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, 2002

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 _____

TWENTIETH ANNUAL PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

OCTOBER 15 - 16, 2002

Oregon Convention Center
Portland, Oregon

TABLE OF CONTENTS

Management Track – October 15

Test Technology Track – October 16

Process Track – October 16

Management Track – October 16

Alternate Papers

Preface

Members,

Hello! I am delighted to welcome you to the 20th Annual Pacific Northwest Software Quality Conference. Our conference is the longest-running conference dedicated solely to the issues surrounding Software Quality. Twenty years. That's almost a "generation".

With the uncertain economic situation at the time of this writing, I recognize the sacrifices and your employers have made enable you to attend our conference. I can assure you that our conference is one of the greatest values in the Software Quality conference realm.

Please make sure to visit our exhibitors this year - they work hard to bring you information, products, and services to help you and your respective companies do their jobs better.

I believe that all of us recognize Software Quality as very important - otherwise we wouldn't be here! I would like to challenge all of you to the following:
If you find that a subject of interest to you is underrepresented at the conference, please submit a paper next year! Writing a paper offers you an opportunity to learn more about the subjects that interest you, and become an expert in your topic. In addition, your decision to share your knowledge will have a positive impact on our conference next year, and going forward. If you don't want to submit a paper, invite your coworkers, or someone else in the software development field to become a part of this conference. Next year is our 21st year - the beginning of the third decade of PNSQC! Help us make it another banner year.

PNSQC is a non-profit organization, staffed primarily by volunteers. My job this year has again been made very simple because of the dedication of the committee chairs, and the wonderful volunteers among those committees. The excellent program being brought to you this year is because of the exceptional submissions from people like yourselves, and the selfless dedication of the Program Committee - this year with three co-chairs - Bhushan Gupta, Paul Dittman and Sue Bartlett.

I would like to thank all members of all the committees and Board of Directors of the conference for all their hard work and all the volunteer hours they have contributed to this conference. Special thanks as well to our conference manager, Terri Moore of Pacific Agenda. She always handles the organizational and administrative tasks very efficiently.

If you find you like what you find, please let others know. If you find you are dissatisfied for any reason, please let us know. After all, how can we improve if we don't know there are problems? Also - please fill out your feedback forms. We do read them, and we do analyze the results.

Again, welcome, and thanks for attending the 2002 Pacific Northwest Software Quality Conference - the best value of it's kind!

Sandy Raddue
PNSQC President and Conference Chair

CONFERENCE OFFICERS/COMMITTEE CHAIRS

Sandy Raddue – PNSQC President and Chair
Cypress Semiconductor

Bhushan Gupta – PNSQC Vice President, Program Co-Chair
Hewlett-Packard

Paul Dittman – PNSQC Secretary, Program Co-Chair

Doug Vorwaller – PNSQC Treasurer
Willamette Industries

Laura Anneker – Workshop Chair
Hewlett-Packard

Sue Bartlett - 2002 – 2003 Keynote Chair, 2002 Program Co-Chair

Rick Clements – Strategic Planning
Cypress Semiconductor

Dennis Ganoe – Web Master
Prodx

Shauna Gonzales - Birds of a Feather
ImageBuilder Software

Ganesh Prabhala - PNSQC Software Excellence Award
Intel Corporation

Cindy Oubre - Exhibits Chair, Volunteer Coordinator, PNSQC Board Member

Erik Simmons – PNSQC Board Member
Intel Corporation

Evan Tradup – Publicity Chair
Hewlett-Packard

CONFERENCE PLANNING COMMITTEE

Tylene Aldassy
Corillian

Hilly Alexander
Comsys

Randy Albert
Oregon Institute of Technology

Chris Baker
CorePolicy Systems

Pieter Botman
True North Systems Consulting

Kit Bradley
PSC Inc.

David Butt
Oregon Health Science University

Kingsum Chow
Intel Corporation

Manny Gatlin
Intel Corporation

Kathy Harris
Intel Corporation

Kathy Iberle
Hewlett Packard

Mukesh Jain
Microsoft

Mark Johnson
Cadence Design Systems

Jonathan Morris
Natural Messaging, Inc.

PS Rao
Intel Corporation

Jean Richardson
Communication Works

Ian Savage
SunGard/Tiger Systems

Eric Schnellman
JanEric Systems

Ruku Tekchandani
Intel Corporation

Maximum Efficiency Testing and the Future of Software Test Automation

Alberto Savoia, Director of Engineering, Google Inc.

Abstract

We have barely scratched the surface of how computers can help us test themselves. Today, test execution is the only area where software test automation is used consistently; except for a few and rarely used tools, most software tests are still designed, developed, and analyzed manually. Capture/playback tools, for example, are the most popular category in the test automation market; but they are mostly used to automate the execution of *manually* developed scripts using *manually* provided test data. Can all this manual work be automated? Alberto Savoia believes so, and in this talk he will share, justify, and explain his predictions on the future of software test automation. The presentation will focus not only on technological advances, but also on the inevitable organizational and human resource challenges that will arise as we try to maximize software testing efficiency.

***Alberto Savoia**, has been a pioneer and innovator in software test automation for nearly 20 years. He is currently co-founder and CTO of a newly formed start-up focused on test automation technology. His previous start-up, Velogic, was the first to introduce Web-based load testing and was acquired by Keynote Systems in 2000. He has also served as Director of Software Technology Research at Sun Microsystems Laboratories, General Manager of Sun's SunTest business unit, Chief Technologist at Keynote Systems, Director of Engineering at Google, as well as co-founder and CTO of Velogic. He is the author of many articles, papers, and patents on software test automation, and has led the design and implementation of several award-winning testing tools. His presentation "The Science of Web Load Testing" won the Best Presentation Award at both the STARWEST 2000 and Software Quality Week 2000 conferences. He can be reached at: **asavoia@ix.netcom.com**.*

Maximum Efficiency Testing and the Future of Software Test Automation

Alberto Savoia

{ TestAgility }

alberto@TestAgility.com

Overview

- Maximum Efficiency Testing
  - Simplification
  - Reuse
  - Automation
- MET and your career

The Birth of MET

- One of the top 5 brands on the Internet
- >1,000,000,000 hits/week
- ~150 developers
- …
- 3 testers!

Maximum Efficiency Testing

- Effective:
  - Doing the right thing
  - Achieving the desired result
- Efficient:
  - Doing the thing right
  - Achieving the desired result with minimum waste
- Maximum Efficiency Testing (MET):
  Achieving the desired testing objectives with minimum waste

Minimizing Waste

- Precious resources
  - People
  - Time
- Semi-precious resources
  - Money
- Cubic Zirconias
  - CPU cycles
  - Disk space

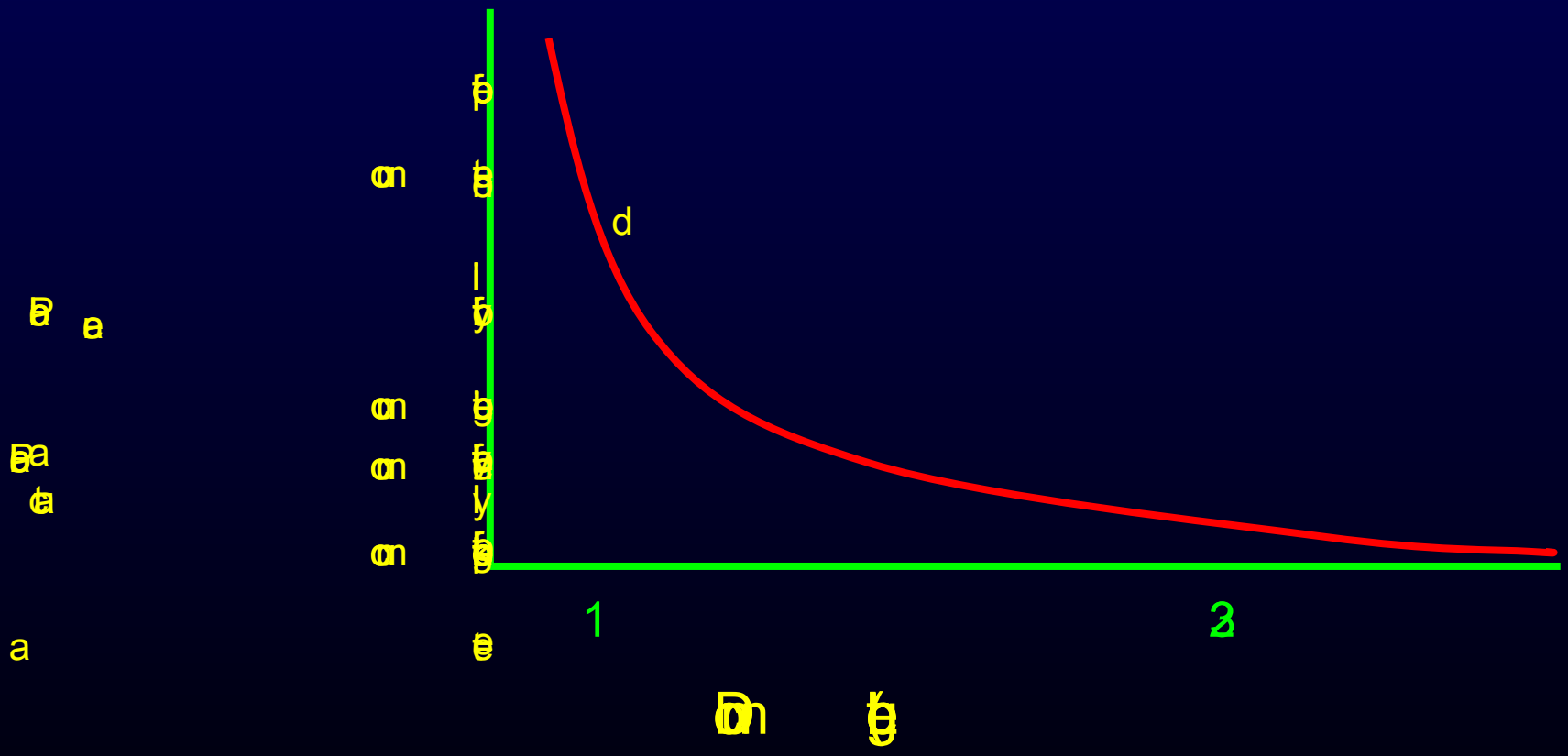
Key MET Strategies

- Simplification
- Reuse
- Automation

K.I.S.S.S.

- Simple and small *things* are:
  - Easier to understand
  - Easier to review
  - Easier to change
  - Easier to accept
  - Easier to follow
  - …
- Complexity and size leads to:
  - Confusion
  - Procrastination
  - Avoidance
  - …

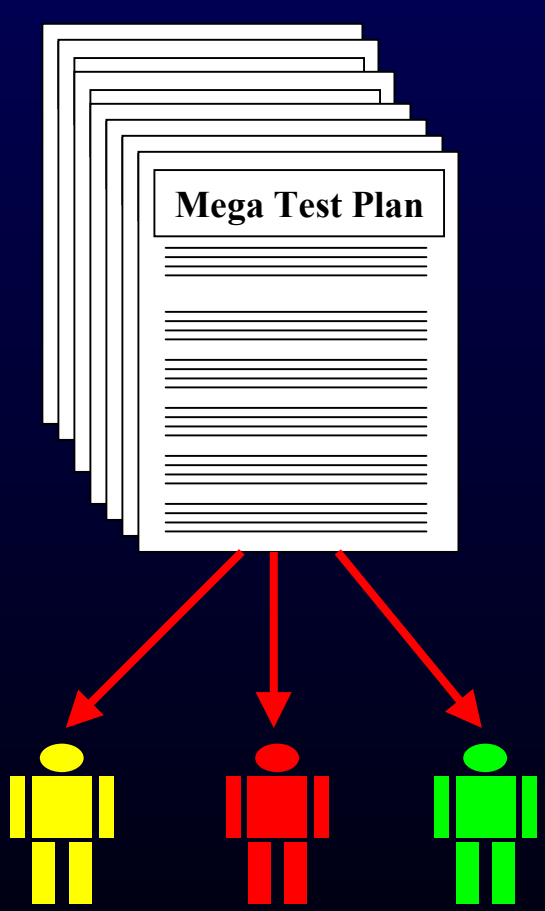
Simplification Example: Test Docs

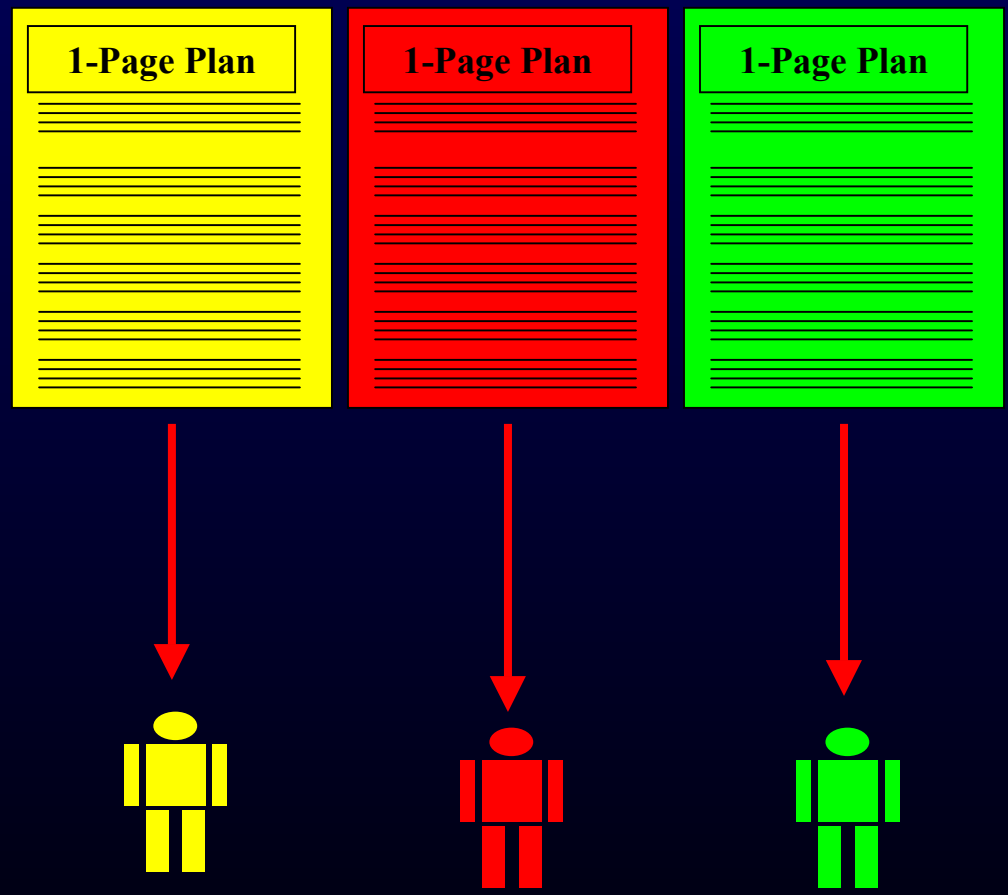


Example MET Solution

- One-page test documents whenever possible
  - 1-page test plan
  - 1-page test reports
  - …
- But how?
  - Split big documents into smaller ones
  - Keep each document tightly focused
    - Focused audience
    - Focused content
    - Avoid frills (re-read Strunk and White's *Elements of Style*)
    - Assume intelligent reader
    - If they don't understand, they'll ask

Avoid the Monolith

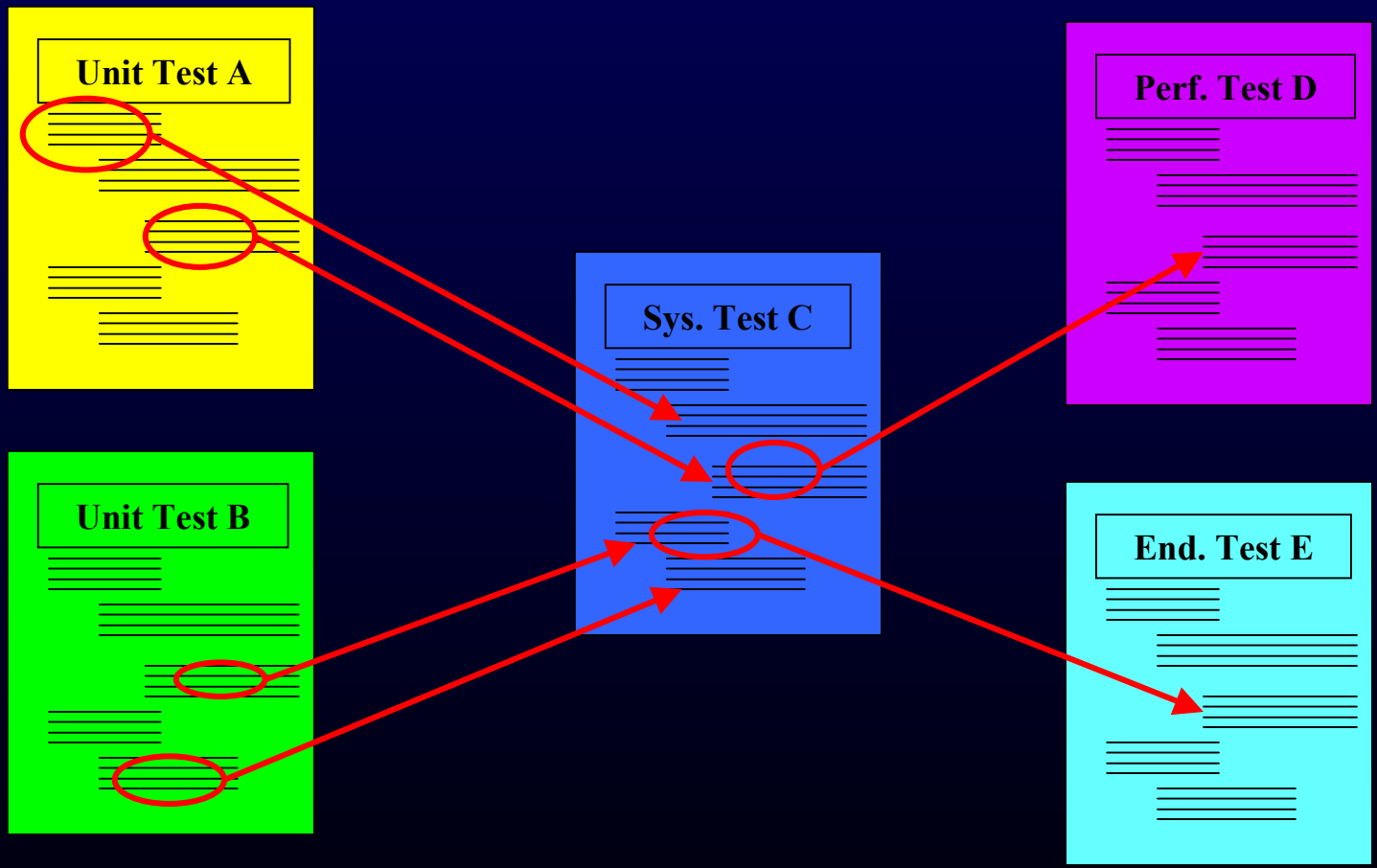




Reusability and Testing

The duplication of efforts in most software testing projects is stunning

Opportunities for Reuse

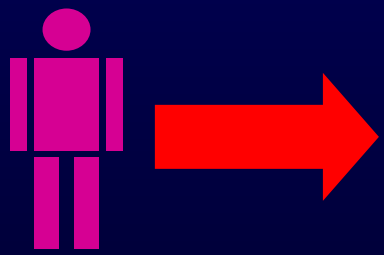


Component Based Testing

- All tests can be partitioned into two basic components:
  - Test data
  - Test assertions
- Testing components can be:
  - Reused
  - Shared
  - Leveraged in dozens of ways to amplify testing efforts
- Huge ROI and gains in productivity and quality

Component Based Testing

Monolithic Testing
(Violates all good software principles)



Personal Test *Infrastructure*

Test Data + Assertions

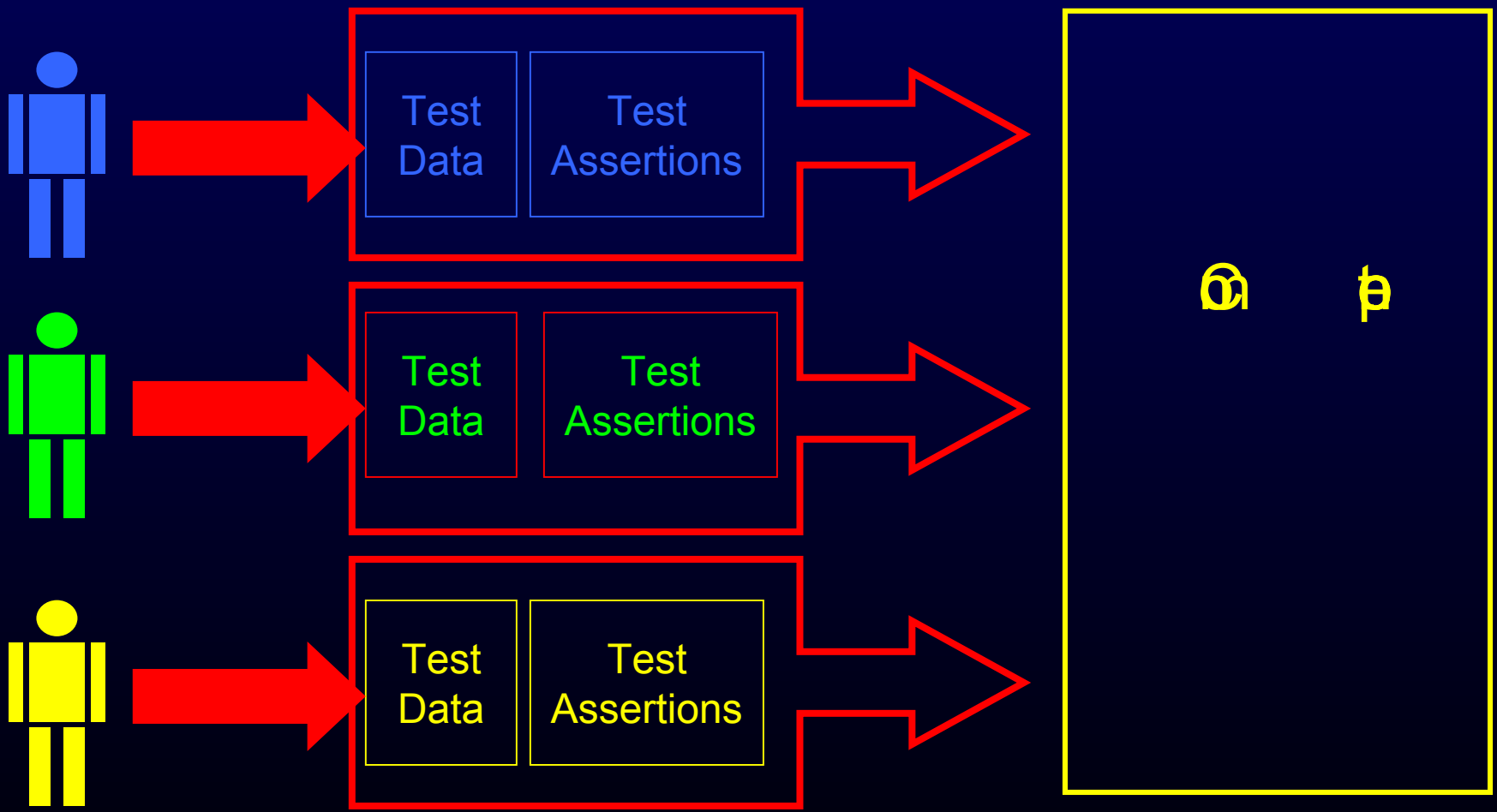
Component-Based Testing

Test Data

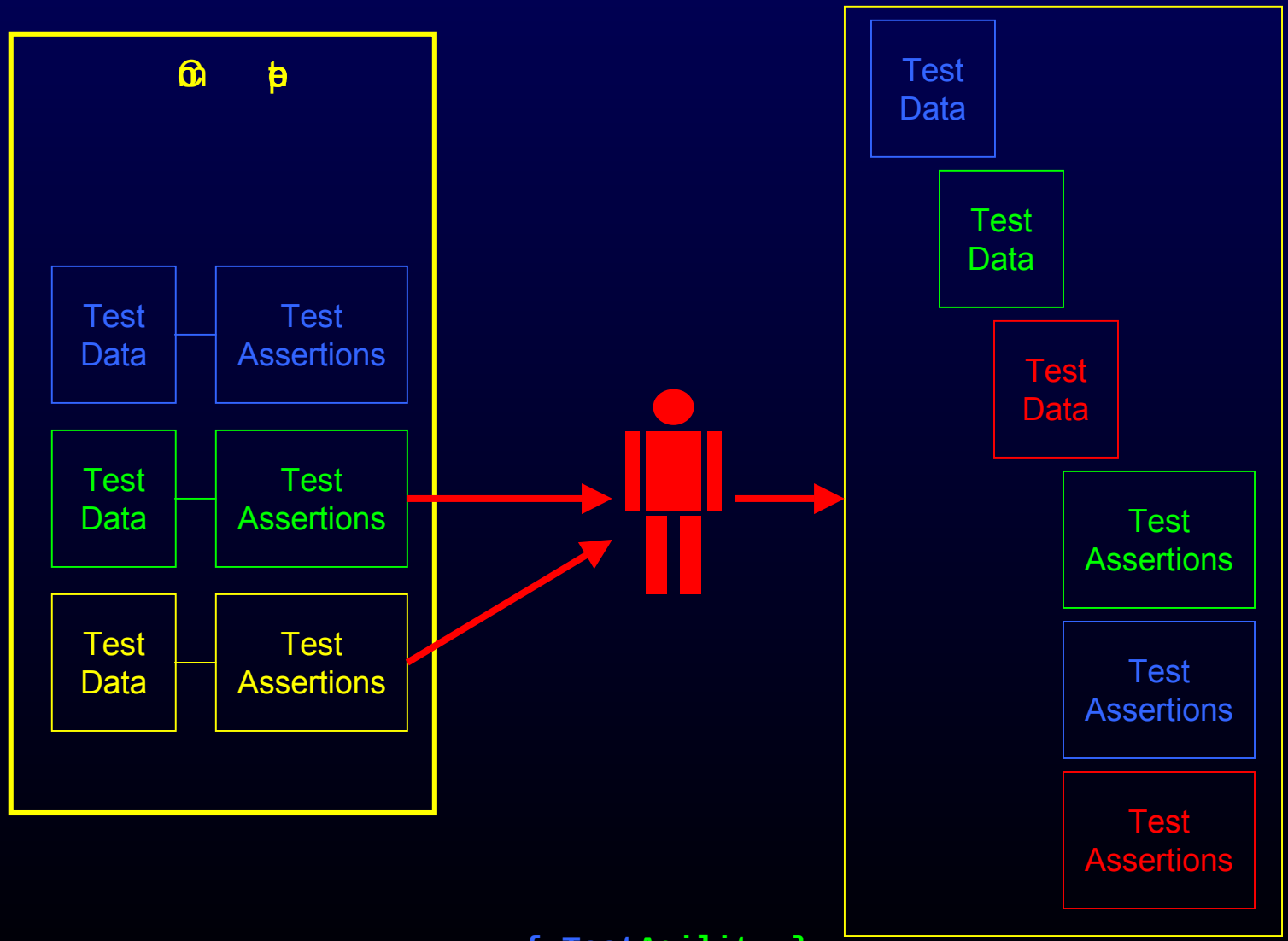
Test Assertions

Component-Based Testing Platform

Test Collaboration



Component Based Testing



Test Automation The Great Efficiency Maximizer

Brief *Personal* History of SW Test Automation

- Pre-GUI Era (pre-1984)
  - FORTRAN, COBOL, C, UNIX, DOS
  - Text based I/O == easy automation
  - Very little in commercial test automation
- GUI Era
  - Pascal, C++, VB, MacOS, Windows, Solaris
  - Why I hated GUIs at first
  - The birth of commercial testing automation tools
- The Internet Era
  - Java, Linux
  - The rise of performance and load testing
  - Software testing becomes a very big business

Why Automate?

The Three Faces of Software Testing

☹	😐	☺
•Boring •Tedious •Repetitive •*A-trained-monkey-could-do-it*	•Somewhat boring •Somewhat tedious •A bit repetitive •Need some human intelligence and creativity	•Fun •Challenging •Need creativity •Need intelligence
Can, and should, be automated for people's sake	Should be automated for the sake of efficiency	May be automated for the sake of efficiency

Test Automation Matrix

Three levels of *automatability*

- Easy to automate
- Somewhat easy to automate
- Difficult to automate

Test Automation Matrix

	Easy to Automate	Somewhat Easy to Automate	Difficult To Automate
	**X**	X	
	X	**X**	X
		X	**X**

Testing Tasks

Testing Task		Automatability	Current Practices
Prioritizing (what to test)		Moderate 2	1 Rarely automated
Test Planning (tasks, schedules, …)		Moderate 2	1 Rarely automated
Test Development		Difficult 3	1 Rarely automated
Test Execution (incl. capture/replay and load testing)		Easy 1	3 Always automated

Automation Opportunities

Task	Automatability Level	Current Practices	Benefits from Automation	Opportunity for More Automation (AL – CP)*Ben
Prioritizing	2	1	3 - High	3
Planning	2	1	2 - Medium	2
Development	2.5	1	3 - High	4.5
Execution	3	2.5	3 - High	1.5

Observations

- <10% of what could be automated is automated
- The *low-hanging* fruits on the software test automation tree have been picked:
  - Test execution
    - Capture/Playback, Load Testing, Test execution frameworks (e.g. Junit)
- Today
  - Test automation ←→ test execution
- Biggest opportunities
  - Test Prioritization
  - Test Planning
  - Test Development

Test Prioritization

- 100% test coverage is too expensive for most organizations
- Prioritizing is key
- How do **you** prioritize testing?
  - What's easiest/fastest?
  - Best guesses based on intuition / experience?
  - Objective metrics?

