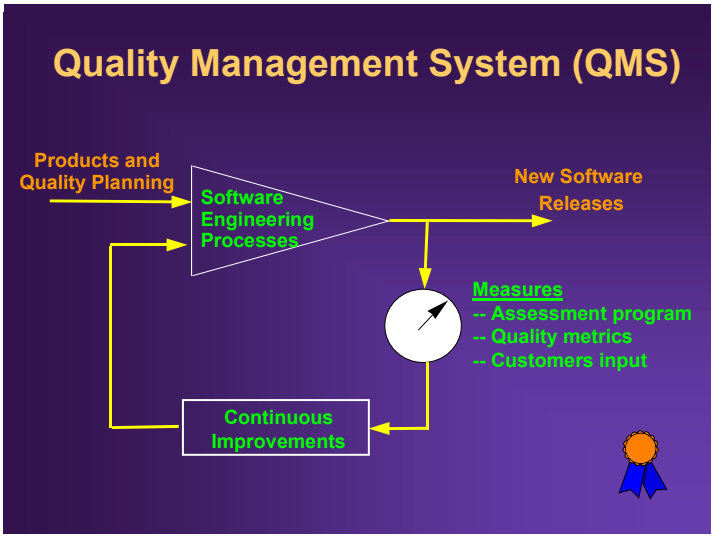


Figure 1: Quality Management System (QMS)

The three key elements of our Quality Management System are:

- Common processes,
- Measures, and
- Continuous improvement

For every software product introduced, including upgrades of previously released products, all the three elements of the Quality management System are essential for meeting and exceeding customer satisfaction. A key element of the Quality management system is the SSQA Quality Assessment methodology that enables a disciplined quality improvements on a product-by-product basis.

SELECTING QUALITY ASSESSMENT SCHEME

Our quality assessment journey started several years ago, as integral part of the "quality partnerships" that we initiated with our key customers. The first step in deployment of a quality assessment program was to select a suitable quality assessment scheme that addresses the quality and business needs of the company. We used the following selection criteria:

- Suitable to EDA industry culture
- Scalability to individual software product teams with varying maturity levels
- Providing a quantitative measure
- Leading to continuous and quantifiable improvements
- Ability to conduct assessments quickly and with minimum effort
- Assessment methodology that support quality partnership with our customers

We evaluated two general quality assessment schemes -- ISO9000 and the Malcolm Baldrige National Quality Award -- and three software industry specific assessment schemes – CMM, SPICE/ISO15501, and the semiconductor industry recommended assessment program – SSQA (Standardized Software Quality Assessment). We reviewed our selection criteria with our customers and the final recommendations were to select the SSQA as the most suitable assessment vehicle in our industry by which different product teams can evaluate and improve the maturity and effectiveness of their software development practices.

In addition, we extended the standard SSQA methodology to include our own methodology for periodic mini-assessments that are very important in driving continuous quality improvements by the different product teams. The two parts of the company assessment program are:

- Full Assessments: Base-line quality assessments of each product team (interview 10-15 engineers from R&D, AE, MKT). For the base-line assessments we use the standard SSQA methodology
- Mini-Assessments: Periodic quality reviews with each product team - Driving continuous quality improvements. For the mini-assessments we use our own methodology which is an extension to the standard SSQA methodology

THE SSQA ASSESSMENT MODEL

We use software quality assessments at the product level to determine the current maturity level of the product software development engineering practices; to foster quality improvement; to share "best practices;" and to ensure that the product software development and support processes are effective in achieving customer satisfaction.

Base-line SSQA Assessments:

Product base-line SSQA assessments are conducted once per product. The intent of base-line assessments is to determine the base line status and scoring of the software development processes, to point out strengths, and to identify opportunities for improvement. Using the SSQA methodology, the Assessment Review Team interviews a cross section of the product team engineers, and collects evidence on the processes that have been deployed in the management, development, rollout, and support of the software product.

During the review process the assessment team compares the current quality system to a "perfect" or ideal quality system, as described in the SSQA 12-element guidelines. These 12 elements are listed below. The assessment team scores the current software maturity level each of the 12 quality elements, evaluating each on four categories: management commitment, approach, deployment and results. Taken together, the 12 assessment elements provide a comprehensive evaluation of the product team commitment to software quality and customer satisfaction.

The 12 SSQA assessment elements

1. **Planning Process**
2. **Specifications and Reviews**
3. **Coding Practices**
4. **R&D Testing**
5. **Regression Suites**
6. **Alpha Testing**
7. **Beta Testing**
8. **Entry/Exit Criteria**
9. **User and Training Documents**
10. **Bug Management**
11. **Support Services**
12. **Customer Feedback**

Base-line quality assessments are conducted to review the software engineering maturity levels of different product teams. Typically, the assessment review team interview 10-15 members of each product team, representing Management, R&D, Marketing, Application Engineers, Tech Pub and Operations. The "checklist" of subjects that we used in the interviews is shown below:

Checklist used in interviews:

Product Life-Cycle

- Establish basis -- product overview, goals, team,...
- Requirements -- defined , reviewed, changed ….
- Plans & progress monitoring
- Functional & design specs, coding, unit tests
- Test process, Alpha, Beta
- Rollout planning
- Phase hand-off criteria, release criteria ….
- Release coordination ….

Customer support

- Defect management process
- Communicating with customers
- Metrics, response time, backlog ….

Support Systems

- Management support
- Staffing, skill, and training
- Quality goals
- Code Reviews
- Customer interactions / feedback
- Release Management
- QA / customer advocate
- Configuration Management
- Tech publications
- Computing Resources & Backup

During the interviews, the assessment team verify deployment against the company software development life cycle process which is illustrated in Figure 2. Following each interview, the assessment review team provide detailed report to each interviewee on his/her interview finding – including strengths and suggested areas for improvements.

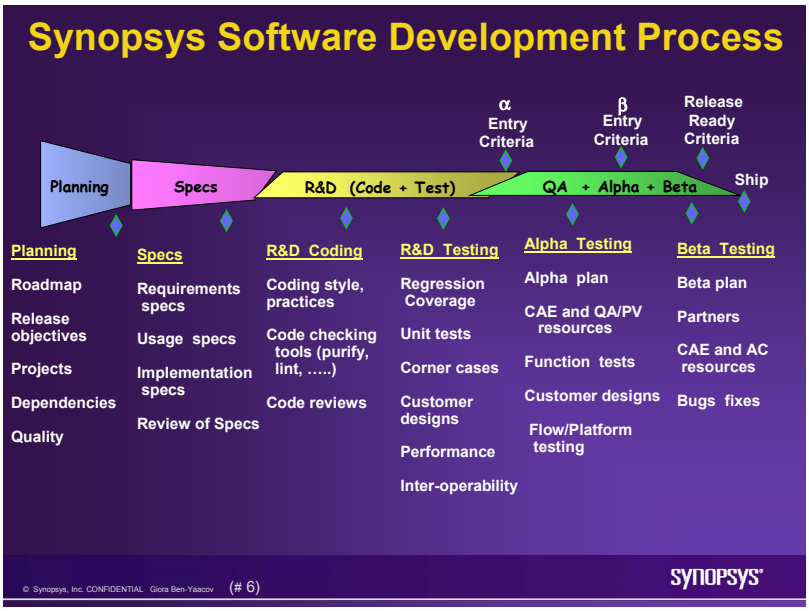


Figure 2: The company software life-cycle development process

The SSQA methodology also includes a detailed guide to be used by the Assessment Review Team to determine objectively the scoring levels for the 12 SSQA elements. A score ranging from 0 (poor) to 10 (outstanding) identifies the maturity level of each quality element, including process definition, process deployments, results obtained, priorities, and impact on the customers.

- **Level 0** - No systematic approach apparent
- **Level 1-2** - Beginning of a process in place; although decentralized and fragmented
- **Level 3-4** - Process direction is being defined; more centralized and less fragmented
- **Level 5-6** - Significant effort underway. Deployment in major areas. Some results being realized.
- **Level 7-8** - Effective quality system fully in place. Significant, positive results. All areas involved.
- **Level 9- 10** - Setting the standard for achieving Total Customer Satisfaction.

Periodic Mini-Assessments:

In our company, one of the main goals of the quality assessment program is to drive continuous quality improvements by the different product teams. For this reason, we extended the standard SSQA methodology for full-assessments of the base-line quality maturity levels of different product teams, to include our own methodology for periodic mini-assessments that are very important in increasing quality awareness and in driving continuous quality improvements by the different product teams.

Typically, every 6-12 months, we conduct for each product a 2-hours mini-assessment review with the product "core team" –- 6-12 key people of the product team that represent the R&D, the Marketing and the Application Engineering functions of the product team. The "template" questions that are covered in the mini-assessments reviews are shown below. Also, for the scoring of the mini-assessments we use our company "standard" scoring, which is a scoring level between –2 to +2 (+2 = very satisfied, +1 = satisfied, 0 = neutral, -1 = weak, -2 = very weak) that is used to measure the process maturity, results obtained, and the impact on customers.

Template for the 12 questions

1. Planning
1.1 Product, team, competition, revenue
1.2 Roadmap, business, partners
1.3 Release objectives, Projects priorities
1.4 Integration, inter-operability
1.5 Effectiveness of reviews

2. Specifications
2.1 Customer requirement specs
2.2 Usage specs
2.3 Functional, implementation specs
2.4 Effectiveness of reviews

3. Coding practices
3.1 Coding -engineers coding skill, training
3.2 Code stability, re-write, commenting
3.3 Documenting error messages
3.4 Code checking tools (glint, purify ..)
3.5 Code reviews

4. Testing by R&D
4.1 Unit test, branches & corners coverage
4.2 Logic is correct, data paths verified
4.2 System testing, large designs
4.3 QoR, Performance
4.4 Inter-operability, platform integration

5. Regression suite(s)
5.1_Nightly regression
5.2 Large regressions

6. Independent testing - Alpha
6.1 Alpha - Plan and resources (CAE, QA)
6.2 Functions, usability, large designs
6.3 Stability, performance, capacity
6.4 Inter-operability, integration
6.5 Effectiveness of Alpha tests

7. Independent testing - Beta
7.1 Beta – Plan, customers/partners
7.2 Customers feedback
7.3 Effectiveness of Beta tests

8. Phase entry/exit criteria list
8.1 Alpha-entry criteria and review
8.2 Beta-entry criteria and review
8.3 Release-ready criteria and review

9. Doc & training materials
9.1 Product and use documentation
9.2 Application notes, release notes
9.3 Training materials (AC, customers)

10. Bug management
10.1 Process for bug triage
10.2 Responsiveness
10.3 Effort on bugs
10.4 WDC, Incoming bugs – trend, value

11. Field support
11.1 ACs – Numbers, knowledge, focus
11.2 Sales – Priority, knowledge
11.3 Evaluations at customers site

12. Customer feedback
12.1 Estimating level of satisfaction

SAMPLE RESULTS

Base-line SSQA Assessments:

Base-line quality assessments are conducted to determine the base line software engineering maturity level for each product. The assessment includes a review of the status of the software development processes, looking at strengths, and identifying opportunities for improvement.

The assessment team scores the software maturity level each of the 12 quality elements, evaluating each on four categories: management commitment, approach, deployment and results. Taken together, the 12 assessment elements provide a comprehensive evaluation of the product team commitment to software quality and customer satisfaction. A numerical score ranging from 0 (poor) to 10 (outstanding) is given to each of the 12 quality elements. The total scoring for the product is an average of the scorings for the 12 elements. An example of the SSQA base-line scoring for a product is illustrated in Figure 3.

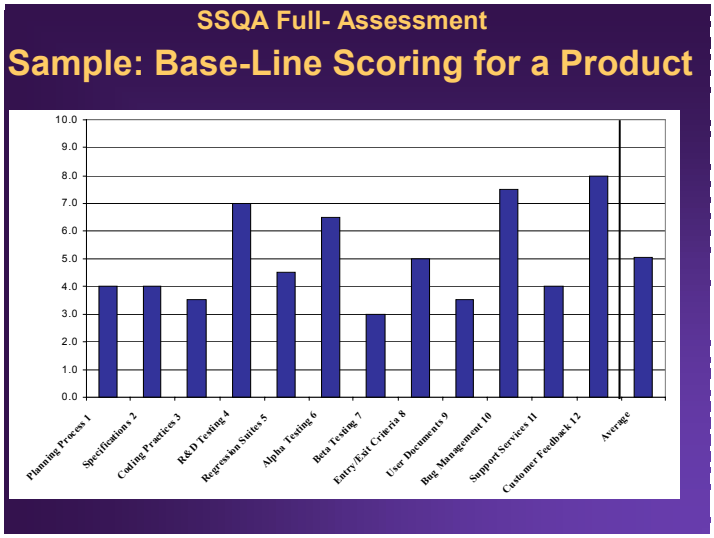


Figure 3: Example of base-line assessment scoring for a product

At the end of the base-line assessment, an assessment report is presented to the product team and their management. The report provides a macro view of the state of the quality system, recognize achievements, point out shortcomings and opportunities, and offers recommendations.

Periodic Mini-Assessments:

Typically, every 6-12 months, we conduct for each product a 2-hours mini-assessment review with the product "core team" –- 6-12 key people of the product team that represent the R&D, the Marketing and the Application Engineering functions of the product team. The main goal is to drive continuous quality improvements by the different product teams.

Figure 4 shows how the initial SSQA assessment scoring for a specific product have been improved over a period of two years with continuous improvements.
A simplified example of the report from the mini-assessment is shown in Figure 5. For each of the 12 elements, we summarize the status, strengths and improvement opportunities that were recommended by the product core team. In addition, the core team recommends the scoring.

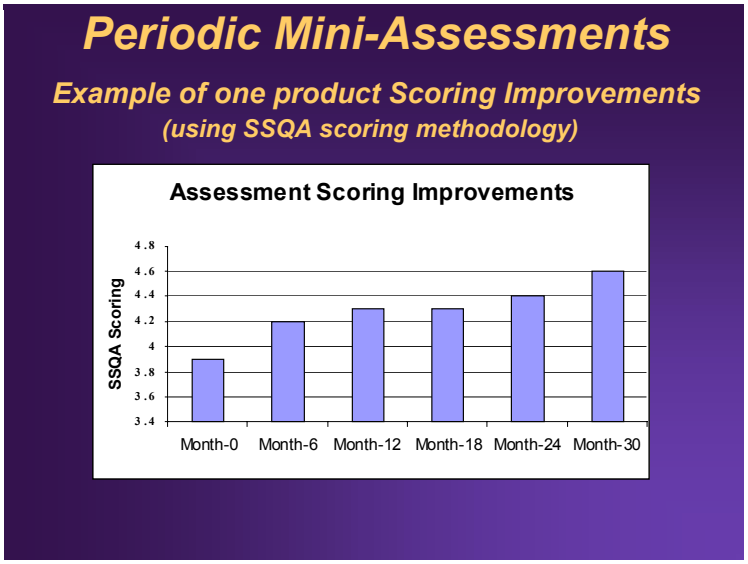


Figure 4: Periodic mini-assessments drive continuous improvements

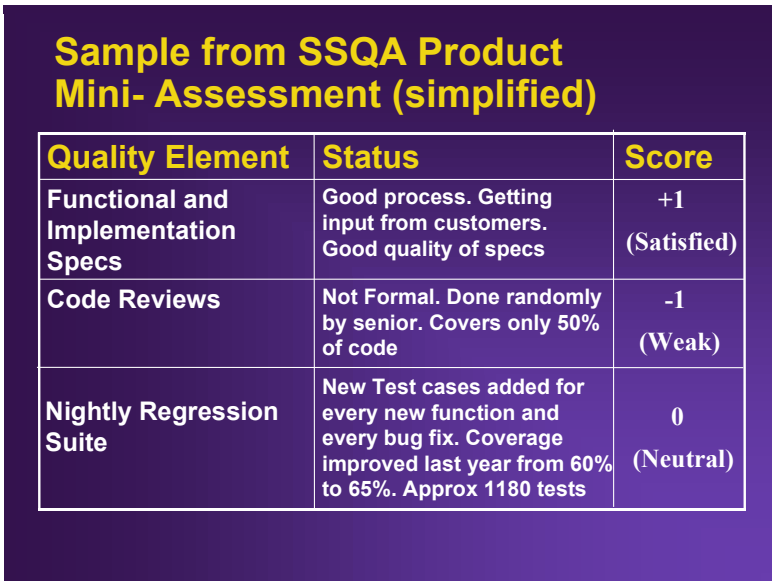


Figure 5: Simplified sample of mini-assessment report

The mini-assessment report also includes a list of the key improvement activities that the core team agreed to carry out. An example of such a list is shown below:

Continuous improvement plan for next 6 months:

(1) # of Bugs: Reduce WDC of backlog by 25%
(2) Vigilance on coding errors: # of Lint errors - target: 0 fatal warnings at code freeze. # of Purify errors - target: 0 at code freeze
(3) Regression Suite: Increase line coverage (PurCov) by 15%
(4) # of undocumented error messages: Reduce # of undocumented messages from 30% to 20%

CONCLUDING REMARKS

As described earlier, our assessment program was initiated under "quality partnerships" with several of the company largest customers. These "quality partnerships" involve conducting detailed assessments of the software engineering maturity for the company key software products (using the standard SSQA methodology and scoring scheme), and periodic mini-assessments (using home grown methodology and scoring practices) that stimulate continuous quality improvement deployments.

The paper demonstrated that the base-line software development maturity levels at different product groups vary significantly, and that continuous improvements in the maturity levels were measured for the software product teams during the periodic mini-assessment reviews.

In our opinion, the assessment deployment has succeeded in increasing quality awareness across the organization and is continuously guiding management and employees to include quality as an important element in the decision-making process.

Our customer quality partners have been pleased with our assessment program, and with our efforts to continuously improve both the software engineering processes and the awareness of the importance of customer satisfaction.

The positive results of the software process improvement effort demonstrated that "investing in quality improvements does indeed pay". This is illustrated in the results of the 2002 EE Times survey shown in Figure 6. The EE Times Electronic Design Automation (EDA) Survey was conducted during May 2002. The survey gathered responses from 506 chip designers who subscribe to EE Times or to Integrated System Design magazine. Among all the EDA companies, Synopsys came out on top in almost all categories.

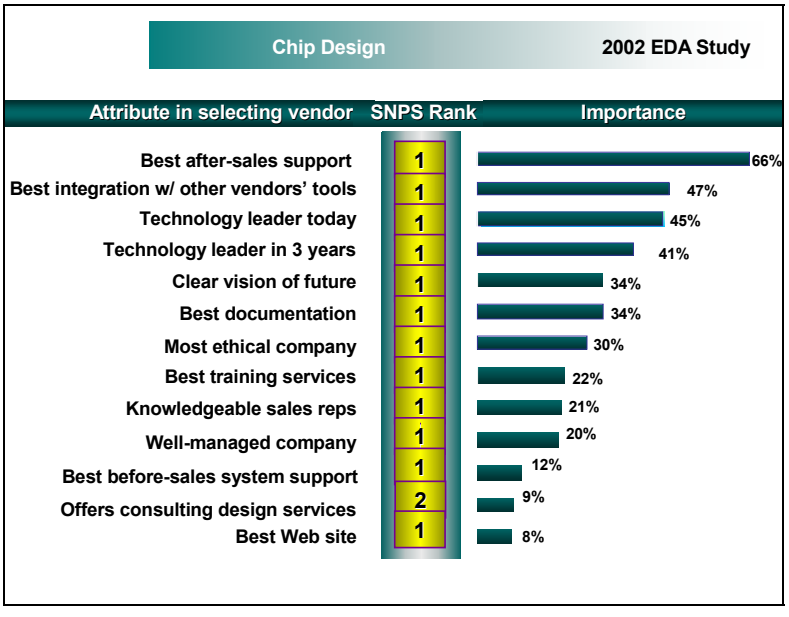


Figure 6: 2002 EE Times Survey

REFERENCES

(1) R.A. Radice, “ISO 9001 Interpreted for Software Organiations”, Paradoxicon Publishing, 1995

(2) G. Ben-Yaacov, P. Suratkar, M. Holliday & K. Bartleson, “Continuous Quality Improvements at a Silicon Valley High-Tech Software Company", Software Quality Professional (SQP), American Society for Quality, Vol 5, No 2, March 2003

(3) G. Ben-Yaacov, P. Suratkar, M. Holliday & K. Bartleson, “Advancing Quality of EDA Software", Invited paper, IEEE ISQED 2002 Symposium on Quality Electronic Design, San Jose, Ca., March 18-20, 2002

(4) M. Holliday, G. Ben-Yaacov, P. Suratkar and K. Bartleson, “Effective Software Metrics Program Drives Quality Improvements and Business Growth”, Pacific Northwest Software Quality Conference, Portland, Oregon, October 15-17, 2001

(5) G. Ben-Yaacov, R. Goldman and E. Stone, "Applying Moore's Technology Adoption Life Cycle Model to Quality of EDA Software", IEEE 2nd International Symposium on Quality Electronic Design (ISQED), San Jose, CA, 2001

(6) R. Grady, “Successful Software Process Improvement”, Prentice Hall press, 1997 edition.

(7) K Schulmeyer and J. McManus, “Total Quality Management for Software”, International Thompson Computer Press, 1996 edition.

BIOGRAPHY

Giora Ben-Yaacov – Giora is a Quality Architect at Synopsys. He holds responsibility for quality leadership and he is working closely with the various software development communities at Synopsys striving to achieve performance excellence. Giora's extensive experience in software development and quality improvement programs include a spectrum of results ranging from that of individual contributor, through project leader, program manager, instructor, consultant, and deployment of industry quality standards such as ISO 9000 and CMM. Giora has B.Sc and M.Sc degrees in Electrical and Nuclear Engineering from the Technion in Haifa Israel, and a Ph.D degree in Engineering from the University of Cape Town in South Africa. He has authored more than 40 technical papers and made presentations at various conferences and industry meetings.

Pramod Suratkar -- Pramod is a Staff Quality Engineer at Synopsys. He has a M.Tech degree from Indian Institute of Technology, Bombay and M.S. and Ph.D degrees in engineering from University Of Wisconsin in Madison. Pramod has 30 years of extensive marketing research, business process reengineering and software quality improvement program management experience. Before Synopsys, he spent 12 years in quality management in telecom, database and energy industries.

Marsha Holliday -- Marsha is a Staff Quality Engineer at Synopsys. She has an extensive software quality background, including software quality improvement program management, product

testing, product development and manufacturing process improvement. Before joining Synopsys, Ms. Holliday was the Software Quality Assurance Director at Intermedics, Inc. At that position, she spent 13 years directing software quality assurance for pacemaker design and manufacturing.

Karen Bartleson -- Karen is director, Quality and Interoperability at Synopsys. With more than 20 years experience in EDA, she is currently responsible for initiatives and programs that further Synopsys' product quality and EDA interoperability. Before joining Synopsys, Ms. Bartleson was CAD manager at United Technologies Microelectronics Center and manager of Logic Analysis at Texas Instruments. Ms. Bartleson is a board member of Accellera and an active member of the EDA Consortium interoperability and quality committees. She holds a BSEE from California Polytechnic University, San Luis Obispo, California.

About Synopsys, Inc.

Synopsys, Inc., headquartered in Mountain View, California, creates leading electronic design automation (EDA) software tools for the global electronics market. The company delivers advanced design technologies and solutions to developers of complex integrated circuits, electronic systems, and systems on a chip. The company has more than 4000 employees world-wide and an annual revenue of over $1 billion. The company has over 1/3 of the total staff (approx 1500 engineers) creating the software products. There are over 60 product teams. Many of the teams joined the company as part of acquisitions. Each of these acquired teams had different level of software engineering maturity and had their own quality processes and measurements.

The Process Before the Process

Tim Lister
Atlantic Systems Guild, Inc

We have been working hard for many years on making the development process more efficient and effective. Could it be that this work does not matter very much because the early process, the one that decides which projects, with what expectations, are funded and staffed, may be more than a bit broken? In this talk Tim Lister will look at the early project-decision process, and will discuss how we can get it to help us succeed at product delivery.

Tim Lister is a principal of the Atlantic Systems Guild, Inc., based in the New York office. He divides his time between consulting, teaching, and writing. Tim is co-author with his partner, Tom DeMarco, of the new book, *Waltzing With Bears: Managing Risk on Software Projects* (Dorset House, 2003). His article, written with Tom DeMarco, *Both Sides Always Lose: Litigation of Software Intensive Contracts*, was the feature article in the February 2000 issue of the journal, CrossTalk.

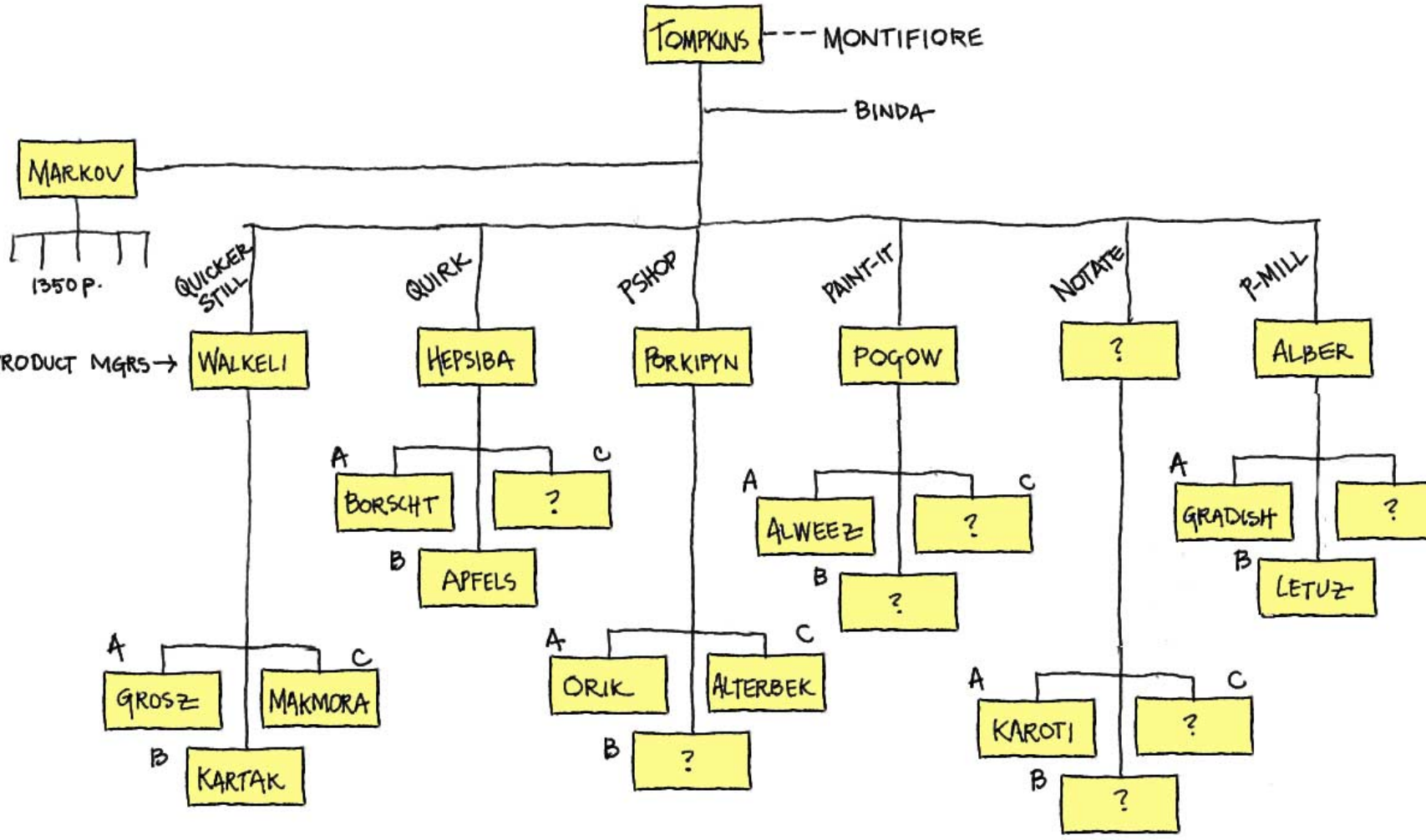
Tim Lister is working on tailoring software development processes using software risk management techniques. He is on the Yucca Mountain Nuclear Repository Technology Advisory Council, and is a member of the Cutter IT Trends Council. For the Cutter IT Trends Council he has written the majority opinion on Internet2, Continuous Partial Attention, and Risk Management.

How much risk is too much risk?

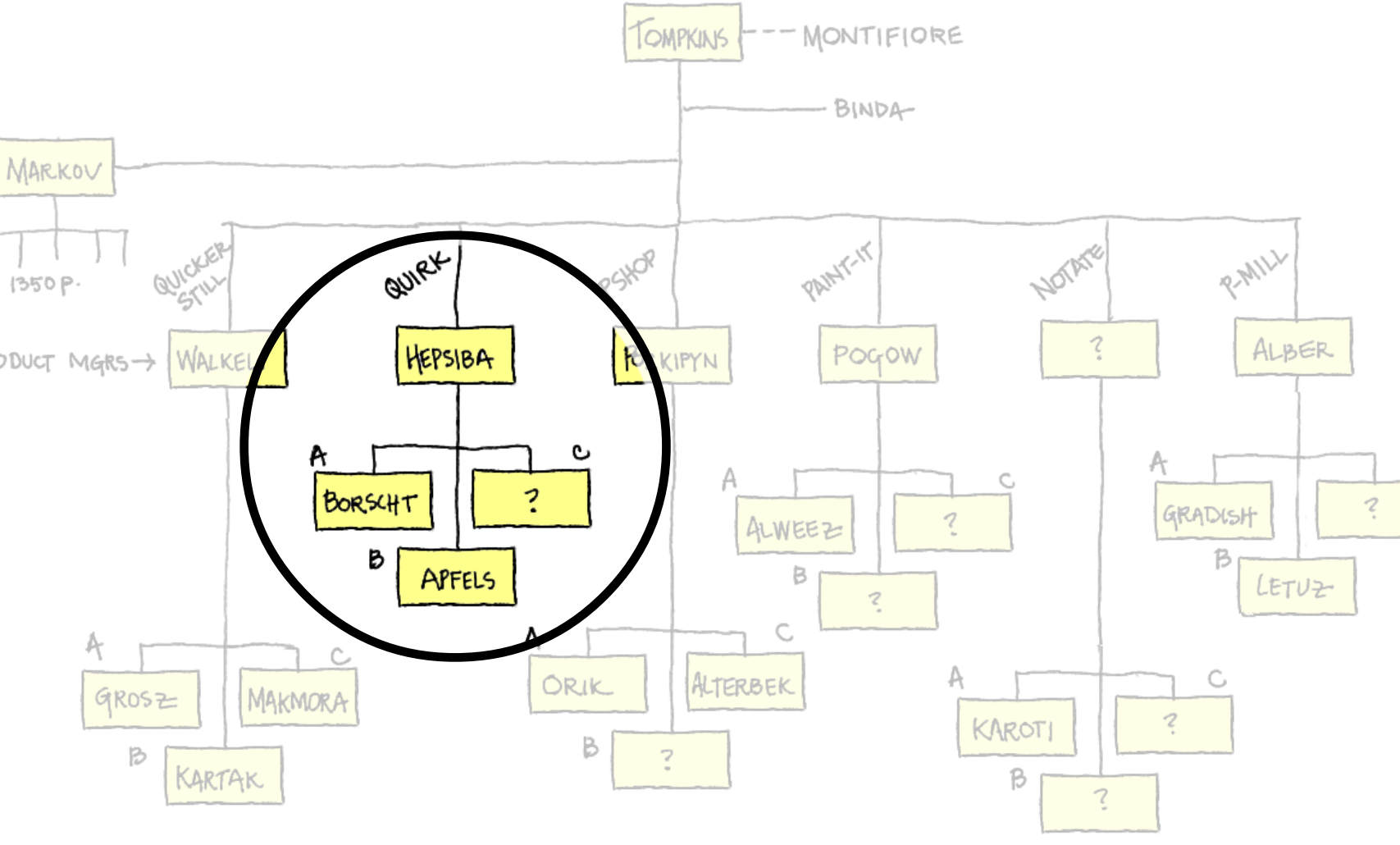
INSIGHTS:

- surprise about where the major risks lie
- surprise about the value of low risk projects
- surprise about risk quantification
- surprise about precision
- surprise about aggressively scheduled projects

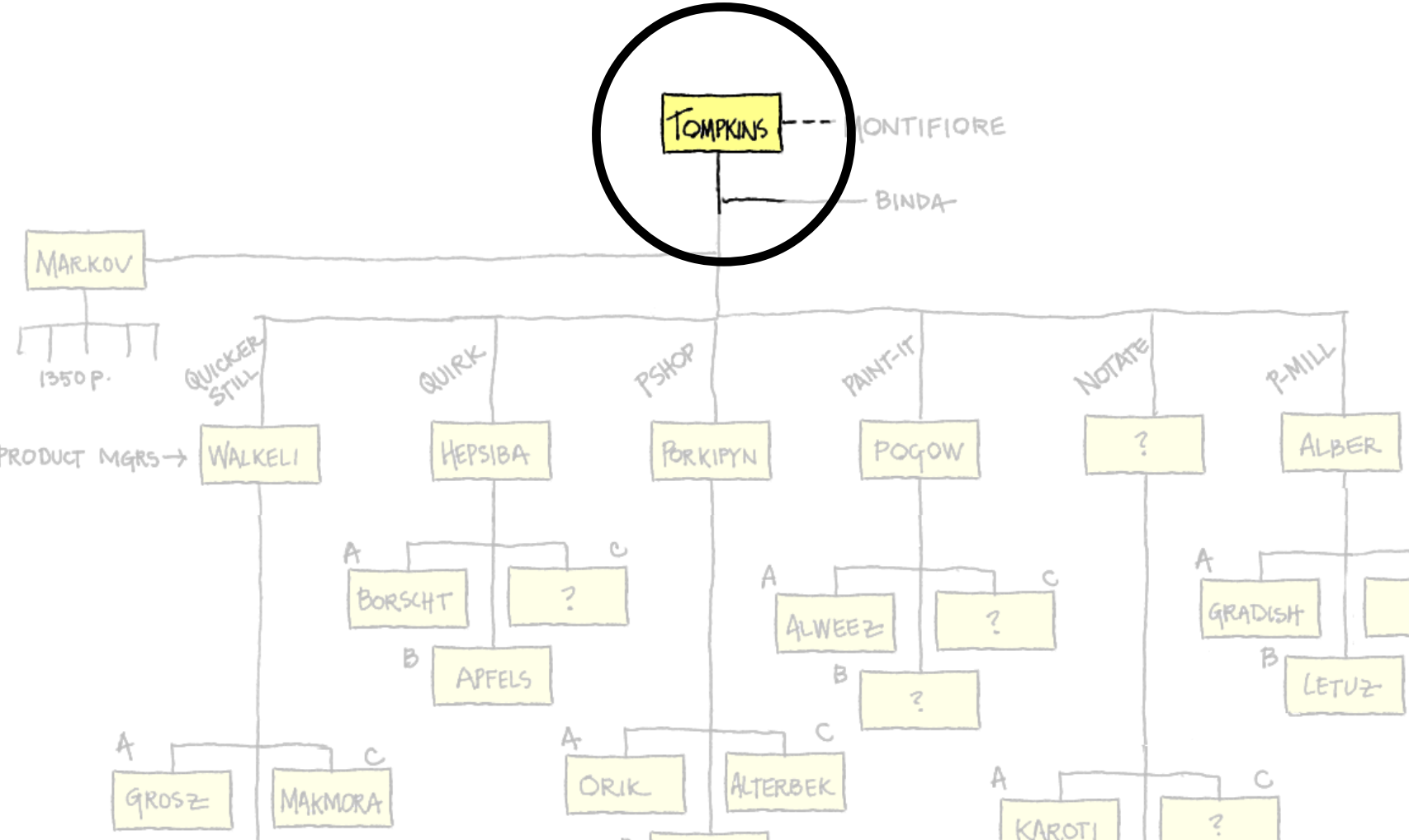
WHERE IS THE BIGGEST RISK?



WHERE IS THE BIGGEST RISK?



WHERE IS THE BIGGEST RISK?



THE FIRST SURPRISE:

The biggest risk an organization faces is lost opportunity, the failure to choose the right projects.

THE FIRST SURPRISE:

The biggest risk an organization faces is lost opportunity, the failure to choose the right projects.

So . . .

- Value is every bit as important as cost (the plusses matter as much as the minuses).

THE FIRST SURPRISE:

The biggest risk an organization faces is lost opportunity, the failure to choose the right projects.