Automating Prioritization

Task	Partially Automatable	Automated Today
Gather objective metrics on the various modules Complexity Popularity Developer history Risk if module fails	YES	Very Rarely
Augment with your intuition	NO	N/A
Do what's easiest/fastest	YES	Very Rarely

Error-Prone Modules

An error-prone module is a module that is responsible for a disproportionate number of defects

Steve McConnell
Rapid Software Development

IBM Example: 57% of errors in 7% of the modules

Caper Jones (1991)

20% of modules in a program, typically responsible for ~80% of the errors

Barry Bohem (1987)

“Those two developers are your best insurance for continued employment”

Alberto Savoia, ~1988, talking to a software tester

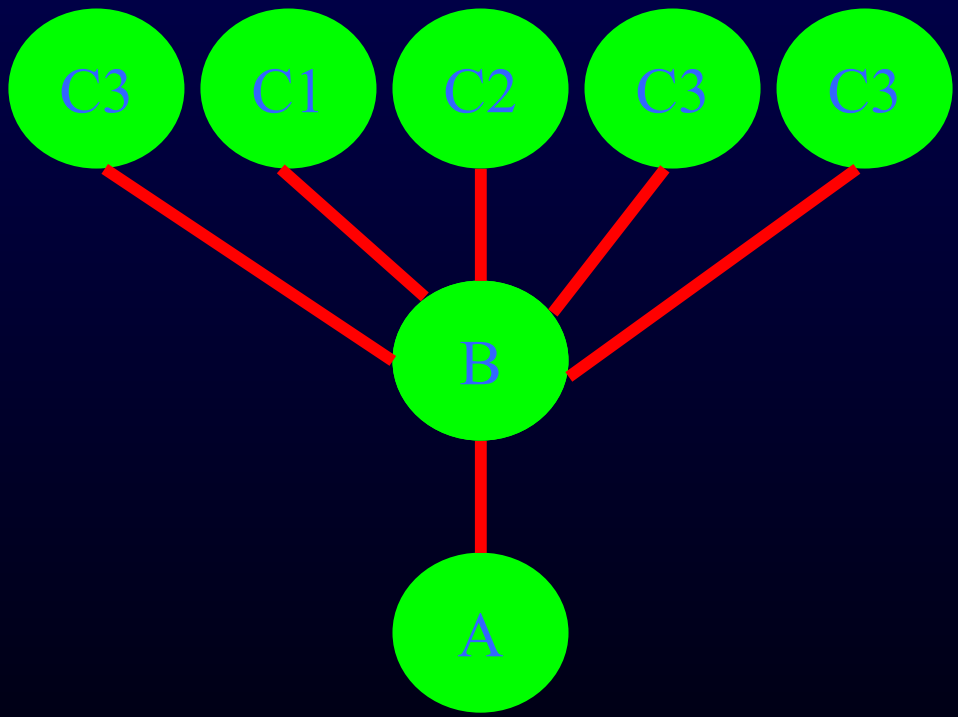
Prioritizing By Developer Risk

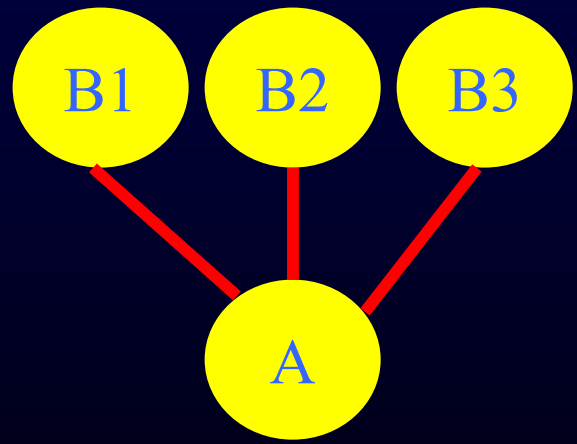
Developer's Name	Historical Defects/Module	Defect Risk Rating
Alan	3.4	3
Betty	1.1	1
Clifford	0.7	1
Dave	7.8	5

Prioritizing By Complexity

Module	Complexity Score	Defect Risk Rating
A	234	2
B	120	1
C	20	1
D	60	3

Prioritizing By Popularity





Automation Example

Automated Test Prioritizer					
Module	Number of Functions	Complexity Score	Popularity Scores 1st/2nd level	Developer Risk Factor 1(low) – 5(high)	Test Priority (Risk)
A	13	234	34/154	1	2
B	34	120	12/67	3	3
C	3	20	1/1	2	4
D	23	60	8/67	5	1

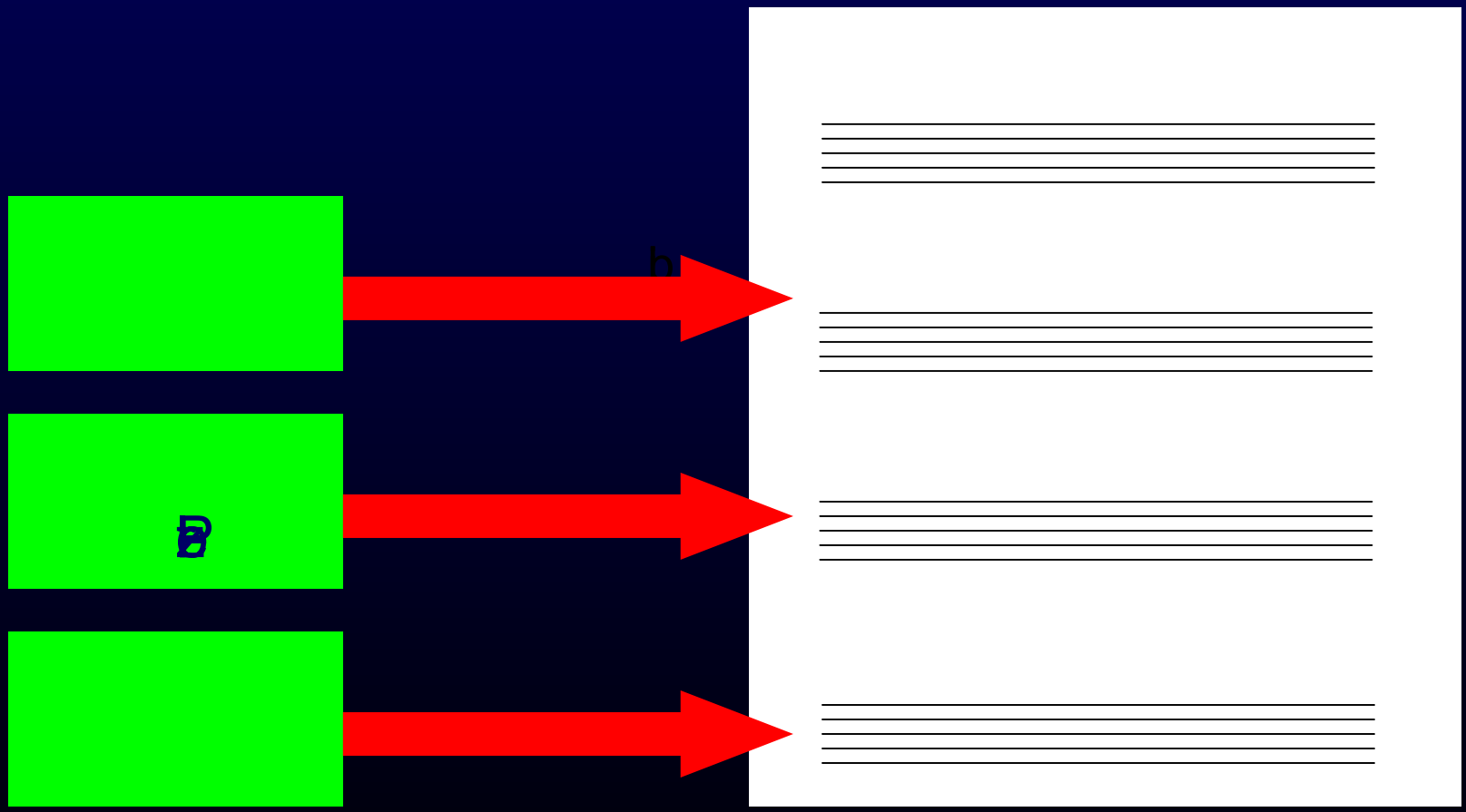
Automating Planning

Task	Partially Automatable	Automated Today
Risk Analysis / Prioritization	YES	Very Rarely
Required Resources • Engineers • Time • Computing Resources	YES	Partially (PERT Charts)
Test Plan Writing	YES	Very Rarely

Automation Example

Automated Test Effort Estimator					
Module	Number of Functions	Test Data Factor	Complexity Score	Time Estimate (hrs) (**50%** branch cov.)	Time Estimate (hrs) (**80%** branch cov.)
A	13	53	234	48	161
B	34	23	120	9	27
C	3	5	20	1	2
D	23	98	60	7	23
Total	**73**	**176**	**434**	**65 hrs 2.1 weeks**	**213 hrs 7.1 weeks**

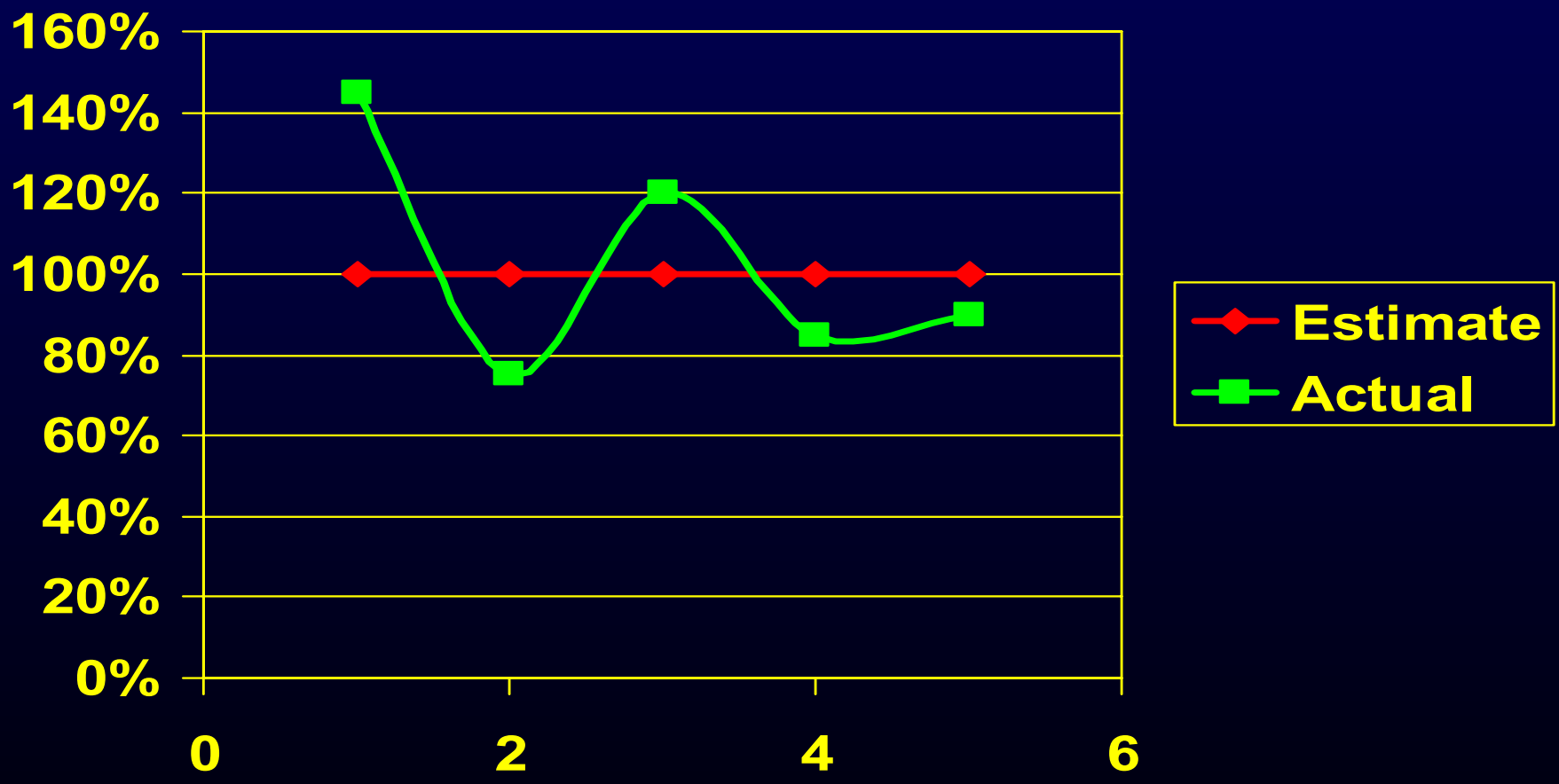
Test Plan Automation



Automating Priority and Planning

- Caveats
  - It takes a few cycles to calibrate this type of automation tools
    - E.g. mapping code-complexity and test data requirements into engineer hours
  - But at least you are keeping record of your educated guesses
  - Ad-Hoc → Methodical
- Automation is not ALWAYS the best path
  - Make sure you do a quick analysis on the automation ROI

Getting Better At It



Automating Test Development

- W
  a
  m a
  ?
  - e
    s

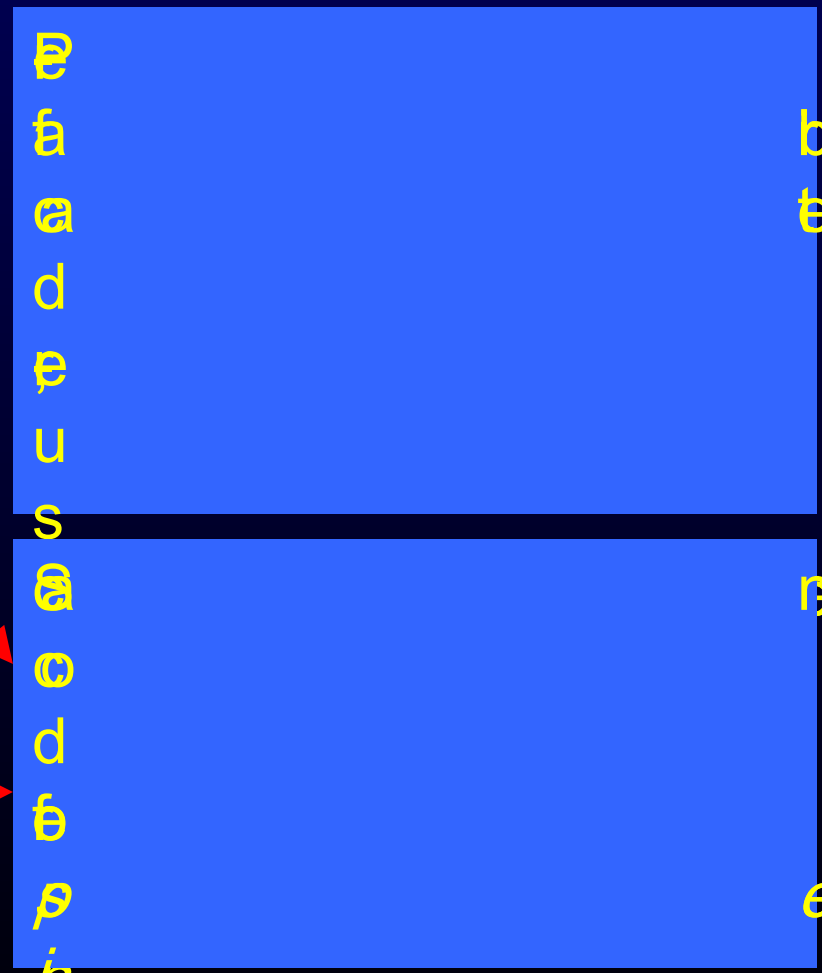
n
t
p

Program Understanding?

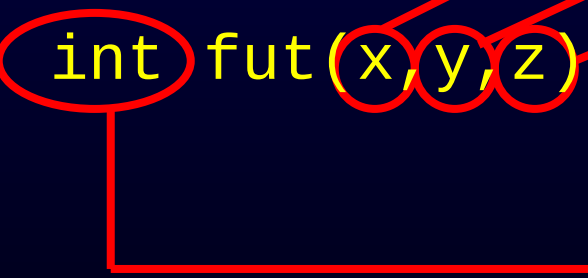
- Full program understanding is, ahem, a *few* years away
- Partial program understanding for testing:
  - programmers leave *clues* that can come in very handy for *testing*

Test Data Selection

```
int fut(int x) {
…
if (x >= 42) {
   // do something
} else {
   // do something else
}
…
   int y = 3;
…
```

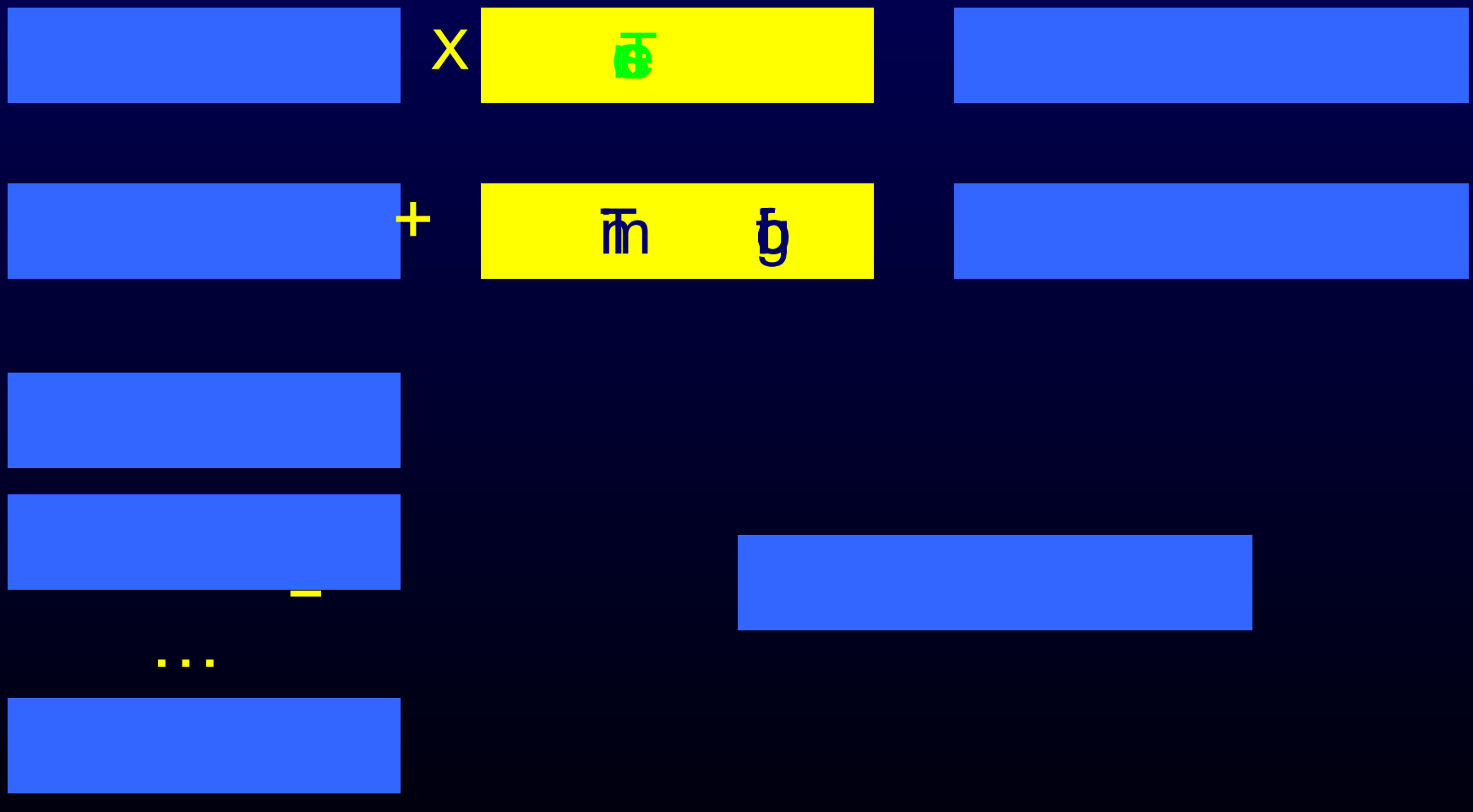


Template Code




```
…
for (test values for x)
 for (test values for y)
  for (test values for z) {
    int result;
    result = fut(x,y,z);
    if result == expected
      pass();
    else
      fail();
    }
…
}
```

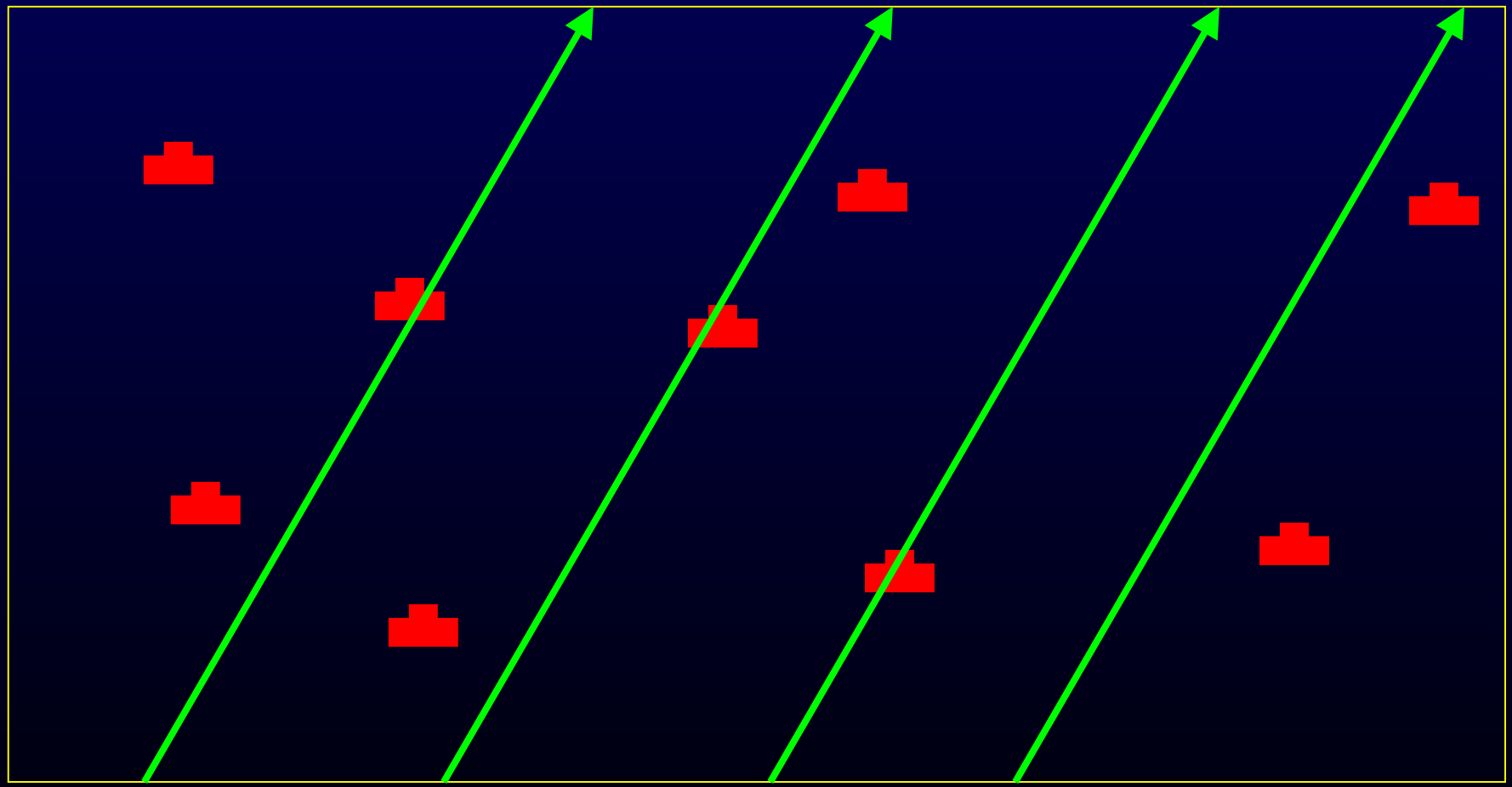
Test Amplification



Beyond Regression Testing

- Regression testing == automated re-execution of previously developed tests
- Regression testing's key limitation
  - Keeps going over the same ground
- Opportunity for for further automation
  - Regression test mutation

Regression Testing

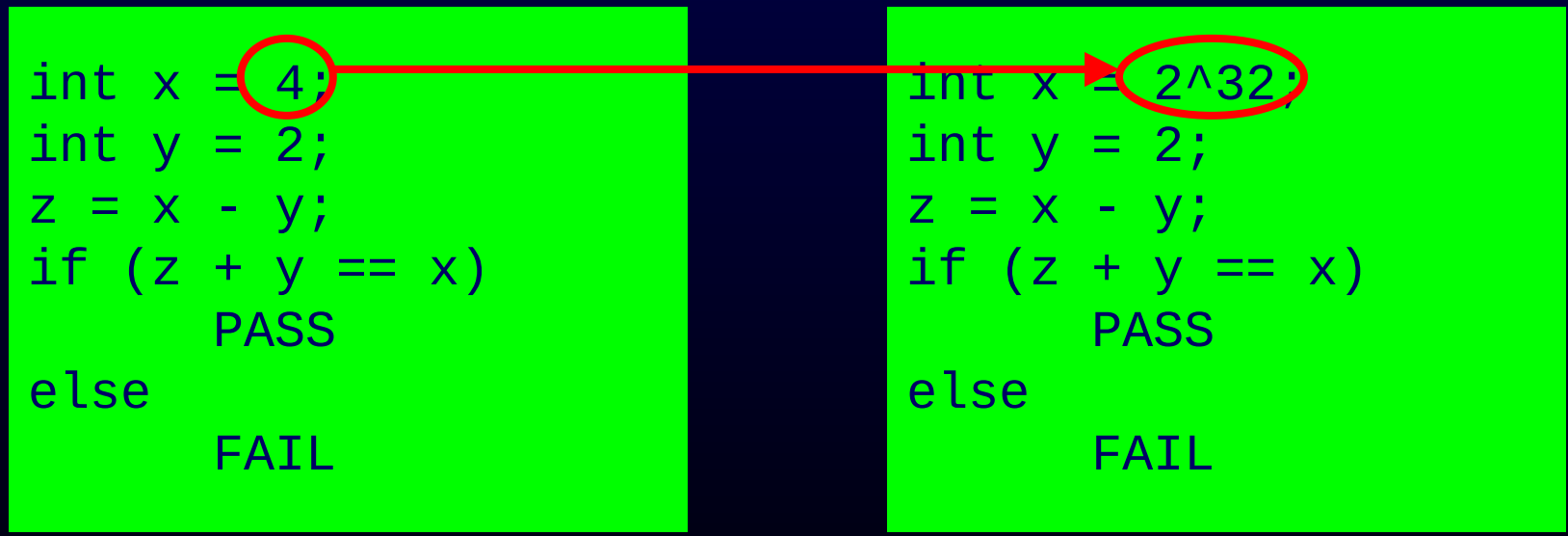


Source: James Bach

Regression Test Mutation

- Take an existing regression test
- Automatically make small modifications to generate tests that cover new ground

Trivial example:



Mutating Automated Tests
Douglas Hoffman, STAR 2000

The Future of Test Automation

- Most of the ideas and technology we presented are not new
- The biggest opportunity for the future of test automation is to make more use of the existing technology
- This will require:
  - Kick in the pants of test automation companies
  - Kick in the pants of professional software testers

Test Automation and Your Career

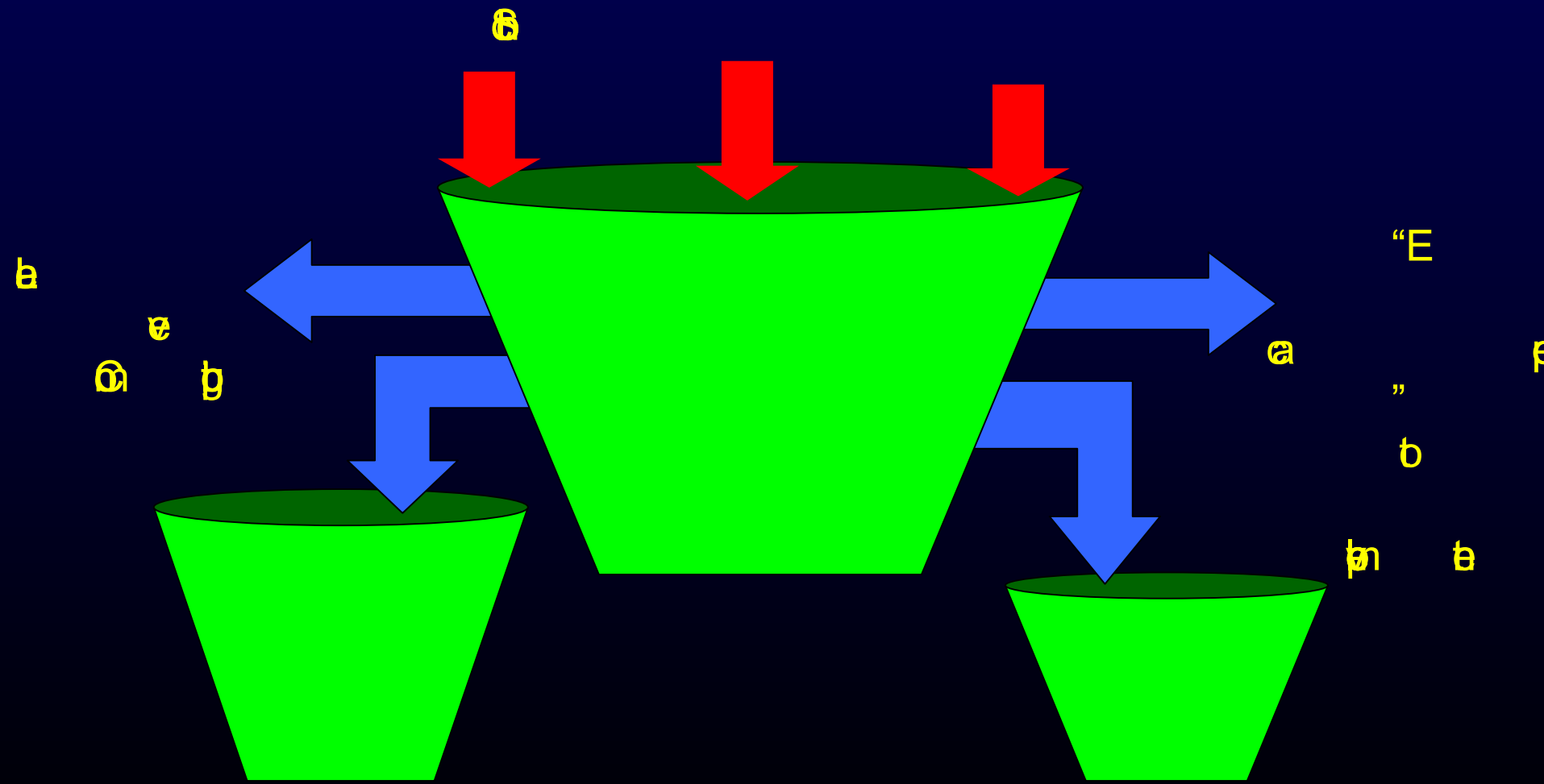
Recent Trends

- From waterfall to iterative software development models
- eXtreme Programming, Agile Development
  - Good
    - More emphasis on testing (esp. unit)
    - Developer testing
    - Testing gets new respect from developers
  - Bad
    - Less/no specifications
    - Frequent changes to functionality
- Implications for testers
  - Think on your feet, leverage automation as much as possible

SW Testing as a Career

- SW testing is a fantastic career choice:
  - As challenging, or more challenging, than development
  - Much less competition than development
  - Huge need, and opportunities for revolutionary new ideas, tools, approaches
  - The market will always be hungry for great software testers
  - If you get very good at test automation, you can get a job in any economy

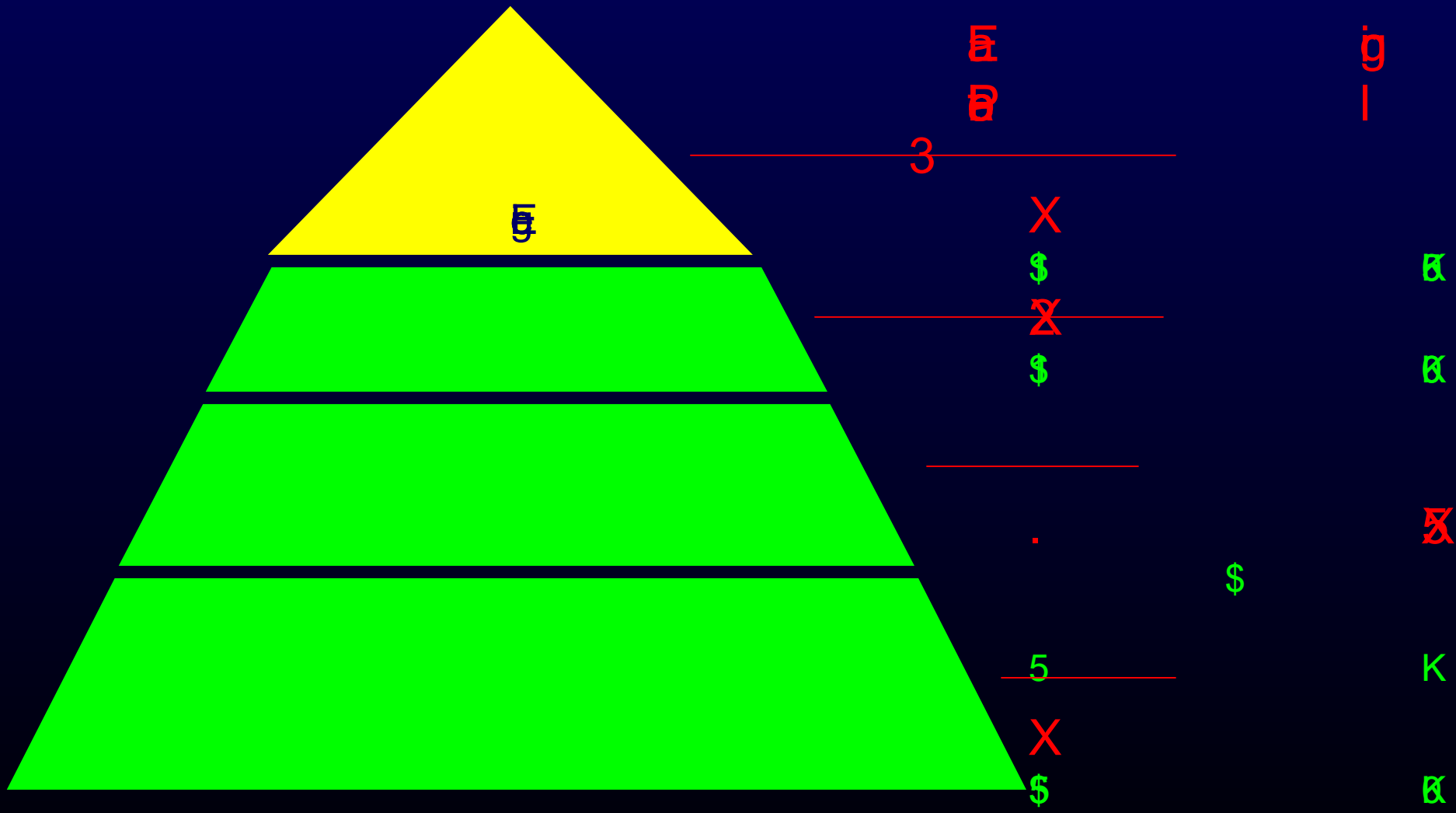
The 3 Buckets of Testers



Testing Pros Categories

- Technical (Non-managers)
  - Individual Test Developer
    - Leverage/Impact = 1X
  - Test Automation Engineer
    - Leverage/Impact = 10X+
- Managerial
  - Tech lead
  - Manager
  - Director
  - VP of Quality
- In each position, your value to the organization will increase significantly if you automate intelligently

The Testing Career Ladder (Technical)



Automation Pays Off

Conclusion

It's very hard to predict, especially the future

Niels Bohr

- To maximize testing efficiency you need to optimize simplicity, reuse, and automation
- We have only scratched the surface in these areas, especially automation
- There are ***many*** opportunities for simplifying, re-using, and automating all sorts of testing tasks
- Don't think *all or nothing*, automate what you can
- Ability to automate → value to organization → great career

Contact Info + Q&A

I love to talk about test simplification, reuse, and automation. If you have questions, or ideas you want to discuss, please contact me:

alberto@TestAgility.com

Q&A

How Testers Think

James Bach, Satisfice Inc.

Abstract

The role of a tester is to penetrate illusions and bring vital information about the product to light. To do that requires not only that we create and execute test cases, but also that we think differently from our non-testing co-workers. This talk examines some of the dynamics of thinking like a tester and some of the challenges of developing a tester mind.

***James Bach** (**http://www.satisfice.com**) spends his time consulting and teaching software quality assurance and testing. He has worked at Apple Computer and other Silicon Valley companies as a programmer, tester, and test manager, and is the author (with Kaner and Pettichord) of Lessons Learned in Software Testing.*

How Testers Think

James Bach, Satisfice, Inc.
james@satisfice.com
www.satisfice.com

Too many textbooks treat testers as clerical wind-up toys.

"This model identifies a **standards-based** life cycle testing process that concentrates on developing **formal test documentation** to implement **repeatable structured testing** on a software or hardware/software system. The general intent is that the test documentation be developed based on a **formal requirements specification** document...**Once the documentation is developed, the test is executed**."