So . . .

- Value is every bit as important as cost (the plusses matter as much as the minuses).
- Your process for deciding which projects to do is more important than your process for how to do them.

THE KEY PROJECT NEGOTIATION:

THE KEY PROJECT NEGOTIATION:

THE KEY PROJECT NEGOTIATION:

THE SECOND SURPRISE:

The real reason we need to do risk management is not so much to survive our risks, but to enable risk-taking.

THE SECOND SURPRISE:

The real reason we need to do risk management is not so much to survive our risks, but to enable risk-taking.

Conversely:

- A failure to manage risks ensures that no one will take any but the most minor risks.

THE SECOND SURPRISE:

The real reason we need to do risk management is not so much to survive our risks, but to enable risk-taking.

Conversely:

- A failure to manage risks ensures that no one will take any but the most minor risks.
- Without credible risk management, it is impossible to pursue meaningful value.

THE THIRD SURPRISE:

The only reason to quantify cost (schedule and budget) is to have something to compare to your quantification of benefit.

THE THIRD SURPRISE:

The only reason to quantify cost (schedule and budget) is to have something to compare to your quantification of benefit.

A failure to quantify benefit assures that:

- there is perfect one-way accountability; the project team is accountable but the client is completely unaccountable.

THE THIRD SURPRISE:

The only reason to quantify cost (schedule and budget) is to have something to compare to your quantification of benefit.

A failure to quantify benefit assures that:

- there is perfect one-way accountability; the project team is accountable but the client is completely unaccountable.
- there is no way to assure that the high value projects get done (prioritization is a charade).

THE FOURTH SURPRISE:

There is no sense making cost quantification more precise than value quantification.

THE FOURTH SURPRISE:

There is no sense making cost quantification more precise than value quantification.

So if all the user is willing to say about benefit is:

"We gotta have it!"

THE FOURTH SURPRISE:

There is no sense making cost quantification more precise than value quantification.

So if all the user is willing to say about benefit is:

"We gotta have it!"

then it's perfectly reasonable to quantify budget and schedule only to this extent:

"It's gonna cost a lot,"
and
"It will be done when it's done."


I'm sorry, Tom, but I simply cannot tolerate delivery any later than December 31, 2003.


I'm sorry, Tom, but I simply cannot tolerate delivery any later than December 31, 2003.
If I limit them to this year they can't possibly spend more than I'm willing to pay.

THE FIFTH SURPRISE:

An aggressive delivery date is often driven by a cost containment motive, not an urgent date motive.

THE FIFTH SURPRISE:

An aggressive delivery date is often driven by a cost containment motive, not an urgent date motive.

- The more aggressive the schedule, the more likely it is that the product is of low value.

THE FIFTH SURPRISE:

An aggressive delivery date is often driven by a cost containment motive, not an urgent date motive.

- The more aggressive the schedule, the more likely it is that the product is of low value.

An inescapable corollary:

> Starting a project late is a
> worse sin than finishing it late.

INSIGHTS:

- surprise about where the major risks lie
- surprise about the value of low risk projects
- surprise about risk quantification
- surprise about precision
- surprise about aggressively scheduled projects

I think we're doing risk management.
Um . . . we are, aren't we?

THE "ARE WE REALLY DOING RISK MANAGEMENT" TEST

(in six parts):

THE "ARE WE REALLY DOING RISK MANAGEMENT" TEST

(in six parts):

1. Is there a census of risks with at least 10-20 risks on it?

THE "ARE WE REALLY DOING RISK MANAGEMENT" TEST

(in six parts):

1. Is there a census of risks with at least 10-20 risks on it?
2. Is each risk quantified as to probability and cost and schedule impact?

THE "ARE WE REALLY DOING RISK MANAGEMENT" TEST

(in six parts):

1. Is there a census of risks with at least 10-20 risks on it?
2. Is each risk quantified as to probability and cost and schedule impact?
3. Is there at least one early transition indicator associated with each risk?

THE "ARE WE REALLY DOING RISK MANAGEMENT" TEST (CONTINUED)

4. Does the census include the core risks indicated by past industry experience?

THE "ARE WE REALLY DOING RISK MANAGEMENT" TEST (CONTINUED)

4. Does the census include the core risks indicated by past industry experience?

5. Are risk diagrams used widely to specify both the causal risks as well as the net result (schedule and cost) risks?

THE "ARE WE REALLY DOING RISK MANAGEMENT" TEST (CONTINUED)

4. Does the census include the core risks indicated by past industry experience?

5. Are risk diagrams used widely to specify both the causal risks as well as the net result (schedule and cost) risks?

6. Is the scheduled delivery date significantly different from the best-case scenario?

Title: The Manager's Role in Starting and Ending Projects: Charters and Retrospectives

Abstract: Managers play a central role in nourishing the conditions for project team success and play their role better by championing the use of charters and retrospectives. Adding two processes, chartering and retrospectives, to start and end the project team development life cycle boosts overall project success and its contribution to desired business outcomes. In this paper the author discusses ways managers can learn how to more effectively optimize the start and end of projects. The author and readers together examine the impact of expected and unexpected change on projects. We look at the value of charters and retrospectives for organizational learning and discuss the advantages for the project team and the organization as a whole.

Author: **Diana Larsen** is a senior organizational development and change management consultant who shares her time between Industrial Logic, Inc. (www.industriallogic.com), and FutureWorks Consulting LLC (www.futureworksconsulting.com). Diana works with clients in the software industry to create and maintain company culture and performance, building workplaces that develop and dignify people while achieving business results. Her focus on large and small changes in organization culture, structure and leadership has served clients through startups, transitions, expansions and alliances. A specialist in the "I" of Industrial eXtreme Programming (www.industrialxp.org), Diana serves as a coach, consultant and facilitator to senior and middle managers, development teams, and others in the project community. She conducts readiness assessments and facilitates processes (including project chartering and retrospectives) that support and sustain change initiatives, attain effective team performance, and retain organizational learning. Diana has special expertise in using appreciative inquiry approaches, Open Space Technology and other large group processes. She is a frequent speaker at XP/Agile conferences, authors articles on XP and organizational change, co-founded the Women's Center for Applied Leadership and serves as a board member of the Earth and Spirit Council.

Diana Larsen
diana@industriallogic.com

FutureWorks Consulting LLC
(http://www.futureworksconsulting.com)
4850 NE 9th Avenue
Portland, OR 97211
503-288-3550

Industrial Logic, Inc.
(http://www.industriallogic.com)
2583 Cedar
Berkeley, CA 97408
866-540-8336

To Begin

Most of us are about as eager to be changed as we were to be born, and go through our stages in a similar state of shock. – James Baldwin

Few projects proceed without facing the challenge of expected or unexpected changes in the work or environment. Organizations, work units, project teams and communities, as well as the individuals who make up these aggregations, display predictable responses to changes in their project environments. In this paper, we will explore ways of thinking about these predictable responses and examine two ways managers can help to steer their projects through the turbulence of change.

The first observation I offer is that change is not unusual or new. "In the Western world the philosophers of science seem to agree that change is such a pervasive and immediate element of our experience that it could become the subject of thought only after the early Greek philosophers had been able to conceptualize the antithetical concept of invariance or persistence. Until then there was nothing that change could be conceptually contrasted with…," writes Paul Watzlawick and his coauthors in their book, *Change*.[i] Change was the way of all life. Through seasons, cycles, migrations and all, change was/is constant. After the Greeks, the tendency became to view persistence or invariance or stability as the natural, normal or spontaneous state of affairs. Thus, as an adversary to the inertia of routine, change became a problem.

It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change. – Charles Darwin

In this instance, a problem is only a problem because we want it to be something else. A book is fine as a book, but if I want it to speak to me, if I want it to be a tape recorder, I have turned the book into to a problem. Change is only a problem if we do not acknowledge its pervasiveness in our work lives and roll with its dynamic processes. My professional advice is to learn about change, get comfortable with it and get good at it. Change is here to stay. What's more, change is the foundation of learning. This is not a message lots of people want to hear, having been thoroughly indoctrinated by the "desirability of persistence" camp. Today in our workplaces we discuss how organizations or project teams can do more with less, make the switch from a waterfall process to eXtreme Programming (XP) or other Agile software development methods; or some other urgent issue. Tomorrow the topic may be different. Whatever occurs, it is certain that change will be a part of it. With that foundation, let's move on to two ways of getting good at managing the changes that are inherent in projects.

Value of Charters

It is change, continual change, inevitable change that is the dominant factor in society today. No sensible decision can be made any longer without taking into account not only the world as it is, but the world as it will be. – Isaac Asimov

It's easy to pay lip service to using software development methods or supporting project teams, and then not walk the walk during implementation. For example, many a development team doesn't get the support it needs from its customer representative because they find it difficult to make time for the team. Project Chartering brings some discipline to the development process. A charter helps the managers and the team to identify a project's vision and mission, along with well-defined organizational and financial objectives, committed resources, critical interactions and more. Often requirements planning alone helps elucidate what features are needed for a given project, though it can fall short when it comes to explaining why. As a result, development teams can create finely crafted, fully tested software that fails to meet unarticulated organizational or financial objectives.

Good planning and clear understandings are essential to an effective software development organization. Developing a project charter is a collaborative exercise among project community members, i.e., between the software project sponsor, the project manager, and the development team members. It is also the opportunity for the project community to develop initial working agreements. As you develop your start-up charter, consider the following categories and questions. Use project chartering as a way of staying on track, or of identifying when the task has changed and may need to be re-evaluated.

The important thing is never to stop questioning. – Albert Einstein

The Eight Elements of a Charter

- VISION – How do the authorizing players/sponsors define the overall vision of the project? How does this project fit within and contribute to the company vision?

- MISSION – In a nutshell, why does the project and project community exist? What will the project accomplish? What value does the work add? What are the desired business deliverables, results, outcomes or effects? Can you describe in 25 words or less what the project is about?

- VALUES – No more than five core values may influence the behavior of a project community. What are the project community's core values? (Consider, for example, values such as communication, courage, enjoyment, feedback, learning, simplicity, trust, quality, etc.). How will these values affect the project work? Do individuals within the project community have their own values and are these in conflict with any of the project community's values? What are the sponsor's values and philosophy regarding the project mission? How will they affect the work?

- PROJECT COMMUNITY – Who is the team sponsor? Who are the authorizing players? Who owns the business obligations? Who makes "go/no go" decisions; i.e., who evaluates "work to date" and makes continuation decisions? Who are the stakeholders; i.e., who will affect or be affected by the decisions, recommendations and of the team, including software users? Who are the customers/users for the product, both internal and external? On what decisions do others want/need input? How will the team be led? By whom? Who are the project manager(s) and program manager(s)? Who are the

individuals with the right mix of skills, knowledge and experience to accomplish the work of the team? Who are the key parties and partners with regard to this task and team? What actions will the team take to insure that key parties and partners are informed of the team's goals, objectives, schedule and progress?

- MANAGEMENT TESTS – Given the project vision and mission, what are the meaningful measures of success? What are the milestones? How does the work get reviewed and how is feedback communicated? How will the project community capture its wisdom at project's end?

- BOUNDARIES, SCOPE & LIMITS – How does this project fit within the context of other projects and other work? What are the limits of authority regarding decision-making, budget impacts, and so on? Are there any expectations about which the project community should know? Are there specific expectations regarding the outcome? For what key events should the project have a planned response?

- COMMITTED RESOURCES – Where/how does the project community secure the resources – money, time, access, environment, training, tools/equipment, permissions – it needs to proceed? What of these are already in place? What is the timeline for securing additional resources?

- WORKING AGREEMENTS – What agreements does the project community need about expectations for professional behavior when its members meet? What are your highest experiences and your best guesses for the future about how you will need to work together to optimize true enjoyment of the work? How do you want to work together? What are the agreements about enabling behaviors? What additional roles does the project community need and who will fulfill them? Is there a metaphor that best fits this project and provides a way of thinking about the work and/or interactions?

A charter is not a static document, written, put on the shelf and never referred to again. A charter persists as a working draft. Thus, we refer to chartering, an active process of continuing to check assumptions throughout the project. However, in the beginning, a project community needs a basic set of agreements in order to move forward with its work. The project management team crafts the first draft of the charter, including the vision, mission, values, project community, evaluation plan, boundaries, resources, and working agreements.

In the chartering process, the manager brings the first draft to the project community for review and re-vision as needed. Oftentimes, the assistance of a professional facilitator can help to move this part of the process along more effectively, in the same way as such a presence can benefit a requirements workshop. Out of the initial chartering session comes a document that everyone can refer to for guidance on decision-making and working relationships, as well as a host of other issues.

Occasionally the working charters may need to be amended because of new circumstances or new information – that's why we call the process chartering.

Example 1, the managers and other project community members may initially define a management test – e.g. By the end of release, 80% of "Super Software 3.0" customers report that task-switching is 2 points easier than it was in 2.0, based on initial usablity rating between 1 and 10. – As the project goes forward, in conversations with the customer, the company may get feedback that customers do not notice or focus on task-switching and a different usability measure, e.g. time to load, would be more significant for them. Managers may want to change the test.

Example 2, projects can be plagued by scope creep. If the initial project charter has clearly defined boundaries and scope, any changes require a discussion to amend the charter, giving the whole project community a chance to understand the trade-offs (time, quality, cost) that changed or increased scope may incur.

Chartering is an active participle to express a dynamic process that adapts and responds to the changes in the project environment.

Value of Retrospectives

If past history was all there was to the game, the richest people would be librarians. – Warren Buffet

In software development, the opportunity for review and learning occur in two instances: first, at the end of each project milestone or iteration; and second, at the end of the project. Each milestone on any development project offers its own unique moment for capturing essential information and understanding, in addition to adding to the capacity for planning for what comes next. A retrospective process is more than a simple post mortem review, going deeper with its analysis and retrieving previously untapped knowledge about what works in a specific environment as well as finding ways to carry forward the new understanding as process improvement on future projects. It is a way in which any organization can become skilled at acquiring and using its collective wisdom in creating software.[ii] Establishing retrospectives as an ongoing ritual part of every development project provides current and future teams with a means to access best practices, collect and compare effort data, avoid faulty decisions, and assess the chances of success of further innovations.

As an example, here is a quote from (we'll call him) "James," a project manager in a large Silicon Valley company. His project community was huge, ultimately involving over 400 in the course of a 28-month project. These remarks were written 6 months after a retrospective that included over 70 people as he reflected on whether the retrospective we did made any difference:

> "After the X Project retrospective there were strong feelings that there needed to be follow-through on the plan we developed for changes in the way that releases were organized, and we have implemented new project teams as a result. We have also pushed accountability and responsibility for the content of the releases down to individual Project teams and managers, which was something that people asked for during the retrospective. We are now holding more cross-functional meetings and Transfer of Information meetings to improve people's understanding of what is being worked on and to find out if there are any concerns or issues

that need to be resolved for the release. We are also forming teams to define process improvements based on best practices across all projects, so I think that there has been a lot of good information transferred to the people working on the next series of releases. It's hoped that they will be able to build on the successes identified in X Project at the retrospective, and these releases will become the basis for even more successes and improvements. We both enjoyed working with you and hope that we will have the chance to work with you again."

Many of the improvements "James" mentions in his remarks were things that people had been talking about among themselves for months, but those discussions by themselves never developed the horsepower to actually make the desired changes happen. It took critical mass with a common understanding, such as is developed in a retrospective where all perspectives are represented, considered and included as a part of the planning process. In addition, if you are considering a change to a new methodology, like adopting some form of Agile software development practices (e.g. customer-developer interaction, test-driven development, iterative development, pair programming and others) the more likely that a retrospective of a non-Agile project will provide the critical incentive to make the leap.

Many organizations that have already adopted Agile practices conduct a retrospective at the end of each iteration as well as at the project's end. Why do both? They find the best, and possibly only, way to ensure that an organization realizes the full benefit of its previous investment in training, coaching, and so forth, is to allow the project community the time and opportunity to learn enough to: a) prevent any discomfort they felt this time from re-occurring on future projects and b) build on successes and best practices that may only be visible by taking the "whole project" view.

There is a huge payoff for teams who continue to learn from their experiences of doing software development well. The way humans learn is through a cycle of action and reflection followed by more action, and so on. (Or as the quality movement folks say: plan, do, check, act.) Retrospective participants include as many of the project community as possible, including managers, analysts, testers, engineers, programmers, coaches, support staff, customers and other stakeholders. A retrospective can help the group see what needs to be adjusted, improved, shifted, adapted, etc.; in order for the development investment to be truly worthwhile and to realize the potential payoff.

Building an internal cadre of facilitators offers one way to institutionalize retrospectives in your organization. Managers are rarely the best person to lead a retrospective for two reasons; 1) the employees' tendency to say what they think the manager wants to hear, and 2) the loss of the manager's perspective in the discussion if they are acting in the neutral facilitator role and giving their attention to group process.

Value of Organizational Learning

We cannot become what we need to be remaining what we are. – Max Depree

Project Charters and Retrospectives build two kinds of organizational learning: survival learning and generative learning. Survival learning increases an organization's ability to remember lessons learned and to adapt to future changing circumstances. Generative learning increases an organization's ability to build on past experience to strengthen the capacity to create, innovate and take advantage of new opportunities. Both are needed for continuing improvement in project processes and products.

Increase your Project Success Quotient (PSQ)

We now accept the fact that learning is a lifelong process of keeping abreast of change. And the most pressing task is teaching people to learn. – Peter Drucker

How do managers influence the inevitability of change and manage the risks on their projects? Managers who champion the use of Chartering and Retrospectives in their organizations communicate their organizational intentions and business objectives to development teams, and thereby achieve more successful projects in business as well as development results. These managers are also better prepared for the change management challenges inherent in doing things a new way and plan for these opportunities to effectively influence the change process.

For instance, after incorporating a short retrospective after each iteration, a project manager arranged for a retrospective review of the entire project up through the first release. Though the team had delivered the release significantly above the target date, during the retrospective they identified the risk that future releases would suffer because the team lacked bench strength in a critical area – database programming – and that the next release would require a steep ramp-up on a new piece of technology. Before the retrospective ended, they had developed strategies and working agreements to address both risk issues. As they began the next release with a charter, they defined the need for additional database expertise as a needed team resource and included the new working agreements on cross-training, so they would not be forgotten. The team is on the path to retain its "significantly above target" track record.

Tom DeMarco and Tim Lister say that project management is risk management.[iii] Effective managers look for ways to discover the possible risks, evaluate their potential impact, prepare a response and watch for their appearance. The role of a participatory chartering session building on a prior project's retrospective cannot be underestimated as a powerful tool for locating and managing risks. By incorporating chartering and retrospectives into the project plan, managers have increased their project success quotient.

Life is change, growth is optional, choose wisely. – Karen Kaiser Clark

---

[i] Paul Watzlawick, John Weakland and Richard Fisch. *Change: Principles of Problem Formation and Problem Resolution.* W. W. Norton and Company. New York. 1974.

[ii] Norman L.Kerth. *Project Retrospectives: A Handbook for Team Reviews*. Dorset House Publishing. New York. 2001.

[iii] De Marco, Tom and Tim Lister. *Waltzing with Bears: Managing Risks on Software Projects.* Dorset House Publishing. New York. 2003.

The Human Side of Risk

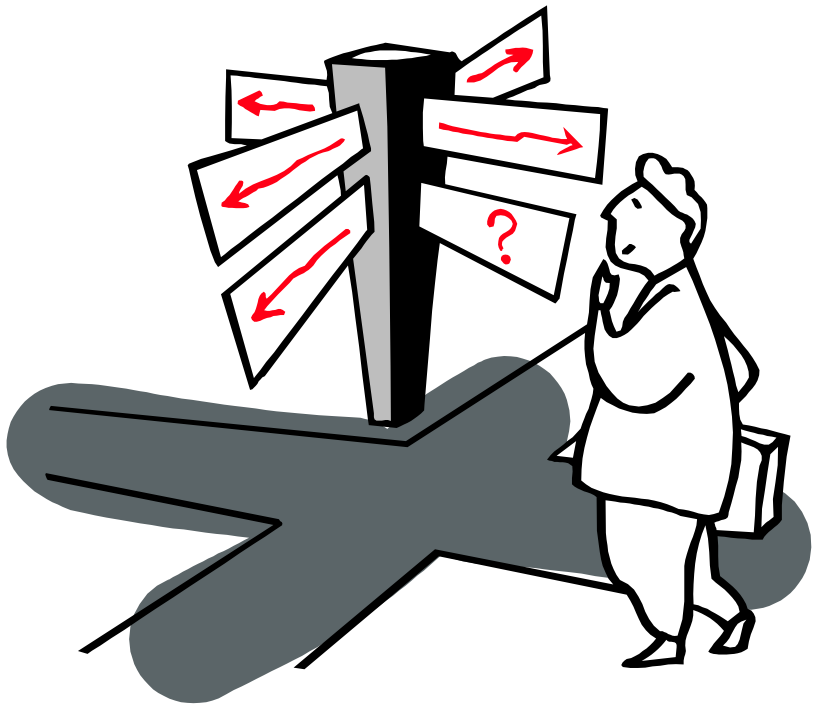
Eric Simmons,
Intel Corporation

A quick search using the terms Risk, Management, and Assessment on an Amazingly large book Website turns up 249 books on the topic, not to mention the chapters and sections devoted to various aspects of risk in texts on project management. Add hundreds of magazines, articles, conference proceedings, presentations, and courses at public and private institutions and you come up with a very large body of knowledge indeed. But, the vast majority of that material concerns the analytical side of risk assessment and management. It's the other side of risk - the human side - where we need to increase our knowledge. If we don't, even the best analytical methods will be ineffective.

Erik Simmons has 16 years experience in software and quality engineering. Erik currently works as Platform Quality Engineer in the Platform Quality Methods group, part of the Corporate Quality Network at Intel Corporation. He is responsible for Requirements Engineering practices at Intel, and lends support to several other corporate software and product quality initiatives. Erik is a member of the Advisory Board for IEEE 1074 (Standard for Software Product Lifecycles), the Quality Week USA Advisory Board, and is a founding member of the Rose City Software Process Improvement Network. He is a past member of the Pacific Northwest Software Quality Conference Board of Directors, and has made invited conference appearances in New Zealand, Australia, England, Belgium, and the US. Erik holds a Masters degree in mathematical modeling and a Bachelors degree in applied mathematics from Humboldt State University in California, and was appointed to the Clinical Faculty of Oregon Health Sciences University in 1991.

The Human Side of Risk

Erik Simmons, Intel Corporation

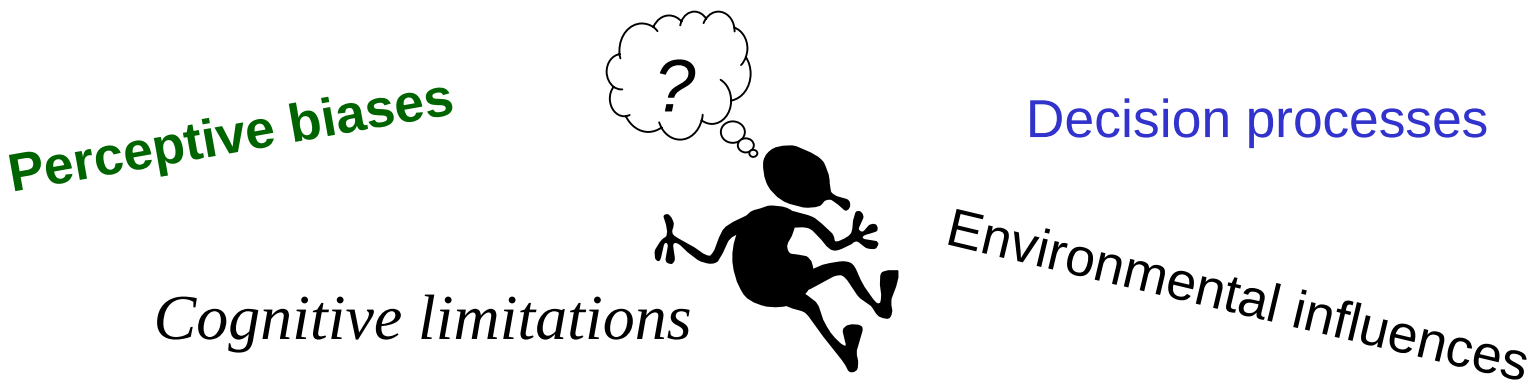


The Study of Risk

Risk is a highly studied subject:

- 249 books found on an Amazingly Large Bookstore's Website
- Hundreds of papers, conference proceedings, tutorials, classes, and book chapters

But, few of us understand the human side of risk

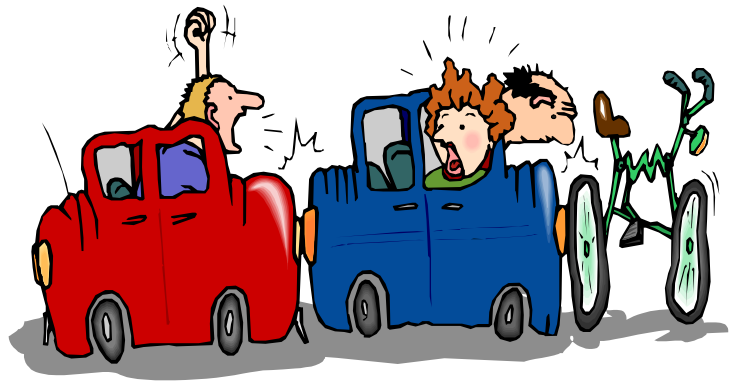


Are We Rational or Irrational?

What's the greater risk of death?

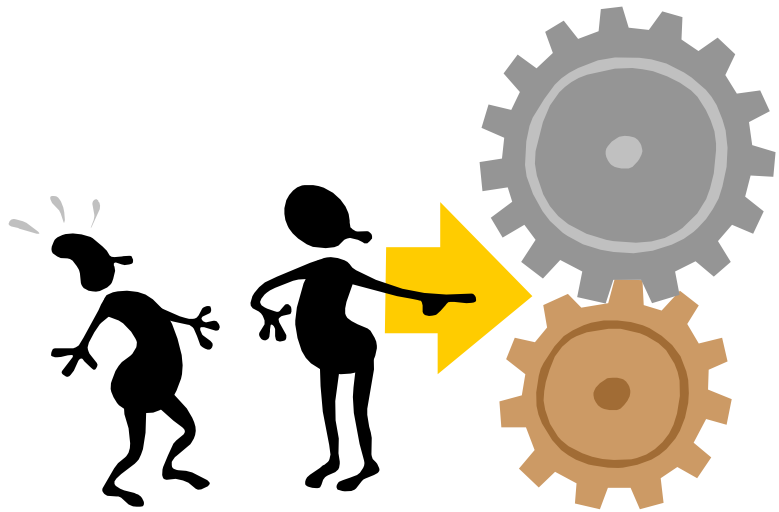
1:659,779

1:100



1:6,585

Risk Perception
Influences and Factors



The Two Sides of Risk

Analytical	Behavioral
• Logical • Data Driven • Scientific	• Emotional • Experiential • Irrational
Failure Modes & Effects Analysis (FMEA)	Subjective assessment or ignorance of risk
Risk Management	Gambling

Some Risk Perception Influencers

- *Dread* associated with the risk
- Perceived **controllability**
- *Imposed* versus **voluntary** risk
- *Immediate* versus **delayed** consequences
- *Low* versus **high** knowledge about the consequences
- **Chronic** versus *acute* consequences
- *Severe* versus **non-severe** consequences
- *Signaling* ability for other similar or more severe events
- **Familiarity** (experience) with the risk
- *Availability* of examples

Heightens Perception

Diminishes Perception

Software Examples

- *Dread:* Security flaws, product recall, loss of life
- *Imposed* versus **voluntary** risk: Schedule reductions by management versus the development team
- *Immediate* versus **delayed** consequences: Loss of fixed bid contract versus renegotiation later on
- *Low* versus **high** knowledge about the consequences: Commitment to support a unreleased OS version
- *Severe* versus **non-severe** consequences: Complete build failure versus a few hours lost to an SCM issue
- *Signaling* ability for other similar or more severe events: Performance issues in the first of many products on a new architecture

Cultural & Environmental Influences

Culture and environment influence the way risk is dealt with, and when. For example, the US culture is marked by*:

- Insistence for choice
- Pursuit of dreams
- Big is better
- Impatience
- Tolerance of mistakes
- Urge to improvise
- Fixation on what's "new"

Various cultures might view risk awareness as pessimism, realism, or exemplary behavior

How does your company's culture and environment affect your perception of risk?

*Adapted from *The Stuff Americans are Made of,* J. Hammond and J. Morrison

The Message Matters

Suppose you had a rare medical condition for which surgery was the only cure

If there were two procedures available, would you chose:

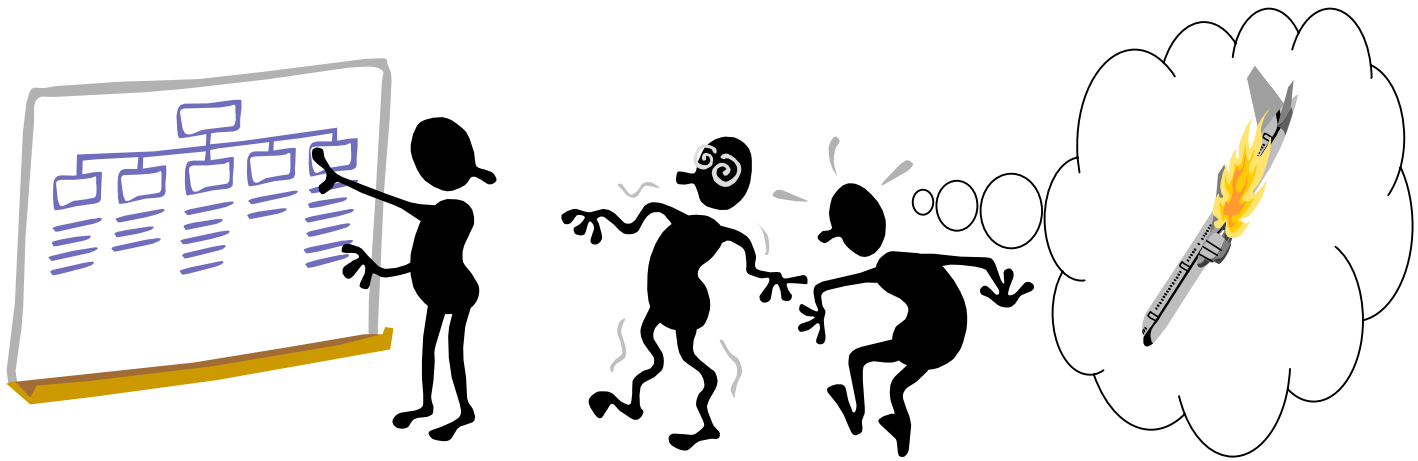
1. A procedure where nearly 1 in 5 patients die in surgery
2. A procedure with 85% probability of survival

Hmmm… Do the options really feel the same?

Unintended Messages

A fear of flying program had an expert explain to participants how unlikely it was for planes to crash

- The expert listed all the catastrophic failure modes and stated the probability of failure for each
- Result: The participants came away *more* afraid, because no one knew until then that there were so many ways a plane could crash!

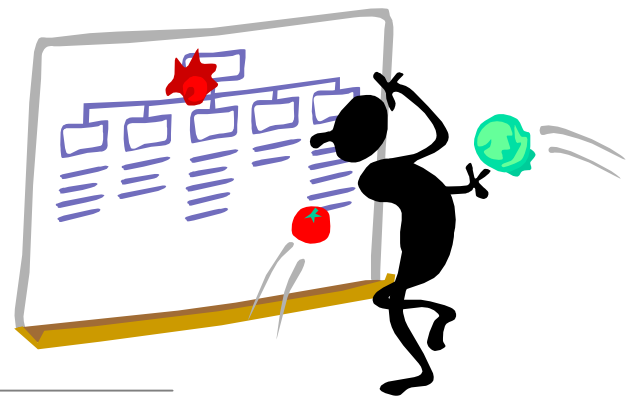


Too Much Information

Software risk spreadsheets can send the wrong message to other team members and management

Rigorous risk analysis and complete spreadsheets can cause:

- Added scrutiny from upper management (risk = problem)
- Individuals who state risks to be branded as “negative”, “not a team player”, etc.
- Avoidance of risks because “it’s too much to deal with”



What Could We Do Instead?

Monitor all the risks, but consider using a format other than the entire spreadsheet to report on them. For example:

- Educate people about risk management!
- Report only the Top 10 risks
- Report rolled-up metrics, such as total exposure (probability of occurrence X impact in time or dollars)
- Rotate the category of risk reported each meeting

Cognitive & Behavioral Limitations

Test Your Knowledge

How many people died as a result of motor vehicle crashes in the United States in 2000?

- Provide a point estimate and the smallest range that you are certain contains the true answer.
- Write your answer below

Point estimate:

Range:

Questions:

Did you come close?

Was the real answer included in you range estimate?

How did you arrive at your estimate?

What influenced your estimation and decision process?

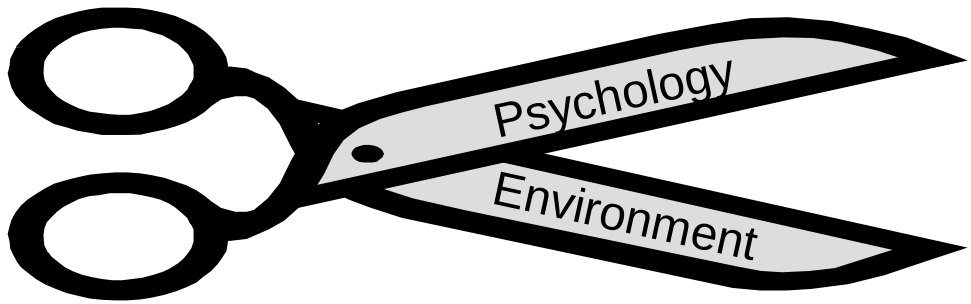
- Incomplete information
- Personal history
- Time pressure
- Cost of being wrong
- Familiarity with the risk
- And more…

Bounded Rationality

A completely rational decision is practically impossible:

- We cannot know everything we would like before making decisions
- Even with incomplete information, the data complexity is often overwhelming
- Emotion also plays a significant and often unpredictable role in decisions

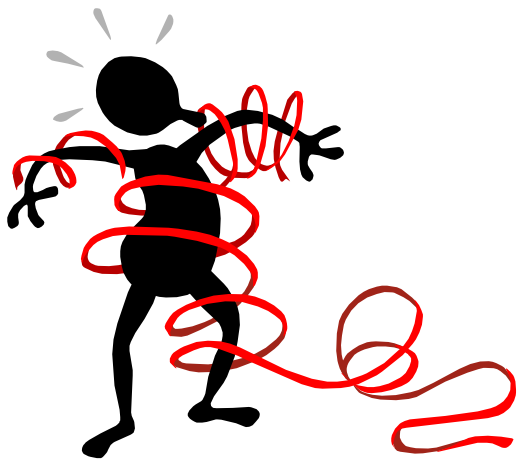
We are forced to create and use simplified heuristics and models to aid decision making



Decisions, Decisions…

Three basic building blocks for a decision heuristic:

- Search rule: How to locate alternatives
- Stopping rule: When to stop looking
- Decision rule: How to decide between alternatives

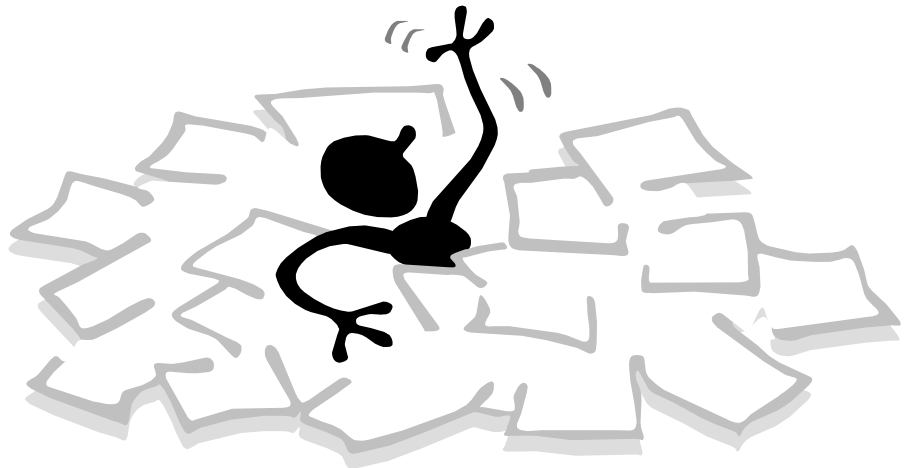


How much time should we spend deciding how much time to spend?