-- from a real article about testing.
(*I added the boldfacing to emphasize **instructions***)

Where's the *thinking* in this picture?

Testers light the way.

This is our role.

We make informed decisions about quality possible.
Because we think critically about software.

By our *thinking*, we can compensate for a difficult project environment.

Instead of this…	consider this.
– complete specs	– implicit specs & inference
– quantifiable criteria	– meaningful criteria
– protected schedule	– risk-driven iterations
– early involvement	– good working relationship
– zero defect philosophy	– good enough quality
– complete test coverage	– enough information

Testing is about questions; Posing them and answering them.

- **Product**
  - What is this product?
  - What can I control and observe?
  - What should I test?
- **Tests**
  - What would constitute a diversified and practical test strategy?
  - How can I improve my understanding of how well or poorly this product works?
  - If there were an important problem here, how would I uncover it?
  - What document to load? Which button to push? What number to enter?
  - How powerful is this test?
  - What have I learned from this test that helps me perform powerful new tests?
  - What just happened? How do I examine that more closely?
- **Problems**
  - What quality criteria matter?
  - What kinds of problems might I find in this product?
  - Is what I see, here, a problem? If so, why?
  - How important is this problem? Why should it be fixed?

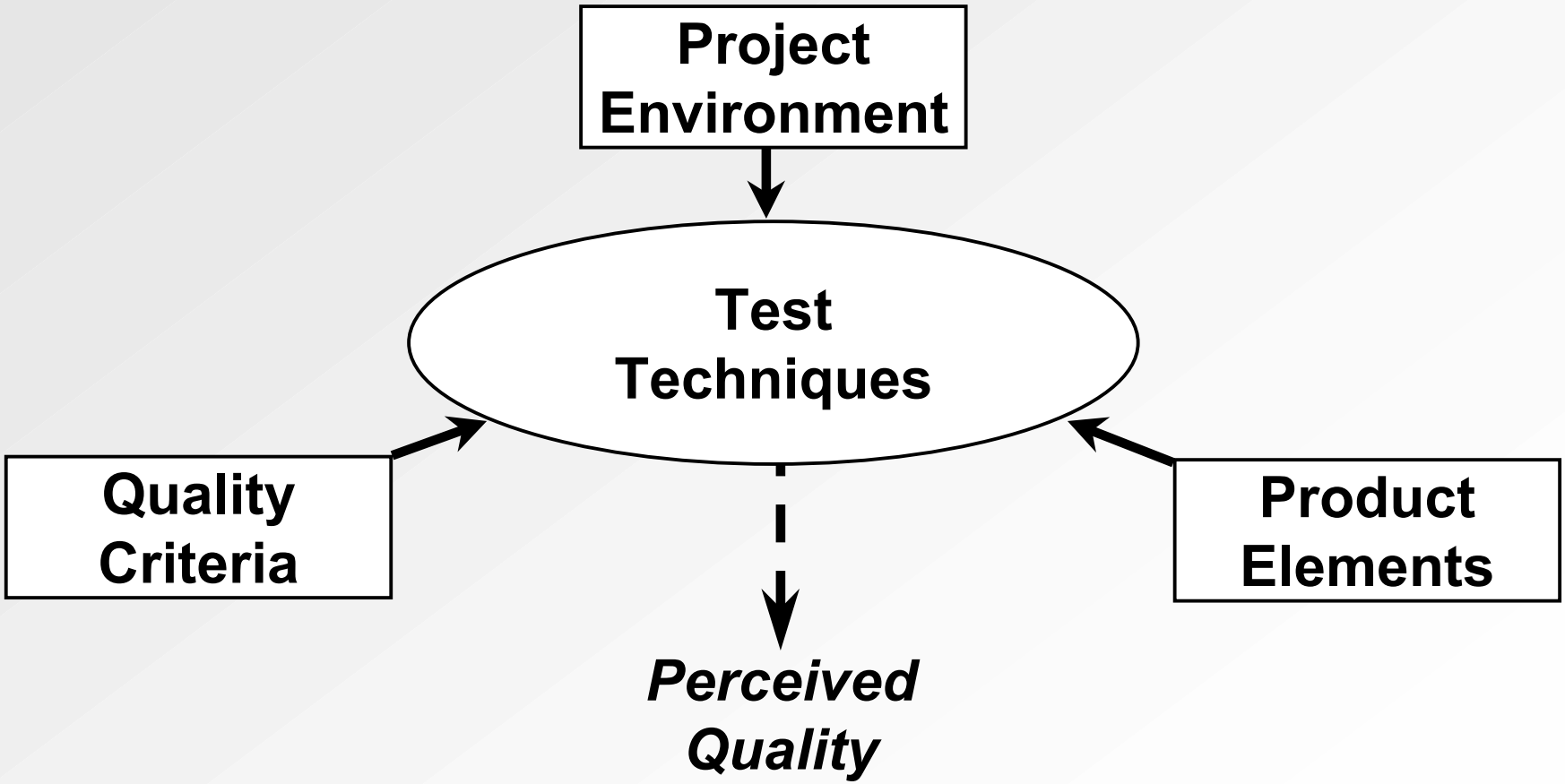
Testing is about ideas. Heuristics give you ideas.

- A heuristic is a *fallible* idea or method that may help solve a problem.
- You don't *comply* to a heuristic, you apply it. Heuristics can hurt you when elevated to the status of authoritative rules.
- Heuristics represent wise behavior only in context. They do not *contain* wisdom.
- Your relationship to a heuristic is the key to applying it wisely.

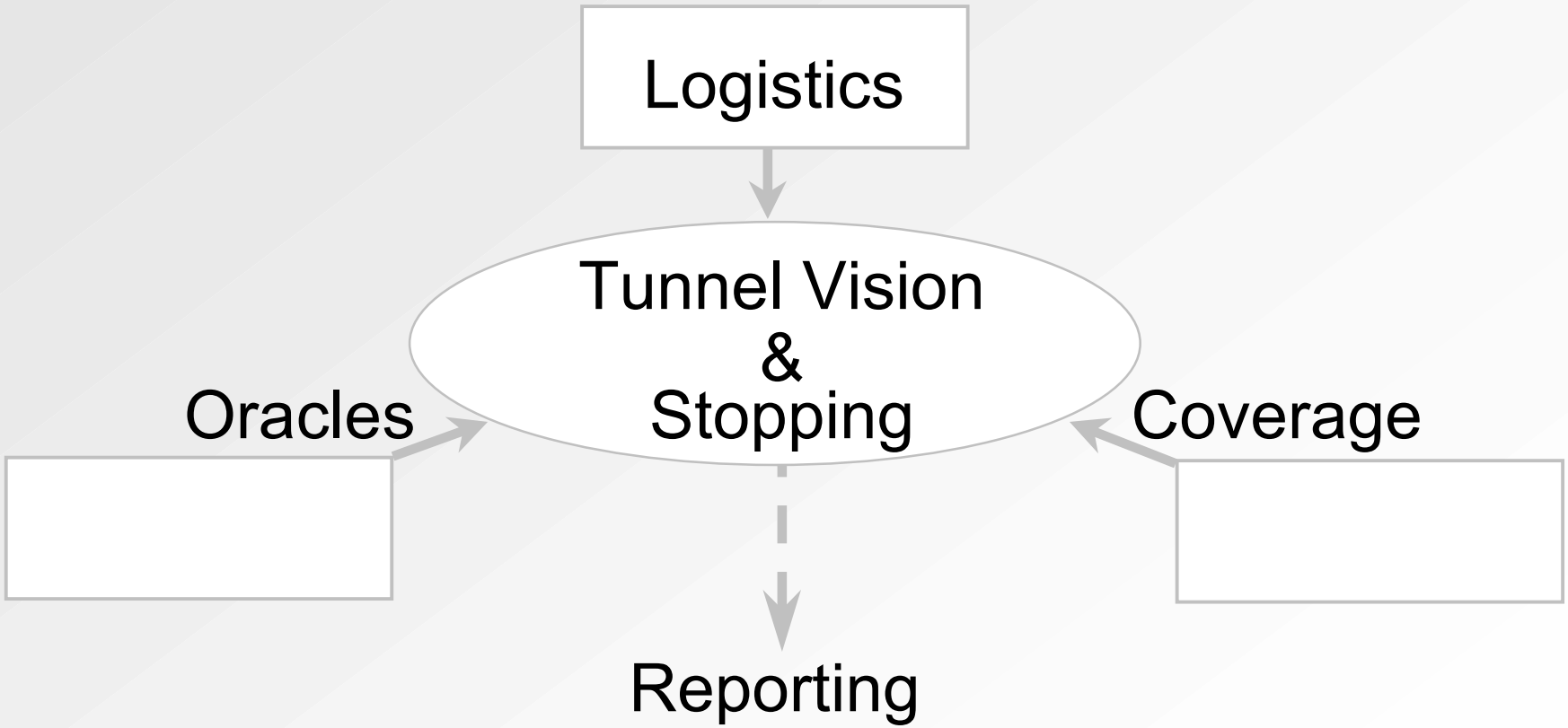
"Heuristic reasoning is not regarded as final and strict but as provisional and plausible only, whose purpose is to discover the solution to the present problem."

- George Polya, *How to Solve It*

Heuristic Model for Test Design



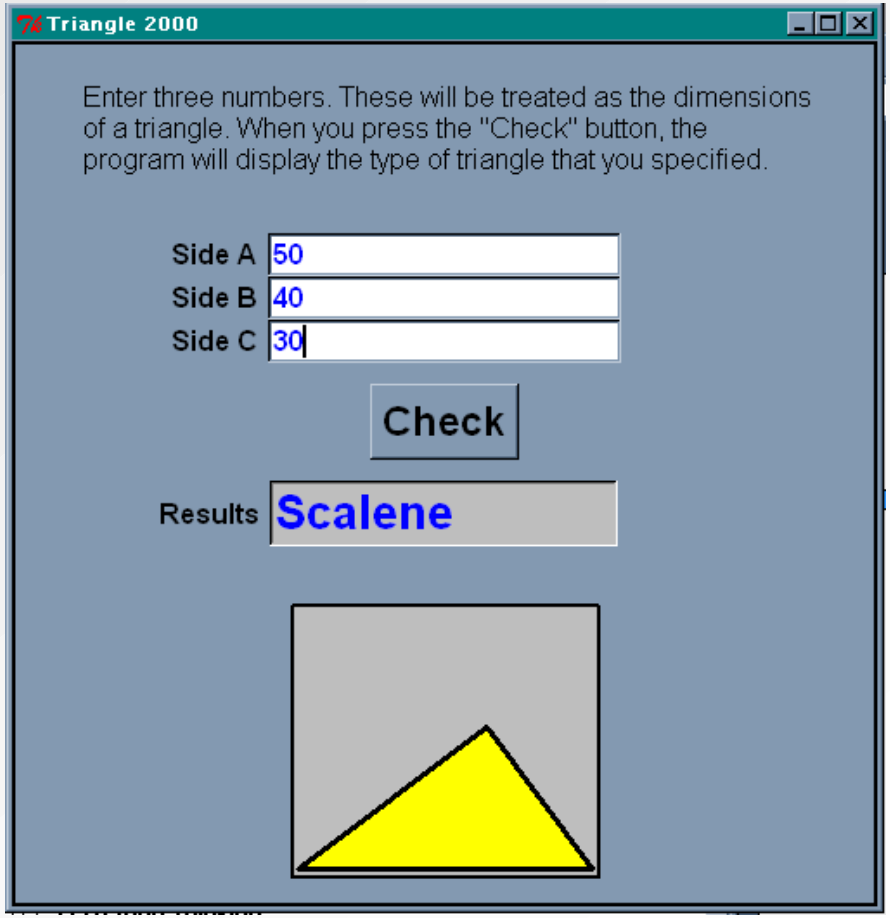
Six Problems of Testing



Cloistered Coverage: The *Triangle* Program

Students in Satisfice's *Rapid Software Testing* class are given 20 minutes to test this program.

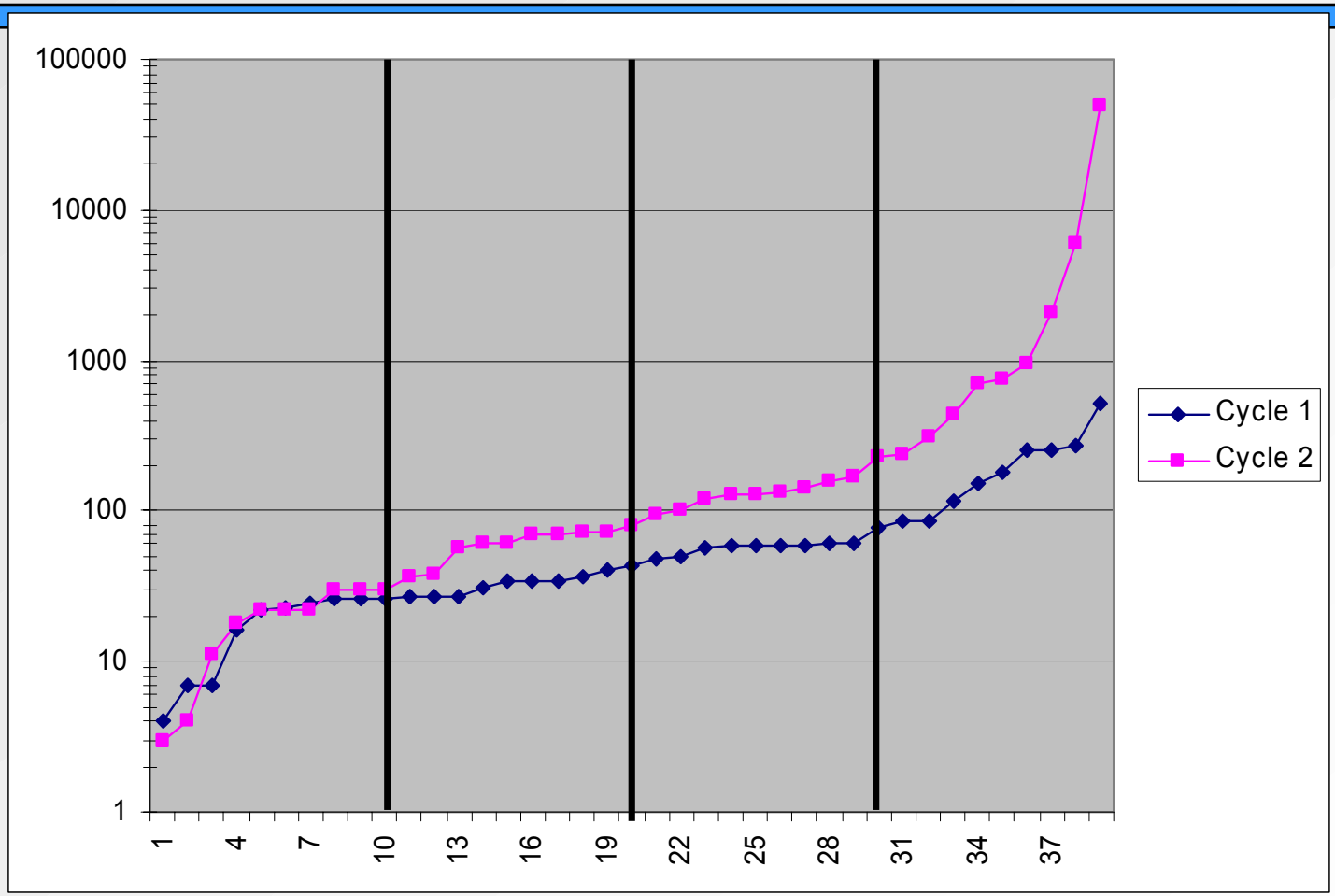
We see interesting differences in how testers approach this task.



How Well Does Triangle Handle Long Inputs?

- What does “long” mean?
- What does “handle well” mean?
- What will users do? What will they expect?
- So what?

Field Lengths Chosen by 39 Testers, Over Two Cycles



Interesting Lengths

- **16** digits & up: loss of mathematical precision.
- **23** digits & up: can't see all of the input.
- **310** digits & up: input not understood as a number.
- **1,000** digits & up: exponentially increasing freeze when navigating to the end of the field by pressing <END>.
- **23,829** digits & up: all text in field turns white.
- **2,400,000** digits: crash (reproducible).

Most of these are unknown to the programmer!

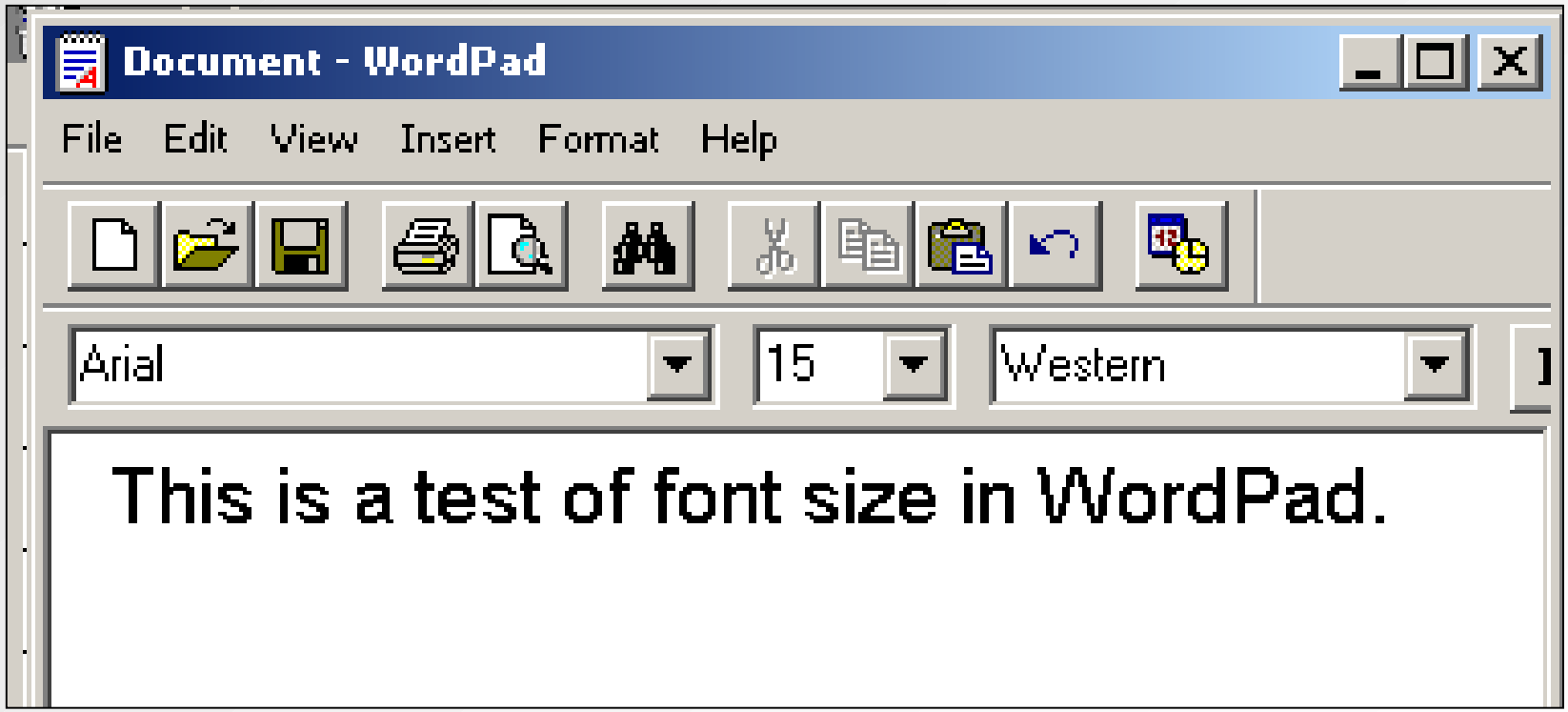
What stops testers from trying longer inputs?

- Seduced by what's visible.
- Think they need a spec that tells them the max.
- If they have a spec, they stop when the spec says stop.
- Satisfied by the first boundary (16 digits).
- Let their fingers do the walking instead of using a program like notepad to generate input.
- Use strictly linear lengthening strategy.
- Don't realize the significance of degradation.
- Assume it will be too hard and take too long.
- Think "No one would do that" (hackers do it)

Thinking About Coverage

- Testers with less expertise…
  - Think about coverage mostly in terms of what they can see.
  - Cover the product indiscriminately.
  - Avoid questions about the completeness of their testing.
  - Can’t reason about how much testing is enough.
- Better testers are more likely to…
  - Think about coverage in many dimensions.
  - Maximize diversity of tests while focusing on areas of risk.
  - Invite questions about the completeness of their testing.
  - Lead discussions on what testing is needed.

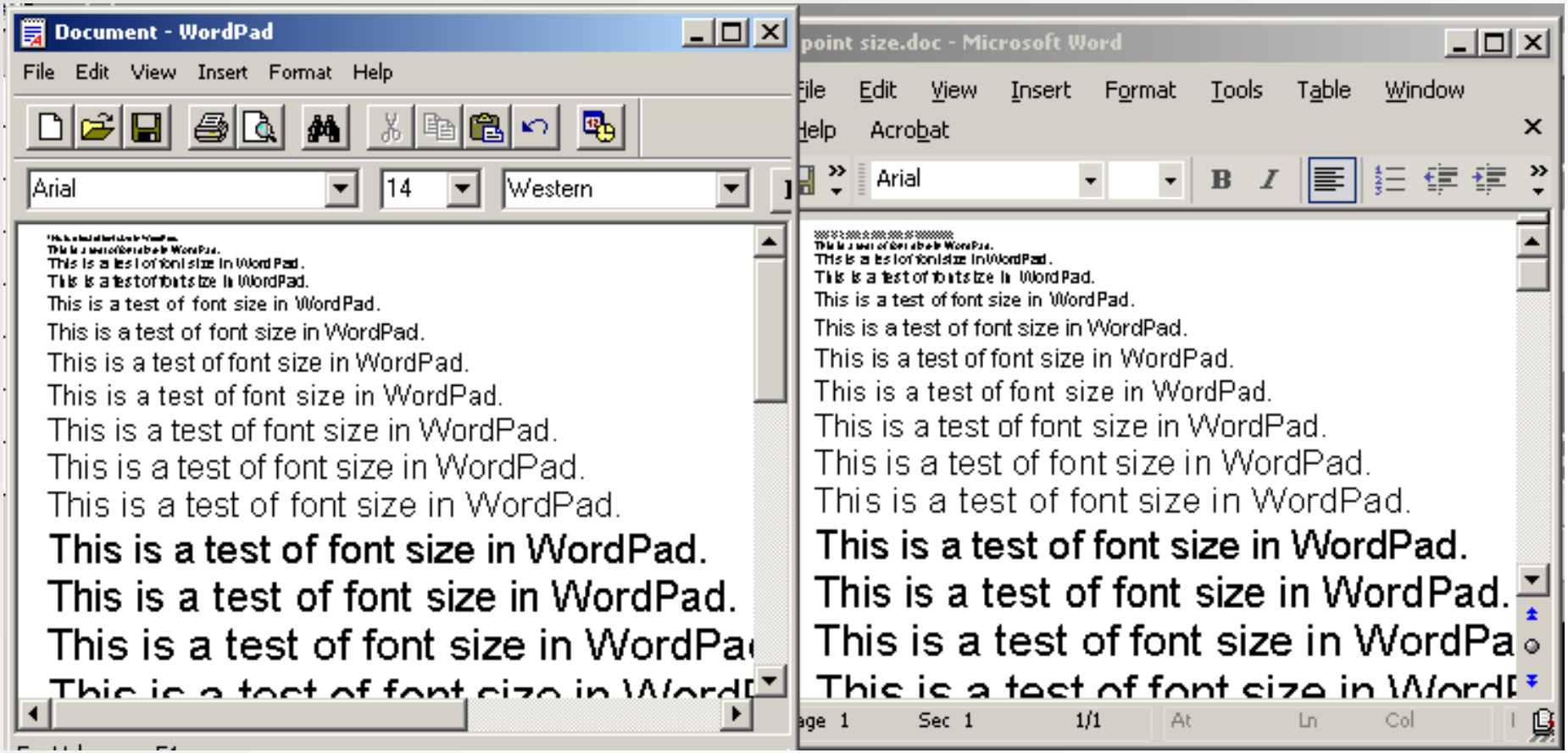
Oblivious Oracles:
Does font size work in WordPad?



Is this the right font size? What's your oracle?

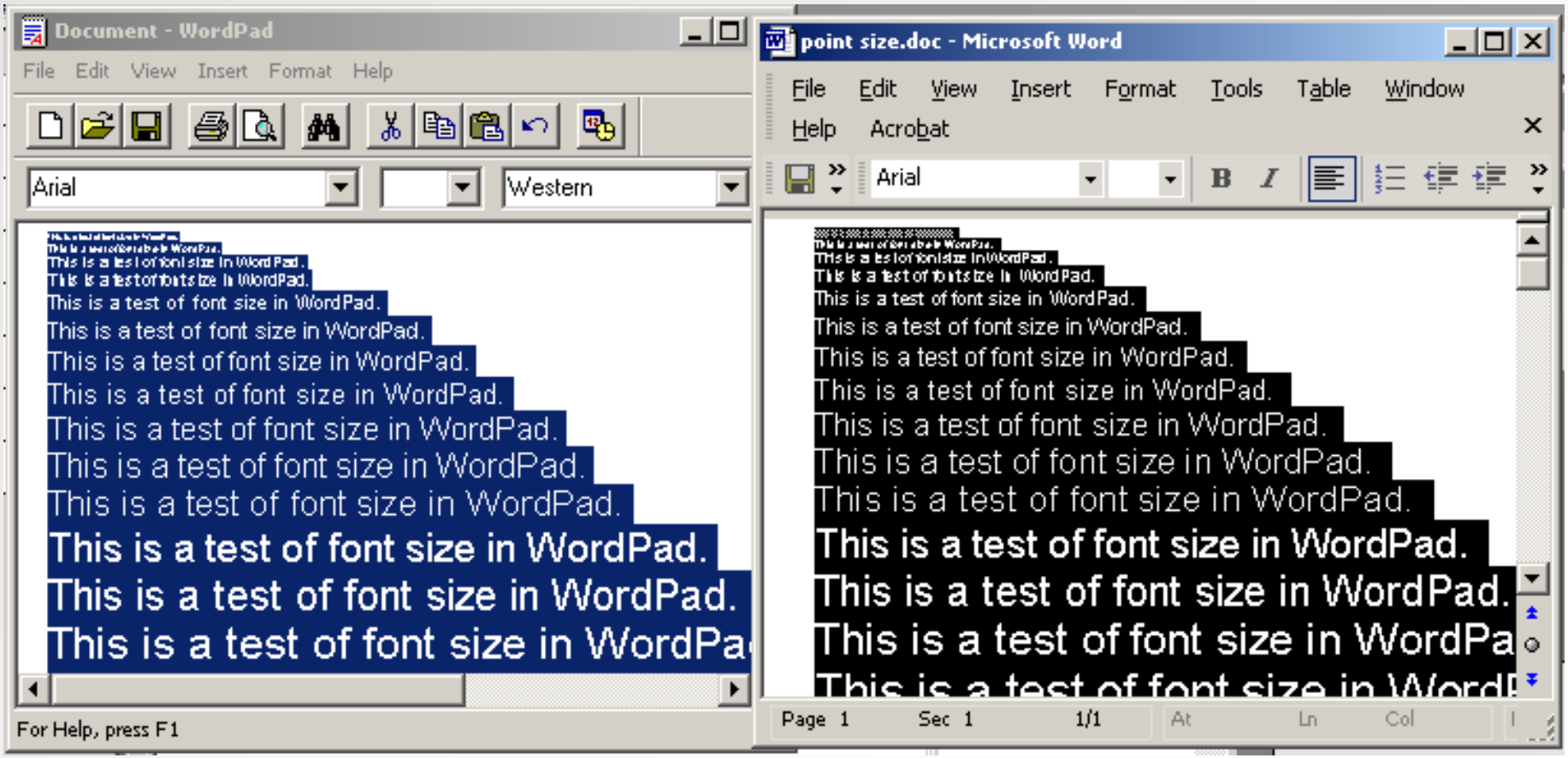
The Comparable Product Heuristic will save us! Or will it…?

WordPad **Word**



What does this tell us?

WordPad **Word**



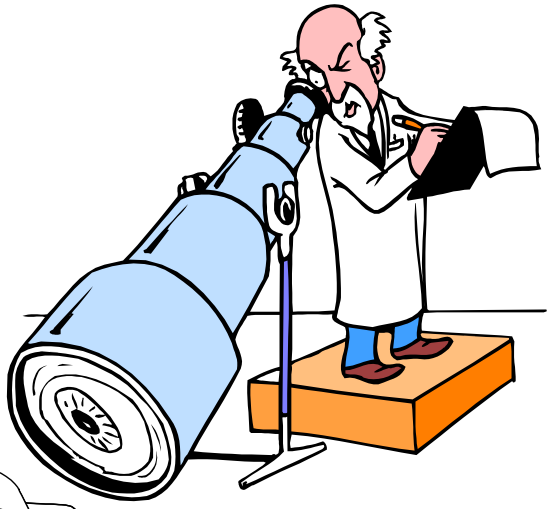
Oracle Heuristics: "HICCUPP"

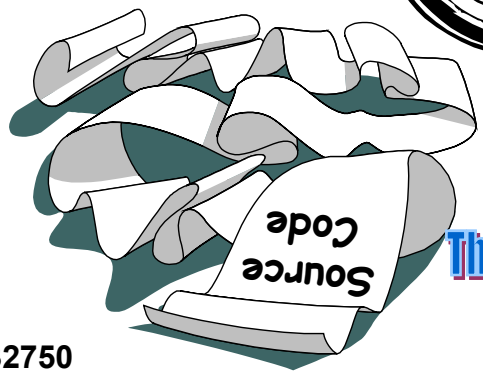
- **Consistent with History:** Present function behavior is consistent with past behavior.
- **Consistent with our Image:** Function behavior is consistent with an image that the organization wants to project.
- **Consistent with Comparable Products:** Function behavior is consistent with that of similar functions in comparable products.
- **Consistent with Claims:** Function behavior is consistent with what people say it's supposed to be.
- **Consistent with User's Expectations:** Function behavior is consistent with what we think users want.
- **Consistent within Product:** Function behavior is consistent with behavior of comparable functions or functional patterns within the product.
- **Consistent with Purpose:** Function behavior is consistent with apparent purpose.

Better Thinking

- *Conjecture and Refutation*: reasoning without certainty.
- *Abductive Inference*: finding the best explanation among alternatives.
- *Lateral Thinking:* the art of being distractible.
- *Forward-backward thinking*: connecting your observations to your imagination.
- *Heuristics:* applying helpful problem-solving short cuts.
- *De-biasing:* managing unhelpful short cuts.
- *Pairing:* two testers, one computer.
- *Study other fields.* Example: Information Theory.

Basis Path Testing





The Westfall Team

Theresa Hunt
1660 Barton St.
Longwood, Fl 32750
phone: (407) 834-5825
fax: (407) 834-2735
Theresahunt@earthlink.net
www.westfallteam.com

The Challenge of Testing

Slide **The Challenge of Testing**

A major challenge in testing is to determine a good starting set of test cases that:

- Eliminate redundant testing
- Provide adequate test coverage
- Allow more effective testing
- Make the most of limited testing resources

Too Many Paths

There are typically many possible paths between the entry and exit of a typical software program. Every decision doubles the number of potential paths, every case statement multiplies the number of potential paths by the number of cases and every loop multiplies the number of potential paths by the number of different iteration values possible for the loop. [based on Beizer-90]

Complete path coverage of even a simple unit is extremely difficult. For example, software that includes straight line code except for a single loop that can be executed from 1 to 100 times would have 100 paths.

The Challenge

Given that testing, like all other development activities has a limited amount of resources (time, people, equipment), the challenge is to select a set of test cases that is most likely to identify as many different potential defects as possible within those limits. To do this, we must eliminate as much redundancy as possible from the testing process while still insuring adequate test coverage.

Basis Path Testing Defined

Slide **What is Basis Path Testing?**

Basis path testing is a hybrid between path testing and branch testing:

Path Testing: Testing designed to execute all or selected paths through a computer program [IEEE610]

Branch Testing: Testing designed to execute each outcome of each decision point in a computer program [IEEE610]

Basis Path Testing: Testing that fulfills the requirements of branch testing & also tests all of the independent paths that could be used to construct any arbitrary path through the computer program [based on NIST]

Path A path through the software is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit. A path may go through several junctions, processes, or decisions, one or more times. [Beizer-90]

Independent Path An independent path is any path through the software that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. [Pressman-01]

Basis Path Testing Defined (cont.)

Slide **What is a Basis Path?**

A *basis path* is a unique path through the software where no iterations are allowed - all possible paths through the system are linear combinations of them.