Cognitive “Limitations”?

Fast and Frugal Heuristics for decision making can outperform more complex decision structures like regression and neural nets in certain environments

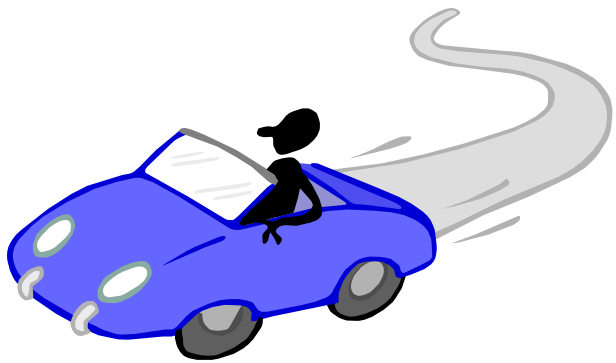
We must consider such heuristics in risk management

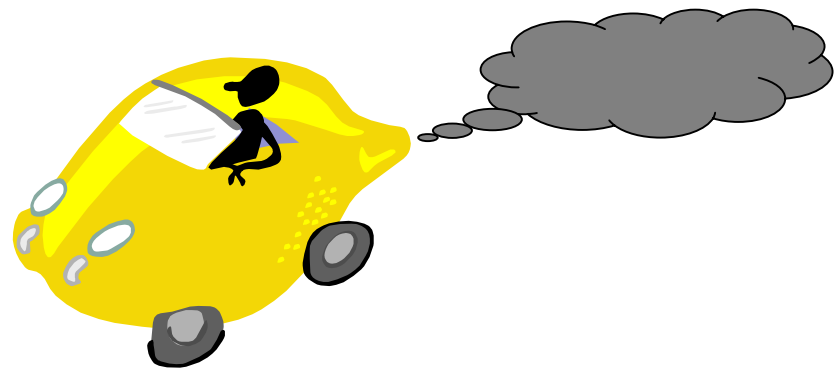


There are times when more information is not better

Which Would You Choose?

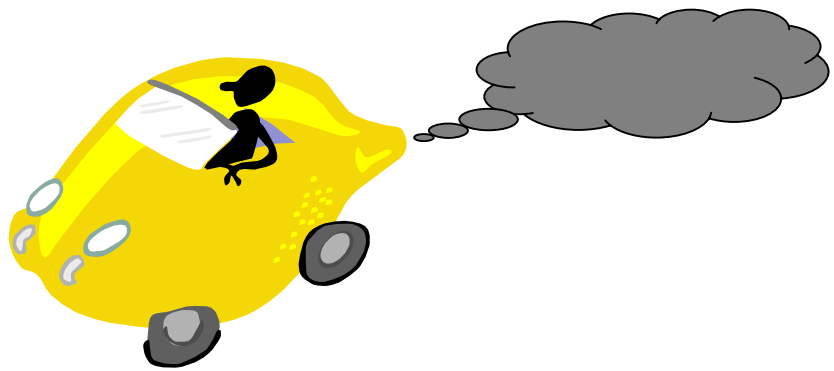
Car Name	Cost	Quality	Performance
Aardvark	Best	Best	Average
Belchfire Mark II	Worst	Average	Best
Conestoga Coupe	Average	Worst	Worst





How About Now?

Car Name	Cost	Quality	Performance
Aardvark	Best	Worst	Average
Belchfire Mark II	Worst	Average	Best
Conestoga Coupe	Average	Best	Worst



Some Fast & Frugal Heuristics

- Take the first solution that meets minimum requirements in the most important dimension
- Take the best solution in the most important dimension
- Take the equally weighted best solution in all dimensions
- Imitate the most successful in cases where no clear best solution exists
- Imitate the majority in volatile or complex environments
- Take the recognized solution over unrecognized solutions

Schedule

Security

Performance

Ease of Use

Cost

On The Other Hand…

Experts sometimes run things intuitively rather than analytically

In those cases, new twists, unseen conditions, or novel combinations of events can be disastrous

Chernobyl happened to a very experienced team

Many reasons behind it, including

- Overconfidence from repeated safety rule violations without negative consequences
- Time pressure
- Perceived controllability

It Happens in Software, Too

An experienced team wants to skip inspecting a requirements document in order to save time. Why?

- *Time pressure & short term thinking:* over-insuring small, likely losses while under-insuring for catastrophic loss
- *Perceived controllability:* Exposures to this risk without catastrophic consequences in the past
- *Over-weighting local experience:* Generalization of local experience to infer a lower risk than really exists

What can you do about it?

- Express the risk over time rather than single incident
- Provide data that describes risk rates and consequences based on large samples and studies
- Advocate a balance between short and long term thinking

Emotion, Risk, and Decision Making

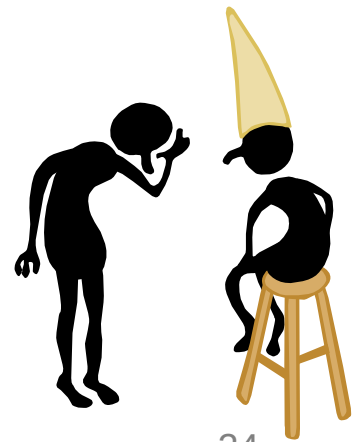
Past success creates pride and self worth

- In this setting, people tend to chose risk minimizing strategies

Failures create stigma, shame and a lack of self worth

- When this happens, people tend to chose reward maximizing strategies and ignore risk

Dread can focus inordinate attention on certain risks and influence decision making



Affect and Risk Perception

Affect is the feelings or emotions elicited by an external stimulus

Affect can strongly influence risk perception

What feelings and emotions do the following terms create?

- Ice cream
- Nuclear waste
- Tropical vacation
- Chemicals
- Public speaking

- Inspections
- Metrics program
- Agile methods
- Process improvement
- Object Orientation

What's the Greater Risk of Death?

1:659,779

1:2

1:373,787

Summary

- We sometimes hold very inaccurate and irrational views about risks
- Risk means different things to different people, and risk perception is influenced in many ways
- Risk definition is an exercise in power. If you control how risk is defined, you can greatly influence which strategy is best in response
- More information is not necessarily better - it may confuse or alarm rather than inform
- Presentation matters!

Summary

- We must better understand our tendencies and limitations as humans when it comes to assessing risks and estimating their severity and probability
- We must better understand how the way we express a risk influences someone's response to it
- We must better understand how culture, values, corporate and personal history, and other similar factors influence risk perception and management

Otherwise, the best analytical tools and methods for risk will have little effect

For More Information

- *The Perception of Risk,* Paul Slovic, Earthscan Publications Ltd, 2000
- *The Logic of Failure,* Dietrich Dörner, Metropolitan Books, 1996
- *Bounded Rationality: The Adaptive Toolbox,* Gerd Gigerenzer and Reinhard Selten, MIT, 2001
- *Elements of Reason: Cognition, Choice, and the Bounds of Rationality,* Arthur Lupia et al, Cambridge, 2000

ENTERPRISE DATA MODELING: The basic building blocks of an EAI initiative

Helen Umberger

PacifiCorp, Portland, OR, USA
Helen.umberger@pacificorp.com

Abstract: The newest buzz-word is Enterprise Application Interfaces (EAI) and everyone wants them now in their organizations. However, reality shows us just hooking together all the corporations' legacies applications together without planning will not buy you any of the promised improvements of the EAI architecture. It is necessary to create a data model of all the data in your enterprise, and map the data in each of your applications to a common information model, before you can realize the promise of EAI architecture. This paper will explain what an enterprise data model (also known as Common Information Model) is, the benefit of its creation, and how it makes the implementation of the EAI architecture possible. Then we will exam a case study of an actual implementation at PacifiCorp

Author Bio: Helen Umberger is currently a Project Lead for the EAI Adapter Group in the Futures Department at PacifiCorp in Portland Oregon. Her duties include designing new EAI Architectures to over lay the existing legacy systems.
They also include designing prototype and benchmarking projects for new technologies.

Past work has included heading the testing effort of large multi-platform, multi-language systems for Blue Cross Blue Shield.

Helen is also currently a Masters candidate in the Oregon Masters of Software Engineering program through the Oregon Graduate Institute.

Introduction:

Why focus on data architecture, when what you are trying to create is Enterprise Application Integration (EAI) architecture? Robinson[1] suggests that when companies design EAI architectures they place the emphasis on the technology problems involved with connecting the systems, but ignore the issue of information integration. This leads to several problems. Information that appears to mean the same thing but doesn't; conflicts with scaling, where different applications use different units of measurement, and naming conflicts where naming schemes differ significantly.

The introduction of EAI and Common Information Model into a business environment requires examining the existing architecture of the company systems, and creating an enterprise-wide Blue Print or corporate architecture design. Without this any attempt to introduce EAI to the corporation will result in point-to-point interfaces, with constant data integrity issues, and none of the expected ROI (return on investment) or goals expected of an EAI roll out:

- Extend the life of legacy systems.
- Able to buy new applications rather than building them.
- Ability of business mangers to see trends/data across the company.
- Able to quickly merge in an acquired company's data/systems.
- Speed, Business wants new functionality when they ask for it.
- Ease of integration (for example: Integrate the e-commerce system with the back-end customer databases and with the call center and customer-relationship management system – most likely all from different vendors, expecting/storing data in different formats.)
- Consistency enterprise wide in how auditing, security, data quality and disaster recovery are handled.

This paper will look at the first basic building block of creating a proper corporate architecture for supporting an EAI, the creation of the data architecture. It will also examine a case study at PacifiCorp (http://www.pacificorp.com/).

Part of the goal at PacifiCorp was to develop a data architecture that not only complies with the standard usages of data architecture, but one that also captures the life cycle of data and manages it effectively and securely. The data architecture needs to include the selected methodologies and guidelines on how we create and maintain our data.

Data architecture is defined as:

> "the structure of the components, their relationships, and the principles and guidelines governing their design and evolution overtime"
>
> Institute of Electrical and Electronics Engineers, IEEE 610.2

The focus of how data architecture is designed will look at the CIM (Common Information Model). There are many different CIM specific to the business, or environment type the company is focusing on. In this paper we are focusing on the CIM based on IEC (International Electrotechnical Commission) Technical Committe 57, Working Group 14 (WG14) standards for the Utility Industries (water, power, and gas). (IEC 61968) [2]

Benefits of CIM: [3]

- Data model driven solutions lead to interoperability.
- Provides common semantics for information exchange between heterogeneous systems.
- NERC mandated (Federal regulations).
- Provides for automatic generation of message payloads in XML.
- Ensures a common language for all messages defined.
- Avoids proprietary message formats from vendors.
- Eliminates work of creating DTD (Document Type Definitions). for each message.
- Uses industry standard modeling notation UML.
- Permits software tool use for defining and maintaining data models.
- Documents data model.

The CIM should be part of a company's strategy of creating corporate data architecture. The data architecture encompasses the activities of defining, structuring and documenting the data resource as well as maintaining data quality.

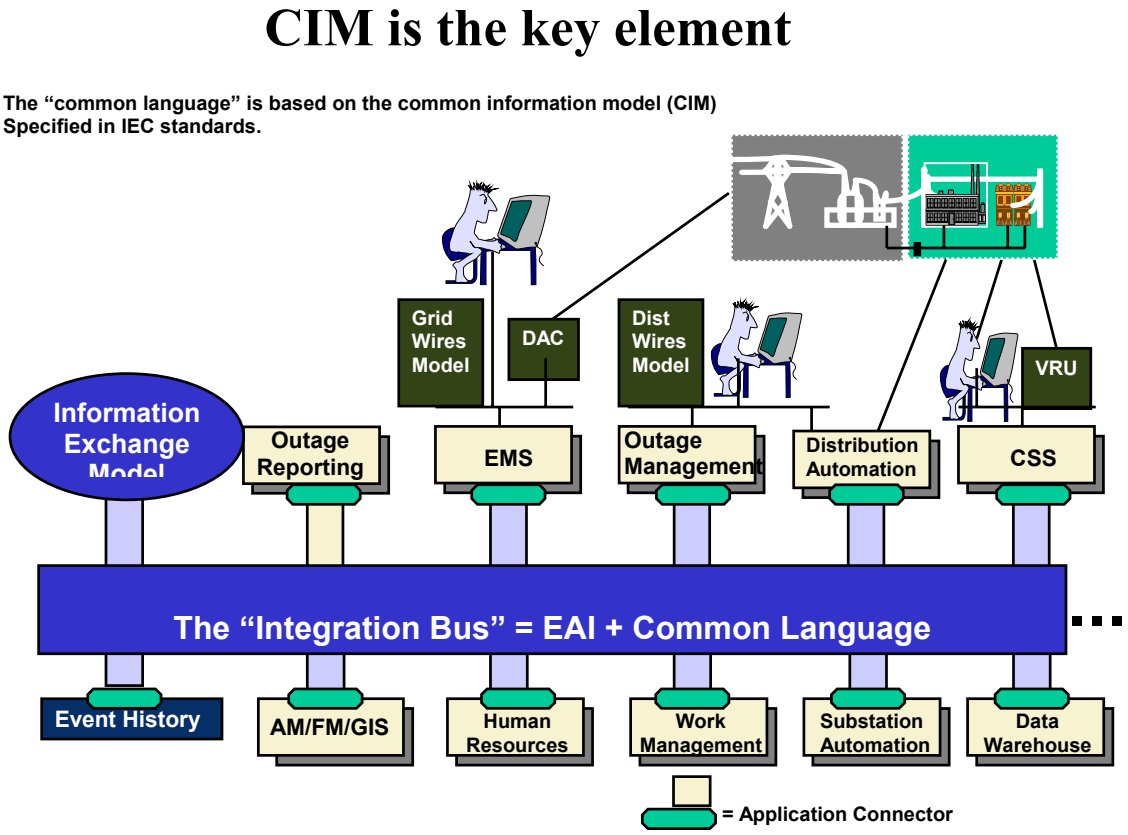


Figure 1 - CIM Conceptual Model of Utility Industry [4]

CIM also directly contributes to the success of the EAI roll out by supporting the over-all functionality by connecting multiple databases resulting in a consolidated view of information. It is necessary for the businesses to have a corporate-wide view of the company. EAI will be providing a layer of abstraction between the applications and the data. The advantage of EAI is that it is irrelevant if the company uses various types of databases, and data formats, since EAI will transform all data to a common format. The EAI structure will take over the responsibility of transporting the data (asynchronously) from one database/application to another in a consistent and repeatable manner, removing that responsibility from individual applications.

Another issue in EAI implementation if the data architecture is ignored, is that most EAI products have proprietary IDL (Interface Definition Language) which can lead to the need of a rebuild every time the an application introduces a new data element. This approach is not very flexible and leads to higher labor cost.

The solution is to use the CIM because it is a complete ontology that connects applications with meta-data and descriptions so that semantic conflicts can be avoided. [1]

What is CIM?

The common information model is a conceptual interface model of objects representing an industry, in our case, a generic public utility (gas, water or power). The goal of the industry CIM is to get buy-in on a common naming structure for elements, to remove redundancy, and to come to a standard for B2B interactions among the utilities as required by deregulation legislation.

The CIM also is a platform for the utilities to come to an agreement on a common messaging subject naming convention, for the use in B2B transactions to improve reuse and development times internally.

The data architectural level built around the CIM will then support the business intelligence layer of the EAI by presenting common definitions for all elements in a business regardless of the company.

Why use the Common Information Model?

Addressing the desire for EAI and B2B architectures in large utility companies whose current system architectures support disparate applications that are already built or new (legacy or purchased applications), each supported by dissimilar runtime environments. [5]

One of the emerging technologies for dealing with the above issue is to use information transformation. According to Robinson it is easier to do high-level information transformations then to write API wrappers for all of the applications you need to integrate. The quality of service advantage of doing high-level transformations of the data verses just writing API wrappers, is that the mapping would already be done in the information transformation, leading to consistency in the high-level data.

Another emerging technology is the use of middleware. This architecture requires that all the data be transformed to a common format before being dropped onto the information bus (see figure 1) ("Information Bus" is a common term to describe an aspect of a middle ware architecture. However in many documents it is also referred to as a hub. In the diagram below the information bus is the vertical on the right side of the diagram). The reason for this is that so any future applications that plug into the bus will know how to subscribe to the messages, and translate the data to it own internal needs. Without a

common information model there would be no reuse of messages and there would be no real advantage over the traditional point-to-point interfaces.

Eliminates the complexities and maintenance costs with hundreds of point-to-point connections…

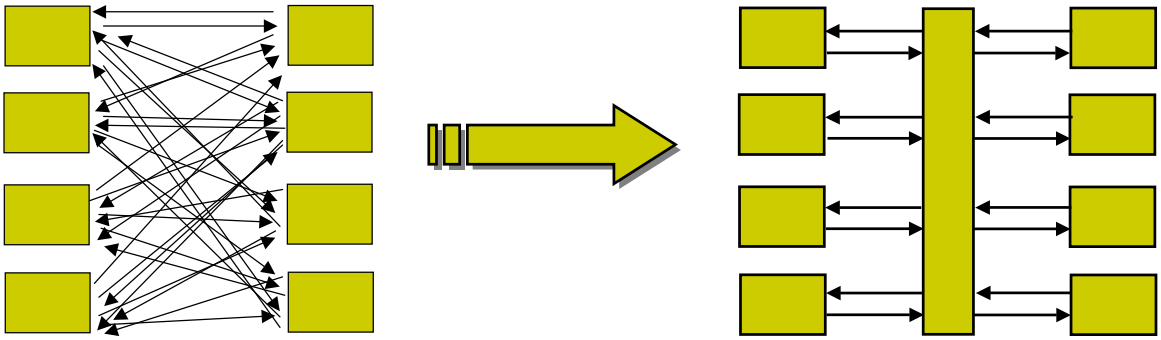


Figure 2 - Standard Interfaces vs. Middleware Interfaces [4]

Messaging can also be used as a standardized data exchange for data between utilities. To integrate n systems in the point to point method it requires n*(n-1) separate interfaces. Then the issue becomes the effort to develop each unique interface and maintain it. Then issues of data integrity creep into the system as they get out of sync, due to different back up schedules, data sources, etc. There is also a cascade effect when one application is replaced on all the other systems downstream and, if applications are more than one interface removed, the developers removing one system maybe unaware of the effect on other application's data.