Slide **McCabe's Basis Path Testing**

Steps:

1: Draw a control flow graph

2: Calculate Cyclomatic complexity

3: Choose a "basis set" of paths

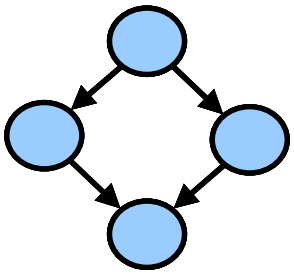
4: Generate test cases to exercise each path

Step 1: Draw a Control Flow Graph

Slide **The Control Flow Graph**

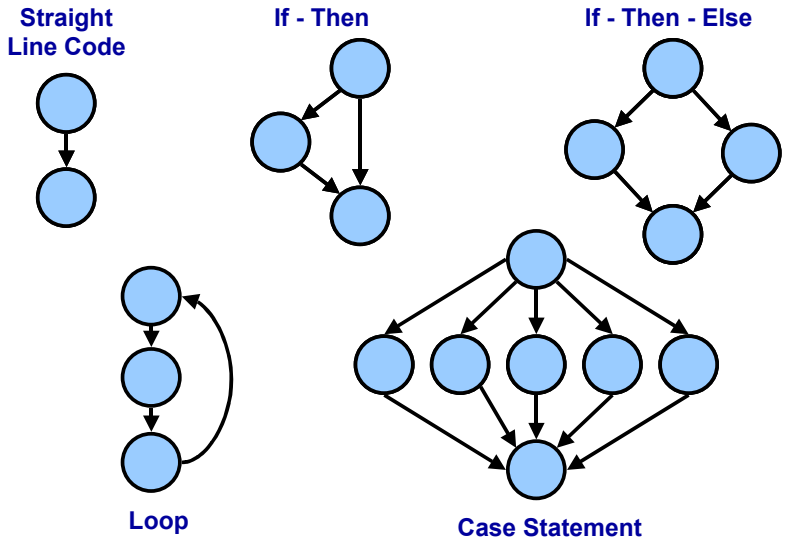
Any procedural design can be translated into a control flow graph:

- Lines (or arrows) called *edges* represent flow of control
- Circles called *nodes* represent one or more actions
- Areas bounded by edges and nodes called *regions*
- A *predicate node* is a node containing a condition



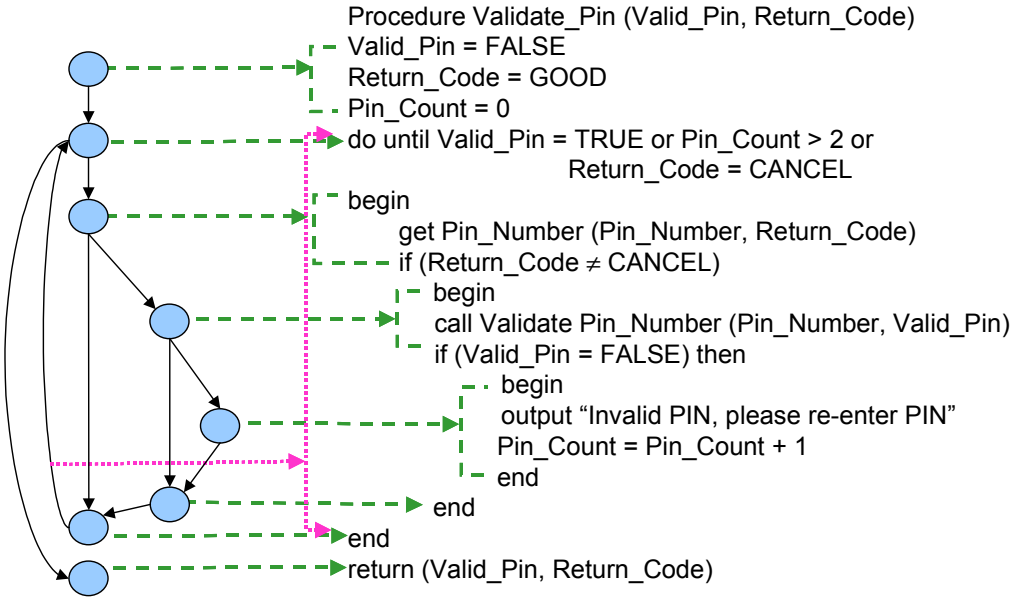
Slide **Control Flow Graph Structures**

Basic control flow graph structures:

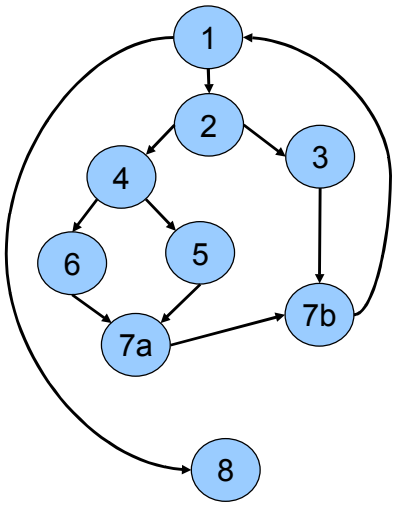


Step 1: Draw a Control Flow Graph (cont.)

Slide **Draw a Control Flow Graph - Example**



Slide **Another Example**




```
1.      do while records remain
            read record;
2.          if record field 1 = 0
3.              then process record;
                    store in buffer;
                    increment counter;
4.          elsif record field 2 = 0
5.              then reset record;
6.              else process record;
                    store in file;
7a.         endif;
        endif;
7b.     enddo;
8.  end;
```

[based on Sobey]

Step 1 The first step in basis path testing is to draw the control flow graph. As illustrated in the examples above, this can be done directly from the source code.

Step 2: Calculate Cyclomatic Complexity

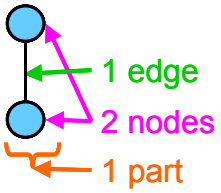
Slide

Step 2: Calculate Cyclomatic Complexity

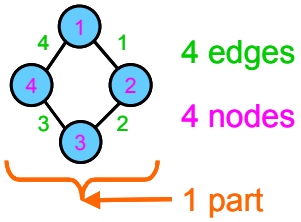
Control flow graph:

Model: V(G) = edges - nodes + 2p

where p = number of unconnected parts of the graph



V(G) = 1 - 2 + 2x1 = 1
Straight line code always has a complexity of 1



V(G) = 4 - 4 + 2x1 = 2

Cyclomatic Complexity

The second step in basis path testing is to calculate the Cyclomatic complexity from the control flow graph. McCabe's Cyclomatic Complexity is a measure of the number of linearly independent paths through the unit/component. It can therefore be used in structural testing to determine the minimum number of tests that must be executed for complete basis path coverage.

Calculation

Cyclomatic Complexity is calculated from a control flow graph by subtracting the number of nodes from the number of edges and adding 2 times the number of unconnected parts of the graph. For example, straight-line code (first control flow graph above) has 1 edge and 2 nodes so its complexity is 1-2+2*1 = 1. This second graph has 4 edges and 4 nodes so its complexity is 4-4+2*1 = 2. What is the complexity of the other two graphs?

Tools

Static analysis tools also exist for drawing control flow graphs and/or calculating Cyclomatic complexity from source code.

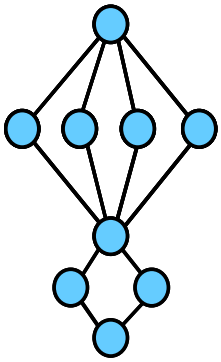
Step 2: Calculate Cyclomatic Complexity (cont.)

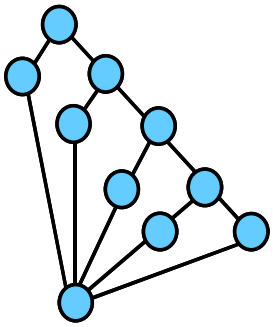
Slide **Cyclomatic Complexity - Exercise -**

Control flow graph:

Model: V(G) = edges - nodes + 2p

where p = number of unconnected parts of the graph





Instructions: Calculate the Cyclomatic complexity separately for each of the two control flow graphs above.

Cyclomatic complexity of first graph = _____

Cyclomatic complexity of second graph = _____

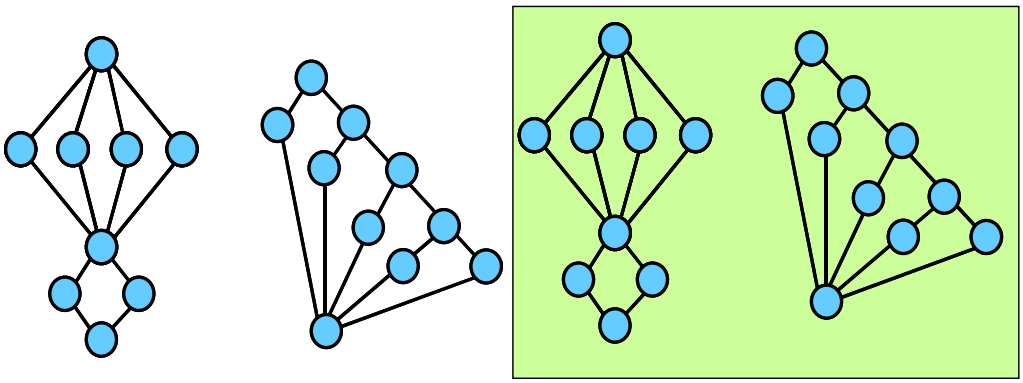
Then assuming that these two control flow graphs are two unconnected parts of the same graph, calculate the Cyclomatic complexity of the combined graph.

Cyclomatic complexity of combined graph = _____

Step 2: Calculate Cyclomatic Complexity (cont.)

Slide **Answer to Exercise**

V(G) = edges - nodes + 2p



12 edges
9 nodes
12-9+2(1)=5

13 edges
10 nodes
13-10+2(1)=5

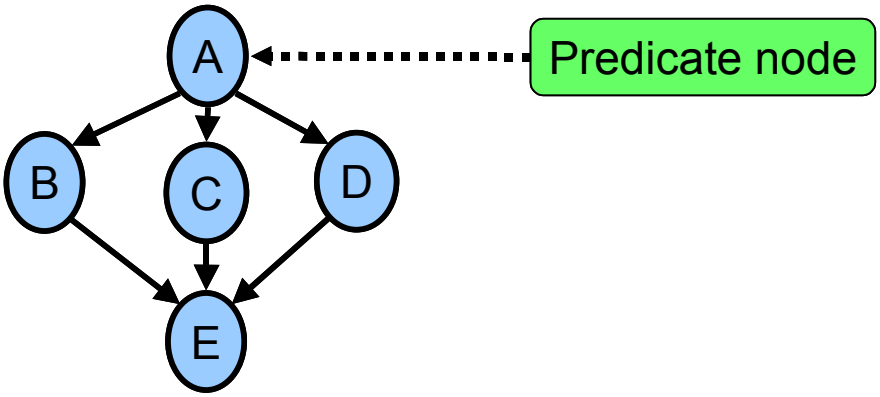
25 edges
19 nodes
25-19+2(2)=10

Step 2: Calculate Cyclomatic Complexity (cont.)

Slides **Predicate Nodes Formula**

One possible way of calculating V(G) is to use the predicate nodes formula:

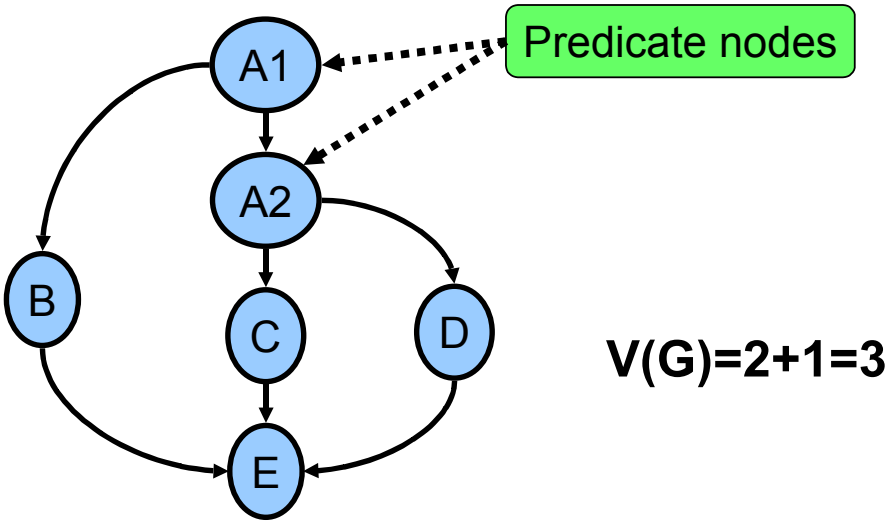
V(G) = Number of Predicate Nodes + 1



Thus for this particular graph: V(G)=1+1=2

BUT**……..** V(G)=6-5+2(1)=3……..

A limitation on the predicate nodes formula is that it assumes that there are only two outgoing flows for each of such nodes. To adjust for this, split the predicate node into two sub-nodes:



V(G)=2+1=3

[based on Chek]

Step 2: Calculate Cyclomatic Complexity (cont.)

Slide

Uses of Cyclomatic Complexity

Cyclomatic complexity can be used to:

- Tell us how many paths to look for in basis path testing
- Help pinpoint areas of potential instability
- Indicate a unit/component's testability & understandability (maintainability)
- Provide a quantitative indication of unit/component's control flow complexity
- Indicate the effort required to test a unit/component

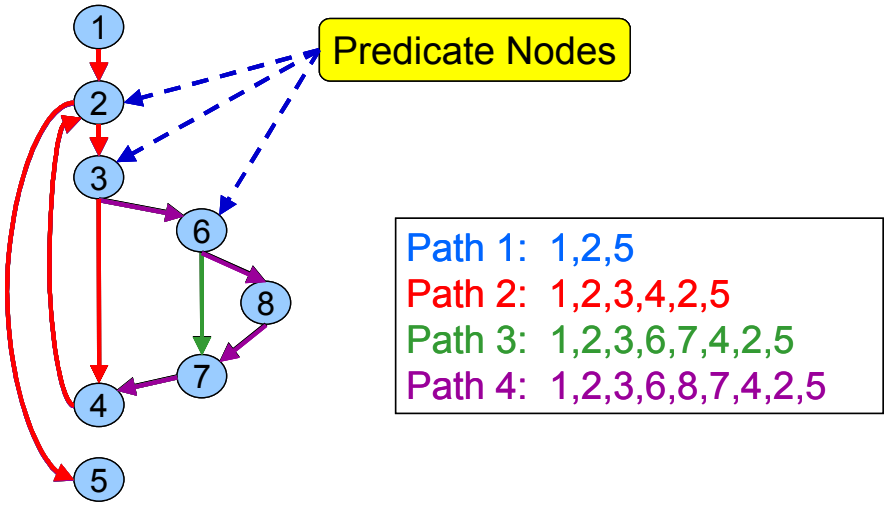
Use

Based on the unit/component's Cyclomatic complexity we can apply appropriate levels of effort on detailed inspections and testing. Those units/components having a high complexity measure need more intense inspection and test, those with lower measures are likely to have fewer defects and thus would require less stringent inspection and test. The Cyclomatic complexity also tells us how many paths to evaluate for basis path testing

Step 3: Choose a Basis Set of Paths

Slide **Step 3: Choose a Basis Set of Paths**

Using a control flow graph:



Step 1 Draw the control flow graph – for this example we will use the control flow graph we drew in the first code example.

Step 2 Calculate the Cyclomatic complexity – Remember there are actually three ways to calculate the Cyclomatic complexity of a control flow graph.

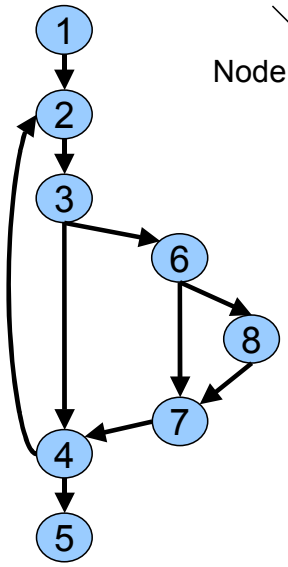
1. V(G) = edges – nodes + 2p. For this example there are 10 edges, 8 nodes and p is 1, so V(G) = 10 – 8 + 2 = 4
2. V(G) = number of regions in the control flow graph. For this example there are 3 enclosed regions plus the outside region, so V(G) = 4.
3. V(G) = number of predicate nodes + 1. A predicate node is a node with more than one edge emanating from it. For this example, nodes 2, 3 and 6 are predicate nodes, so V(G) = 3 + 1 = 4.

Step 3 Choose a set of basis paths – Determining the predicate nodes can help identify a set of basis paths. If test cases can be designed to cover the basis path set, it will result in complete decision (and statement) coverage of the code. Each basis path that you select must in effect test at least one new untested edge, in other words it must traverse at least one new edge. Otherwise it is considered a redundant path and does not belong in the basis set. This aids in eliminating redundant testing and ensures validity of each test case. For each subsequent basis path selected try to keep the number of new edges added as low as possible – progressing on until you have covered all possible basis paths.

Step 3: Choose a Basis Set of Paths (cont.)

Slide **Step 3: Choose a Basis Set of Paths (cont.)**

The Graph (Connection) Matrix



Node	Connected to node 1	2	3	4	5	6	7	8	
1	0	1	0	0	0	0	0	0	1 – 1 = 0
2	0	0	1	0	0	0	0	0	1 – 1 = 0
3	0	0	0	1	0	1	0	0	2 - 1 = 1
4	0	1	0	0	1	0	0	0	2 – 1 = 1
5	0	0	0	0	0	0	0	0	---
6	0	0	0	0	0	0	1	1	2 – 1 = 1
7	0	0	0	1	0	0	0	0	1 – 1 = 0
8	0	0	0	0	0	0	1	0	1 – 1 = 0

3+1=4

Graph matrix A graph matrix, also called a connection matrix:

- Is square with #sides equal to #nodes
- Has rows and columns that correspond to the nodes
- Has non-zero value weights that correspond to the edges

Note For additional information refer to the National Bureau of Standards special publication "Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric. [McCabe-82]

Link Weight Can associate a number with each edge entry. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist).

Use a value of 1 (indicating that a connection exists) to calculate the Cyclomatic complexity:

- For each row, sum column values and subtract 1
- Sum these totals and add 1

Some other interesting link weights:

- Probability that a link (edge) will be executed
- Processing time for traversal of a link
- Memory required during traversal of a link
- Resources required during traversal of a link

Step 4: Generate Test Cases

Slide **Step 4: Generate Test Cases**

Generate test cases that will force execution of each path in the basis set - for examples:

Path 1: 1,2,5
Test case 1
Path 1 can not be tested stand-alone & must be tested as part of path 2, 3 or 4

Path 2: 1,2,3,4,2,5
Test case 2
Press cancel in response to the "Enter PIN Number" prompt

Path 3: 1,2,3,6,7,4,2,5
Test case 3
Enter a valid PIN number on the first try

Path 4: 1,2,3,6,8,7,4,2,5
Test case 4
Enter an invalid PIN on the first try & a valid PIN on the second try

---

Test Case 3 It should be noted that in order to follow path 4 this test case executes the loop a second time so the path is actually 1,2,3,6,8,7,4,2,3,6,7,4,2,5.

Basis Path Testing During Integration

Slide **Basis Path Testing During Integration**

Basis path testing can also be applied to integration testing when units/components are integrated together

McCabe's Design Predicate approach:

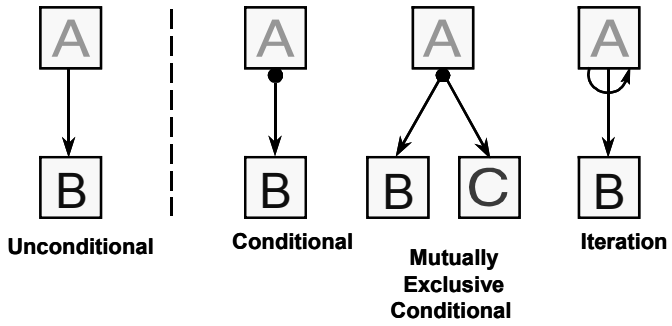
- Draw a "structure chart"
- Calculate integration complexity
- Select tests to exercise every "type" of interaction, not every combination

This approach can be used to predict the integration test effort before coding begins

---

Slide **Types of Interactions**

Types of unit/component interactions:



Unconditional Unit/component A always calls unit/component B. A calling tree always has an integration complexity of at least one. The integration complexity is NOT incremented for each occurrence of an unconditional interaction.

Conditional Unit/component A calls unit/component B only if certain conditions are met. The integration complexity is incremented by one for each occurrence of a conditional interaction in the structure chart.

Mutually Exclusive Unit/component A calls either unit/component B or unit/component C (but not both) based upon certain conditions being met. The integration complexity is incremented by one for each occurrence of a mutually exclusive conditional interaction in the structure chart.

Iterative Unit/component A calls unit/component B one or more times based upon certain conditions being met. The integration complexity is incremented by one for each occurrence of an iterative interaction in the structure chart.

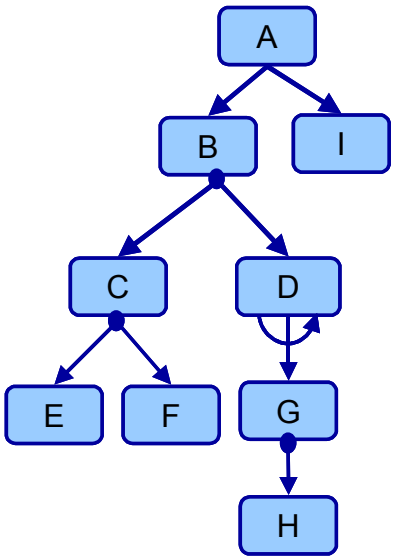
Basis Path Testing During Integration (cont.)

Slide **Integration Basis Path Test Cases**

Basis path testing can also be applied to integration testing when units/components are integrated together:

Complexity: 5

Basis set of paths:
1. A,B,C,E & then A calls I
2. A,B,C,F & then A calls I
3. A,B,D,G (only once) & then A calls I
4. A,B,D,G (more than once) & then A calls I
5. A,B,D,G,H & then A calls I



Basis Path Testing

Basis path testing is a white-box testing technique that identifies test cases based on the flows or logical paths that can be taken through the software. However, basis path testing can also be applied to integration testing when software units/components are integrated together. The use of the technique quantifies the integration effort involved as well as the design-level complexity. A basis path is a unique path through the software where no iterations are allowed.

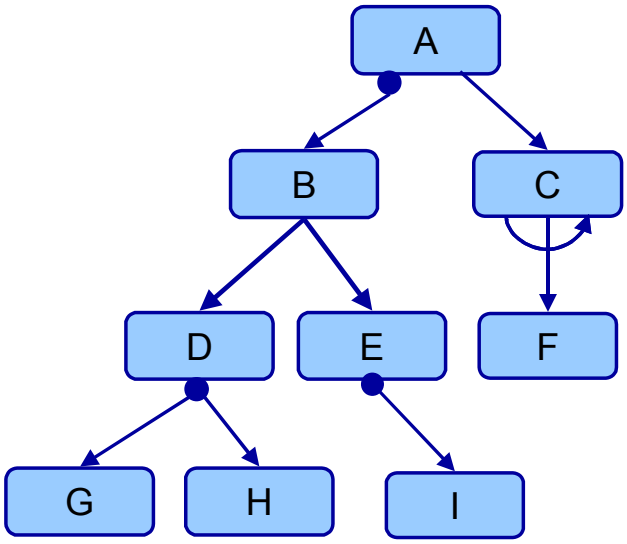
Basis Path Testing During Integration – Exercise

Slide **Integration Basis Path Testing - Exercise**

Instructions:

1.Calculate the integration complexity for the calling tree

2.Select a set of basis paths to test the calling tree



Integration Complexity = _____

Basis Paths:

Basis Path Testing During Integration - Answer

Slide **Integration Basis Path Testing - Answer**

Interaction types:

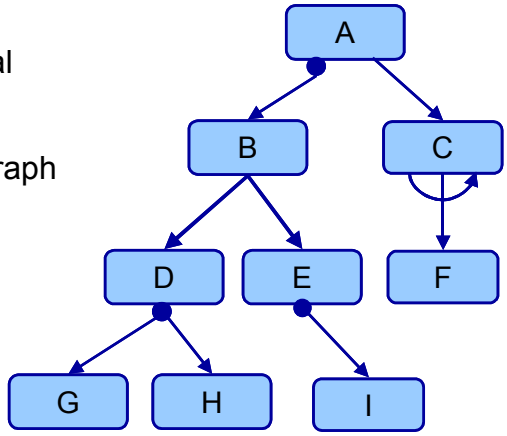
2 conditional

1 mutually exclusive conditional

1 iteration

1 simply because we have a graph

Complexity: 5



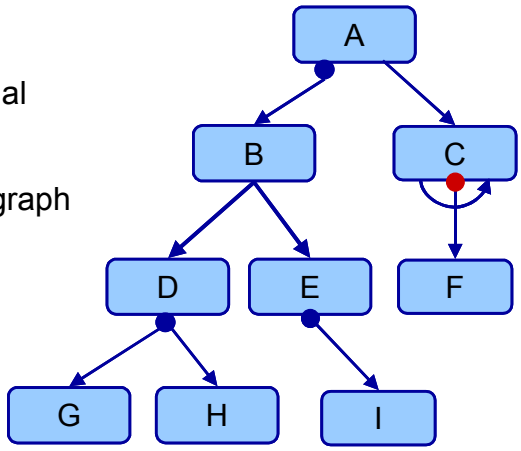
Basis set of paths:

1. A,C,F (only once)
2. A,C,F (more than once)
3. A,B,D,G & then B calls E & then A calls C
4. A,B,D,H & then B calls E & then A calls C,F
5. A,B,D,H & then B calls E, I & then A calls C,F

Answer - Integration Basis Path Testing (cont.)

Slide **And a "What If"**

Complexity: 6



Interaction types:

3 conditional

1 mutually exclusive conditional

1 iteration

1 simply because we have a graph

Basis set of paths:

1. A,C,F (only once)
2. A,C,F (more than once)
3. A,B,D,G & then B calls E & then A calls C
4. A,B,D,H & then B calls E & then A calls C,F
5. A,B,D,H & then B calls E, I & then A calls C,F
6. **A,C**

Addition If the call from unit C to unit F is both conditional and iterative, you add one to the integration complexity for the iteration and one for the conditional. This would increase the integration complexity of this diagram to 6. The additional basis path would be A,C.

Conclusion

Slide **Conclusion**

Benefits of basis path testing:

- Defines the number of independent paths thus the number of test cases needed to ensure:
  - Every statement will be executed at least one time
  - Every condition will be executed on its true & false sides
- Focuses attention on program logic
- Facilitates analytical versus arbitrary test case design

References

Beizer-90 — Boris Beizer, *Software Testing Techniques*, Van Nostrand Reinhold., New York, 1990, ISBN 0-442-20672-0.

GSAM — Department of the Air Force Software Technology Support Center, *Guidelines for Successful Acquisition and Management of Software-Intensive Systems: Weapon Systems Command and Control Systems Management Information Systems*, *Version 3.0*, May 2000, Hill Air Force Base, Utah 84056-5205, https://www.stsc.hill.af.mil

IEEE-610 — IEEE Standards Software Engineering, Volume 1, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std. 610-1990 , The Institute of Electrical and Electronics Engineers, 1999, ISBN 0-7381-1559-2.

McCabe-82 — Thomas J. McCabe, Structured Testing: *A Software Testing Methodology Using the Cyclomatic Complexity Metric*, NBS Special Publication, National Bureau of Standards, 1982. http://www.mccabe.com/nist/nist_pub.php

Pressman-01 — Roger Pressman, *Software Engineering, A practitioner's Approach, Fifth Edition*, McGraw Hill, Boston, 2001, ISBN 0-07-365578-3.

On-Line Resources (note: The use of all on-line resources is subject to applicable copyright laws and may change with time and publication status. The net locations given below should also be considered dynamic, but are accurate at the completion of this paper.)

Unpublished notes on basis path testing are available at:

Check Yang — http://www.chekyang.com

Dr. A.J. Sobey — http://louisa.levels.unisa.edu.au/se1/testing-notes/test01_3.htm

Joseph Poole — NISTIR 5737 - http://hissa.nist.gov/publications/nistir5737/

Contact Information

Slide **Contact Information**

The Westfall Team

Theresa Hunt
1660 Barton St.
Longwood, FL 32750

phone: (407) 834-5825
fax: (407) 834-2735
Theresahunt@earthlink.net

www.westfallteam.com

Minimizing Performance Test Cases for Multi-Tiered Software Systems

Mahesh Bhat, Kingsum Chow, and Jason Davidson
(Mahesh.Bhat, Kingsum.Chow, Jason.A.Davidson@intel.com)

Abstract
Software performance testing for multi-tiered software is a daunting task – the test cases configurations are immense when fully specified; and actual test cases often require the tiers to reach a specific running state before a test can be performed.

This paper proposes a method that
1. Establishes a performance model.
2. Demonstrates that a few experiment that can be afforded for the daily run. The performance model is used to check if these data indicate that we are going to meet the rest of the requirements.
3. Designs a set of test case experiments for the automated weekend runs to check the overall performance and also to recheck the performance model to make sure it is still valid.

We demonstrate the use of the method by applying it to a workload based on J2EE framework. This workload exhibits good performance prediction problem. QA engineers can then use this method to quickly find out the performance of the web application under various conditions without testing each case.

Biographies
Mahesh Bhat is a performance engineer working with Intel's Software Solutions Group. Mahesh received his Ph.D. from the Applied Mechanics department, Indian Institute of Technology in 1996. Mahesh has focused on middleware software development and performance since that time. He has worked for with Intel for last two years. Mahesh's email address is: Mahesh.Bhat@intel.com

Kingsum Chow is a performance engineer working with Intel's Software Solutions Group. Kingsum received his Ph.D. from the Computer Science department, University of Washington in 1996. He worked for Intel since then. He specialized in performance modeling and data analysis in multiple areas such as web applications, compilers and computer architecture. Kingsum's email address is: Kingsum.Chow@intel.com

Jason Davidson is a software engineer working with Intel's Software Solutions Group. Jason received his B.S. in Computer Science at Western Oregon University in 1999. Since that time he has worked for Intel on various projects related to ease of use, knowledge management, middleware, and performance engineering. His work has been presented at various trade shows, and incorporated into various products – including the recently released "Intel® Architecture Software Development Kit for RosettaNet". Jason's email handle is: Jason.A.Davidson@intel.com

1 Introduction

Performance has become a major aspect of software quality. The software not only is supposed to do what is intended to, but it is also expected to complete a task within a time limit. It is not easy to predict the performance of a software application when all of its components are running on the on a single machine. With the emergence of distributed architectures for web applications, it becomes a real challenging job to check and validate software performance for quality.

Distributed architectures are widely used for developing web applications for performance and reliability reasons. Apart from the complexity of distributed architectures, there is an additional complication in validating the performance of the web applications built on distributed architectures. These applications typically allow users to do multiple tasks simultaneously. As a result, a large number of users may be carrying out many different activities at the same time. The mix of their activities may not be known *a priori*. This may be a cause of concern as this greatly affects the ability to validate the performance of the web application. A quality assurance (QA) engineer needs to make sure that the application meets its performance criteria under different mixes of user activities. For example, some requests may need to be handled within X seconds under certain load conditions while other requests may need to be handled within Y seconds, a different time constraint. The QA engineer should try to find out performance statistics early in the development process, as fixing performance problem may require design changes and it is cheaper to make design changes in the early phase of software development. To understand the performance characteristics of a software application, the QA engineer may carry apply load and stress testing. In some cases, they may have to test many combinations of request mixes. In this paper, we propose a method based on design of experiments to effectively employ a small number of experiment runs that need to be made to help increase the productivity of the QA engineer to validate performance criteria are met.

The performance of the web application can be measured by the overall successful operations it is able to carry out in the specific time (throughput) and average time required to complete each operation (response time). Response time is particularly important in the case of the web application, as users may not like to sit in front of the computer for a long time waiting for transaction to complete. Many people have quoted that they would abandon a site if it takes too long (2, 4 and 8 seconds have all been quoted) to complete transactions. In this paper, we will show how to apply a simple mathematical model to predict the performance of a web application and in turn how to use it to be more productive in performance testing. This model will help QA engineers to quickly find out the performance of the web application under various conditions. Additionally this will demonstrate how QA engineers can make very few runs to successfully perform the same job and thus making them more productive. We will present our method in a case study applying it to a J2EE based web application.

2 A performance validation problem

The performance of the web application is measured by the throughput as well as by measuring the response time. Initially it was believed that the response time for a Web Application Server is

simply a function of the number of requests. However, even if the *number* of requests is the same, the response time still can vary a lot.

The problem is that, as stated earlier, there are many different kinds of requests that may go into a web application server. Depending on the *kind* of request, the server may take more or less time to execute that request. In some cases, a limit for the response time of a particular request is required. In some other cases, instead of focusing on one kind of request, a limit for the average response time is required. Yet in other cases, some mixture of these requirements may exist. In all these cases, it is necessary to have the ability to measure the response time for all required usage scenarios. It is possible to save some measurement time by developing a response time performance model and use the model to derive response times for other scenarios that are not measured. However, it is important to note that such a model needs to take into the account of the requests mixture in order for it to be successful. For example, Figure 1 shows the response time of a transaction (known as New Order in the figure) with different transaction mixes for

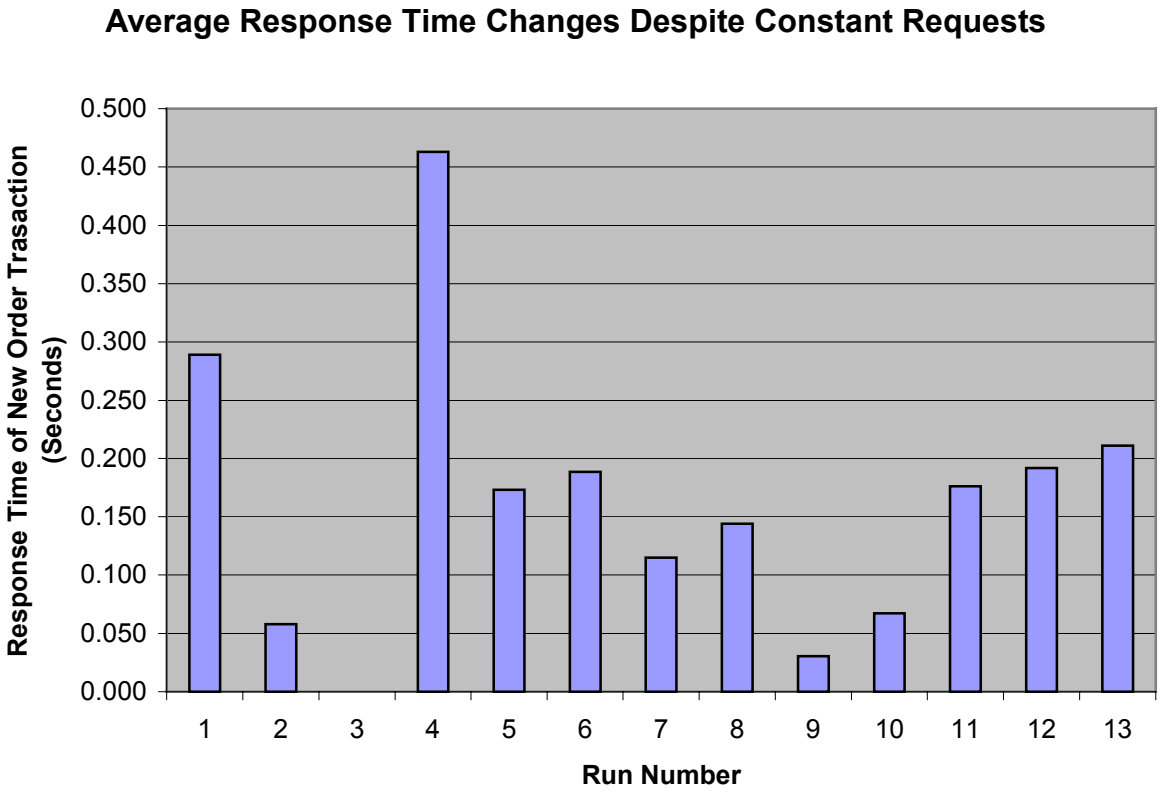


Figure 1. Response Time varies depending on the Type of Request. 13 experiments runs of different mixes of 4 different kinds of transactions were studied. While the total numbers of transactions were kept the same, the proportions of transactions were varied for each run. On the vertical axis, the average response time of one kind of transaction (New Order) was plotted and compared with varying the mixes. This figure shows the variation of response time is highly dependent on the transaction mixes.

different experiment runs. The response time is shown on the vertical axis while the run numbers are shown on the horizontal axis. In these experiments, the total numbers of transactions were kept the same. However, we can see that depending upon the mix, the response time of the particular transaction varies a lot. For example, in Run Number 3 it was zero when there were such transactions involved in the requests.

Performance prediction of a software system is well known to be a difficult problem. According to Menasce[1], a reasonable performance model may have a 10-30% margin of error. In the case of software load and stress testing, the error of prediction may even be higher as when resources are used at stressed conditions that in turn make modeling difficult. There are many factors that affect the performance, e.g. disks, network, OS etc. These components interact with each other their interactions often make performance prediction a challenge.

Despite these difficulties in predicting the performance of web application and thus applying it to reduce QA effort in performance validation, we hypothesized that a simple performance model with moderate accuracy can be developed and it can be useful for the general QA activities. By applying it regularly to software releases such as daily and weekly tests, software performance issues might be caught early in the development and thus might be corrected with a lower cost.

We will illustrate the concept of applying a simple performance modeling method to help QA effort for performance validation using the following example. A software system is being developed with a requirement that the average response time of all requests needs to fall within 2 seconds under all load conditions where the total number of requests does not exceed 70 requests per second. In one approach, a typical configuration is picked by the developer or the QA engineer, and the response time is measured for this particular load configuration for the 2 seconds requirement. The advantage of this approach is clearly simplicity. However, this approach really does not tell us if the response time may exceed the required 2 seconds under other load conditions. A similar but better approach would use multiple load configurations for testing and would validate the software system does not exceed 2 seconds in those cases. While this approach gives us more confidence than the first one, it is still not clear if the software system can indeed perform in less than 2 seconds under ALL load conditions. Since enumerating all load conditions is close to impossible, we employ an approach to develop a simple performance model that can be used to compute the response time for a range of configurations. If we can validate such a performance model, with certain acceptable error of margin, we can use this model to further increase the confidence for performance validation.

In our approach of using a simple performance model, we consider a typical QA process that comprises of weekly and nightly tests. While more tests can be conducted in the weekend, the nightly runs are greatly constrained by the number of tests that can be run. In our approach, we divide the performance validation process in 2 phases: the model validation phase and the test phase. The model validation phase requires more experiment runs and is conducted as part of the weekend runs to validate the performance model in addition to testing for performance validation. It is necessary to validate a model on a regular basis as software changes may, naturally, affect the performance of the software system. The test phase requires fewer experiment runs as it simply applies the most recent model from the most recent weekend runs.

In the validation phase, we apply a method of design of experiments based on a mixture model. Many methods of design of experiments are available and they are described in detail in the book written by Box, Hunter and Hunter[2]. A mixture model is chosen here as it correlates to fixing the number of transactions but varying the different kinds of activities. A mixture model can be readily constructed by using software such as JMP[3], among others[4,5]. Some low cost software alternatives[6,7] may also be adequate to assist the user in modeling. The mixture model employs multiple variables as multiple kinds of transactions. Many kinds of multivariate analysis have been described in the literature. Among them, Kachigan[8] described the concept in an easy to understand way. Readers that are more mathematically inclined may also consider an alternative book by Montgomery[9].

In our application of the mixture model, each kind of transaction constitutes one kind of ingredient. First, we run each ingredient without mixing. These experiments help estimate the primary effect of these main ingredients. Then we run experiments by mixing 2 ingredients in equal proportions and do that for all ingredients. These later experiments help estimate if there are any interaction effects due to mixing these ingredients. Then we run experiments by mixing 3 ingredients at a time, again by mixing equal proportions of each of them. We continue doing that until all ingredients have been chosen in last experiment run.

Table 1 illustrates the 7 configurations that can be used to construct a simple performance model. Each row of the table specifies the amount of ingredients to be used in each experiment. For example, the first row specifies that while 100% of the ingredient X1 is used, no X2 and X3 should be used. The first 3 rows of Table 1 correspond to the main ingredients X1, X2 and X3 without other mixes. The next 3 rows correspond to the 3 combinations of mixing 2 ingredients each. The last row corresponds to the only one combination of mixing all 3 ingredients in equal proportions.

Table 1 The configuration of the samples runs for 3 ingredients used to construct a simple performance model.

X1	X2	X3
100.0%	0.0%	0.0%
0.0%	100.0%	0.0%
0.0%	0.0%	100.0%
50.0%	50.0%	0.0%
50.0%	0.0%	50.0%
0.0%	50.0%	50.0%
33.3%	33.3%	33.3%

As we can infer from Table 1, the number of experiment runs that are required to construct a performance model may increase significant when more ingredients are considered. In such cases, some high level interactions may be dropped to trade accuracy for productivity.

As described earlier, after the measurements were made for the mixture model of experiment design, there are still many methods to construct a model. Our approach uses a general linear model with 2-factor interactions to construct a performance model. In the case of a 3-ingredient model, it will transform to the following equation:

$$\mathbf{R = a_0 + a_1X_1 + a_2X_2 + a_3X_3 + a_{12}X_1{*}X_2 + a_{13}{*}X_1{*}X_3 + a_{23}{*}X_2{*}X_3}$$

The model, as expressed by the above equation, attempts to extract a simple relationship between the response time and the mixture of ingredients. We have also considered other models such as a univariate linear model, a univariate non-linear model and a response surface method. A univariate linear model attempts to construct a model using only one factor while multiple factors participate in the activities. One would not expect a good performance model by considering only one factor while ignoring the rest of factors entirely. A univariate non-linear model extends on the univariate linear model by adding non-linear terms, e.g. the square of the value. While such a model is sometimes adequate for certain kinds of workloads, we prefer a model that takes into considerations of interaction effects as these activities share computing resources from the software system. To keep the model simple, we do not use non-linear factors, i.e., we made a trade off for simplicity so that the model is generally applicable to software applications. However, the accuracy of the model still needs to be assessed.

After a model is constructed, there are two ways to assess the "goodness" of the model. We can use the coefficient of correlation as an indicator. We can also make additional experiment runs that are outside the scope of the sample runs and compute the difference between their measured performance values and the expected performance values from the model that was constructed earlier. Our approach applies both methods. In the extra experiment runs (also called validation runs), the user, the developer and the QA engineer can also help pick the configurations to run. If the coefficient of correlation is high and close to 1, the confidence that a good model is found is increased. If the differences between the values predicted by the model and the measured values are also small, the confidence that a good model has been found is further increased.

A method to minimize performance test cases

In this section, we describe our approach to adapt the performance validation method to a typical web application server. Our approach contains four steps:

1. Categorize the web transactions into ingredients
2. Design and run the experiments and collect measurements
3. Construct a performance model, and
4. Validate the model

These 4 steps are described in the following sections.

3.1 Categorize the web transactions into ingredients

While a web application server supports multiple kinds of requests simultaneously, one needs to identify the main kinds of requests for the model construction. For a system that supports only a few kinds of transactions, all of them can be included in the model. For a system that supports many kinds of transactions, some kind of selection criteria can be used when not all of them can be included in the design of experiments due to QA resource constraints. When not all transactions can be included in the design, one can disregard transactions that only happen rarely (e.g. < 1%) of the time. However, it is important to check the assumption that the assumption is indeed true when the system is operational in the future. One can also combine multiple requests into reduce the number of ingredients for the model. The model could be weakened by this approach. But this may still result in an acceptable model by trading accuracy for time.

The end result of this step is a set of ingredients, X1, X2, … Xn that will be used in the next step for the design of experiments. The response variable, R, is also identified in this step. R is the performance indicator that the user cares about when the system is in operation. A typical response variable is the response time or the throughput of the system.

3.2 Design and run the experiments and collect measurements

The design of experiments has already been partly covered in section 2 for a specific instance of a web validation problem that the requirement is to meet a certain response time while supporting a certain number of simultaneous transactions. Later in section 4 we will illustrate a case study of such an example. For other kinds of requirements, a different design of experiments may be required. For example, the software specification document may require the response time not to exceed M seconds when the load is X and not to exceed N seconds and the load is Y. In such cases, one can choose to have separate sets of mixture model for X and Y. An alternative is to include X and Y as variations in the model. It would take some experiments to understand which approach is better for a given workload. However, the validation method can be used to judge the goodness of the model.

The end result of this step is a set of experiments for both sample runs and validation runs and the measurements associated with the experiments.

3.3 Construct a performance model

After the experiment data have been collected, the measurements from the sample runs were used to construct a performance model. Methods of constructing performance model were partly described in section 2 for the problem that a specific response time needs to be met for a given maximum number of transactions. For other scenarios of performance requirements, different performance models should be considered. It is perhaps a good idea to try a simple model first before attempting any difficult ones. A simple model may be a good tradeoff between accuracy and time to run experiments so far as it provides adequate accuracy.

The end result of this step is a mathematical representation of the performance model. Such a model can often be expressed in terms of some equations. The equations here usually relate some independent variables to the response variable of interest. The independent variables are

categorized by the amount of ingredients while the response variable of interest is a measurement for the performance requirement.

Validate the model

In this step we establish a method to measure the goodness of fit of the model. It is a good idea to check with the author of the software specification if the method is acceptable. In section 2, we described two such methods, namely, the coefficient of correlations and the differences between measured and expected values from the model. We prefer to use the later approach as it directly demonstrates the predictive power of the model. It is based on the predictive power of the model that we can reduce the number of test cases for performance QA.

The end result of this step is to accept or reject the performance model. If a performance is not acceptable, several iterations of the approach can be tried. For example, perhaps a different performance model need to be constructed, i.e., go back to step 3 (Construct a performance model). If there is some doubt to the experiment data collection or the design of the experiments, then one can go back to step 2. If one doubts sufficient kinds of transactions have been identified, one can go back to step 1 and perhaps consult with the user or the requirement documents again.

In the next section, we describe a case study applying our approach to minimize test cases for performance QA activities.

4 A case study to minimize performance test cases

A case study was conducted to illustrate how a performance model can help to minimize performance QA test cases. The case study is divided into 2 phases. In phase 1, we demonstrate that an adequate simple performance model can be obtained for a web application server. In phase 2, we sketch the potential productivity improvement that can be achieved with the approach.

We applied the approach to a Java 2 Enterprise Edition (J2EE) based workload that is deployed on BEA WebLogic Server ™. J2EE is a framework that defines standard for developing and deploying enterprise level Web application. We considered such a workload for our approach as it mimics some web application servers. For applying this method, we need an application (workload) that is representative of the real world systems. The workload should be complex enough to capture intra-, extra- and inter company business processes. It must be scalable. The ECperf ™ workload meets all these requirements.

ECperf[10] is an Enterprise Java Beans (EJB)™ benchmark developed by SUN Microsystems to test the performance of J2EE based application servers and containers. It represents manufacturing, supply chain management and order/inventory business problem. It consists of two applications modeled in it, the order entry and the manufacturing applications. The order entry application is responsible for entering new orders, changing existing orders, retrieving

order status and checking the status of the customer. The manufacturing application is responsible for running assembly lines in the plants. It is a 24x7 as well as distributed workload. It also has well defined scalability rules.

As an example of one of the four order entry transactions mentioned earlier, Figure 2 shows a typical new order transaction in the order entry application in the ECperf workload.

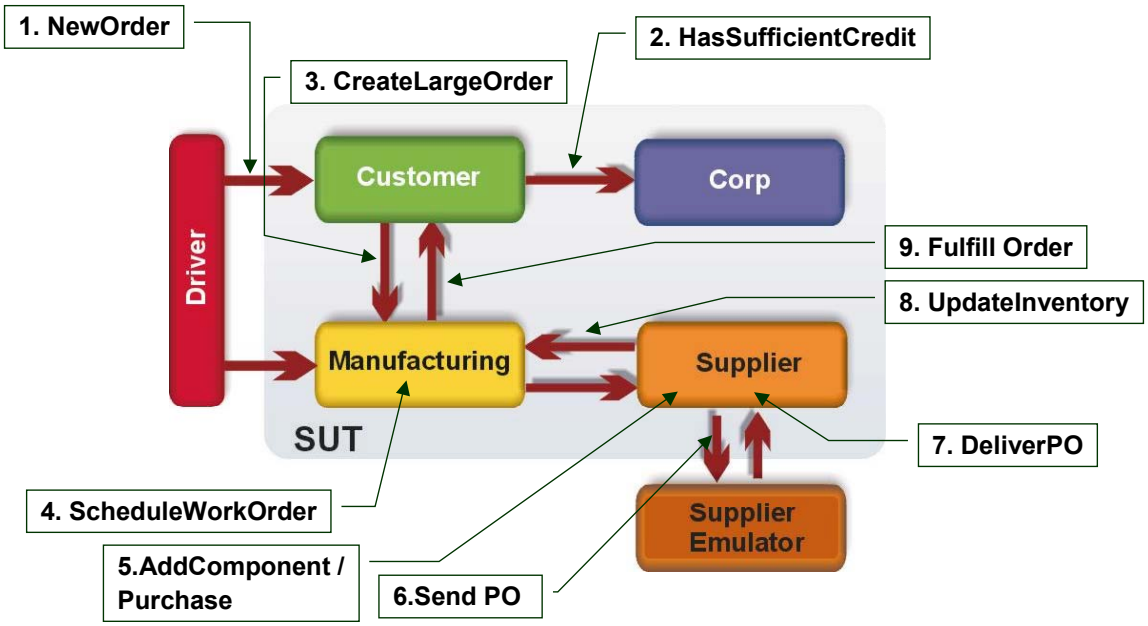


Figure 2 An illustration of the flow of activities for a typical New Order Transaction in the case study of the workload that mimics ECperf activities.

When a customer places a new order (1), the application gets price of all items ordered, taking into account the item discount and calculate the order total. The application then computes the customer's discount for this order and calls for corporate domain to check whether the customer has sufficient credit to cover his purchase (2). If the customer does not have enough credit, it returns an error. If the order is a customer order, it triggers a LargeOrder to be created in the Manufacturing domain (3). The manufacturing domain then in turn schedules a work order (4), identifies components that make up this assembly in the Bill of Material (BOM) and assigns the required parts from the inventory (5) This also calls Suppliers (6) if it needs any parts with low inventory. If the purchase order is placed then the supplier delivers the placed order (7). Once work order is complete, it changes all the records to indicate that it has fulfilled the order (8,9).

Other transaction kinds are outside the scope of this paper. Despite the complexity of the transitions and the involvement of multiple system resources to complete each task, we are going to show later that an acceptable performance model can be obtained by using only simple mathematical modeling.

Figure 3 illustrates some hardware configuration that can be used in our case study for a workload that mimics ECperf activities. As seen from Figure 3, as in the real production system, there are 3 tiers in this workload, Client, Mid-Tier Application Server and Database. This can also have cluster of application servers that can be used for load balancing and fault tolerance purposes.

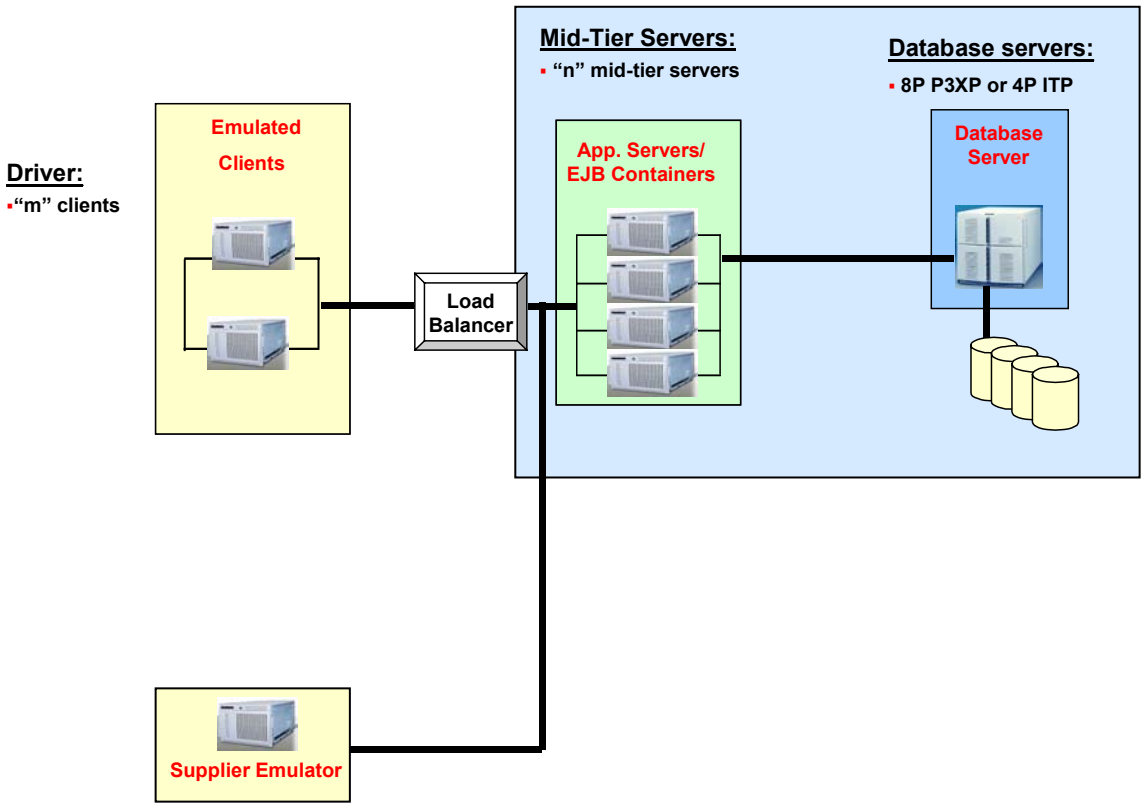


Figure 3 An illustration of some hardware configuration that can be used in our case study for a workload that mimics ECperf activities.

The reader may see some resemblance of the software system and hardware configurations to their own software development in Figure 2 and Figure 3.

In phase 1 of the case study, we now demonstrate our approach by applying it to the order entry application of the ECperf workload. We measured the weighted average response time for the order entry application and evaluate a simple performance model by the coefficient of correlation and its ability to predict responses for samples it has not seen. We first describe how we applied the four-step approach in section 3 in this case study.

Categorize the web transactions into ingredients

In this step, we need to find out what kinds of end user requests are supported by the web application. For example, in a typical web application, some of the concurrent users may be placing the new orders, some may be changing the orders they have already placed or some may be checking the status of the order or some may be checking the status of the customers. In our case of the ECperf workload, this step is relatively easy as four kinds of transactions are readily derived from the workload specification. They are:

New Orders (NO)

- Change Orders (CO)
- Order Status (OS)
- Customer Status (CS)

Since only 4 ingredients were found. There is no need to exclude any factor and we proceeded to the next step. Using all 4 ingredients.

4.2 Design and run the experiments and collect measurements

In step 2 of the approach, 4 ingredients need to be included in the design of experiments. As described in section 3.2, we developed a mixture model at high load rate. This mixture model – had experiment runs that are consisted of 100% of each ingredient as shown in Table 2.

Table 2 Test cases correspond to 100% of each ingredient. The table shows the contribution in percentages of each ingredient (in columns) for the experiment runs (rows).

NO	CO	OS	CS
100	0	0	0
0	100	0	0
0	0	100	0
0	0	0	100

In addition to single ingredient contributions, we also included two ingredient mixtures of equal proportions, i.e., 50% of each pair of requests. Table 3 shows the 6 sample runs that correspond to these two ingredient mixtures.

Table 3 Test cases correspond to 2 ingredient interactions. The table shows the contribution in percentages of each ingredient (in columns) for the experiment runs (rows).

NO	CO	OS	CS
50	50	0	0
0	50	50	0
0	0	50	50
0	50	0	50
50	0	50	0
50	0	50	0

We then proceeded to finish the rest of the test cases, including 4 runs for 3 ingredient of each proportions and 1 run for 4 ingredient of each proportions. The details of these configurations can be inferred from Table 4 that will be described later and thus are not shown separately.

Based on our experience with the workload, we also chose Six test case for validation. All data, including both the sample and the validation sets are shown in Table 4. The average response time (WtRA) was chosen to be the response variable as it is part of the performance requirement for the workload.

Table 4 shows the dataset used for these experiments. First 14 rows are the sample set this is used for constructing the model and last six rows is a validation set. Columns WtRA and TP are response variables.

Table 4 The entire data set comprising the sample set and the validation set. The first 14 rows of data belong to the sample set. The sample set has only 14 rows of data instead of the expected 15 rows because one data point was thrown out, as there were some system failures in that run. The first 4 columns correspond to the rate of transactions per second for each of the 4 ingredients. The last column is the response variable. The model is to predict WtRA from the transaction rates of NO, CO, OS and CS. The last 6 rows of the table contain the validation set.

	NO	CO	OS	CS	WtRA
Sample set	0	23.05	23.56	23.19	0.07
	22.99	0	23.65	23.09	0.22
	0	34.88	34.86	0	0.29
	23.23	23.36	0	22.77	0.50
	0	0	35.23	35.17	0.13
	30.87	30.72	0	0	2.42
	0	35.00	0	35.61	0.20
	22.98	23.81	22.88	0	0.53
	35.18	0	35.17	0	0.50
	17.42	17.46	17.41	17.39	0.34
	34.65	0	0	34.78	0.59
	0	0	0	69.99	0.15
	0	0	70.49	0	0.17
	0	69.8	0	0	0.23
Validation set	33.37	13.66	13.40	6.77	1.19
	34.54	13.63	6.99	13.72	0.77
	34.43	6.99	13.85	13.75	0.69
	14.053	34.92	14.16	6.94	0.36
	6.97	34.90	14.12	14.09	0.30
	14.00	14.02	34.94	6.94	0.36

4.3 Construct a performance model

Multivariate Linear (ML) model is a general model that will be attempted to apply for this workload. In order to apply it, we need to determine the independent variables and response variables.

In this case, independent variables are number of transactions per second for new order, change order, order status and customer status. These are NO, CO, OS and CS columns of Table 2. The response variable is weighted average response time, wtRA column of Table 2. After identifying the independent variables and response variable, regression analysis is carried out using Excel.

Validate the model

We used two criteria, the regression of the sample set and the predictability of the validation set, to judge the goodness of the model.

Table 5 shows the regression statistics of the sample set as given by Excel. These key statistics indicate that the model should be acceptable as the "R Square" value is more than 0.95.

Table 5 Regression statistics

Multiple R	0.99
R Square	0.97
Adjusted R Square	0.96

Table 6 shows the weighted average response predicted by using a multivariate linear model described earlier. The differences between the predicted values and the measured values were all found to be less than 0.1 sec.

Table 6 Differences in measured and predicted average response time in the case study

Measured Response Time (sec)	Predicted Response Time (sec)	Difference (sec)
1.19	1.11	0.08
0.77	0.72	0.05
0.69	0.69	0.00
0.35	0.30	0.05
0.30	0.24	0.06
0.36	0.34	0.02

Based on the two criteria, we accepted that the model is adequate for performance prediction.

4.5 Minimizing test cases

The section describes the phase 2 of the case study. In our workload, it took about half an hour for an experiment run. The specification required a maximum of 2 seconds for an average transaction under all loads of the maximum number of transactions per second. If we enumerate all possible experiment combinations with steps of 10% (e.g. testing with 10% NO, 10% CO, 10% OS, 70% CS, etc), we would need a total of well over 250 test cases, i.e., more than 5 days of test time. Applying the approach described here, we reduced the validation experiments to (15

runs of sample set and 6 runs of validation set), a 21 run effort, or less than half a day's of effort over the weekend for model validation and testing. Then for the nightly test cases, we could select just 6 runs for a total of 3 hours per night. We turned a tremendous effort of running with test cases into something manageable with weekly and nightly tests by trading off a little accuracy with a great gain in productivity. In this case study, the accuracy that is sacrificed is barely the order of 0.1 sec, which should be acceptable for a requirement of 2 seconds.

Summary and discussions

This paper described an approach that with some mild effort, it can be applied to some performance test scenarios. We have shown that it is possible to use our approach to get acceptable performance prediction for one web application workload. We have also sketched how the results can be useful for performance validation by reducing the number of test cases. The contributions of this paper are:

- Demonstrate that simple mathematical models can be useful for performance analysis.
- Demonstrate that performance analysis can be made useful for QA engineers.
- Apply common methods to construct performance models for web application servers.

In the literature, some early successes applying simple mathematical models to web applications were also reported. Chow[11] described applying a model to achieve 2% prediction errors on a backend application in a data center. Further, Chow and Bhat[12] compared the application of simple models to some sample Java server applications. The work described in this paper continues their investigation on applying simple models to solve real life software performance problems.

There are still quite a number of limitations in our approach. For example, we have only done a case study for a response time requirement where a maximum number of simultaneous transactions can be present and also for a small number of kinds of transactions. More work needs to be done to understand a more general approach, which may involve a more complex requirement, such as multiple performance requirements at multiple load conditions, or a more complex workload, such as multiple kinds of transactions but not entirely independent of each other.

It is perhaps more beneficial to automate such an approach, perhaps by combining with data mining methods. That is yet another interesting piece of future work.

References

[1] Daniel A. Menasce and Virgilio A. F. Almeida, "Capacity Planning for Web Services", Prentice Hall, 2002.
[2] George E. P. Box, William Hunter and J. Stuart Hunter, "Statistics for Experimenters An Introduction to Design, Data Analysis and Model Building", John Wiley & Sons, 1978.
[3] JMP, from SAS Institute Inc. (www.sas.com)
[4] Splus from Insightful Corporation. (www.insightful.com)
[5] Systat, from SPSS Inc. (www.spss.com)
[6] Xlstat (www.xlstat.com)
[7] Excel Analysis Toolpak, from Microsoft (www.microsoft.com)
[8] Sam Kash Kachigan, "Multivariate Statistical Analysis A Conceptual Introduction", 2nd edition, Radius Press, New York, 1982.
[9] Douglas C. Montgomery, "Design and Analysis of Experiments", John Wiley & Sons, 1991.
[10] ECperf: www.theserverside.com/ecperf
[11] Kingsum Chow, "Methods for e-Commerce Web Performance Prediction And Capacity Planning", Proceedings of the Intel Quality and Reliability Conference, Oct 16-18, 2000, San Diego, California, USA.
[12] Kingsum Chow and Mahesh Bhat, "Methods for Web Performance Prediction And Capacity Planning", BEA eWorld 2002.

"Testing in the .NET Maze"

Thomas Arnold, tom@xtenddevelopment.com
2002 Pacific Northwest Software Quality Conference

Introduction

Microsoft has created a new environment that promises to ease the process of software development. As test engineers, it is up to us to figure out how the new .NET Framework applies to us, and our efforts, in testing the resulting applications.

Because of the enormity of the .NET Framework this paper moves from the large, high-level view of .NET down to specific examples in ASP.NET. That is, *WebForms* are explored instead of *WinForms*, two pieces of the puzzle that will become clearer as you read on.

This paper is provided for the 2002 Pacific Northwest Software Quality Conference in Portland, Oregon. For more information about PNSQC visit www.pnsqc.org. For the electronic version of this document, visit www.XtendDevelopment.com/pnsqc2002/ .

Topics

In this paper I will introduce you to a high-level view of what Microsoft .NET is, issues – such as migrating to the .NET solution – that could result in bugs, inherent challenges for software test engineers, approaches to testing deployed projects, and where to find more information to continue to learn about testing .NET applications.

Author/Speaker Background

My background is in software development and automated testing. I started my professional career in the software industry as a test engineer in the Seattle, Washington, area in 1990. Since that time I've continued to be involved in software testing (focusing mostly on test automation), development (C, C++, VB, and most recently Java), and managing software development projects.

I started using Microsoft .NET in August 2001 and have found it to bring some very exciting things to the table for developers. I was happy to see that Microsoft kept software test engineers in mind as they created this new solution, as you'll see.

Why .NET

Why Microsoft .NET? Sun Microsystems is one reason. Sun has been building on its Java solutions since 1995 when Java was first released, and they've been running hard for these past 7 years. Their solution is J2EE (Java 2 Enterprise Edition) that allows the Java language to work within multiple operating systems as well as with many databases. Sun has also come out with J2SE (Java 2 Standard Edition) and J2ME (Java 2 Micro Edition). These solutions allow Java users to work in simple web environments (J2SE), Enterprise (J2EE), and with handheld devices (J2ME). Very exciting, and all bundled up in a very nice package with many developers excited about the prospects.

Enter Microsoft, a company that has long dominated the software development industry, suddenly seeing some of its development supporters casting their gazes

upon Sun Microsystems' solutions to Internet applications and multiple platform support. Sun, with its popular Java programming language that is secure and easy to use compared to C++.

Microsoft took the next logical step in the evolution of their development approaches and brought together many of its development solutions to be placed into a bucket named ".NET." Does that mean .NET is entirely new? No. Microsoft has taken all of its existing functionality, added in some additional bits (albeit some rather large and important bits), and pulled it into a solution that will compete (very well) with Sun Microsystems.

This is a good thing, why? Because Microsoft .NET brings a new focus on how to approach Windows and Internet development, an approach that not only opens up the architecture to allow a host of new languages to be supported in the .NET development environment, but operating systems as well. The .NET Framework is setting the scene to allow applications to be developed and deployed in many environments, and additional support for such deployment created by third party vendors.

.NET Framework

This framework, at first glance, seems more like a huge puzzle or maze. Just when we're getting things figured out, yet another enigma in the software industry presents itself, this time in the form of Microsoft .NET.

Fear not, .NET is not so overwhelming after all. Remember that it's an encapsulation of a number of pre-existing Microsoft technologies with a few new ones thrown in for good measure. This, as well as a common thread – or framework – that pulls it all together, is what makes up Microsoft .NET.

Common Language Specification

The top layer shown in **Figure 1** (on the following page) illustrates the default languages already supported by .NET: Visual Basic, C++, JavaScript (known at Microsoft as "JScript"), and Microsoft's new C# (pronounced, "C Sharp").

Microsoft Visual Basic .NET

Visual Basic now offers full object-oriented language features, including implementation inheritance. It also allows developers to create highly scalable code with explicit free threading and highly maintainable code with the addition of modernized language constructs like structured exception handling.

Microsoft Visual C# .NET

Microsoft also created its own Java-like language called C#. C# is very much like Java in that it handles all garbage collection, provides security, and is fairly easy to use compared to C++. It was built from the ground up with the .NET Framework in mind and is a modern, object-oriented, type-safe language. C# "is designed to bring rapid development to the C++ programmer without sacrificing the power and control that have been a hallmark of C and C++."

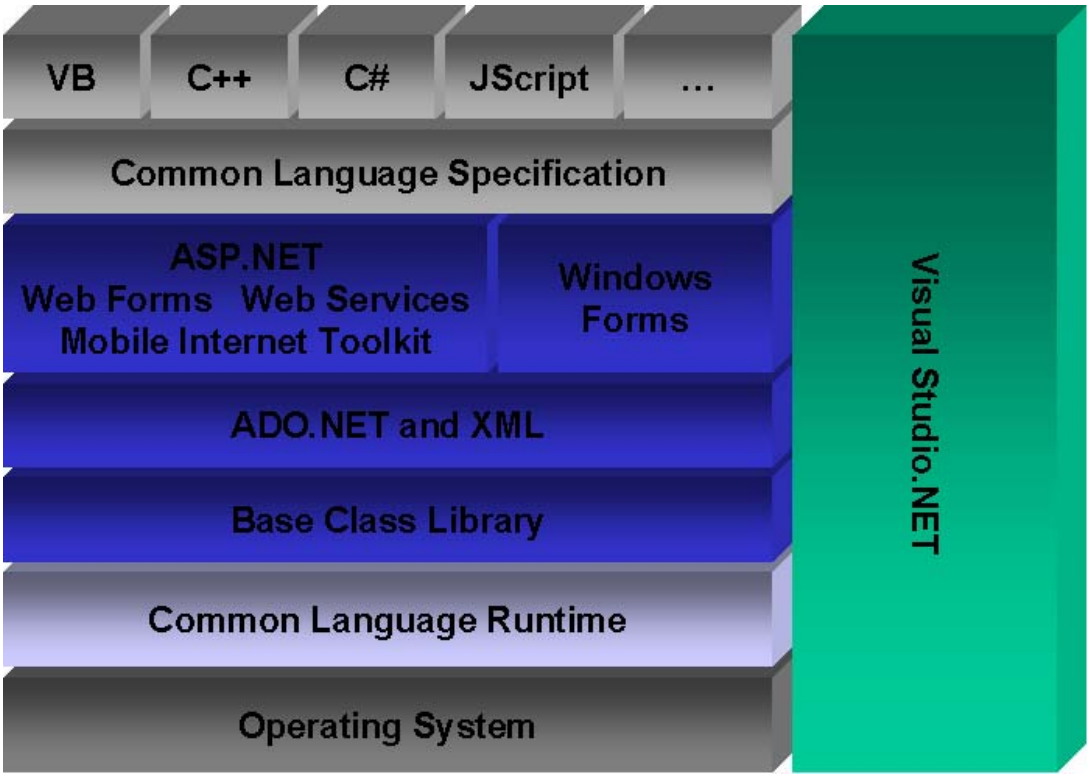


Figure 1 - *Microsoft .NET Framework*

Microsoft Visual C++ .NET

Traditional *unmanaged* (outside of .NET) C++ and new *managed* (within .NET's structure) C++ code can be mixed freely within the same application. Existing components can be wrapped as .NET components by using the managed extensions. Most importantly, providing support for C++ preserves investment in existing code while integrating with the .NET Framework.