The EAI method uses an "information bus" or publish-and-subscribe approach, where all the applications in a corporation talk directly to a single "information bus" (A middleware construct). This approach significantly reduces the amount of code needed to add or replace an interface. For this method to work, all the data must be mapped to a common and agreed upon format. It also creates a standardized data exchange for data between utilities. Companies with large numbers of stand-alone systems tend to have isolated, redundant databases, which leads to redundant data, out of sync data, and difficulty locating the true database of record for an element. Having all of your data mappable to the CIM allows for the future clean up of your corporate data, and increased control over the growth, security and integrity of the data.

How is data exchangeable in an EAI system?

Data is exchanged through messaging, instead of point to point interfaces. How do you create and name messages? The messages should be made up of data that is agreed upon by the sending and receiving application. However, one of the benefits of messaging is reuse – so in theory all of your corporation's applications need access to the data definitions. The only way to do that is to have a corporate level data dictionary to which all data in the corporation can be mapped.

Part of the CIM is a naming convention agreed upon by the IEC for messages. Use of the standard messaging naming convention has several benefits. Traffic of messages on the network can be managed and load balanced, by assigning different routing and network paths by higher level qualifiers in the message subject name. Messages tend to be in real time, instead of a batch mode like point to point interfaces, even if one of the applications being integrated is a legacy batch applications. Also in a B2B transaction with another utility, they would understand the higher level qualifiers from your message subject name, and be able to route it to the correct application(s) in their environment.

How is data mapped to the CIM?

CIM is machine-readable meta-data that can be accessed by applications and/or message definitions and thus avoid semantic conflicts. [5] CIM uses object-oriented modeling techniques and is expressed using UML (Unified Modeling Language) notation. CIM is divided into packages, and each package provides a model of the given item (different aspects of the utility business). Each package contains one or more class diagrams showing graphically all the classes in the package, their attributes, and their relationships. [5] Creating a common language for exchanges of data, between applications, or between utilities.

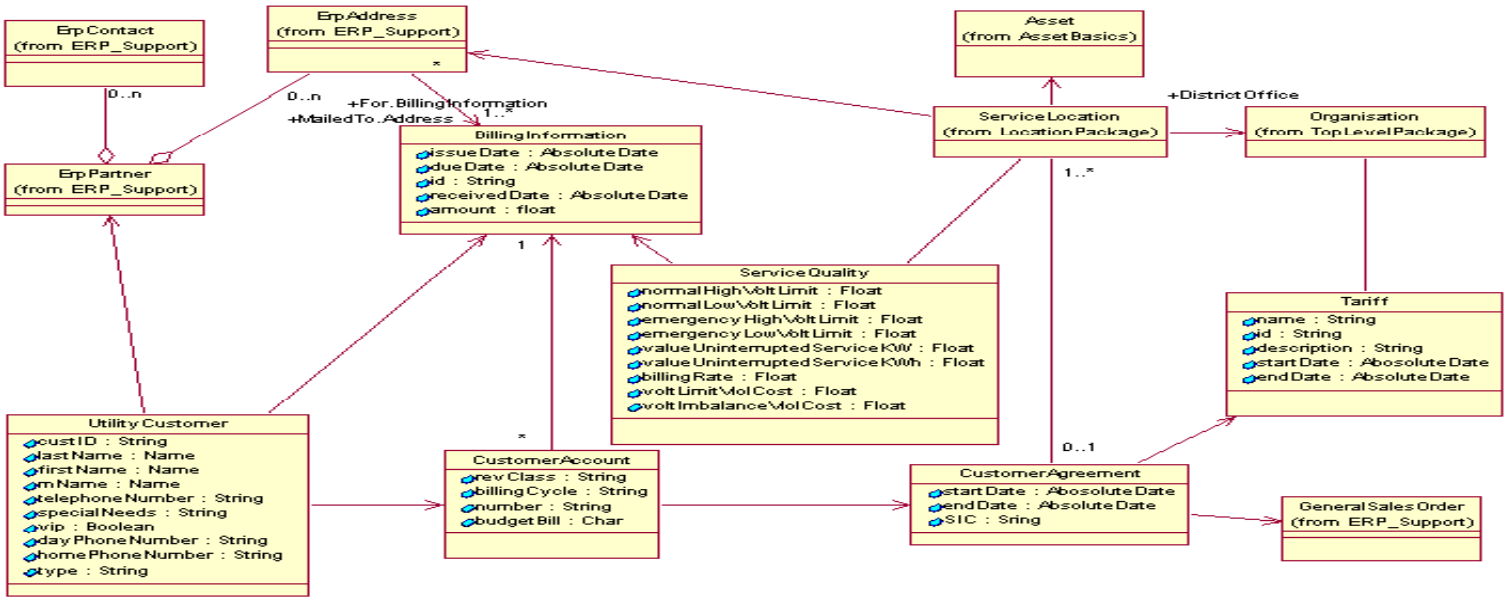


Figure 3 –. UML model for customer related data in the CIM [6]

The implementation of CIM in your environment does not a have dictated messaging format – in the case of PacifiCorp messages are exchanged into XML (Extensible Markup Language) and DTD (Document Type Definitions).

So the first step is creating corporate wide data – not application level data.

First the data in each application has to be understood, using the conventional data models as a starting point, looking for data that spans the corporation.

An enterprise-wide model is then created that spans multiple application and database domains. Next, a metadata level for the data is created which should tell you what each element is (what does it mean, what is its database of record, where is it used, how does it get there, and when is it modified).

The above information will let you know the relationships among data elements and identify where there is have redundant data stores and/or interfaces as well as point out issues in data quality. This is important when identifying the financial, legal, security, and regulatory value of each data element.

Once implemented CIM will then help individual projects focus on their scope, knowing that their objects can be leveraged by other projects. It then provides a neutral vehicle for achieving agreements among different parts of the organization that may not have worked together in the past. The content and relationships of various types of data can be discovered. And it will lower the life-cycle costs of those applications using the CIM.

It also solves the problem when 3 or more systems need to be integrated have different data formats. The total number of transformations is ***n*(n-1)*** for ***n*** systems with disparate data formats, which previously resulted in to many interfaces to be maintained.

The data now has an owner, the corporate data modelers, when previously ownership of an interface and its data transformations may not have had a clear IT or business owner, or had been orphaned to different owners.

Once the EAI framework is in place, the above CIM data is identified and is imported as a DTD into the Integration tool (this varies by vendor). Usually developers import the data schema of the source application and then the data elements are then dragged and dropped or mapped directly to the corresponding element in the CIM. The definition of that element specifies the translation parameters. The exported XML can then be sent as a message through the information bus with a WG14 CIM compliant message name that any other application can now subscribe to.

CASE STUDY: PacifiCorp

PacifiCorp uses the IEC supplied CIM using Rational Rose to render XML giving them an almost complete IEC 61968 compliant model. The IEC standard also includes naming conventions for messages.

PacifiCorp's Goal:

Part of the goal at PacifiCorp was to develop a data architecture that not only complies with the standard usages of data architecture, but one that also captures the life cycle of data and manages it effectively and securely. The data architecture needs to include the selected methodologies and guidelines on how we create and maintain our data.

The Problems:

Achievement of this goal for logical consistency requires cooperation among historically separate and often competing business units. Some aspects of logical consistency require process, as well as technical standardization, to address business unit objections. The increasing use of packaged applications constrains the ability to use, for example, common formats and names. While new applications may be required to conform to logical consistency standards, the cost of legacy systems compliance can be prohibitive, dictating long-term migratory effort. Physically distributed, logically consistent databases have a strong tendency to become inconsistent over time without constant vigilance and oversight.

PacifiCorp was unable to manage redundant data. Without data management, application databases continue to proliferate and the organizational and functional boundaries between them continued to be a barrier to achieving corporate or agency goals. The descriptive data in the application databases that support functionally bound systems tend to be redundant. For example, within the enterprise, customer name and address data are in many systems. Other data is stored in multiple shapes, forms and locations.

The solution:

Descriptive data should reside in the corporate data model, where they would occur only once, thus being manageable and still accessible to the application systems through the application program interface.

The goal is to move the descriptive data from the individual application systems up to the corporate data model where they are stored independent of their use and where they may be accessed by many business systems. The application-specific data will continue to be stored in the application data layer where they are access by a single business system.

A CIM should contain functionally independent data. This data does not rely on a business relationship to exist. It is dynamic and subject to change.

The application data layer contains functionally dependent data. This data exist to support a business function or activity. It exists as a record of a business transaction. This data is stable and rarely ever changes, unless derived data is also being stored.

The objective is to produce sets of data that can be trusted by the enterprise and used across many business units, for improved overall data integrity.

The first actual project using the new EAI infrastructure and the CIM was the Retail Access project. The Retail Access project was required by new state legislation deregulating utilities conducting business in Oregon. (SB1149) [7]. This resulted in PacifiCorp having to drastically change its business model and integrate a large number of standalone applications in a tight time-frame.

This project was chosen as the first project to use the EAI/CIM solution because it would comply with new corporate goals:

- Move away from creating new in-house solutions. New systems would be purchased if available.
- Pick the minimal solution.
- Require tighter integration of existing applications, and newly purchased applications.
- No new interface will be point to point. (Because applications data needs span multiple platforms)
- EAI technology would be used for all new interfaces.
- Interfaces would be mapped to a CIM to make reuse of messages easier.

Additionally, SB1149 required PacifiCorp to share data with other utilities and power re-sellers. Under the new law PacifiCorp would have to use a standard format based on the industry CIM.

Project histories taught us that at PacifiCorp the cost of developing or modifying a traditional point-to-point interface was estimated at $15,000. The in-house estimate on the time required for a traditional point-to-point interface was 21 weeks per interface.

For this particular project the need to purchase one new API was identified. The project had 15 interfaces, 8 of which were reused three times each within the project, the equivalent to 31 new point-to-point interface programs. The estimated cost of 31 new interfaces in the point-to-point system was $465,000. Cost for each new adapter was $20,000. The cost of the 15 interfaces using the new EAI/CIM modeling was $297,000. This resulted in a **savings of $148,000 (32%)**. Future reuses of existing messages are already being planned for future phases of this project, so savings will continue to increase.

The estimation of EAI savings was developed using the following formula:

> Cost avoidance = (acquisition + configuration + programming avoidance – reuse cost) * reuse count

This resulted in an estimate for the entire interface development of 651 person weeks. The average development time for the EAI interface messaging was five man weeks to design, configure, and test each interface. The final time to completion was 75 person weeks. Timesaving was critical to this project, which had a State mandated installation date. **The timesaving was estimated to be 576 person weeks.**

Note: with out the timesavings, our Retail Access project would not have made the state mandated installation date.

The ability to get instant reuse of messages without an interface program. If the message was required by more than one platform, the applications just had to register with the CIM version of the required message to get the data. This ability to automatically register with an existing message is a built-in function of MessageBroker – the Tibco mapping utility. (Tibco was the vendor of the mapping tools used to implement the CIM in PacifiCorp)

The ability to re-map OS390/MVS flat files to XML formats with drag and drop technology through the Tibco mapping tool MessageBroker. (No programming was required for the conversion.)

Two of our newly purchased applications had the requirement of inputting data through XML formatted data. The legacy publishing system was a mainframe application; the traditional point to point application program written in Cobol on the mainframe would not have been able to translate the data to XML. The traditional method would have required a custom program in the receiving application's environment (Unix) to translate the data to XML. By being able to automatically reformat flat files from the OS390/MVS system into XML messages and feed them to the new applications, we avoided having to develop the new applications in-house, or purchasing a translator for the files, or creating complicated interface programs that reformat flat files into XML messages. This was a mapping function of our middleware products (the suite from

TIBCO), which imported the XSD (XML Schema Definition) from the CIM , and a file layout of any inputted data and transformed the incoming data for inclusion in the CIM.

Additionally, by using the CIM to map all data crossing multiple platforms, they were able to get obvious reuse of code and configurations thereby increasing the ability of the Adapter team to meet tight deadlines.

Consequential benefits include avoiding point-to-point interface programs and unmonitored traffic on the network (e.g.; ftp), as well as avoiding complicated database extract programs. The Tibco adapters can access the data directly, map and send it out without any programming needs.

Which applications were integrated with this Retail Access project? The legacy billing system on the OS390/MVS system is now integrated with our new meter reading system (MVSTAR) and our new Electronic Partners – resellers system (MVPBS) on Solaris 2.7 platforms.

The MVPBS system also now interfaces with wholesale partners by sending out e-mails when scheduling changes occur or when a new electronic partner signs up with our system. The e-mail is just a subscription to an existing message under triggered conditions on the MVPBS system.

This new EAI system gave us the ability to solve the difficulty of integrating disparate business systems within and among business units. We can now:

1. Acquire and fully implement best-in-class packaged applications from multiple vendors. This fulfills the corporate directive to 'buy before build'; reducing the number of in-house developed custom applications.
2. More quickly append and absorb another organization's critical data into our infrastructure (mergers or acquisitions).
3. Adapt to market transforming events such as deregulation, energy crises etc.
4. Create 'virtual enterprises' encompassing different companies across a single supply chain.
5. Re-engineer business process and streamline operations quickly.

After this successful project, future plans include replacing dedicated and mostly undocumented point to point interfaces, and integrating all new systems.

Enterprise Application Integration technology has enabled standards-based interface methodology among the multitude of backbone and ancillary systems in PacifiCorp.

Conclusion

A complete enterprise-wide integration strategy requires implementation of a standard corporate-wide data model that transcends individual application systems.

This is achieved by:

- Creating stability with standardized data abstractions.
- Allowing the corporate meta data to be created/defined and controlled by a centralized data modeling group.

Giving the following benefits:

- Improved ability to integrate business processes across commercial-off-the-shelf applications.
- A simplified means for internal and external organizations to share information.
- Less dependence on individual vendors
- Improved usefulness of existing applications.
- Lower overall life-cycle costs of applications.
- Support for event-driven business processes.
- Data congruence.
- Ability to identify data that is a corporate asset (either to protect it or place value on it).

Epilogue:

To date PacifiCorp has integrated more than 25 of its systems, and investments payback is meeting expectations. Improved volume through put, lower maintenance, quicker turn around expectations have been met.

In conjunction with other components of the PacifiCorp strategic IT infrastructure, such as data warehousing, Web technologies, process modeling, data modeling and application code reuse, projects are being implemented in much smaller chunks. This improves delivery quality, budget and milestone delivery. This approach also allows business value to be delivered sooner and project risks to be managed more effectively.

REFERENCES

[1] Robinson – Paper "Model Driven Integration (MDI) for Electric Utilities" by Greg Robinson http://www.ece.msstate.edu/~schulz/research/d_sim/dist02_panel.html

[2] The CIM - 61968 – www.wg14.com . The actual Utility CIM

[3]Saxton – CIM Market Extensions (CME) by Terry Saxton – paper can be viewed at https://www.nerc.net/esc/files/escosc-1102.pdf - if you are unable to open the file with this link, go to google and type in CIM Market Extensions Terry Saxton.

[4]Dietz – Paper "Outage Call Handling with the CIM" by Janet Dietz at DistribuTech2003 http://dt2003.events.pennnet.com/conference_program.cfm

[5] Robinson – Paper "Key Standards for Utility Enterprise Application Integration(EAI)" by Greg Robinson – KPMG Consulting http://www.ece.msstate.edu/~schulz/research/d_sim/dist02_panel.html

[6]Dietz – Paper " DataModeling and MetaData" by Janet Dietz – PacifiCorp White Paper

[7] Senate Bill 1149 70th OREGON LEGISLATIVE ASSEMBLY--1999 Regular Session http://www.leg.state.or.us/99reg/measures/sb1100.dir/sb1149.en.html

ENTERPRISE DATA MODELING

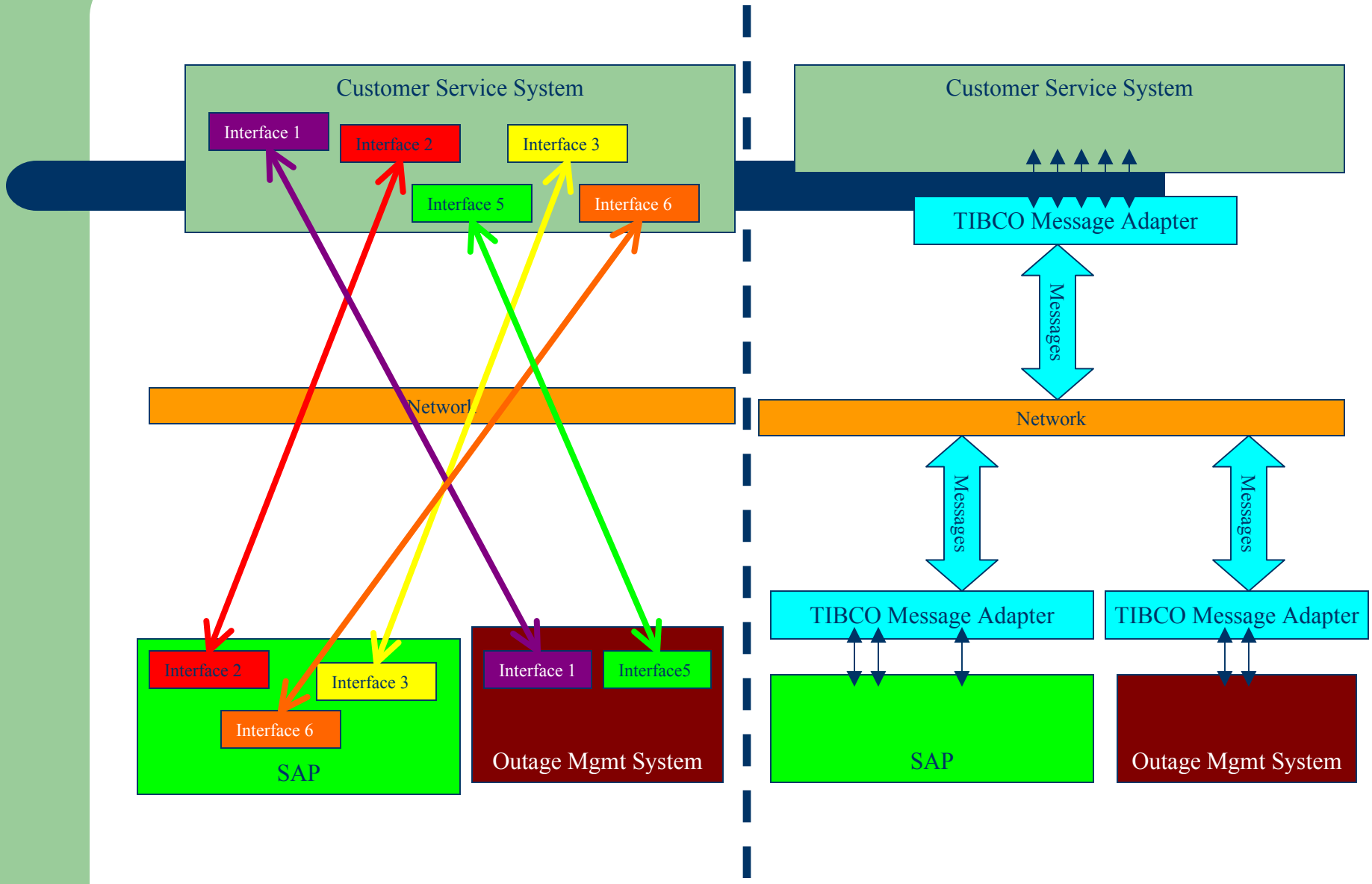
Helen Umberger

The Goal : EAI

- Why?
  - Businesses are hoping for :
    - Extend the life of legacy systems
    - Able to buy vs. build new applications
    - Ability of business mangers to see trends/data across the company.
    - Able to quickly merge in an acquired companies data/systems
    - Speed, Business wants new functionality when they want it
    - Consistency enterprise wide in how auditing, security, disaster recovery are handled.