JavaScript / JScript

JavaScript is the common language used for web development. This is because older versions of Netscape Navigator supported only JavaScript, while Microsoft Internet Explorer supported JavaScript and VBScript. If a website is being created with client-side scripting, maximum compatibility can be maintained by using JavaScript. It's no wonder the popular language is supported in the .NET Framework.

Other Languages

This is not a complete list of the languages supported by .NET, however, not by a long shot. By creating and publishing a *Common Language Specification* (CLS), third party vendors can take new or existing languages and fit them into the .NET puzzle. This means COBOL, FORTRAN, Java, and many other languages can now be used to program Windows (WinForms) and Web (ASP.NET) applications and services. Therefore the list will never be complete as vendors continue to add to the roster of supported languages. And, if these languages follow the rules laid down by the CLS, they can then access the libraries provided with .NET, as shown in Figure 1's middle layer.

The Middle Layer

The middle layer looks nice and neat the way it's divided up in Figure 1, but really it all exists in a grouping of over 90 collections, each one referred to as a *namespace*. All languages that follow the CLS can work with these namespaces and thereby use ADO.NET (database support) and the other libraries necessary to create Windows and Web applications. It is also possible to create namespaces outside of what Microsoft has already provided.

ASP.NET

The rewrite of ASP (Active Server Pages) – called ASPX for no other reason than they were focused more on creating something cool than trying to figure out a catchy name – became known as ASP+, and later renamed to ASP.NET. It addressed many shortcomings of ASP. While ASP pages (.asp) are still supported by the ASP.NET server, they must use the new file extension (.aspx). ASP.NET focuses more on separating the HTML from the code. By using a *code behind* approach, the HTML is kept in an .aspx file, and the new general practice is to place the blocks of code into separate files to be included by the .aspx files (such as pagename.aspx.vb for VB.NET code, or pagename.aspx.cs for C# code, for example).

In addition, ASP.NET works closely with Microsoft NT/Win2K servers' security settings. It works within those policies to make changes to permissions, rights, and more, much easier. It also allows a user session to be shared over multiple servers so that load balancing is easier and more effective. And, should one of the servers go down, the user remains blissfully unaware and is able to carry out his transaction because his session does not live on any one server.

And last, ASP.NET compiles its pages *just in time* so that execution is much faster than ASP. The first time an .aspx page is accessed after being saved to the web server, the page is compiled into a pseudo code form. This doesn't increase actual execution speed of the code (that is, it's not compiled into machine language), but it does allow the ASP.NET Server to avoid the compilation step for each and every user before spitting out the generated HTML. Execution speed seems faster to the end-user since the compilation step occurs only when the page is modified.

ADO.NET

Open Data Base Connectivity (ODBC) is an old tried and true standard for accessing data. It was designed to provide a common set of routines to programmers. These routines remained unchanged regardless of the type of database being accessed (e.g. Access, SQL, Oracle). The next step in the evolution of the anonymous data store was OLE-DB that not only supports ODBC, as well as its own methods for working with Access/SQL/Oracle, it also works with Exchange, Excel, and other applications (no, they don't have to be Microsoft applications, just support OLE-DB).

ADO.NET (ActiveX Data Objects) is a friendly interface to OLE-DB. It provides a set of objects to the languages working within the guidelines of the CLS. It's yet another level of abstraction to keep things simple and common to the programmer, and allows ADO.NET to deal with the bit twiddling behind the scenes.

Base Class Libraries

ASP.NET and ADO.NET are part of the base class libraries provided in the .NET framework. These namespaces are what provide common objects and methods used by all CLS-compliant languages in the .NET framework.

Common Language Runtime

The final layer is the Common Language Runtime, or CLR. This piece sits on top of the operating system and executes the compiled code. So here's where it gets really cool. Because all languages that want to work with .NET must conform to the CLS, and these *managed* languages all use the base class libraries (including ASP.NET and ADO.NET), everything can be compiled down to a common set of metadata or an *Intermediate Language*. This is the most basic level of data and at this point it doesn't matter what language the instructions were written in. VB, C#, C++, Java, COBOL, whatever, it all looks the same at this intermediate language level.

This is a wonderful thing because this means that all languages can (and do) share the same class definitions and objects defined further up the ladder. A namespace can be created and used by all of the languages because they all eventually end up at this very basic level so that the runtime engine can interpret them.

It gets better. Because the programming languages use the objects created via the .NET namespaces for file manipulation, and therefore the CLR separates the operating system from those languages, different versions of the CLR can be written for the Macintosh, Linux, and so on. When OS-specific versions of the CLR are rolled out, it will be possible to write your program once and have it deployed on multiple operating systems without any extra work. (In theory). Sound familiar? (Hint: Sun Microsystems' goal with Java).

Challenges

As with any new approach, there is always a price of entry, whether it's the learning curve or bringing your now-Legacy-code along into the new system. In the case of the .NET Framework there are two obvious challenges from the start: Migration of old code into the new environment and understanding how much control .NET wants to exercise in an attempt to hold our hands and make things easier.

Migration Issues

The Common Language Specification requires all languages to follow specific guidelines to be allowed to participate in the .NET Framework. This applied to creating a .NET version of Visual Basic as well. The result is changes that are easy to accept when creating new applications, but can be more involved when migrating code.

In the case of Visual Basic (VB.NET), for example, migration issues exist in regards to the introduction and handling of new data types, renaming/moving of functions, and the discontinuation of keywords.

What this has to do with software testing is that the code base that once worked "good enough" to share with the user-base gets touched, and in an invasive way. Opening a code base after Testing has blessed it is already a tricky business, but to modify code so that it can work within the new framework – replacing type declarations, using new functions, and more – will require a full test pass to verify nothing breaks in the process. (Software test automation that's already in place will come in handy).

New Types & Keywords

With the creation of a more generalized approach allowing many languages to work together comes the need to tighten and redefine past approaches. **Table 1** reflects just a few of the differences between VB6 and VB.NET.

Visual Basic 6	**VB.NET**
Fixed length strings were declared as: Dim Name as String * 30	Fixed-length strings are not allowed
An Integer type is 16-bits	A Short type is 16-bits
A Long type is 32-bits	An Integer type is 32-bits
No support for a 64-bits integer type	A Long type is 64-bits
Any data can be set to a Variant variable	The Variant type is unsupported
"Dim X, Y as Long" results in X declared as a Variant and Y as a Long	"Dim X, Y as Long" results in X and Y declared as a Long
Keyword "Empty" indicates an un-initialized Variant variable. "Null" indicates that a variable contains no valid data.	"Null" and "Empty" have been replaced by the keyword "Nothing"

Table 1 - *Visual Basic 6 & VB.NET differences (Types and Keywords)*

Moved Functions

To follow the new object-oriented approach that .NET utilizes through its libraries – known as *namespaces* – functions have been relocated. Let's take Visual Basic's Math functions for example. They have all been moved into the `System.Math` group, so now:

```
X = Cos(Y)
```

Becomes:

```
X = System.Math.Cos(Y)
```

A tool does exist for migrating Visual Basic 6 projects over to VB.NET. Before you breathe a sigh of relief, however, know that most people who have used this tool say that it is not that helpful on large conversion efforts. If you have a simple application to convert, it will provide you with some assistance. However, the changes between version 6 and VB.NET are great enough to make a re-write of the application worth considering, depending on the type of application and its features. For more details about upgrading your Visual Basic 6 applications to VB.NET, I recommend the following MSDN article as a very good starting point:

http://msdn.microsoft.com/vbasic/techinfo/articles/upgrade/vbupgrade.asp

No Direct Mapping of ASP.NET to HTML Controls

One of the exciting features of ASP.NET is its ability to spare the developer from the hassles of tracking which browser a visitor is using during a web session and providing different HTML based on the visitor's browser's capabilities. ASP.NET will issue the HTML it thinks best suits the client. This is a great concept, but in practice the results are not yet clearly known. It remains to be seen how well this type of handholding will work and if it results in workarounds being that much more challenging.

An example is the text box placed on a web page. In ASP.NET it looks like this:

```
<asp:TextBox id="SearchTextBox" runat="server"
MaxLength="25"></asp:textbox>
```

It generates the following HTML:

```
<input name="ModuleSearch:SearchTextBox" type="text"
maxlength="25" id="ModuleSearch_SearchTextBox" />
```

However, depending on the browser, it could also generate this HTML:

```
<textarea name="ModuleSearch:SearchTextBox" rows=1
maxlength="25" id="ModuleSearch_SearchTextBox"><textarea>
```

You will note that these are two different control types, yet they can resemble each other depending on the browser being used. The theory is that ASP.NET knows best, and this remains to be seen. To be sure, as feedback comes in ASP.NET will become much more robust as Microsoft builds on its goal of helping testers and developers alike worry less about browser compatibility.

Testing in .NET

Now that we have a high-level view of what Microsoft .NET is, some of the challenges that programmers face, and some of the issues testers need to be aware of, let's look at a sample ASP.NET application and some of the things we should consider in its testing.

The web application we'll use in this example comes with Microsoft Visual Studio .NET and is called "Duwamish 7.0." Its home page is shown in **Figure 2** on the following page. This application is for a fictional on-line bookseller and demonstrates the concepts of modular development, working with controls, searching, an e-commerce shopping cart, and working with a Microsoft SQL Server database. (Microsoft was kind enough to fill the database for us with sample data).

Black Box Testing Remains Crucial

Although .NET makes technical testing more accessible to software test engineers, the non-programming aspects for software testing remains key. Usability issues remain important, of course, as does verifying that an application behaves, as an end-user would expect. There is nothing new to be introduced to software test engineers in the realm of black box testing in regards to an ASP.NET web application. Browser compatibility testing remains an important part of the process, especially since ASP.NET generates HTML unique to a user's operating system, browser and browser version.

In regards to taking more technical approaches, there are many opportunities available to test engineers, which the remainder of this document will explore.

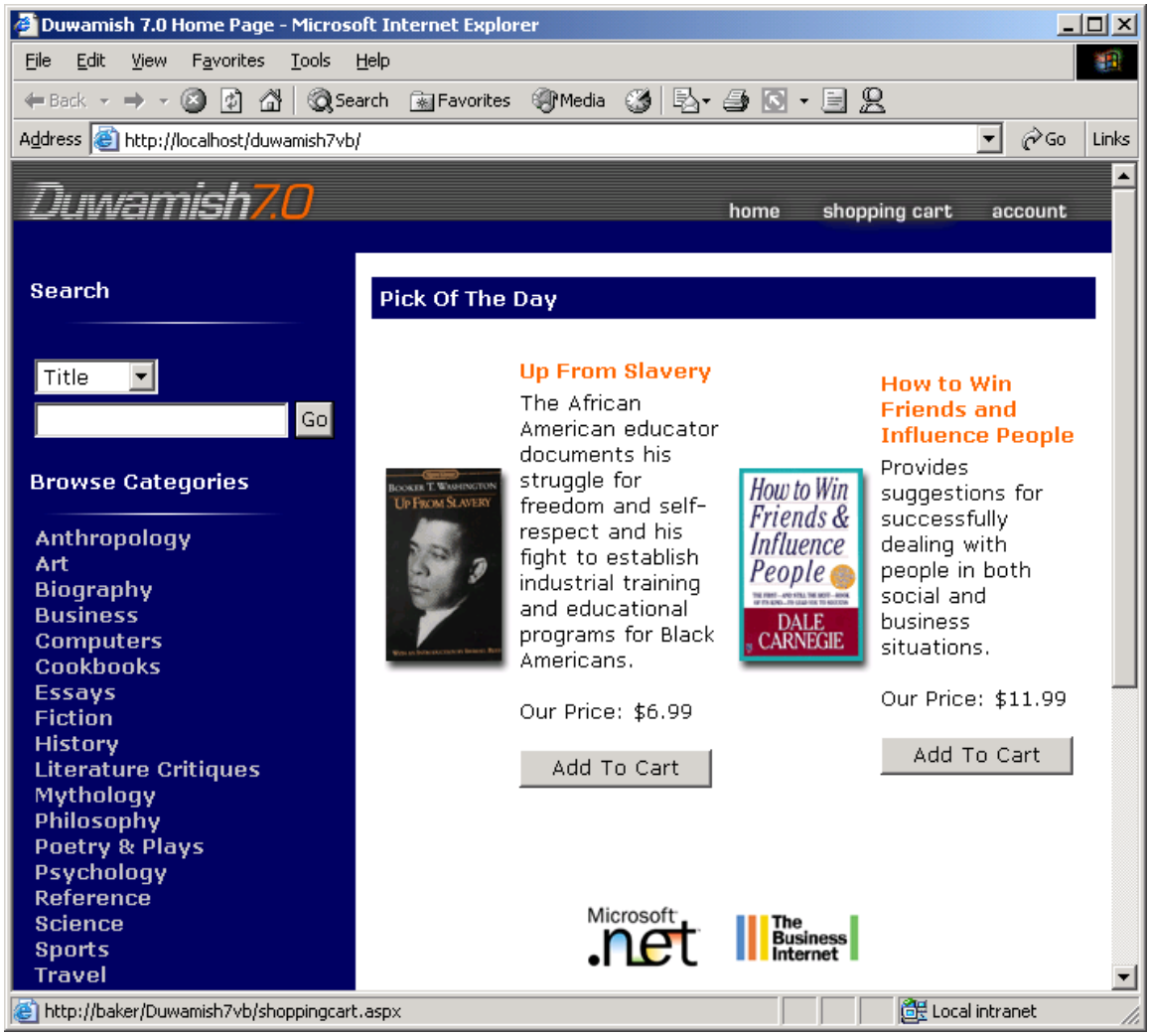


Figure 2 *– Sample application that invokes many of ASP.NET's features.*

Technical Testing

Microsoft has provided the tools necessary to test and debug ASP.NET applications that are not only useful for developers but testers as well. Some of these tools move into the realm of gray and white box testing, however, which some testing organizations are against. The concern of these organizations is that it steps too far away from what the end-user will experience. It also requires a more technical (and hence, typically more costly) test engineer. I am of the opinion that while black box testing is extremely important, the more technical a tester can be in their efforts, the more effective they can be in diagnosing and tracking down issues and bugs and communicating those problems to their programmer counterparts.

In this section we will introduce the `<trace>` setting that can be added to an application's `web.config` file and how it is used in debugging. We will also look at some of the tools available for automating the testing on ASP.NET applications, both in functionality and load/stress testing.

Web.config

ASP.NET has what Microsoft refers to as a *configuration system*. This system is an extensible infrastructure that enables all ASP.NET applications' configuration settings to be defined when an application is first deployed, and modified any time thereafter. The root configuration file is `machine.config` and configures the entire web server. Another file – `web.config` – can appear in multiple directories throughout the ASP.NET web application server. The `web.config` file affects the directory it is in, as well as its directory's sub-directories. In addition, a `web.config` file in a lower child directory can override or modify those settings of its parent.

Each `web.config` file contains a nested hierarchy of XML tags and sub-tags. These tags have attributes that specify the configuration settings. There are over 60 elements that make up the configuration schema that controls how ASP.NET web applications behave. You can even add your own, if you like. **Listing 1** shows an example of a `web.config` file.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
 <configSections>
  <section name="ApplicationConfiguration"
   type="Duwamish7.SystemFramework.ApplicationConfiguration,
   Duwamish7.SystemFramework" />
  <section name="DuwamishConfiguration"
   type="Duwamish7.Common.DuwamishConfiguration,
   Duwamish7.Common" />
  <section name="SourceViewer"
   type="System.Configuration.NameValueSectionHandler, System,
   Version=1.0.3300.0, Culture=neutral,
   PublicKeyToken=b77a5c561934e089" />
 </configSections>
 <system.web>
  <customErrors defaultRedirect="errorpage.aspx" mode="On" />
  <compilation debug="true" />
  <sessionState cookieless="false" timeout="20" mode="InProc"
   stateConnectionString="tcpip=127.0.0.1:42424"
   sqlConnectionString="data source=127.0.0.1;
   user id=sa;password=" />
  <globalization responseEncoding="utf-8"
   requestEncoding="utf-8" />
  <!-- security -->
  <authentication mode="Forms">
    <forms name=".ADUAUTH" loginUrl="secure\logon.aspx"
     protection="All">
    </forms>
  </authentication>
  <authorization>
    <allow users="*" />
  </authorization>
 </system.web>
</configuration>
```

***Listing 1**: Example of a web.config file.*

First and foremost, don't sweat it. This isn't as scary as it looks, and not only that, we only want to work with a very small section of the file. Specifically, we will insert a `<trace>` tag directly under the `<system.web>` tag and place it into the directory containing the pages we want to work with. In this case, to keep it simple, and to avoid multiple copies of `web.config` that could go forgotten, it will be the root version of `web.config` that we modify. The following line is added:

```
<trace enabled="true" pageOutput="true" requestLimit="15"
 traceMode="SortByCategory" />
```

Inserting this simple XML tag and its properties has dramatic effects on your ASP.NET web application. In standard ASP pages it was necessary to insert statements to print out the current status of variables or track what branches of code were executed. This was done by strategically placing `Response.Write()` functions throughout the .asp files. The problem was that these statements had to be removed later (and not forgotten). ASP.NET's solution to this is the *trace* functionality. With the above line added to `web.config` in the Duwamish 7.0 sample application, navigating to its home page (Figure 1) tacks on the additional information shown in **Figure 2**.

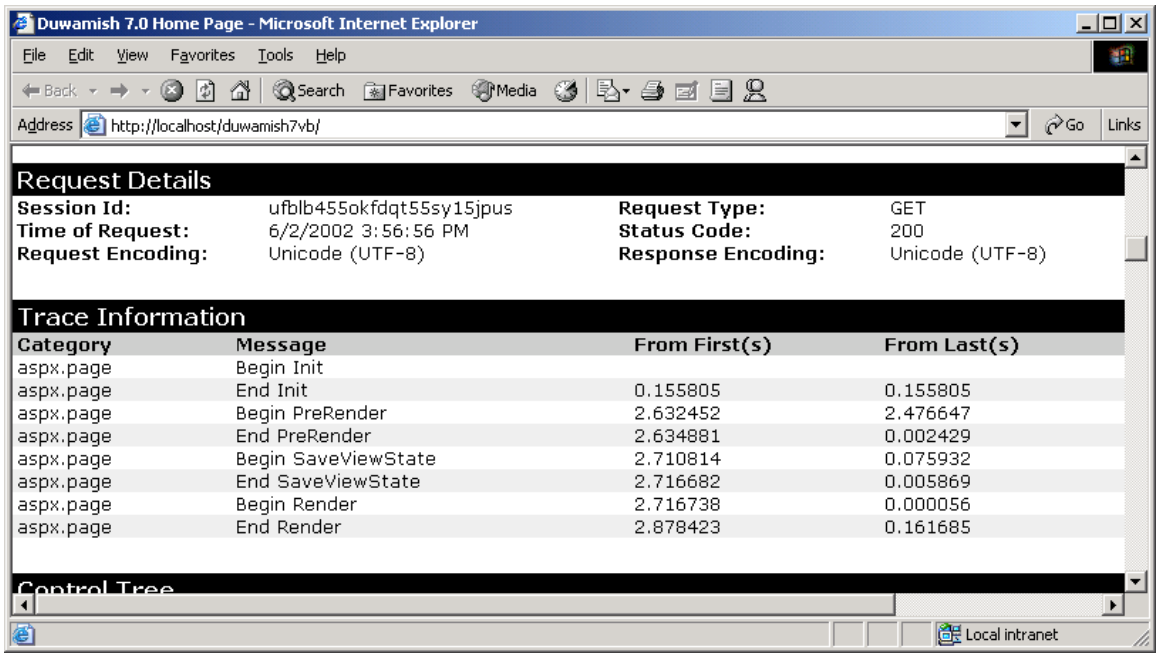


Figure 2 *– The <trace> tag is added to web.config in the Duwamish 7.0 web directory.*

As you can see, ASP.NET's new tracing functionality allows us to view verbose information about an application with minimal intrusiveness. In the past it was necessary to sprinkle `Response.Write()` routines throughout the code. Now, only one file needs to be modified lowering the likelihood this setting will be accidentally left enabled. In addition, `Trace.Write()` routines may be used throughout the code where `Response.Write()` methods might have been used for debugging purposes in the past. Because `Trace.Write()` is only invoked when the `<trace>` tag is in place, those statements may remain in place without ill effect.

The output provided when using <trace> is provided in six sections:

- **Request Details:** This provides such basic information as the visitor's unique session ID, the time & date the request came into the server, HTTP request type and its status code.
- **Trace Information:** In addition to ASP.NET-generated information about the execution of the application, this is where `Trace.Write()` values are printed. The two parameters taken by `Trace.Write()` are displayed here as Category and Message values.
- **Control Tree:** This section provides information about the controls used within the page. ID, type, render size, and view state information is provided.
- **Cookies Collection:** Any cookies sent by the client in its request to the server are displayed.
- **Headers Collection:** HTTP header values sent by the client to the server are displayed here in a simple Name/Value pairing.
- **Querystring Collection:** This shows up only when the `GET` method is used to submit a form. It shows the variables and values sent with the request. (These same values can be found in their raw form in the `QUERY_STRING` entry of the Server variables section).
- **Server variables:** The Name and Value of server side variables are displayed in this table. This section includes a lot of the same information listed in the other sections, just not as nicely formatted.

This information is helpful to test engineers in a number of ways. The *Request Details* section, for example, is important in showing how the server responded back to the client. Specifically, showing a status code of 200 means that no errors were encountered. An error of 400 is a *Bad Request*, 401 is *Required Authorization*, 403 is *Forbidden Directory*, 404 is *Page Not Found*, and 500 is *Internal Server Error* (nothing new). Being able to print a page that shows the request string that resulted in an Internal Server Error can be very helpful to development. The *Trace Information* section can provide helpful information about which branches of code were executed if the `Trace.Write()` method was used by the programmers. The *Headers Collection* shows the type of browser and operating system used to access the page, which can be helpful in verifying the browsers your group identified as important are actually used in testing. The *QueryString Collection* makes it easy to see what values a form sent to the page currently being displayed.

Microsoft ACT 1.0

Purpose of Microsoft Application Center Test 1.0 – or ACT – is to stress test web servers and analyze performance and scalability problems with web applications. This includes ASP.NET applications and their components. This type of testing is accomplished by opening multiple connections to the server and rapidly sending HTTP requests, thereby simulating a large group of users. Although high-load stress testing over long periods of time is ACT's main purpose, it can also be used for functionality testing. Lastly, Application Center Test will work with any web server or web application that adheres to the HTTP protocol.

Putting Application Center Test into use on a testing project allows you to see how your web server reacts when several hundred users access your application at the same time. This simulates peak periods and not only provides performance and

scalability information, but also tests databases in regards to such issues as concurrency, transactions, number of users supported, locks, pooling, and so on.

This tool comes with Visual Studio .NET Enterprise Developer and can be used within the Visual Studio .NET's Integrated Development Environment (IDE), but more options are available when the stand-alone ACT program is used.

```
Sub SendRequest1()
    Dim oConnection, oRequest, oResponse, oHeaders, strStatusCode
    If fEnableDelays = True then Test.Sleep (0)
    Set oConnection = Test.CreateConnection("localhost", 80, false)
    If (oConnection is Nothing) Then
        Test.Trace "Error: Unable to create connection to localhost"
    Else
        Set oRequest = Test.CreateRequest
        oRequest.Path = "/duwamish7vb"
        oRequest.Verb = "GET"
        oRequest.HTTPVersion = "HTTP/1.0"
        set oHeaders = oRequest.Headers
        oHeaders.RemoveAll
        oHeaders.Add "Accept", "image/gif, image/x-xbitmap, " + _
             "image/jpeg, image/pjpeg, application/msword, " + _
             "application/vnd.ms-powerpoint, application/vnd.ms-excel, */*"
        oHeaders.Add "Accept-Language", "en-us"
        oHeaders.Add "User-Agent", "Mozilla/4.0 (compatible; " + _
             "MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)"
        'oHeaders.Add "Host", "localhost"
        oHeaders.Add "Host", "(automatic)"
        oHeaders.Add "Cookie", "(automatic)"
        Set oResponse = oConnection.Send(oRequest)
        If (oResponse is Nothing) Then
            Test.Trace "Error: Failed to receive response for URL to " + _
             "/duwamish7vb"
        Else
            strStatusCode = oResponse.ResultCode
        End If
        oConnection.Close
    End If
End Sub
```

Listing 2 *– Example of a single request generated by ACT 1.0's recorder.*

Listing 2 shows an example of a subroutine generated by ACT's recorder. When a script is generated, requests are broken up into individual subroutines that are then called one-by-one by a `Main()` subroutine.

Just like the rest of .NET, this application relies on an object-oriented approach and provides a number of objects that you can use. In Listing 2, note that we're looking at the `SendRequest1()` subroutine. This is the first routine generated when the webapp-under-test was navigated to. The line of interest is the one that says, `oRequest.Path = "/duwamish7vb"`. This is the root of the web directory or site we've selected for testing. The other pieces specify the type of request coming in (in this case "GET" instead of "POST" or "HEAD), the MIME types your browser declares that it will accept and understand, your browser information, and so on. (You'll be happy to know that cookie handling is built in to ACT 1.0). After completing the `Request` information, it is dispatched to the web site with a call to the

`Connection.Send()` method (`oConnection` is a Connection object, which was created when a call was made to `Test.CreateConnection` at the front part of the subroutine). The `oConnection.Send(oRequest)` call sends off the Request information that was filled out in the middle of the script. (Lots of stuff just to generate a single request, eh?)

Subsequent `SendRequest`*N*`()` routines are created (where *N* equals the next number in the sequence) based on what the displayed page relies upon. This includes a request for the cascading style sheet (/duwamish7vb/css/duwamish.css), images (bannerlogo.gif, bannerhome.gif, bannercart.gif, banneraccount.gif, line.gif, etc.), and the page itself (Default.aspx).

Finally, down around the 15th request, in the `SendRequest15()` routine, we get to the request that was generated when typing in a search string and clicking the submit button. The resulting request path looks something like this:

```
oRequest.Path = "/Duwamish7vb/searchresults.aspx" + _
"?type=0&fullType=Title&text=how+to+win+friends"
```

This is a direct call to the .aspx (ASP.NET) file using the `GET` method (the `GET` method causes the submitted form's values to be part of the URL, as opposed to `POST` which embeds the variables and their paired values into the HTTP header and goes unseen by the user). This call goes through the whole process previously described, sending off the above request to `searchresults.aspx` with the query string shown above, hoping for a reply in HTML by the server. The CSS file is downloaded as are the GIFs and JPEGs. Whew! Lot's of traffic going on just for a single request!

When all is said and done, 34 separate requests are generated, and all we did was:

1. Navigate to http://localhost/duwamish7vb
2. Type "how to win friends" into the search box
3. Clicked the "Go" button to submit the form
4. Clicked on the link of the book found by the query

The main routine that fires off each of these requests is simple enough, and shown (with some abbreviation) in **Listing 3**.