- How do we get there?
  - Do just start integrating disparate applications? And the benefits magically appear?

Standard Interfaces Vs. TIBCO Interfaces

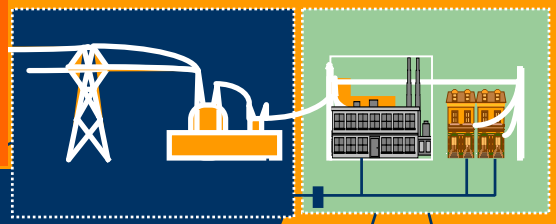


How do we get to EAI from a Traditional Environment?

- First the data has to be understood.
- Standard model provides a “jump start.”
- Enterprise-wide model spans multiple application and database domains.
- Individual projects can focus on their scope, knowing that their objects can be leveraged by other projects.
- It provides a neutral vehicle for achieving agreements among different parts of the organization.
- The content and relationships of various types of data can be discovered.
- Lowers the life-cycle costs of those applications.

CIM is the key element

The "common language" is based on the common information model (CIM) Specified in IEC standards. The meta data for this language is maintained In the information exchange model (IEM). The physical implementation of The IEM varies based on selected technologies.



Information Exchange Model

Grid Wires Model

DAC

Dist Wires Model

VRU

Outage Reporting

EMS

Outage Managemen

Distribution Automation

CSS

The "Integration Bus" = EAI + Common Language

Event History

AM/FM/GIS

Human Resources

Work Management

Substation Automation

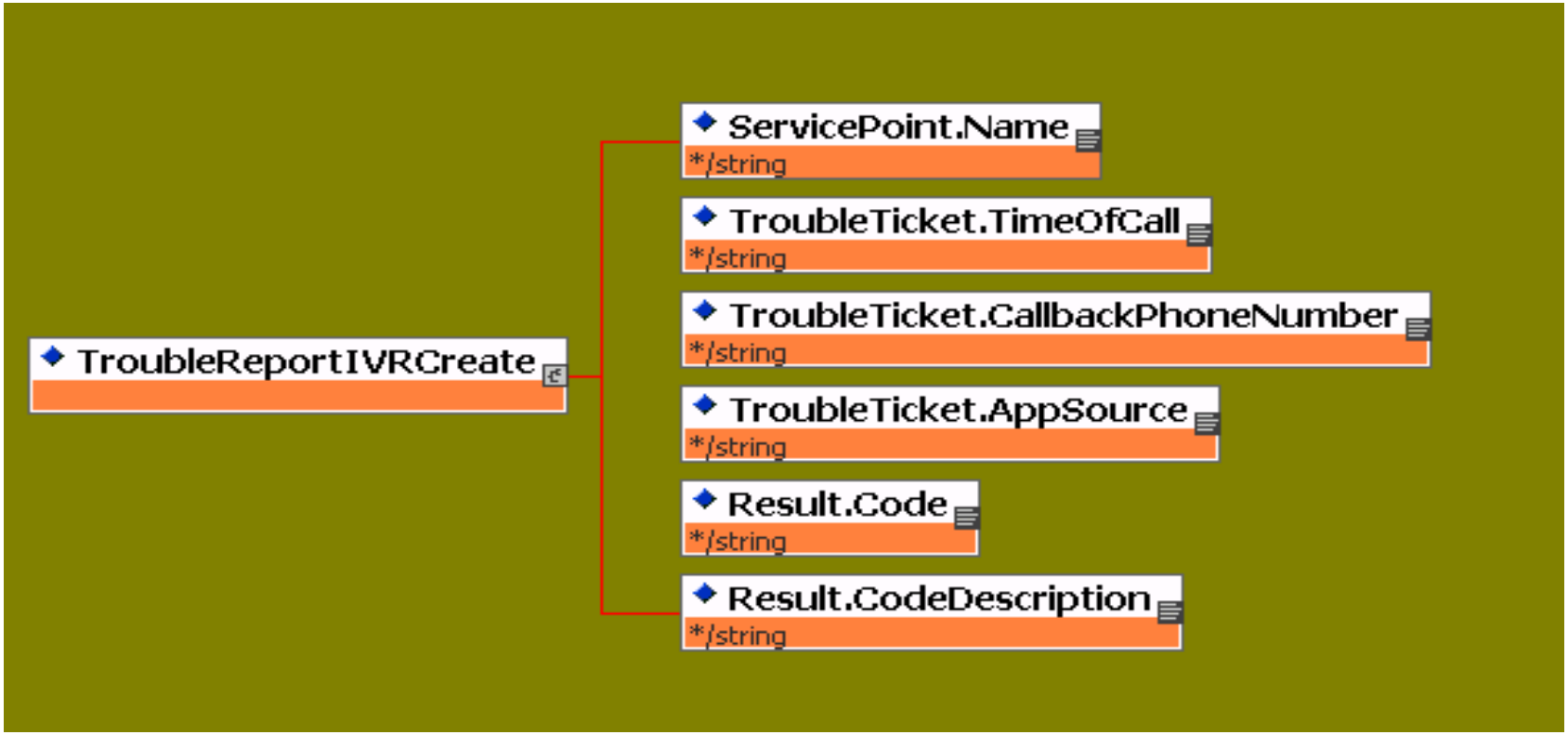
Data Warehouse

= Application Connector

CIM is the KEY

- CIM solution
- Moving to a canonical data structure
- Problem
  - When more than 3 systems to be integrated have different data formats, the total number of transformations is n*(n-1) for n systems with disparate data formats
  - Result – too many interfaces to be maintained
- Solution
  - Develop each interface as a CIM message as an XSD
  - Pair each source or target data flow with that CIM-formatted message
  - Reuse messages whenever possible
  - Use a standard data format pattern to represent data when
    - Integrating applications across the message bus
    - Designing custom applications and databases
    - Building the Enterprise Data Warehouse

Example of an CIM mapping



CASE Study: Retail Access

- PacifiCorp's first use of CIM and EAI
- 2001 Mandated by state legislation, deregulation of the Utilities.
- **Why did we use EAI and CIM for this project?**
  - ✓ Moving away from creating in-house solutions.
  - ✓ Pick minimal solutions.
  - ✓ Requiring tighter integration of existing applications, and newly purchased applications.
  - ✓ No new point to point interfaces – since applications data needs spanned multiple platforms, and systems.
  - ✓ Data needed to be shared with other utilities and power re-sellers need industry standard for data.
  - ✓ Need common data standards for EAI to work – benefits of EAI shown next slide.

Benefits

Cost avoidance = (acquisition + configuration + programming avoidance – reuse cost) * reuse count)

Tradition:

- History on traditional projects showed us the cost of developing a traditional point to point interface was estimated at $15,000. ($465,000 total)
- History on traditional projects showed us the time of developing a traditional point to point interface was 21 man weeks of effort. (651 person weeks total)

This project:

- Estimate was that this project would have 15 system interfaces, 8 of which would go to 3 or more systems. The equivalent of 31 new point to point interfaces.
- Estimate was five man weeks to design, configure, and test each interface. (75 person weeks)

Results:

- Per the formula above Cost for each new adapter was $20,000 (Tibco software) Final cost for all interfaces 297,000 – a savings of 148,000 or 32% savings.
- Per above a savings of 576 person weeks.
- With out the time savings, we would never have made the States legislation deadline – additional cost in penalties was avoided.

* Without the CIM model of the data, we won't have gotten reuse out of 8 of the interfaces. Or had the same level of confidence in the data.

Conclusion

- A complete enterprise-wide integration strategy requires implementation of a standard corporate-wide data model that transcends individual application systems.

  This is achieved by:
  * Creating stability with standardized data abstractions.
  * Allowing the corporate meta data to be created/defined and controlled by a centralized data modeling group.

- Giving the following benefits:

  * Improved ability to integrate business processes across commercial-off-the-shelf applications.

- * A simplified means for internal and external organizations to share information.

  * Less dependence on individual vendors

  *· Improved usefulness of existing applications.

  *· Lower overall life-cycle costs of applications.

  * Support for event-driven business processes.

  *· Ability to identify data that is a corporate asset (either to protect it or place value on it).

Epilogue

- To date PacifiCorp has integrated more than 25 of its systems, and investments payback is meeting expectations. Improved volume through put, lower maintenance, quicker turn around expectations have been met.

- In conjunction with other components of the PacifiCorp strategic IT infrastructure, such as data warehousing, Web technologies, process modeling, data modeling and application code reuse, projects are being implemented in much smaller chunks. This improves delivery quality, budget and milestone delivery. This approach also allows business value to be delivered sooner and project risks to be managed more effectively.

Common Software Testing Risks:
Common-Sense Mitigation Approaches

Claudia Dencker
Software SETT Corporation
233 Oak Meadow Drive
Los Gatos, CA 95032
408-395-9376
cdencker@softsett.com

Abstract
With a lot of diligence and hard work, we can design a test strategy that is thorough and complete, but not risk free. Problems happen, and some forethought is prudent. This paper presents creative and common sense mitigation approaches to 10 common software testing risks. Today mitigation approaches that don't necessarily cost money or require radical change are ideal. Focusing on common sense ideas that are affordable, that work, are creative and sometimes, fairly obvious are paramount strategies that help get the job done smoothly and efficiently.

Certainly the reader's context (organization, politics, overall test charter, technology of software under test) will determine the flexibility he or she has in applying the suggested mitigation approaches. At a minimum, this paper will help a Test Lead get started or think of additional options to common software testing risks.

Bio
Ms. Dencker is President of Software SETT Corporation, established in 1987 and specializing in SQA services for client/server applications, embedded systems, and web applications. She has taught classes worldwide through the IEEE and to major Silicon Valley companies. She also participates on software testing projects in either an advisory or hands-on role. Ms. Dencker received a Bachelors degree and a teaching credential from San Jose State University.

Introduction

No test strategy or test plan is risk free, no matter what the confidence level may be. Problems happen, and some forethought is prudent. The list of what can go wrong is limitless, but software testing problems tend to fall into some basic and familiar categories, such as resources, scope and time. This paper presents creative and common sense mitigation approaches that provide the reader the options to successfully manage or participate in a software testing event. Certainly the reader's context (organization, politics, overall test charter, technology of software under test) will determine the flexibility he or she has in applying the suggested mitigation approaches, but, at a minimum, this paper will help a Test Lead get started.

The 10 top software testing risks are addressed in terms of articulating their full impact to the software testing project. Each risk is defined in terms of what it means and the specific mitigation approaches available. The following risks are discussed.

- Risk 1: Lack of staff availability.
- Risk 2: Lack of staff knowledge.
- Risk 3: Lack of test environment availability.
- Risk 4: Lack of test environment stability.
- Risk 5: Lack of test coverage across product scope.
- Risk 6: Lack of test depth across conditions.
- Risk 7: Lack of realistic schedule estimates.
- Risk 8: Schedule mismanagement.
- Risk 9: Defect mismanagement.
- Risk 10: Lack of testing progress.

Risk 1: Lack of Staff Availability.

Recruitment challenges of yesterday are replaced with staff loading risks today. Yesterday we couldn't find "warm bodies" quickly enough, whereas today finding enough hours in the day for the remaining staff is the primary challenge. Existing staff is being tasked to do more with less resources and time. This results in some of the following effects:

- Fragmented staff where team members cut across more than one project at a time resulting in a loss of focus and effectiveness.
- Normal time-off plans that conflict with project demands leading to overworked staff.
- Staff shortages jeopardizing project schedules and scope (formerly a team of testers would do the job, now only one remains).
- Untrained staff being solicited to assist resulting in increased training and increased management.

Individual testers may not mind having too much to do, since being busy in a down economy translates into job security. But, the Test Lead has a different perspective. He or she is faced with the challenge of getting the job done with the visibility and attention that is needed.

Risk Mitigation Approaches: So what is the Test Lead to do? How can they get the job done as well as before? Are the right people left or will the lead require additional assistance? Aside from negotiating more time, here are some risk mitigation approaches to consider.

- Document team procedures. In years past, projects moved so fast that proper team documentation was a dream; today it is a necessity. With reduced and decimated test teams, the remaining group must find a way to capture the knowledge that just left the

door. In fact, some would say it is too late. But, better late than never. In fact, documenting team procedures is a good way to learn how things were done in the past and to improve the future. The major benefit of documented processes and procedures is to ensure repeatability of past methods. The kinds of procedures that need to be documented are:

1. Whom to contact during a project.
2. Important meeting times and other key test dates and "windows".
3. Locations of project files and archival directories.
4. Access and security information.
5. Testing guidelines unique to the upcoming test event.
6. Definition of the DTS (defect tracking system which comprises use of the bug tracking tool and supporting processes, such as escalation, review and final disposition of defects).

- Use offshore teams. Here is a solution that has become quite popular with management over the years. Whether we like it or not, offshore test teams have become a reality (as has offshore development and offshore customer support) and here are a few words of caution:
    1. Be sure to identify a project manager whose sole purpose is to monitor the relationship and who will be involved personally with travel and constant communications. Face-to-face is important.
    2. Establish strong feedback mechanisms that are frequent, such as in major and minor milestones.
    3. Communicate, communicate, and communicate! This is important for local teams, but increases in urgency for remote teams.
    4. Understand cultural context. Humans interact under a wide range of cultures, which makes life interesting. But, this interest can lose its allure when we try to meet deadlines together.
    5. Keep assignments very well defined, even if the assignment is conceptual or architectural in nature. The idea of a concept scribbled on a restaurant napkin over lunch and from which develops a product that makes it to market won't cut it here.
    6. Plan to work odd hours to map to the offshore team's schedule.
    7. Be as selective as you would be with hiring a local vendor.

Risk 2: Lack of Staff Knowledge.

Another staffing risk is lack of staff knowledge. In a tight job market, junior staff members (i.e., lower paid employees) are called upon to do more senior level work (i.e., higher paid employees). Or, employees from other departments (such as support, technical writing, training, development) may be called upon to step in and fill the shoes of a tester who was just let go. The Test Lead is presented with the following effects of this risk:

- Inexperienced staff resulting in increased training and decreased efficiency during testing. General experience in software testing does count and so does experience in testing the software under test from past releases, but anything less places a greater burden on the Test Lead.
- Untrained staff in the technologies- and product-under-test resulting in a longer ramp-up on the project and lack of effectiveness. While good testing skills are universal and can be applied across a wide range of technologies, some of the ramp-up time can be minimized when you have testers who are knowledgeable in the technology or software (product or system) under test.

Risk Mitigation Approaches: When money is tight and the Test Lead has the challenge of working with staff that doesn't have depth of experience, some options are available.

- Conduct selected training. Training takes many forms with some being very expensive. In our company, Software SETT Corporation, we no longer send employees to learning opportunities that cost in the $1,000's of dollars but rather look for opportunities in the $10's or $100's of dollars. Outside training opportunities that make sense are:
    1. Internet-based training.
    2. Local chapter meetings of national associations such as ASQ (Association of Software Quality), ACM (Association of Computing Machinery), IEEE (Institute of Electrical and Electronics Engineering) and IEEE Computer Society.
    3. Local organizations dedicated to networking and technological developments such as SDForum (Software Developer's Forum) in San Jose.

- Conduct internal training. Bringing an individual up to speed in a down market typically results in on-the-job training, under pressure. But here are some other ideas:
    1. Establish a buddy system (informal back-up designee) or a more formal approach of rotation and cross training. This helps to ensure continuity of work and increases knowledge within team.
    2. Build a rotation scheme between testing and development to cross-train individuals. This helps to build better unit testing skills in the developers and build better technical skills in the testers.
    3. Build retention schemes, such as appreciation day, peer reviews, informal pats on back and bonuses, if you can afford it. To honor Software SETT employees we have given gift certificates to local specialized stores, such as Whole Foods.
    4. Establish a quick training/demo from development at test kick-off time to give the testers a head start.
    5. Encourage testers to "play" or get hands-on familiarity with the product on the developer's environment, on a proto-type or on an early Alpha version.

Risk 3: Lack of Test Environment Availability.

Of all the risks in software testing that have remained consistent throughout the years, one stands out, test environment availability. The loss associated with this risk is not having the environment ready in time and as a result impacting the test schedule or worse, impacting the overall project schedule.

Today with IT budgets slashed, not only must the test team make do with what they have, the critical technical support may also be missing. If money has to be spent, equipment purchases undergo multiple levels of approval before sign-off, and there is no longer the ready assurance that the test team will get what they need when they need it. In fact, existing equipment is often reused and upgraded (new memory, larger disk, etc.); even parts off decommissioned equipment are raided, before new equipment is bought.

A test environment not only consists of hardware, but also software. Here is another eye-popping restriction. Software licenses sometimes need to be renewed which can run the test department into big dollars or underlying software layers need to be replaced by competing products since the company of the original layer is out of business or no longer supports the product.

Another aspect to the test environment is the controls that manage the environment. For example, source or version control is essential for the test team to proceed with assurance. How many of you have tested on the wrong version of software? …

Risk Mitigation Approaches: In these times, challenges are significant and force the team to look for creative mitigation approaches when equipment is limited and budgets tight. Scheduling the environment during off-hours has been a common approach, but here are two more:

- Revitalize existing equipment by making less expensive upgrades or raiding parts. One of the legacies of the good 'ole days is an oversupply of equipment. With staff reductions, business downsizing or closures, we continue to have a lot of equipment. Just go to eBay and see what your equipment sells for and you too will realize the impact of the down economy. To stretch our capital equipment dollars, we can
  1. Take parts off graveyard equipment.
  2. Upgrade the less expensive parts that will give you the best possible value. For example, many of our servers had their memory increased to help boost performance because memory is now cheap.
  3. Use existing equipment longer.
  4. Find ways to use equipment smarter. For example, rather than to buy four different computers to create four unique OS/Browser combinations for our web testing, we have one computer with four swappable hard disks that have the OS/Browser combinations that we want.
  5. Buy at the lowest possible price by aggressively comparison-shopping on the web or through local outlets. And, always ask if a discount can be applied.

- Create an environment checklist. During planning, put together a checklist of tasks to assure full access of the test environment, especially if setup is complex. Be sure to account for browser versions that cannot be downgraded. Most browsers only can be upgraded. Also, don't forget version control setup. If an objective marker cannot be set indicating the version is ready to be tested, be sure to establish a manual procedure. There is nothing worse than to have the test team start on the wrong version of software.

Risk 4: Lack of Test Environment Stability.

Another environment risk is lack of stability. With head count having been reduced, IT technical support can be rare and expensive. Problems will arise during testing and often the cause is the test environment itself and not the software under test. In fact, some teams encounter this problem so often, that they track environment risks in the bug base along with all the software defects.

Lack of full environment stability translates into the following testing impacts:

- Equipment failure of interim equipment, downtime or slow performance leading to possible skewed test results or slowed testing progress.
- Additional management tasks with respect to monitoring equipment delays resulting in increased Test Lead burden.
- Denied access to test equipment due to sharing with another, higher priority task or group resulting in further delays.
- Invalidated lab setup leading to false testing starts. This can mean that the first test cycle not only exercises the software under test, but the test environment as well, which is typically a waste of time.
- Unavailable software or hardware test builds which could impact the schedule.

Risk Mitigation Approaches: An unstable test environment is a given on many testing projects. Rather than to deal with it on the back-end when your team is in the heat of testing, consider the following:

- Check out the environment before testing even starts. Many test environments are set up without undergoing a final checkout. Don't fall into this trap. Instead, be sure to ensure that everything works as expected by running some simple tests. Turn off the machines and reboot to make sure configuration files are correct, access is granted and basic operations can be completed.

- Track configuration problems or environment-related risks in the bug base. Be sure to track environment failures in your bug base in order to analyze the hit to the schedule and test results for the next round of testing.

Risk 5: Lack of Test Coverage Across Product Scope.

Test sets should achieve two goals: breadth across the product under test and depth of test conditions. This breadth and depth criterion is not always achieved for a range of reasons, but the most common is lack of time and lack of expertise. Designing test cases represents one of the most creative and serious aspects of a tester's responsibilities. Regardless of whether the test strategy is structural or exploratory, tests must meet the challenge of finding bugs.

When a test set lacks coverage across the product, this means that functionality and features may be omitted from the tests. As such problems will arise:

- Potential omissions in the testing, as some features will be skipped. This is especially problematic for those items coming late into the test cycle.
- Extra testing of items considered out-of-scope since there is unclear visibility of items to be tested and not to be tested.
- Undue emphasis on some items resulting in an inefficient use of time.

Risk Mitigation Approaches: Your options will depend very much on your team's skill set, company or project context, and the influence that you may or may not have. Aside from encouraging the creation of project and technical documentation, here are two approaches you might want to consider.

- Establish priorities early. Priorities should be set in the beginning of the project or no later than when test design starts. When scope changes, priorities may be the only way by which a course of action can be set. Through the setting of priorities, the test designer can focus his or her efforts by creating tests for those items that are most important to the release. Lower priority items will be addressed if time permits. For example, on many of the Software SETT projects, we organize the feature set into high and low priority items and document this in the test plan.

- Establish exit criteria. Exit criteria help to establish the guidelines for the release and can be another vehicle for maintaining control during change. Exit criteria help set the standard for the test set and the overall testing effort which in turn will guide the test designer to create the correct balance between test breadth and depth.

  For example, on a past project we created a section, Product Coverage, in our exit criteria and listed the specific user capabilities and the features and components needed to assure that the capability worked. This list of features and components helped us to focus our efforts correctly and minimize the risk of not covering the "right" scope. A specific example from the Exit Criteria document follows:

  > "Capability 1. Being able to enter and update imported, structured content through the input tool and being able to view it in the editor, on the company website, and in

the out-going data feed. ***Components:*** Automated and manual import processes, input tool, document repository, company web site and publishers."

Risk 6: Lack of Test Depth Across Conditions.

When a test set lacks depth across conditions, this means that the testing that follows is not robust, but rather thin. There is a flipside to this problem whereby tests may be too detailed (i.e., take too much time to execute or don't match the project objectives) or just not interesting enough with tests that range from simple to complex. Tests that lack depth can be characterized as follows:

- Testing is time driven (indicating what can be done) or personality driven (do as you're told or test around the bugs) resulting in a loss of quality (indicating what should be done).
- Tests are redundant, have holes, duplicate one another, or are too simple or confusing resulting in a poor effort and a costly release with a high number of support calls.
- Expected results are weak, not extensive enough resulting in missed opportunities for further bug identification.
- Data set is not rich enough or doesn't address real-life examples resulting in a lost opportunity for simulating real usage.

In order to accommodate test set problems, the Test Lead may have to make mid-stream corrections. This can happen to the best of test strategies as unforeseen events dictate change. But, this is not a good thing when starting out.

Risk Mitigation Approaches: Some of the mitigation approaches for poor breadth also apply here. Listed below are some additional ideas unique to poor depth of testing.

- Extend your expected results. A robust expected result should answer the following questions:
    1. WHERE to look? Identify the screen, report, database query, log file that the tester should review.
    2. WHAT to look for? Identify the specific value or string, condition or state or lack thereof that the tester should review.
    3. WHEN to look? Identify the specific point in the process when the tester should conduct their validation.

  Sometimes several features will need to be checked to verify that a test has passed. Occasionally the expected result of a test is that nothing should happen upon its execution.

- Assign test case filters to establish a meaningful subset of tests to run. To keep your test set meaningful throughout the test event, consider selecting your subset using one or more of the filters below.
    - **Costs.** Some tests are more expensive to run or to setup; therefore, you may or may not want to pick these tests. Conversely, cheap, low cost tests may be tempting to run, but not necessarily wise or appropriate.
    - **Technical context.** Pick tests that relate to and support the release, technical objectives, i.e., what is new or complex in the given release, what platforms are supported/unsupported?
    - **Business context.** Pick tests that relate to and support the release, business objectives; i.e., what the release attempts to solve for its users, what the level of quality must be in order to release.

- **Project context.** Pick tests that relate to and support the project objectives, i.e., what is practical to do given the resources, schedule and staff available. For example, if there are no junior testers, then pick tests that leverage off of senior knowledge and talent. What is the skill competence and/or arrogance factor of programmer(s) and pick tests that exploit this.
- **Baseline functionality.** Pick tests that exercise a baseline of functionality. This assumes that the baseline has been defined. For a release where installation is critical, running the install tests will answer the baseline question: Does it install?
- **Risk-based.** Pick tests that target the highest risks on a project. These should be clearly articulated in the requirements document or lower level specifications. Some areas of typical risk are**:**
    1. Areas of code where the product has changed due to fixes, new features, or code "clean-up".
    2. Probable failure points due to poor programming or complex programming.
    3. Unacceptable behavior on or impact to the customer.

Risk 7: Lack of Realistic Schedule Estimates.

In the past when market pressures were enormous, getting enough time was a major concern. Now that we have a slower economy, other concerns divert us, such as cutting costs to stay in business and finding ways to still bring value to the marketplace.

With goods and services being bought at a slower rate, we can use this "break" to focus on doing it right. Though, successful companies will realize that, even with this slowdown, pressures to continue to deliver value to our customers is unrelenting and necessary for long-term viability. There is truly no rest for the weary.

Historically, test schedules have been under-estimated because individuals not knowledgeable in testing, development or in the technologies set the schedules early on through "broad brush" scheduling. Some recent development methodologies have attempted to right the wrong in scheduling, by releasing more frequently and in a cross-functional manner where all the players are involved from Day One.

Even if the "right" people set the schedules, teams may still suffer from lack of realistic schedule estimates. Not enough time may be estimated for certain testing activities, because:

- Test execution is taking longer than anticipated (too many bugs, tests take longer, instability in the test environment, staff changes, etc.).
- Retest time was overlooked and not scheduled at the beginning of each test build resulting in a rushed effort in order to fit all testing in.
- Upfront test activities such as test planning, test design and implementation were skipped altogether leading to an ungrounded test effort.
- Unanticipated impacts from test tools are causing delays (ramp-up, implementation, integration into test environment, etc.) and resulting in distractions from overall testing goals.
- Too many tests that need to be run are causing the definition of the test subset to be rushed.

Risk Mitigation Approaches: We can use our past experience and objective measures to set our schedules, but this doesn't always guide us correctly. So what is a Test Lead to do? Just work faster? Work harder? Here are some ideas on getting the time you need.

- Buffer your time. This is a time-honored way of scheduling. Identify your schedule and then double it to account for the unexpected. But, doubling your time may be unacceptable. Try these ideas:
    1. Increase each item within the schedule by an acceptable percentage based on your confidence level of the item or ramp at 50% effectiveness. For example, if your team consists of new people and your product contains cutting-edge technology (never been tried before) consider making your first test cycle longer by 50%+ than the following ones (or cut your team's effectiveness by 50%) to account for learning curve and initial test set-up.
    2. Serialize your testing tasks with later adjustment to parallel/overlapping tasks. This means that all testing tasks follow one another where follow-on tasks are not started until the prior task has completed. Unfortunately not all tasks complete as expected. An option is for you to extend some tasks that need more time by having part of the team continue, but still start the scheduled follow-on tasks with the remaining team members. Now you're running tasks in parallel. Serializing tasks during initial scheduling will give you some elbowroom later on.
    3. Create a "bogus" test cycle or final regression test that can act as your "slop." Many of my schedules have a final regression cycle that not only acts as a final run through of designated or all tests, but it also gives us time to clean-up or catch-up in initial runs that may remain. Keep in mind that during testing, some testing tasks may be added thus complicating your already tight schedule. Scheduling a final regression run will help give you that time to squeeze something in.

- Get commitments early. This means that during your test planning, you should identify your dependencies, needs, and requirements. Work with the project team during this phase to get your commitments. Don't wait until testing is well underway, because then it becomes more complicated. Here are more ideas:
    1. If you have staffing risks, see if development can assist by running some of the tests either under your direction or the direction of the development manager or lead. On past projects we have used development resources to great success where a few of the junior developers were assigned to the test team. They took on the testing mindset and were exceptionally good at running tests, identifying defects, and retesting. A note of caution, the developers did not test functionality and code on which they had worked. As an aside, these "devtesters" moved on in their careers and became outstanding development engineers whose expertise was sought by many.
    2. Even though a schedule may be "set in concrete", if your analysis indicates that you need more time, do not hesitate to present your case. Years ago we led a testing effort for a large data warehousing project that included major schema changes, data conversions, and added tables. The testing schedule was two months in duration. After analyzing all the requirements and designing a test strategy that fulfilled the requirements, we realized that we needed three months. We presented our case and facts to upper management and actually got the three months!
    3. Last, but not least, be sure to manage carefully your team's strengths in order to minimize the weaknesses. Rather than to throw resources (staff) at the schedule, even if there is budget, look for efficiencies in the current team and adjust assignments accordingly.

Risk 8: Schedule Mismanagement.

Another scheduling problem once the project is underway is schedule mismanagement. This means that there are:

- Missed milestones (software delivery is late due to development slippage, third-party vendor not delivering on time, late incoming specifications, or fixes not completed on time) resulting in the Test Lead losing control or having to define workarounds on the fly.
- Lack of management support for missed milestones causing sleepless nights for the Test Lead. Management may be too rigid in enforcing the schedule or conversely, management may be too flexible which allows too many exceptions through.
- Poor project management in that coordination with other teams is missing resulting in additional burdens being placed on test and/or impacting their schedule.
- Too many builds coming into test increasing the "churn" or rework in test, thus impacting forward momentum.

Risk Mitigation Approaches: Schedule mismanagement is an all too common of a problem in most projects as evidenced by schedule slippage and missed milestones. Here are two mitigation approaches available to the Test Lead to help with this risk.

- Schedule to details. By keeping your assumptions and unknowns to a minimum, your schedule will have a far better chance of remaining intact and reflecting true efforts. Scheduling often involves several passes as the details get identified and refined. Be sure you know what your team will be doing on a daily or even hourly basis so that if milestones are missed, you know exactly what to do next or what to skip.

- Be selective of the builds you test. Sometimes the pace at which development is able to provide new builds outpaces the ability of the test team to test. The best way to handle this is to indicate the pace that works for your test team. So, if development can provide daily builds, but you find yourself churning in retest, don't accept all builds. Rather, establish an acceptance schedule that works for you, such as a semi-weekly build schedule, but that also doesn't slow development down.

Risk 9: Defect Mismanagement.

Once testing is underway, a couple of typical problems arise: defect mismanagement and lack of testing progress. The first risk has been with us for years, whereas the causes for the latter have changed more recently due to downsizing and the resulting fear of job loss.

Defect mismanagement is one of the more politically charged risks that a Test Lead has to manage. This risk entails a number of things with all impacts resulting in a loss of forward momentum during test and in a loss of quality at release:

- Reclassification of defect severity or priority by developer or defect review committee without fully understanding the defect report or understanding its full impact.
- Poor fix quality where the fix is incorrect or not complete enough or breaks something else. Defects can often go into "churn" where the defect is alternately tagged open and fixed for a period of time.
- Unknown defect state because there are missing release notes, no one takes the time to update the bug base or worse, a bug base isn't being used. A bug that sits in the "open" queue for a long time is not a good thing.
- Unrepeatable defects clogging up the fix pipeline and/or bug base causing a loss of forward momentum in the fix process.

- Production defects diverting the attention of the test team causing a loss of focus. **Note:** While production issues should always take a priority, the Test Lead may want to designate an individual to handle these issues, rather than to have the entire team refocus their attention.

Risk Mitigation Approaches: During active testing, a Test Lead will need to ensure that defect mismanagement is kept to an absolute minimum. A couple of approaches are available:

- Establish a DTS (defect tracking system) that has full review processes created, owners and reviewers identified and escalation processes fully documented. A DTS system not only ensures progress on the fix process, but it protects the work that is done by the entire team. It helps to establish the ground rules for logging, reviewing and fixing defects. In addition to the DTS, be sure to also define the severity, priority and status values that are available when tagging your defects. If everyone goes off the same definitions, the variance in tagging and incorrect downscaling is kept to a minimum.

- Run regular reports. As a Test Lead, you should be receiving or creating regular reports that allow you to observe trends, but to also see what is happening at a detailed level. Running queries in the bug base to pick up defects that haven't had any activity is important. On a past Software SETT project, we run an Aging Report on a weekly basis, and then notified the owner that he or she hadn't worked on a particular bug over the past week. Another idea is to look at the defect history to identify "churn" that basically caused delays or incorrect routing of the defect.

Risk 10: Lack of Testing Progress.

The second test execution risk, lack of testing progress, has many, many sources of which a few are:

- Tester morale leading to a loss in forward momentum. As companies continue to find ways to work more efficiently, management makes significant changes by either cutting costs (lay-offs, delay in equipment upgrades, frozen salaries, reduced benefits) or in various internal restructuring (new organizational charts, merged profit centers). All these changes can result in lowered morale and employee uncertainty. The question, "What's next?" takes on greater significance than in years' past.
- Invalidated test cases/procedures entering test execution resulting in inefficient testing. Because much of our test design is done before we can fully check out the tests, starting our testing with shaky tests is not unusual. Most test professionals are used to this and adapt their testing on the fly. This risk becomes more serious when testers are junior and inexperienced.
- Poorly estimated test cycles resulting in tests that take longer than expected. Sometimes testing is more complex or just takes longer than originally scheduled. This also leads to a lack of progress and could probably be remedied in the future by scheduling to a more detailed level.
- Early build versions that are slow in coming, resulting in an idled test team. Certainly in this situation some offline tasks could be completed such as reviewing test cases, adding new ones, or doing a more complete job of test preparation. But, an idled test team is not a good thing.

Risk Mitigation Approaches: Mitigation approaches that help keep momentum positive and forward moving are important. When money is tight, we are required to think more creatively and make do with what we have. This is not necessarily a bad thing. In fact, some amazing things have been done when we are forced to economize. Here are some approaches you can use:

- Educate development or management as to the cause/effect of changes. Help them to understand the controls and/or management processes that need to be in place to get a better handle on defects.

- Prioritize tests by first reviewing project objectives, conducting a risk analysis individually or through a collaborative team effort. Once priorities are set, some level of filtering can be done to determine the best subset of tests to run. The test matrix can be reviewed to fit to the revised priorities.

- Establish "No Fix" meeting. When scope changes, certain defects, test cases, test cycles may be escalated or down graded. "No Fix" meetings help establish the boundaries of what the technical team can fix and cannot do when scope is changing.

Conclusion

In summary our 10 top risks present challenges to the Test Lead. In fact, many of the risks such as staffing and test environment challenges have been with us for years. The only difference is that their mitigation approaches will vary depending on company culture, general health of the economy and project goals. Today mitigation approaches that don't necessarily cost money or require radical change are ideal. Focusing on common sense ideas that are affordable, that work, are creative and sometimes, fairly obvious are paramount strategies for the Test Lead in these more difficult times.

In general managing the risks for a Test Lead requires vigilance and knowledge. Knowing what's going on both formally and informally will allow the lead to pull from their toolkit of knowledge. Perhaps even a few ideas presented in this paper will help the Lead do a better job.

Acknowledgment: The ideas and concepts presented for the test case filters are the result of the LAWST 15 workshop, March 2003 attended by the following individuals: Mark Johnson, Brett Petticord, Claudia Dencker, Doug Hoffman, Elizabeth Hendrickson, Geordie Kit, Harry Robinson, Joe Webb, Ross Collard, Chris Sepulveda, Neal Kuhn, Mary McCann, Brian Lawrence.