```
Sub Main()
    call SendRequest1()
    call SendRequest2()
    ': (3-33 removed for brevity)
    call SendRequest34()
End Sub
Main
```

Listing 3 *– Each Request is sent in turn by calls to their corresponding subroutines.*

Running the script is as simple as clicking a *Play* button found on the toolbar. As the test executes, the status can be viewed and looks similar to **Figure 3**. This status box communicates the current state (for example, "The test is now running"), time

elapsed, time remaining, average requests per second, and total requests made so far. It also lists the number of errors currently encountered allowing you to decide if the test needs to be aborted or allowed to continue.

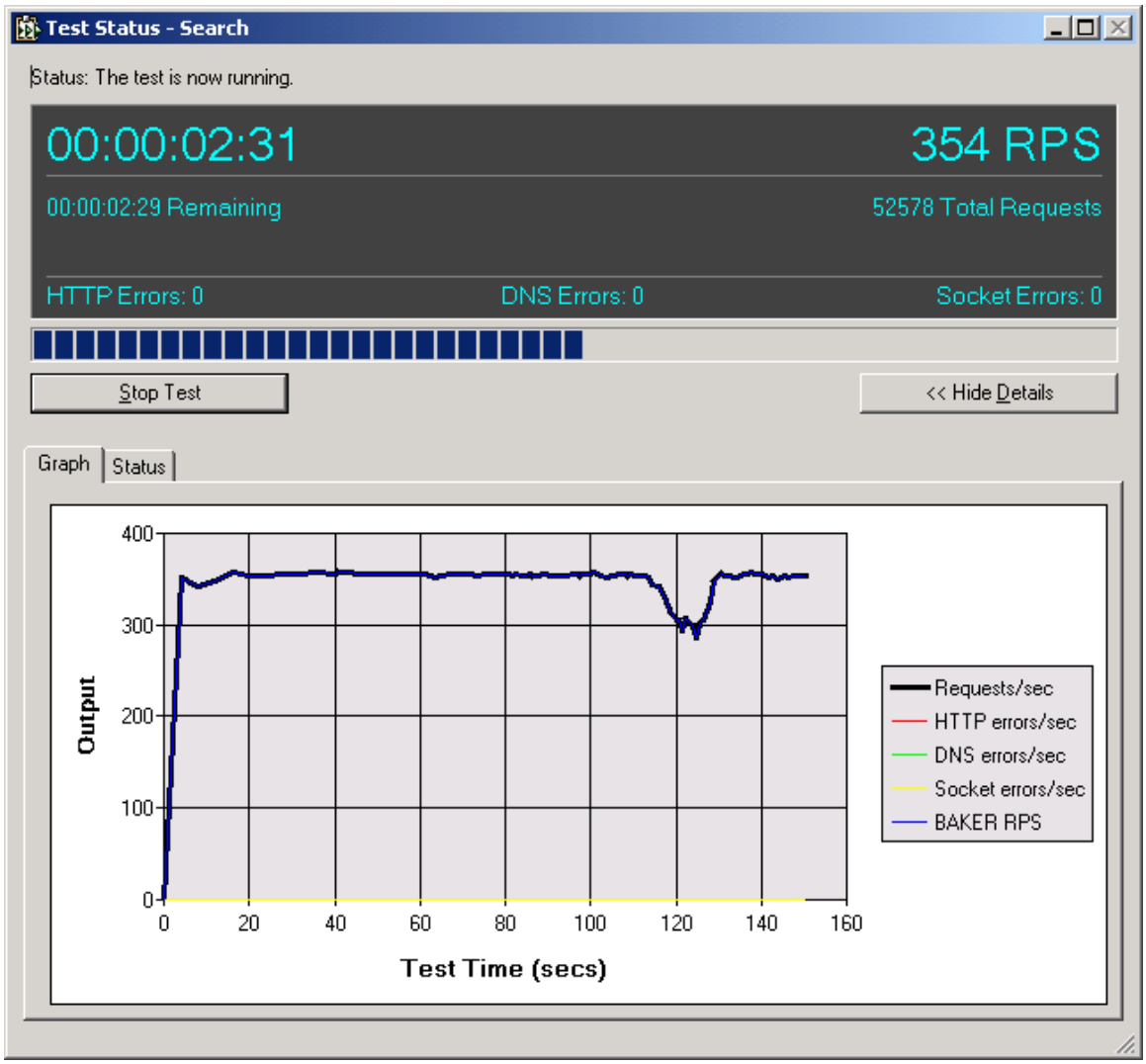


Figure 3 *– Execution of the ACT script halfway through its test run.*

At approximately 2 minutes and 5 seconds into the test run, I jumped to my browser and tried 3 searches, resulting in the dip in Requests Per Second (RPS) generated by the load test, shown in Figure 3. (Hey, the straight line was looking boring). This allowed me to see how peppy the site was even under load.

When a script completes, its results can be viewed by clicking on the *Results* object in the test project. The results are displayed in a list based on the name of the script being executed, and the time/date of its execution. Results show the total number of requests generated during the run (in this case, it was 5 minutes and generated 104,227 requests, an average of 347.42 requests per second). It also gives the test engineer an idea of how responsive the system was for those requests. Responsiveness is measured by tracking the "Average Time to First Byte" (how long until the server started sending back a response) and "Average Time to Last Byte" (total time

to send the requested item). Errors are also listed (HTTP, DNS and Socket), as are network statistics.

Considering my test example ran on the same computer as the web server, bandwidth was pretty good (1,227,864.16 bytes/second) on my 1.2ghz laptop. And last, response codes are tracked allowing us to understand the quality of the requests and responses. The return result of "200" communicates the request was received and a full response was returned. Using multiple computers on the network is one way to raise the number of requests per second experienced by the server. Getting a faster test machine with a big fat pipe direct to the server is another.

There is much more to discover about this program, including tracking server performance, test machine performance, and more. I've used it more in a recorder/playback capacity, but the language is based on VB Script and isn't difficult to master. There are only about 5 objects (with many methods each) that comprise the model: `Test`, `Request`, `Connection`, `User` and the `Response` object. Real-world use of this tool is likely to be: Record a few actions to generate a script, move the common bits into separate routines, and use constants based on the type of machine being used and browser being emulated. There is definitely room for structured programming to be put into practice to make it less bulky and more responsive to the evolution of a web site.

All in all, it's a great tool that provides a lot of capability, and it's "free" (as long as you have the full-blown version of Visual Studio .NET).

Other Considerations

There are other tools out on the market worth considering. These include *Red-Gate*'s new ANTS ("Advanced .NET Testing System") product, *Rational Software*'s new Java-based tool for testing Java and HTML applications, and *PushToTest*'s free open-source load & monitoring tools. These are new or less known, so I won't bother to list the other tool vendors and tools of which you are probably already aware.

- **ANTS** appears to be very similar to Microsoft Application Center Test 1.0 in what it provides. It can be used to generate HTTP requests as well as test Web Services through HTTP requests enriched with the SOAP protocol. They have a 14-day free trial that's worth a look (only allows 10 simultaneous connections in the demo, however). At present they're pricing their tool at around $2,000+.
- **Rational Software**'s new Java-based testing tool is very intriguing. Look for my white paper about it on the AutomationJunkies.com site in July when it releases. By taking advantage of an object-oriented language like Java, this new tool provides flexibility by treating each item on a web page or Java applet as an object. This allows the object to be modified in the script later if it is modified on the website or in the applet. Broken scripts due to UI changes are fixed more quickly (a common automation nightmare). In addition, this new tool has a level of intelligence built in allowing it to weigh the likelihood that a modified control is still the control it wants to interact with, further improving maintainability.
- **PushToTest** has an automation tool whose UI is in Java – allowing it to work anywhere – and the scripting language is based on *Python*, a language that's easier to learn than C or C++. Using the Java framework allows test engineers to

deploy their Python test scripts and generate loads against their server under test. Best of all, it's open source, so you have the code base and are free to modify its functionality and capabilities to your heart's content.

Summary

This is a lot of information to absorb, to be sure. We started by looking at the .NET Framework and getting a 30,000-foot view of what it is and some of the things it has to offer. To continue learning about .NET I suggest visiting such sites as **GotDotNet.com** and **DotNetJunkies.com**. These are two great sites for tutorials and other information about working in .NET. Unfortunately, they're mostly programmer-centric, so you should also check out **StickyMinds.com** and **AutomationJunkies.com**. Be sure to also visit **QAForums.com** for a great list of discussion groups.

The next thing we looked at were some of the challenges development teams face – concerns to programmers and testers alike – when moving into the .NET realm. These include the modification of variable types between VB6 and VB.NET, the placement of functions within the .NET object model, and how .NET does some handholding, which could prove to be problematic when it comes to tweaking how a page is displayed in different browsers. (This could be especially problematic to automation tools that rely on specific controls to be in place and don't tie themselves to a single browser for testing).

We then looked at an ASP.NET example and a tool that comes with Microsoft Visual Studio .NET (Application Center Test 1.0), as well as intrinsic support for ASP.NET deployed applications (the `<trace>` tag in `web.config`). Let's not forget the other new tools that are coming to market to help support .NET testing, as well as new tools that weren't necessarily targeting .NET, but can be used regardless.

This paper scratches the surface, but still provides you with a strong starting point with some of the options available to you when testing in the .NET maze. Good luck and have fun!

# # #

Please direct questions or comments to Tom Arnold at tom@xtenddev.com.

JUnit Extensions for Documentation and Inheritance

Sarah Wilkin and Daniel Hoffman
Department of Computer Science
University of Victoria
PO Box 3055 STN CSC
Victoria, B.C. V8W 3P6, Canada,
orque@uvic.ca
dhoffman@csr.uvic.ca

Abstract

While object oriented programming offers great solutions for today's software developers, this success has created difficult problems in class documentation and testing. In Java, Javadoc helps by allowing class interface documentation to be embedded as code comments and JUnit supports unit testing by providing assert constructs and a test framework. This paper describes JUnitDoc, an integration of Javadoc and JUnit which provides better support for class documentation and testing. With JUnitDoc, test cases are embedded in Javadoc comments and used as both examples for documentation and test cases for quality assurance. Using the Javadoc doclet and taglet features, JUnitDoc extracts the test cases for use in HTML files serving as class documentation and in JUnit drivers for class testing. JUnitDoc uses doclets to traverse the class hierarchy and accumulate test cases for each method, creating a test hierarchy that parallels the inheritance class hierarchy.

Biographies

Sarah Wilkin is a graduate student at the University of Victoria, with research interests in user interfaces, and software documentation and testing.

Dr. Daniel Hoffman received the B.A. degree in mathematics from the State University of New York, Binghamton, in 1974, and the M.S. and Ph.D. degrees in Computer Science in 1981 and 1984, from the University of North Carolina, Chapel Hill. From 1974 to 1979 he worked as a commercial programmer/analyst. He is currently an Associate Professor of Computer Science at the University of Victoria His research area is Software Engineering, emphasizing the industrial application of software documentation, inspection, and testing.

JUnit Extensions for Documentation and Inheritance

Sarah Wilkin* Daniel Hoffman†

Abstract

While object oriented programming offers great solutions for today's software developers, this success has created difficult problems in class documentation and testing. In Java, Javadoc helps by allowing class interface documentation to be embedded as code comments and JUnit supports unit testing by providing assert constructs and a test framework. This paper describes JUnitDoc, an integration of Javadoc and JUnit which provides better support for class documentation and testing. With JUnitDoc, test cases are embedded in Javadoc comments and used as both examples for documentation and test cases for quality assurance. Using the Javadoc doclet and taglet features, JUnitDoc extracts the test cases for use in HTML files serving as class documentation and in JUnit drivers for class testing. JUnitDoc uses doclets to traverse the class hierarchy and accumulate test cases for each method, creating a test hierarchy that parallels the inheritance class hierarchy.

1 Introduction

In 1968, McIlroy proposed a software industry based on reusable components, serving roughly the same role that VLSI chips do in the hardware industry [14]. After 30 years, McIlroy's vision is becoming a reality. Components in the form of class libraries now exist for many object-oriented languages. While class libraries provide valuable solutions, their use gives rise to difficult problems in both documentation and testing.

Class libraries and frameworks provide large and complex APIs, making effective documentation essential for successful use. While the method names and prototypes are expressed in the implementation language, the method behaviour must be documented as well. Typically, this is done with brief prose descriptions, focusing on the situations that commonly arise in API use. Such documentation is inevitably imprecise and incomplete, leading to costly misunderstandings between API implementors and API users. The formal methods community recommends mathematically precise specifications, because they can be complete and unambiguous. Unfortunately such specifications are expensive to write and maintain. Worse, few developers are willing or able to read formal specifications.

Traditionally, unit testing has focused on individual software functions. With the advent of object-oriented technology, the unit-under-test has become the class. Inheritance provides the developer with a powerful tool for factoring out redundant code: put the common code in a superclass and handle special cases in subclasses. The resulting class hierarchies make life difficult for the tester. Ideally, the tester would develop a test hierarchy paralleling the class hierarchy. Unfortunately, this is difficult to accomplish manually and there is no tool support available.

*Dept. of Computer Science, University of Victoria, PO Box 3055 STN CSC, Victoria, B.C. V8W 3P6, Canada, orque@uvic.ca

†Dept. of Computer Science, University of Victoria, PO Box 3055 STN CSC, Victoria, B.C. V8W 3P6, Canada, dhoffman@csr.uvic.ca

```
public interface List extends Collection {
    void add(int index, Object element);
    List subList(int fromIndex, int toIndex);
    ...
}

public class LinkedList implements List {
    public Object removeFirst() {...}
    public void add(int index, Object element) {...}
    ...
}

public class ArrayList implements List { ...  }
```

Figure 1: `List`, `LinkedList`, and `ArrayList` method prototypes

JUnitDoc provides an elegant solution to the problems of class documentation and testing by integrating JUnit and Javadoc. The underlying idea is simple: augment traditional prose documentation with test cases designed specifically for use in documentation. Typically, there are a few cases for each likely question about API behavior. In practice, the test cases serve roughly the same role that FAQs ("frequently asked questions") do on many web sites. The test cases are embedded in the code as Javadoc comments. Using the Javadoc doclet and taglet features, JUnitDoc extracts the test cases in two forms: (1) Javadoc HTML files for class documentation and (2) JUnit drivers for class testing. JUnitDoc provides support for inheritance testing by using doclets to traverse the class hierarchy and accumulate test cases for each method.

2 Driving example: the `List` family

The `Collection` framework in `java.util` provides a way to group multiple objects in a single object. We use one small branch of the elaborate `Collection` hierarchy, starting at the `List` interface.

The `List` interface is used for indexed collections. Elements can be inserted or retrieved at a particular position or searched for by value. Our examples use two methods from `List`: `add` and `subList` (shown in Figure 1). `Add(`*i*`,`*e*`)` allows the user to insert *e* at index *i*, shifting subsequent elements to the right. `SubList(`*begin*`,`*end*`)` retrieves the sublist from index *begin* to index *end*. Some users are surprised to learn the element at *begin* is included in the sublist but *end* is not. The same open/closed index behaviour is found in other methods such as `StringBuffer.replace`.

In the Java API, `LinkedList` and `ArrayList` are subclass implementations of `List`. Both subclasses implement all `List` methods and also adds methods related to their internal operation. `LinkedList` supports specialized access to the first and last elements and `ArrayList` manipulates the internal array size. Our examples use the `LinkedList` methods shown in Figure 1. `LinkedList.add` implements the specification of `List.add` and allows for null elements. `LinkedList.removeFirst` removes and returns the first element from the list.

3 Tool support

3.1 Javadoc, taglets, and doclets

Javadoc is a powerful tool for generating class documentation from code comments. Javadoc comments begin with "`/**`" (rather than just "`/*`") and can precede each class, interface, field, and method. The Javadoc tool generates HTML documentation containing the declarative portions of the code with the comments interleaved. Javadoc-generated HTML is now the accepted means for documenting both the standard libraries and custom code. Typically, only the public classes, fields and methods are documented, for readers who wish to use the class but are not interested in the implementation internals.

To control the appearance of the generated documentation, Javadoc comments typically contain HTML commands. Because these commands are often repetitive, Javadoc provides "tags": a mechanism for generating the HTML for commonly occurring items. A tag is identified by a leading "`@`". For example, the `@param` and `@return` tags generate HTML to format method parameter and return value comments. The Javadoc taglet API allows a user to introduce new tags and to write Java code to generate the desired HTML commands.

By default, Javadoc generates HTML. Some users prefer a different format, e.g., rich text format or XML. The doclet API gives users access to the class, field, and method declarations of a Java file, and to the Javadoc comments associated with each tag.

3.2 `GenDoc` and `GenDriver`

`GenDoc` is a taglet that appends `@testcases` test cases to their associated methods. `GenDriver` s a doclet that takes `@testcases` tags and writes a JUnit test driver with their contents.

With both tools, the `@testcases` tag is used to identify statements to be processed. The tools extract every statement following `@testcases`: one copy is modified for use in the driver and another in the documentation. Test templates are provided as an option to supplement test statements. Using templates results in more robust test cases and readable documentation. To demonstrate JUnitDoc we add test cases to two related classes in the Java standard library.

Adding test cases to a class is straightforward. Statements applicable to the whole class are written above the class declaration; we call this a "class block." Similarly, tests for each method are written directly above the method declaration in a "method block." When multiple `@testcases` tags are used for a single class or method they are still considered one block. Most unit testing tools make the user write and maintain a separate test file; a single change is cumbersome. As shown in Figure 2a, JUnitDoc test cases are edited in the same place as their associated method. The proximity makes maintenance simple and encourages small, frequent updates.

`GenDoc` writes each block under a "Test cases" heading. Class blocks display at the end of the class overview; method blocks display directly below all other documentation for the method.

`GenDriver` extracts test cases and generates a JUnit driver, as shown in Figure 2b. For every method *m(parameterList)*, `GenDriver` extracts the test cases and writes a test method *testm_parameterList_* with them. The order of test methods is identical to the order of source methods. For example, in `LinkedList` in Figure 2a, the `addFirst` cases are written before the `add` cases. The generated driver of Figure 2b shows this order. When a superclass or super-interface has method or class blocks, `GenDriver` extracts and adds them to the driver. All blocks with the same method signature are gathered and written in one test method, with superclass blocks preceding subclass blocks. Since the entire class/interface hierarchy is traversed, every ancestor test case will be included. In Figure 2, the test method for `addFirst` is written first, followed by `add` with test cases from `List` then `LinkedList`.

```
public interface List {
    /**
     * @testcases
     *     list.add("a");
     *     assertTrue(list.contains("a"));
     */
    boolean add(Object o);
    ...
}
/**
 * @testcases
 *     List list = new LinkedList();
 *     LinkedList llist = new LinkedList();
 */
public class LinkedList implements List {
    /**
     * @testcases
     *     llist.addFirst("first");
     *     assertEquals((String)llist.get(0), "first");
     */
    public void addFirst(Object o) {...}
    /**
     * @testcases
     *     llist.add("b");
     *     assertTrue(llist.contains("b"));
     */
    public boolean add(Object o) {...}
    ...
}
```

(a) Test cases

```
...
public class LinkedListTest extends TestCase {
    ...
    List list = new LinkedList();
    LinkedList llist = new LinkedList();

    public void testaddFirst_java_lang_Object_() {
        llist.addFirst("first");
        assertEquals((String)llist.get(0), "first");
    }
    public void testadd_java_lang_Object_() {
        list.add("a"); assertTrue(list.contains("a"));
        llist.add("b"); assertTrue(llist.contains("b"));
    }
}
```

(b) Extract of generated driver for `LinkedList`

Figure 2: Test cases and generated driver

Extracted class blocks are written before test methods in the generated driver; therefore, the class block in the driver is global to the test methods. In Figure 2a, the class block of `LinkedList` shows creation and initialization of two variables used in the method test cases. Note the placement of this block in Figure 2b. When a class's superclass or interface has class blocks they are added before the child's blocks.

JUnitDoc provides a command line option to override ancestor test cases. When enabled, only ancestor methods with the same signature of methods in the class under test are overridden; other ancestor test cases are still inherited.

3.3 Test case templates

While the JUnit assert constructs are useful they have weaknesses regarding exception handling. First, exceptions that occur during evaluation of an assert will cause termination of the driver; sometimes it is important that the driver continue executing. Second, and more important, there is no support for test cases that are supposed to produce an exception, e.g., a call to `add` with index -1. JUnitDoc provides test case templates to handle situations where better exception support is needed.

Test case templates are embedded in Javadoc comments and are identified by keywords preceded by the `#` character, tagged for manipulation by the JUnitDoc preprocessor. There are two types of templates: value-checking and exception-monitoring. The form of a value-checking test case is:

`#valueCheck` *actualValue* `#` *expectedValue* `#end`

where *actualValue* and *expectedValue* are expressions of the same type. For such a test case template, JUnitDoc generates code to compare *actualValue* and *expectedValue*, while monitoring the exception behaviour. An error message is displayed if *actualValue* and *expectedValue* are different or if an exception is thrown during the comparison. In either case, driver execution continues.

The general form of an exception-monitoring test case is:

`#excMonitor` *action* `#` *expectedException* `#end`

where *action* is a fragment of Java code and *expectedException* is a Java exception object. The generated code executes *action*, while monitoring the exception behavior. An error message is displayed if *expectedException* is not thrown or if another exception is thrown. In an exception-monitoring template *expectedException* can be omitted, in which case an error message is printed if any exception is thrown.

To add flexibility, JUnitDoc supports user-defined comparison for `#valueCheck` templates. For a standard type, such as `int` or `String`, JUnitDoc uses its own comparison routine to compare *actualValue* and *expectedValue*. The tester can override this comparison routine, or provide a comparison routine for a new type, by specifying a *type* field in a value-checking test case. The tester then has to provide one method for comparing values of that type and another for printing an error message if the two values are not equal. This can, for example, be used to provide a case-insensitive comparison for strings (the normal comparison is case-sensitive). JUnitDoc also supports user-defined comparison for `#excMonitor` templates. By default, two exceptions are considered equal if the exception objects are of the same type. As with value-checking templates, the tester can override this comparison and provide a custom comparison routine.

3.4 Discussion

Writing test cases with JUnitDoc is fast and easy; running the driver produced by `GenDriver` rewarding. With `GenDoc`, documentation is kept accurate and up to date. In addition, if the order and scope of `@testcases` blocks are kept in mind, the testing framework gains flexibility and power.

A class block is designed for helper methods and variables. For example, those familiar with JUnit may want to put `setUp` and `tearDown` methods here.[1] An object shared and manipulated between tests can also be useful.

Inherited cases provide a way to build up a test method: a setup written high in the hierarchy can be extended and refined lower down. As a result, fundamental test cases must be passed first.

By writing tests for abstract methods and interfaces a strong contract results: descendants have to provide the correct minimal functionality. Without writing a single new line of test code, a descendent class already has a whole suite of tests to pass. This supports the popular XP convention of writing test cases before implementation.

When using interfaces with JUnitDoc a few extra lines of code are required. Since interfaces cannot be instantiated, a concrete implementation must be instantiated instead. In its test cases an interface assumes the existence of variables it needs. For example, `List.add` in Figure 2a uses a variable called `list`. This variable is not declared in `List`, but in the class block of `LinkedList`.

Test templates distill test cases into a short, readable form. Without having to run the cases or be burdened by implementation details, action-result pairs show a documentation reader how a method works. A similar benefit exists for the test writer: she can focus on specifying the correct behaviour of her method instead of writing supplementary code to deal with the unexpected.

4 List class case study

To illustrate the practical application of JUnitDoc we provide a short case study using `List` and `LinkedList`. The test cases for `List` and `LinkedList` are shown in Figure 3 and Figure 4, respectively.

We have taken a question and answer approach to writing test cases. Without burdening the documentation reader or test writer, this style covers the majority of cases. Each question is answered by a concise set of examples and counter-examples.

In an inheritance structure we add test cases starting at the root. Consider the `List` interface shown in Figure 3. As an interface, variables of the interface type must be instantiated with a class that implements the `List` interface. It is useful to declare all of these variables in one location: the class block. The commented lines become part of the `List` documentation, so implementing classes simply copy and paste the variable declaration lines, filling in their class name where appropriate. We also create exception objects in the class block to shorten test cases.

The cases for `List.add` test the bounds of its index. First we set up the list by adding two elements. Valid indexes for adding to a list start at 0, so we test one past the bound by adding at -1. An `#excMonitor` template wrapping the call shows a `BoundsException` is expected. Next the extremes of valid `add` indexes are tested: 0 and `list.size`. After adding, we remove the object in a `#valueCheck` template to check it has the correct value. Adding at `list.size+1` is tested identically to the -1 test. Finally, we return the list to its original state by clearing it.

Tests for `List.subList` are similar to those of `add`, but with two indexes to consider. The generated documentation for this method is shown in Figure 5. Valid lists are returned when

[1] JUnit runs the `setUp()` method before every test method and the `tearDown()` method after

```
 * ...
 * @testcases
 * // required variables:
 * // List list = new [implementation]();
 * Exception BoundsException = new IndexOutOfBoundsException();
 * Exception NoElementException = new java.util.NoSuchElementException();
 */
public interface List {
    ...
     * @testcases
     * // Where can we add?
     *     list.add("0,0");
     *     list.add("1,1");
     *     #excMonitor list.add(-1,"a"); # BoundsException #end
     *     list.add(0,"a");
     *     #valueCheck list.remove(0) # "a" #end
     *     list.add(list.size(),"a");
     *     #valueCheck list.remove(list.size()-1) # "a" #end
     *     #excMonitor list.add(list.size()+1,"a"); # BoundsException #end
     *     list.clear();
     */
    void add(int index, Object element);
    ...

    ...
     * @testcases
     * // What are legal values for fromIndex?
     *     list.add(0,"0");
     *     list.add(1,"1");
     *     #excMonitor list.subList(-1,1); # BoundsException #end
     *     #valueCheck list.subList(0,1).size() # 1 #end
     *     #valueCheck list.subList(0,1).get(0) # "0" #end
     *
     * // What are legal values for toIndex?
     *     #valueCheck list.subList(1,list.size()).get(0) # "1" #end
     *     #excMonitor list.subList(1,list.size()+1); # BoundsException #end
     *
     * // Can fromIndex be greater than toIndex?
     *     #excMonitor list.subList(1,0); # BoundsException #end
     *     list.clear();
     */
    List subList(int fromIndex, int toIndex);
}
```

Figure 3: Test cases for `List`

```
 * ...
 * @testcases
 *     List list = new LinkedList();
 *     LinkedList llist = new LinkedList();
 */
public class LinkedList implements List {
    ...
     * @testcases
     * // What can we legally remove?
     *     llist.add(0,"0");
     *     llist.add(1,"1");
     *     #valueCheck llist.removeFirst() # "0" #end
     *     #valueCheck llist.removeFirst() # "1" #end
     *     #excMonitor llist.removeFirst(); # NoElementException #end
     */
    public Object removeFirst() {
        ...
    }
    ...

    ...
     * @testcases
     * // Can null be added to the list?
     *     #excMonitor llist.add(0,null); #end
     *     llist.clear();
     */
    public void add(int index, Object element) {
        ...
    }
    ...
}
```

Figure 4: Test cases for `LinkedList`

`fromIndex` is less than `toIndex`, so we test the left bounds of `fromIndex` and the right bounds of `toIndex`. We end our tests with the illegal case, when `fromIndex` is greater than `toIndex`. Constructing our test from the prose documentation supplied by Sun, we were surprised when an `IllegalArgumentException` was thrown instead of an `IndexOutOfBoundsException`. Many similar inaccuracies are found in plain text documentation. With test cases, discrepancies can be revealed automatically by running the cases.

An interface must have at least one implementor for its cases to run. `LinkedList` implements `List`, so it instantiates variables required for `List` tests. Shown in Figure 4, the variable declarations need to go in `LinkedList`'s class block to be available to `List` tests.

Every test written for `List` is inherited by `LinkedList`, so we only write tests for new methods and new functionality in an existing method. One new method is `removeFirst`. To check that the object removed was the first in the list, we add two objects then make the `removeFirst` call in a `#valueCheck` template. The call is repeated, testing removal of the remaining object. With the list empty, the third call to `removeFirst` throws a `NoElementException`.

`LinkedList.add` supplements the cases from `List.add` by adding a test for `null` elements. We show `null` can be added to a `LinkedList` by leaving out an expected exception in the `#excMonitor`; any thrown exception causes the case to fail.

Though Figures 3 and 4 demonstrate a benefit from inherited test cases, the advantage is more evident when we consider writing tests for all methods. `List` specifies twenty-five method interfaces; `ArrayList` and `LinkedList` only add three and seven new methods respectively. Therefore, the test code that must be maintained is considerably reduced. In addition, any new implementing classes can use the parent test cases by adding at most a few lines of code.

4.1 Discussion

When writing test cases, either JUnit methods or test templates may be used. From a coding perspective, the difference between writing a value check using templates or JUnit is negligible. Consider the second `#valueCheck` in `Lists.subList` in Figure 3. Rewritten in JUnit syntax, the line becomes:

```
assertEquals(list.subList(0,1).get(0), "0");
```

Expected and actual values are reported in a similar manner; however, if an exception occurs during the check they follow different routes. When using JUnit, the exception is reported and all further tests aborted. With templates, the exception is reported but the remaining test cases are run. Further, JUnit's `assert` statements assume all classes provide `equals` and `toString` methods. Without them, objects are compared by reference and printed as a class name plus object number. Even if provided, the method of comparison may be undesirable for testing. With test templates, a custom print or comparison routine can be written for any class while leaving the original class unmodified.

Exception checking is handled differently between templates and JUnit. Figure 6a shows the JUnit equivalent of the first `#excMonitor` template of `List.subList` in Figure 3. At eight lines, the JUnit equivalent is unsuitable as documentation. The first issue addressed in the JUnit FAQ [13], *"How do I implement a test case for a thrown exception?"* suggests two other methods. The first is shown in Figure 6b. With this method the failure text must be maintained to reflect the exception being checked and the failure message omits the name of the actual exception thrown. Also, if an exception is thrown other than the one expected, all further test cases are aborted. With these deficiencies the comment after the second method is disappointing: *"Looking at this again, the first way is simpler. Sigh..."* In other words, neither method is good.

subList

```
public List subList(int fromIndex,
                    int toIndex)
```

Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive. (If `fromIndex` and `toIndex` are equal, the returned list is empty.) The returned list is backed by this list, so changes in the returned list are reflected in this list, and vice-versa. The returned list supports all of the optional list operations supported by this list.

This method eliminates the need for explicit range operations (of the sort that commonly exist for arrays). Any operation that expects a list can be used as a range operation by passing a subList view instead of a whole list. For example, the following idiom removes a range of elements from a list:

```
list.subList(from, to).clear();
```

Similar idioms may be constructed for `indexOf` and `lastIndexOf`, and all of the algorithms in the `Collections` class can be applied to a subList.

The semantics of this list returned by this method become undefined if the backing list (i.e., this list) is *structurally modified* in any way other than via the returned list. (Structural modifications are those that change the size of this list, or otherwise perturb it in such a fashion that iterations in progress may yield incorrect results.)

Parameters:

`fromIndex` - low endpoint (inclusive) of the subList.
`toIndex` - high endpoint (exclusive) of the subList.

Returns:

a view of the specified range within this list.

Throws:

`java.lang.IndexOutOfBoundsException` - for an illegal endpoint index value (fromIndex < 0 || toIndex > size || fromIndex > toIndex).

Test Cases:

```
// What are legal values for fromIndex?
list.add("0");
list.add("1");
#excMonitor list.subList(-1,1); # BoundsException #end
#valueCheck list.subList(0,1).size() # 1 #end
#valueCheck list.subList(0,1).get(0) # "0" #end

// What are legal values for toIndex?
#valueCheck list.subList(1,list.size()).get(0) # "1" #end
#excMonitor list.subList(1,list.size()+1); # BoundsException #end

// Can fromIndex be greater than toIndex?
#excMonitor list.subList(1,0); # BoundsException() #end
list.clear();
```

Figure 5: Documentation for `List.subList`

```
try {
    list.subList(-1,1);
} catch (Throwable exception) {
    if (BoundsException.getClass() != exception.getClass()) {
        fail("Actual exception: "+exception.getClass().getName()+
            " Expected exception: "+BoundsException.getClass().getName());
    }
}
```

(a) Template equivalent

```
try {
    list.subList(-1,1);
    fail("Should raise an IndexOutOfBoundsException");
} catch(IndexOutOfBoundsException e) {
    // pass
}
```

(b) JUnit FAQ suggested style

Figure 6: Testing for an exception in JUnit

The `#excMonitor` template provides two other benefits. First, code wrapped in an `#excMonitor` is enclosed in a `try-catch` block when the template expands. This allows the user to write robust cases with a minimal amount of code. Second, templates can use custom exception comparison and printing. For example, equivalence testing on messages or other parameters passed with an exception is useful. Also, if a method ambiguously throws different exceptions they could be considered the same.

5 Related work

JUnitDoc is based on the JUnit tool for test automation [4] and the Javadoc tool for embedded documentation [18], and on the considerable previous work in unit test automation. Early work by Panzl [17] provided tools for the regression testing of Fortran subroutines. The PGMGEN system [6] generates drivers for C modules from test scripts and the Protest system [7] automates the testing of C modules using test scripts written in Prolog. The ACE tool [16] supports the testing of Eiffel and C++ classes, and was used extensively for generating drivers for communications software. The ClassBench framework [8] generates tests by automatically traversing a *testgraph*: a graph representing selected states and transitions of the class under test.

The use of examples in documentation is an old idea. In previous papers we explored the idea of using test cases as examples in documentation [5], and introduced the test templates described in Section 2 [3]. Today, use cases [12] are probably the best known technique for software documentation based on examples. While use cases are usually informal and not executable, they can be made executable, as research on SCR requirements documents has shown [15]. Our test cases can be thought of as executable API use cases. Hsia et al. present a systematic, formal method for scenario analysis that supports requirements analysis and change, and acceptance testing [10]. The method is extended to serve as a starting point for a formal model for scenario-based acceptance testing [9, 11]. The systematic approach allows a set of complete and consistent scenarios to be derived for acceptance testing. Similarly, Chang et al. describe a method for generating test scenarios for integration and system testing from formal, Object-Z specifications

and usage profiles [1, 2].

6 Conclusions

The success of object oriented programming has introduced difficult problems as well. It is hard to write class documentation that is both precise and readable, and to code test suites without significant repetition when testing class hierarchies.

JUnitDoc addresses these problems by integrating JUnit and Javadoc. Test cases serve as documentation examples. Consistency between the examples and the class implementation is checked automatically by running the generated drivers. Test cases coded in a superclass are automatically extracted in subclass drivers.

The resulting documentation is easily read and written by today's developers and is quite precise: the test cases are formal, albeit partial, specifications. JUnitDoc supports a test hierarchy that parallels the inheritance hierarchy, minimizing repetition in test cases.

References

[1] K.H. Chang, S-S. Liao, S.B. Seidman, and R. Chapman. Testing object-oriented programs: from formal specification to test scenario generation. *The Journal of Systems and Software*, 42(2):141–151, 1998.

[2] C-Y. Chen, K.H. Chang, and R. Chapman. Test scenario and test case generation based on Object-Z formal specification. In *Proceedings of SEKE'99*, pages 207–211, 1999.

[3] N. Daley, D. Hoffman, and P. Strooper. A framework for table-driven testing of Java classes. *Software - Practice and Experience*, 32(5):465–493, April 2002.

[4] M. Fowler. *Refactoring – Improving the Design of Existing Code*, chapter 4: Building Tests. Addison-Wesley, 1999.

[5] D. Hoffman and P. Strooper. Prose + test cases = specifications. In *Proceedings* 34^{th} *International Conference on Technology of Object-Oriented Languages and Systems*, pages 239–250. IEEE Computer Society, 2000.

[6] D.M. Hoffman. A CASE study in module testing. In *Proc. Conf. Software Maintenance*, pages 100–105. IEEE Computer Society, October 1989.

[7] D.M. Hoffman and P.A. Strooper. Automated module testing in Prolog. *IEEE Trans. Soft. Eng.*, 17(9):933–942, September 1991.

[8] D.M. Hoffman and P.A. Strooper. Classbench: A framework for automated class testing. *Software Practice and Experience*, 27(5):573–597, 1997.

[9] P. Hsia, J. Gao, J. Samuel, D. Kung, Y. Toyoshima, and C. Chen. Behavior-based acceptance testing of software systems: A formal scenario approach. In *Proceedings of COMPSAC'94*, pages 293–298. IEEE Computer Society Press, 1994.

[10] P. Hsia, J. Gao, J. Samuel, D. Kung, Y. Toyoshima, and C. Chen. A formal approach to scenario analysis. *IEEE Software*, 11(2):33–41, 1994.

[11] P. Hsia, D. Kung, and C. Sell. Software requirements and acceptance testing. *Annals of Software Engineering*, 3:291–317, 1997.

[12] I. Jacobsen. *Object-Oriented Software Engineering.* Addison-Wesley, New York, 1992.

[13] JUnit, http://junit.sourceforge.net/doc/faq/faq.htm. *JUnit Frequently Asked Questions.*

[14] M.D. McIlroy. Mass-produced software components. In J.M. Buxton, P. Naur, and B. Randell, editors, *Software Engineering Concepts and Techniques (1968 NATO Conf. on Software Engineering)*, pages 88–98, 1976.

[15] S. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *2nd ACM Workshop on Formal Methods in Softwre Practice*, 1998.

[16] G. Murphy, P. Townsend, and P.S. Wong. Experiences with cluster and class testing. *Commun. ACM*, 37(9):39–47, 1994.

[17] D.J. Panzl. A language for specifying software tests. In *Proc. AFIPS Natl. Comp. Conf.*, pages 609–619. AFIPS, 1978.

[18] Sun Microsystems, http://java.sun.com/j2se/javadoc/index.html. *Javadoc Tool*, 2002.

Agile Testing. What Is It? Can It Work?

Bret Pettichord, Pettichord Consulting

Abstract

People are talking about Extreme Programming and other agile development methodologies that favor speed over process. Do they provide better ways to deliver software? Or are they simply rationalizations for hacking? Bret Pettichord will review these trends and discuss his experiences working as a tester on agile projects. He'll present what testers can learn from agile techniques and values regardless of whether they are working on agile projects themselves.

Should responsible quality professionals try to adapt to these new methods? Or should they refuse to cooperate with this madness? Pettichord believes they present a unique opportunity to develop a practice for agile testing that uses tests as a way of communicating intention and expectation throughout the project team. Agile methods currently say little about the role of the tester.

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 Segue Software.

Traceability 101

Debra Schratz

Abstract

Have you ever delivered a software program that did "ALMOST" everything the customer wanted? One of the major problems for software development teams is managing the constant changes to software requirements. Keeping an eye on changes and ensuring that the product requirements trace from the customer's need, to the requirements, and then down to the work products in both a forward and backward direction, is not a trivial task.

This paper will describe what is traceability, why is tracing requirements important (benefits), and what happens if you don't do requirements traceability.

In addition, this paper will offer suggestions on how to get started and how to gain acceptance from your software development team to implement requirements traceability so you will deliver products that do EVERYTHING the customer wants!

Biographical Sketch

Debra Schratz has 6 years experience in quality engineering. Debra currently works as a Platform Quality Engineer in the Platform Quality Methods group, part of the Corporate Quality Network at Intel Corporation. Since January 2000, Debra has delivered over 200 training classes on requirements engineering for Intel worldwide. Prior to her work in quality, Debra spent 8 years managing an IT department responsible for a 500+ node network for ADC Telecommunications. Debra is a member of the Rose City SPIN Steering Committee. She holds a Bachelor's of Arts degree in Management with an emphasis on Industrial Relations.

Email address: Debra.Schratz@intel.com

Mailing address: 2111 NE 25th Ave, JF1-46, Hillsboro, OR 97124

Traceability 101

Have you ever had any of these experiences?

- Delivered a system that did “almost” everything the customer wanted?
- Development knew what to code first, based on what was most “important” to the customer/end user?
- You were “pretty sure” the major defects were fixed before you released the product?
- Changes to requirements were “understood and managed” effectively?

One of the key problems for software development teams is managing the continual changes to software requirements. Keeping an eye on changes and ensuring that the product requirements trace from the customer's need, to the requirements, and then down to the work products in both a forward and backward direction, is not a simple task.

This paper will describe what is traceability, why is tracing requirements important, and what happens if you don't do requirements traceability. This paper will offer suggestions on how to get started and how to gain acceptance from your software development team to implement requirements traceability so you will deliver products that do **EVERYTHING** the customer wants!

What is traceability?

There are numerous ways to define traceability. A good place to start is with the term “traceability item”. A traceability item is “Any textual, or model item, which needs to be explicitly traced from another textual, or model item, in order to keep track of the dependencies between them” [Spence & Probasco 98]

A common definition of traceability item is all “project artifacts” such as requirements, use cases, design documents, test cases, documentation, etc.

The IEEE definition of traceability is: “The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another”. [IEEE 90]
The most common definition of traceability is “the ability to describe and follow the life of a requirement, in both a forward and backward direction.” [Gotel & Finkelstein 94]. See Figure 1 below to show the different types of traceability.

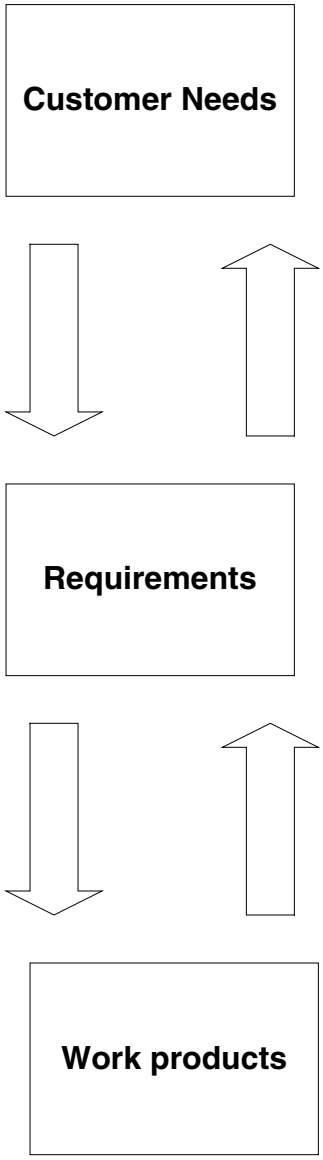


Figure 1

Following a requirement from its origin (from the customer) all the way to when it is implemented is not a minor undertaking! For example: Tracing a requirement from its origin, through development and specification, to its subsequent deployment and use, and through the on-going refinement and iteration through out all phases of development. Most development teams don't understand the various types of traceability until they go to test their products, then it can be too late.

There are four types of traceability "links" that ensure traceability:

1. Forward to requirements: This link maps the customer's needs to the requirement so when a change is made to the vision of the project or system, all requirements can be reviewed to understand the impact of the change [Jarke 98].
2. Backward from requirements: This link verifies the system meets the needs of the customer. Looking backward from the requirements to the customer's needs allows the development team to determine whether or not the customer needs have or have not been met.
3. Forward from requirements: This link evaluates the components built from requirements have the correct requirements assigned to them [Jarke 98]. For example, this is very helpful for a team to be able to re-evaluate the impact of a change made at a high level requirement to the lower level requirements that were derived or have dependencies on that requirement. The purpose of this link is to guarantee any changes made are communicated and evaluated throughout development.
4. Backward to requirements: This link ensures no "gold plating" has been done. By going backwards from each design element to it's original requirement guarantees no unnecessary features have been built.

Maintaining all upstream and downstream linkages allows a project team to quickly and easily understands the scope of any changes. "Traceability has the most value in a methodology when the links are bi-directional" [Ecklunch, Delcambre & Freiling 96]

Why are tracing requirements important?

Requirements tracing helps the project team to understand which parts of the design and code implement the user's requirements, and which tests are necessary to verify the user's requirements to validate all requirements have been implemented correctly. [Foster 01]
Traceability of requirements, when used, offers considerable benefits for a software project [Leon 00]. For example:

- Enables verification that all features are captured in the requirements
- Makes the relationship between product features, use cases, requirements, etc. clear
- Helps ensure full test coverage of product features
- Helps guarantee that no unnecessary features are built
- Eases the requirements change process by:
    1. Simplifying impact assessment
    2. Identifying associated items to be changed

What happens if you don't do requirements traceability?

What will happen is you will deliver a system that you THINK had "almost" everything the customer wanted, but you won't be sure. If you do not have all the key requirements in your products, your customers and end users may not use your application. They will most likely go elsewhere to find the product that has it all. Traceability will allow the team to verify all features are contained in the requirements so you will **KNOW** the system had everything the customer wanted ***BEFORE*** you release it.

Avoid development "guessing" what to work on first. With traceability, the team will know what is most important to the customer and can implement those features first. Traceability will ensure development will focus on what to code first, because the customer needs as well as the relationship between product features, use cases, and requirements are crystal clear.

If you don't do requirements traceability, you won't know if the major defects were fixed until ***AFTER*** you release the product. Traceability will prove full test coverage of product features, which will reduce the number of major defects from getting into the hands of your customers and end users.

Last, changes to requirements won't be fully understood and won't be managed effectively. Traceability will greatly ease the requirements change process by simplifying impact assessment and identifying associated items to be changed.

How to get started: how to gain acceptance from your team

It can be very difficult to convince teams to implement a formal traceability practice. Here are some ideas to encourage your team to give the practice a try:

1. Develop a process
2. Create a matrix
3. Try it on a Pilot project
4. Gain acceptance by showing results

Develop a Process

Think about what needs to be traced. Expand your list from not only requirements, but also perhaps business rules, business processes, use cases, and test cases. Most teams lack the discipline to develop a process, document it, and then train all the team members on how to use it. On the surface, traceability seems very simple. And it can be once you take the time to get the process in place and the team develops the discipline to follow it from start to finish.

After you have a process defined. Spend the time to brainstorm what linkages need to be made. Figure 2 is a simple traceability flow diagram, adapted from Bill Councill, 2001 Sticky minds website that shows how to connect the links:

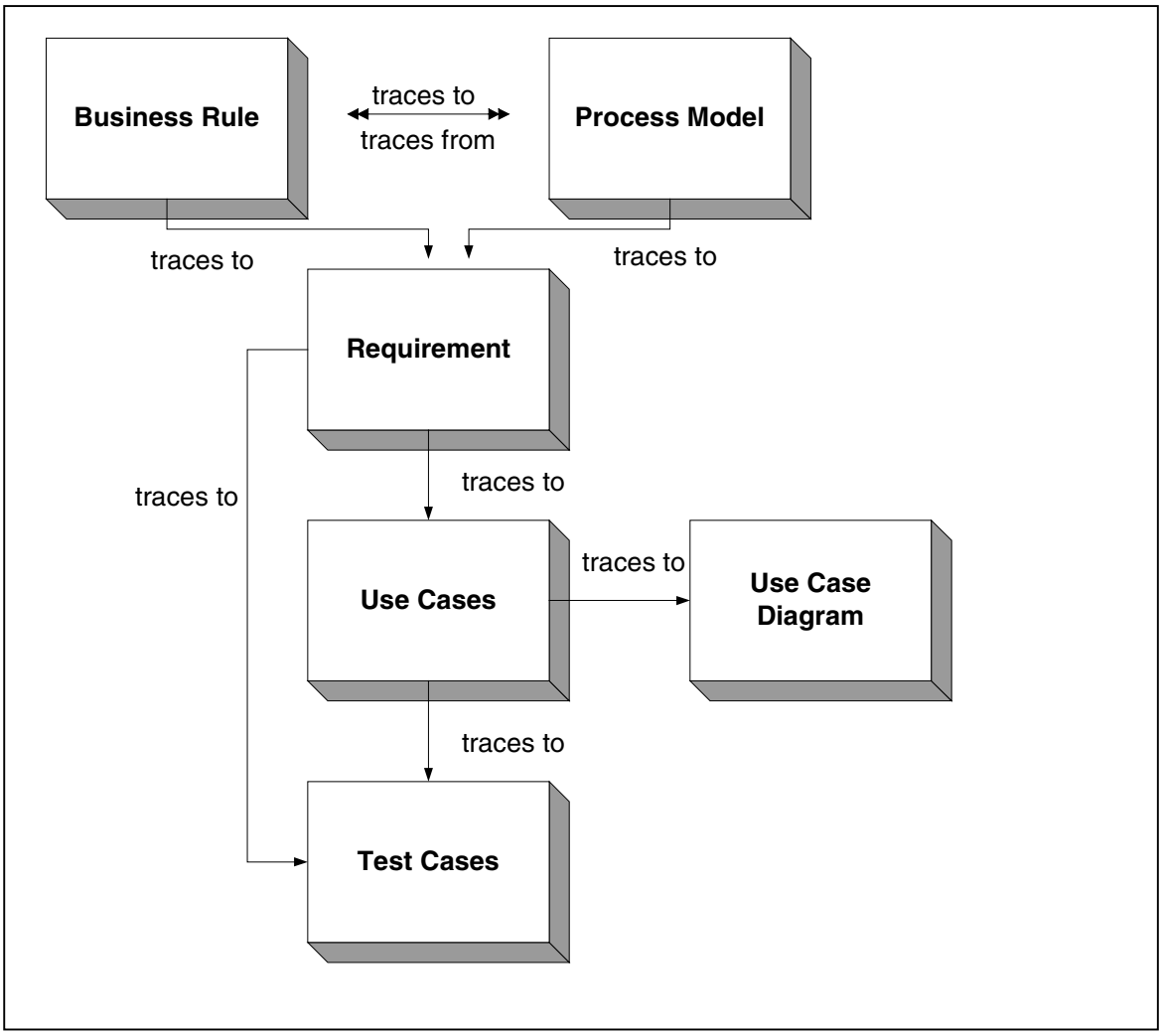


Figure 2 [Councill 01]

Once your team has all the links specified, it is worth the time to identify a system to tag very requirement with a unique and persistent identifier. A tag is typically alpha/numeric, for example: MTTF, OS, or R1, R2, etc. If a hierarchy is used, then MTTF.01, OS.03, or R1.1, R2.1, etc is assigned at the time the requirement is captured. Once documented, the requirement tag must not be changed! Why? If other work products are referencing this tag and the tag changes, all the dependent work products are incorrect.

Create a simple matrix

A easy way to start implementing traceabiity is to create a simple matrix. This matrix can be a spreadsheet where the key requirements are traced to the test cases that satisfy each requirement. Figure 3 is an example of a **simple matrix**:

	MTTF	OS	COL	CLI
TC1	X			
TC2			X	
TC3				

Figure 3

From this matrix you can quickly see if you have a requirement without a test case. Most development teams do not know if they have full test coverage, this type of double-checking will ensure everything gets tested.

Test teams are smart, from the simple matrix you can identify test cases that are being run that don't have a requirement associated with it. This type of discussion will prevent development teams from missing a requirement that the testing team knows is needed to make the application complete. The requirements team will have a more complete list of requirements based on input from the validation team. Also, from this type of matrix, the team may decide to write another requirement to satisfy the test case if it makes sense or they may determine the test case is not needed, therefore using their testing time more effectively.

At some point you may want to expand the simple matrix to include linking back from user requirements to a specific use case and be able to link forward to one more design, code, or test cases. Figure 4 is an example of an **expanded matrix** adapted from Karl Wiegers *Software Requirements,* MS Press 1999, page 303.

Use case ID #	User Requirement	Design Element	Code	Test Case
UC-28	catalog.query.sort	Class catalog	catalog.sort()	search.7
UC-29	catalog.query.import	Class catalog	catalog.import()	search.8
UC-30	catalog.validate.report	Class catalog	catalog.valid()	search.9

Figure 4 [Wiegers99]

To successfully maintain traceability, most teams will implement a requirements management tool to help ease the work of maintaining a manual tagging system. There are hundreds of requirements traceability tools on the commercial market

today [STSC 98]. A good requirements management tool can dramatically ease the task of tracing requirements.

To get you started, below are three tools you might want to become familiar with:

Name of the tool	Vendor	Description	Website URL
Caliber-RM	Technology Builders, Inc. (TBI)	Mid-range priced traceability tool	http://www.tbi.com/products/caliber.html
DOORS	Telelogic (previously QSS)	High-end requirements traceabiity tool	http://www.telelogic.se/
RequisitePro	Rational Software	Cost effective requirements traceability tool	http://www.rational.com/products/reqpro/docs/

All requirements management tools will enable teams to link requirements to their origin, to test cases, to changes to requirements, and to any design documents. The key is to be able to maintain traceability between all the work products used to develop, design and implement the features. A tool will make these links easier to manage.

Before you go out and spend thousands of dollars on a requirements traceability tool, spend time to understand your team's needs. Ask: What is it the team wants the tool to do for them? Brainstorm with your team what functionality you want in the tool. Then get a demo copy of the application and try it out on a small, low-risk project (more on piloting these ideas later in this paper) to validate the vendor's demo. Talk with others who have used the tool before to get their input on the strengths and weaknesses of the tool. Some tools are easier to implement and administer than others. Last, base your buying strategy on a documented list of "requirements" for your requirements traceability tool.

Once you identify a tool, determine who will take care of the daily care and feeding of the application. A designated database administrator will be the go to person when the database gets corrupt, someone deletes a record, a new person comes on the team that needs the application installed locally, and will handle all the upgrades to the tool.

This person may also help the project manager establish and maintain the information in the tool. Most teams forget to spend time identifying whom, how, and when the data will be maintained. Save your team a lot of time by identifying these roles early. Preferably before the tool is purchased or as close to when it is implemented as possible.

Education is the key! Get everyone (including management) on the team to spend the time to understand the benefits of traceability and why he or she should support the new practice. Take the time to hold an "executive summary meeting" with the managers to ensure the managers are up to speed on what they need to do to support traceability within their groups.

Try it on a Pilot project

Work directly with the team members to educate them on the success of other groups who have implemented traceability successfully. Show problems from other teams development activities where difficulties could have been prevented if they had used traceability techniques. These comparisons will help them make the leap from theory to practice. With this knowledge, try traceability on a "pilot" project.

If your whole organization is resistant to implementing traceability across every business group, try to get one project team to "pilot" a few techniques such as uniquely tagging every requirement, avoid bulleted lists and long narrative paragraphs, and populating a simple matrix to see if all requirements are satisfied by the test cases. By choosing a small, less visible, low risk project from a targeted business group you can ensure the team's success.

Gain acceptance by showing results

Within a few weeks after traceability has been "piloted", team leaders should audit the teams practices to ensure they are using the new traceability process, make adjustments to the process to make it more effective, and reward those who are doing it right! After a few months, team leaders need to let managers know of the results. Case studies indicate when just one or two traceability practices are implemented, such as requirements traced to test cases, show test coverage was excellent and the programming was well above average. [Stout01]

Team leaders and managers need to periodically check to make sure results are supporting the time and energy it takes to do traceability right. The team leaders need to audit the new process to make sure it is still being followed and begin to build examples of where traceability has increased design efficiency, reduced time to market, or increased the quality of their products.

After the initial pilot has finished, hold a Retrospective (sometimes called Post Project Reviews or Post Mortems) to reflect on what went well, what worked so

you can reinforce it, learn what needs to be improved, and capture project information for future reference. Norm Kerth in 2001 published a book titled: *Project Retrospectives: A Handbook for Team Reviews*. This is a delightful book, easy to read, and understand. His book covers all the information you will need to put retrospectives into practice immediately. He offers a formal method for safeguarding the important lessons learned from the successes and failures of any project. These lessons and the information they provide will promote stronger teams and savings on following projects.

By far the best way to get a team to embrace traceability is to show **results**. A retrospective will aid in flushing out what happened during the pilot. Design efficiency is the result of requirements tracing. Results are apparent, as "best" designs are likely when complete information is stated up front in a requirements document the developer uses to code from.

This will result in less re-work, which means teams will reduce their time to market. Traceability permits the developer to discover potential problems before they get to the validation team and fix them. The old adage: "An ounce of prevention is worth a pound of cure" is what you want to strive for.

Another benefit of implementing traceability is increased quality. Change management is improved when a change can be traced from the customer needs, to the requirements, to various work products where the impact of the change is evaluated from a business and technical perspective.

These benefits ultimately result in shorter development cycles and more on time launches [Leon00].

Summary

Every development team has their challenges! Reduced cycle times (getting products out the door in less time), budget constraints (with less funding), and constantly changing requirements are just a few. To ensure development teams are able to deliver products that do **EVERYTHING** the customer wants traceability should be implemented. By maintaining traceability throughout a project the following will result:

- The project team will understand which parts of the design and code implement the user's requirements
- Ensure full test coverage of product features
- Development will know what to code "first"
- You will know if the major defects are fixed ***BEFORE*** you release the product
- Changes to requirements will be fully understood and will be managed effectively

Try implementing traceability on your next development project! It will make a difference in the quality of your product resulting in higher customer satisfaction.

Sources for more information

If this subject is of further interest, below is a list of sources for more information on how to implement traceability in your organization:

- *Software Requirements,* Karl E. Wiegers, MS Press 1999.
- *Managing Software Requirements: A Unified Approach*, Dean Leffingwell and Don Widrig, Addison Wesley 1999
- *Requirements Engineering: Processes and Techniques*, Gerald Kotonya and Ian Sommerville, Wiley 1999
- *Exploring Requirements: Quality Before Design*, Donald Gause and Gerald Weinberg, Dorset House 1989
- *Software Requirements Engineering, 2nd Ed.,* Richard Thayer and Merlin Dorfman, IEEE Press 1997 *Systems Engineering: Coping With Complexity*, by Richard Stevens *et al*, Prentice Hall 1998 (good chapter on prototyping)
- *Customer-Centered Products*, Ivy Hooks and Kristin Farry, Amacom 2001
- *Process for System Architecture and Requirements Engineering,* Derek Hatley *et al,* Dorset House 2001
- *Project Retrospectives: A Handbook for Team Reviews,* by Norman Kerth, 2001
- StickyMinds.com has many good articles on Traceability

Reference List

[Councill 01]	Councill, Bill. *Automating Requirements Traceability.* June, 2001 StickyMinds.com
[Ecklund, Delcambre & Freiling 96]	Ecklund, Earl F., Delcambre, Lois M. L. and Freiling, Michael J. "Change Cases: use cases that identify future requirements. *Proceedings of the 11th Annual Conference on Object-Oriented Programming systems, languages, and applications, 342-358.* 1996.
[Foster01]	Foster, Henrietta, "Why Trace Requirements? Overcoming resistance to requirements tracing." *StickyMinds.com* (Oct 2001)
[Gotel & Finkelstein 94]	Gotel, Orlena & Finkelstein, A.W. An analysis of the requirements traceability problem. *In the proceedings of the International Conference Requirements Engineering, IEEE Computer Society Press.* Colorado Springs, CO (1994)
[Jarke 98]	Jarke, Matthias. Requirements Tracing. *Communication ACM* (December 1998): 32-36
[Leon 00]	Leon, Marco. "Staying on Track." *Intelligent Enterprise* (September 2000): 54-57

[IEEE 90]	Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* New York, NY: (1990)
[Stout 01]	Stout, Glenn, "Requirements Traceability, the Effect on the SDLC". *StickyMinds.com* (July 2001)
[Spence & Probasco 98]	Spence, Ian & Probasco, Leslee. "Traceability Studies for Managing Requirements with Use Cases". Available on the web: http://www.rational.com/products/whitepapers/022701.jsp
[STSC 98]	Software Technology Support Center. *Requirements Management Tools* [online]. Available on the web: http://www.stsc.hill.af.mil/RED/LIST.HTML (1998).
[Wiegers99]	⋆Software Requirements, Karl E. Wiegers, MS Press (1999).

Achieving Software Quality through Process Integration

Brian Bryson, Rational Software

One way to improve software quality is to focus on the individual disciplines of Design, Development, QA, etc. However, more significant gains can be made by optimizing communication between these tasks. This presentation will focus on how to improve communication with better techniques, tools & data.

- Communication happens with common goals, tools & data.
- QA & Development can do work for each other to improve testing.
- QA must report progress in the language of their customers.

Achieving Software Quality Through Process Integration

Brian Bryson
Technology Evangelist, Rational Software

bbryson@rational.com

Agenda

- Why we need to talk
- Process Integration
  - Requirements Management
  - Visual Modeling
  - Developer Testing
  - Automated System Testing

Agenda

- Why we need to talk
- Process integration
  - Requirements Management
  - Visual Modeling
  - Developer Testing
  - Automated System Testing

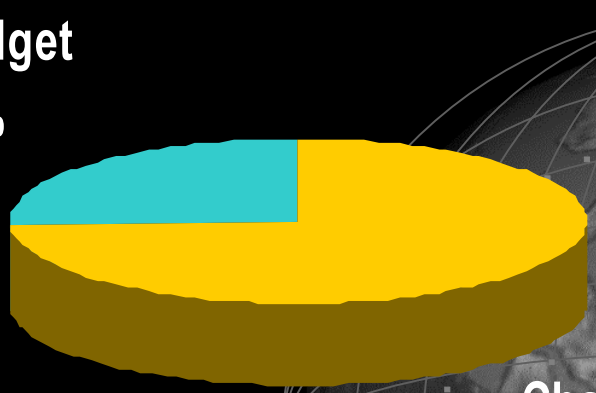
A Software Development Analogy

- Easy to plan
- Easy to build
- Easy to test

A Software Development Analogy

- Requires architecture and process
- Requires a master plan
- Communication between teams is essential

Project Success is Requirements-related



- 53% of projects over-ran their original estimates
  - Average budget overrun: 189% ($59 billion)
  - Average schedule overrun: 222%
- 31% of projects canceled before completion ($81 billion)

Standish Group, *CHAOS: A Recipe For Success, 2000*

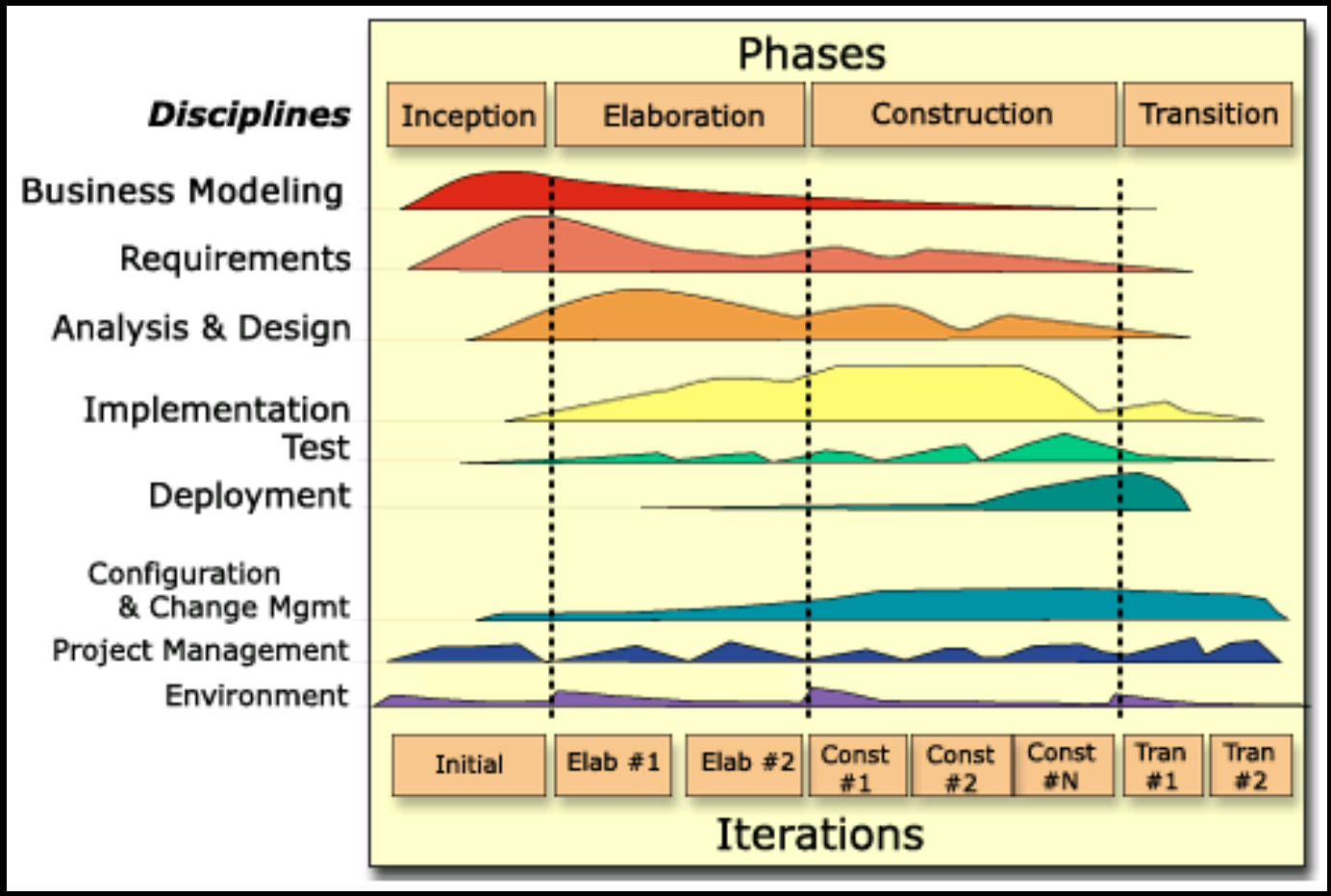
Agenda

- Communication happens when we have
  - Common Goals
  - Common Tools
  - Common Data
  - Common Processes
- Pre-Requisite Ingredient
  - Accurate and Relevant Information to share

Agenda

- Why we need to talk
- Process Integration
  - Requirements Management
  - Visual Modeling
  - Developer Testing
  - Automated System Testing

Software Development Overview



- Integrated disciplines that overlap throughout the lifecycle

Process Integration

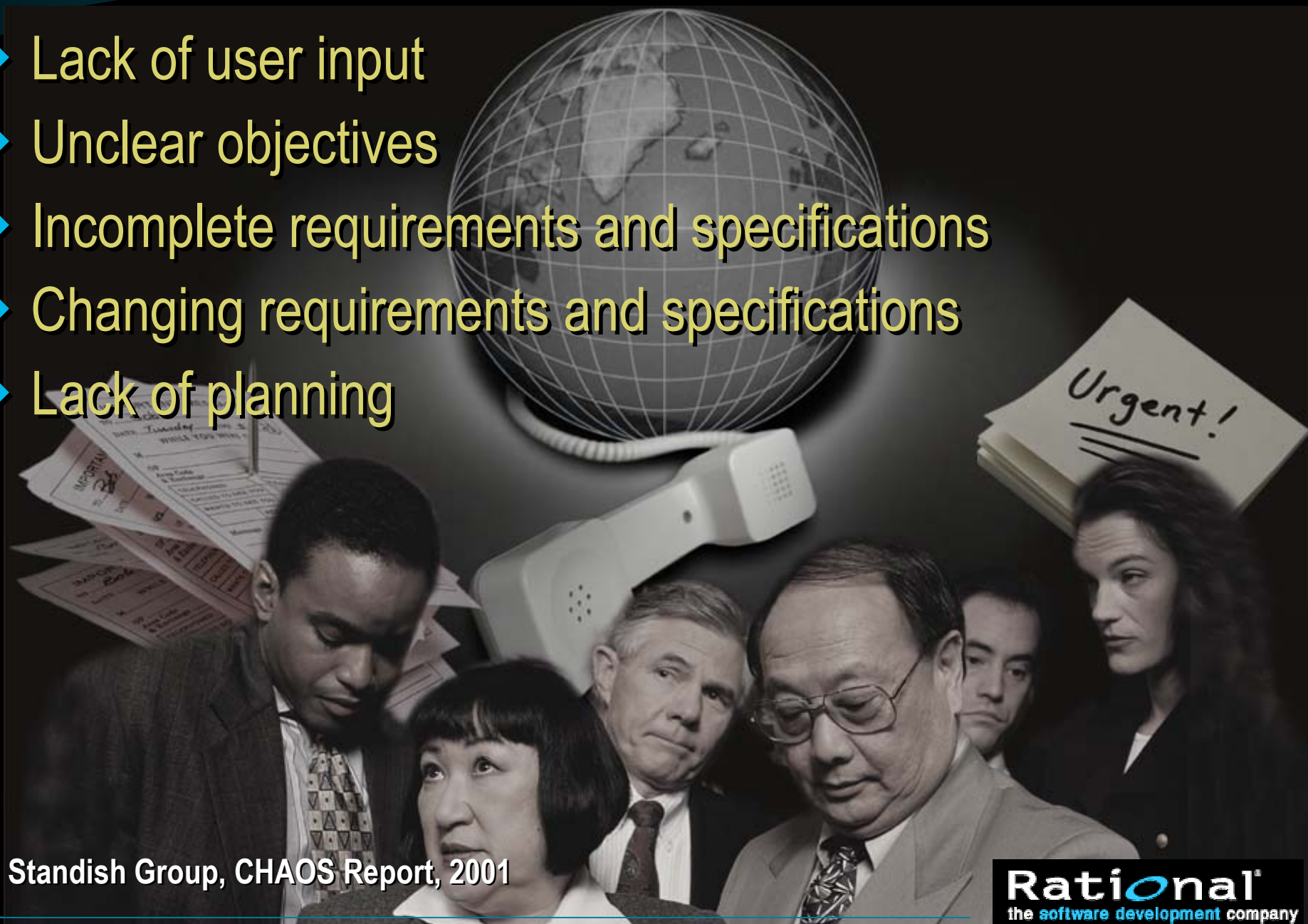
- Individual disciplines in isolation are manageable
- Integration between disciplines is the key

Agenda

- Why we need to talk
- Process Integration
  - Requirements Management
  - Visual Modeling
  - Developer Testing
  - Automated System Testing

Project Failure Factors

- Lack of user input
- Unclear objectives
- Incomplete requirements and specifications
- Changing requirements and specifications
- Lack of planning



Standish Group, CHAOS Report, 2001

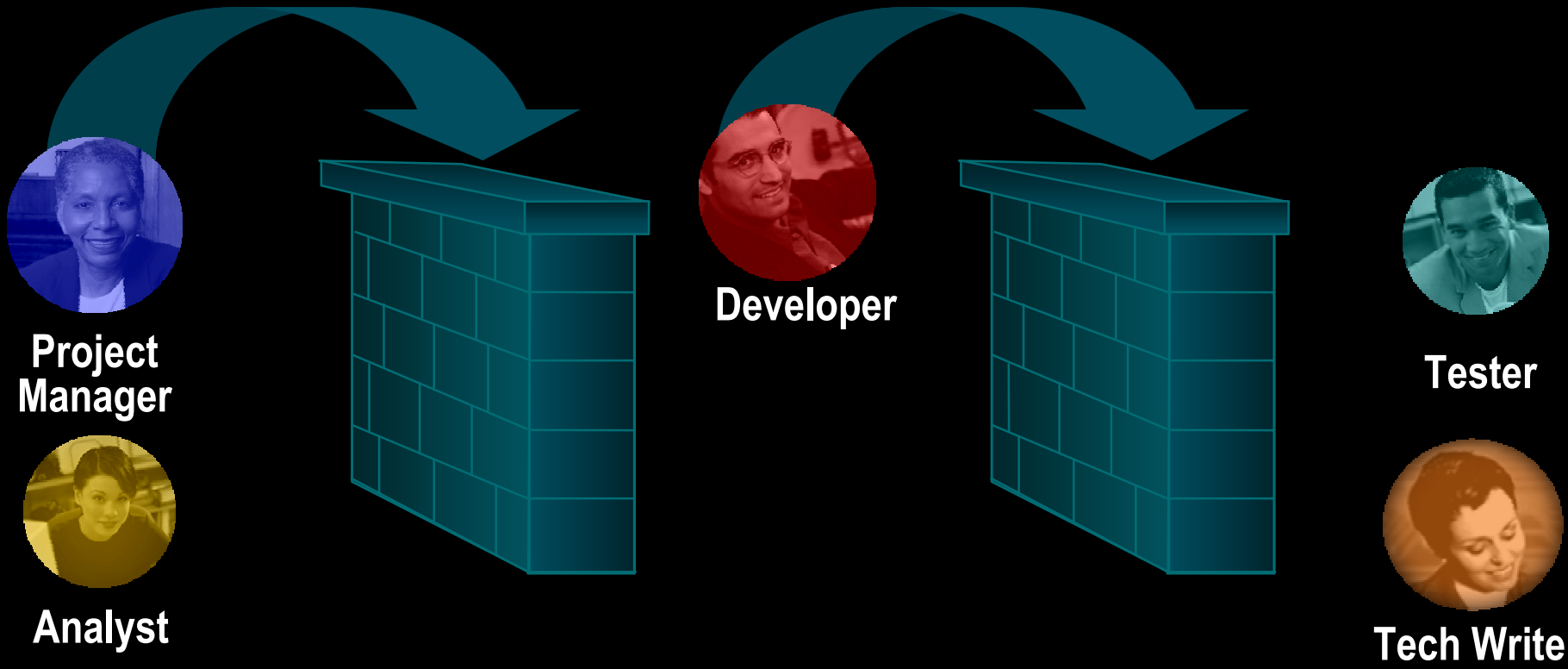
What is Requirements Management?

A systematic approach to eliciting, documenting, organizing, and tracking changing requirements

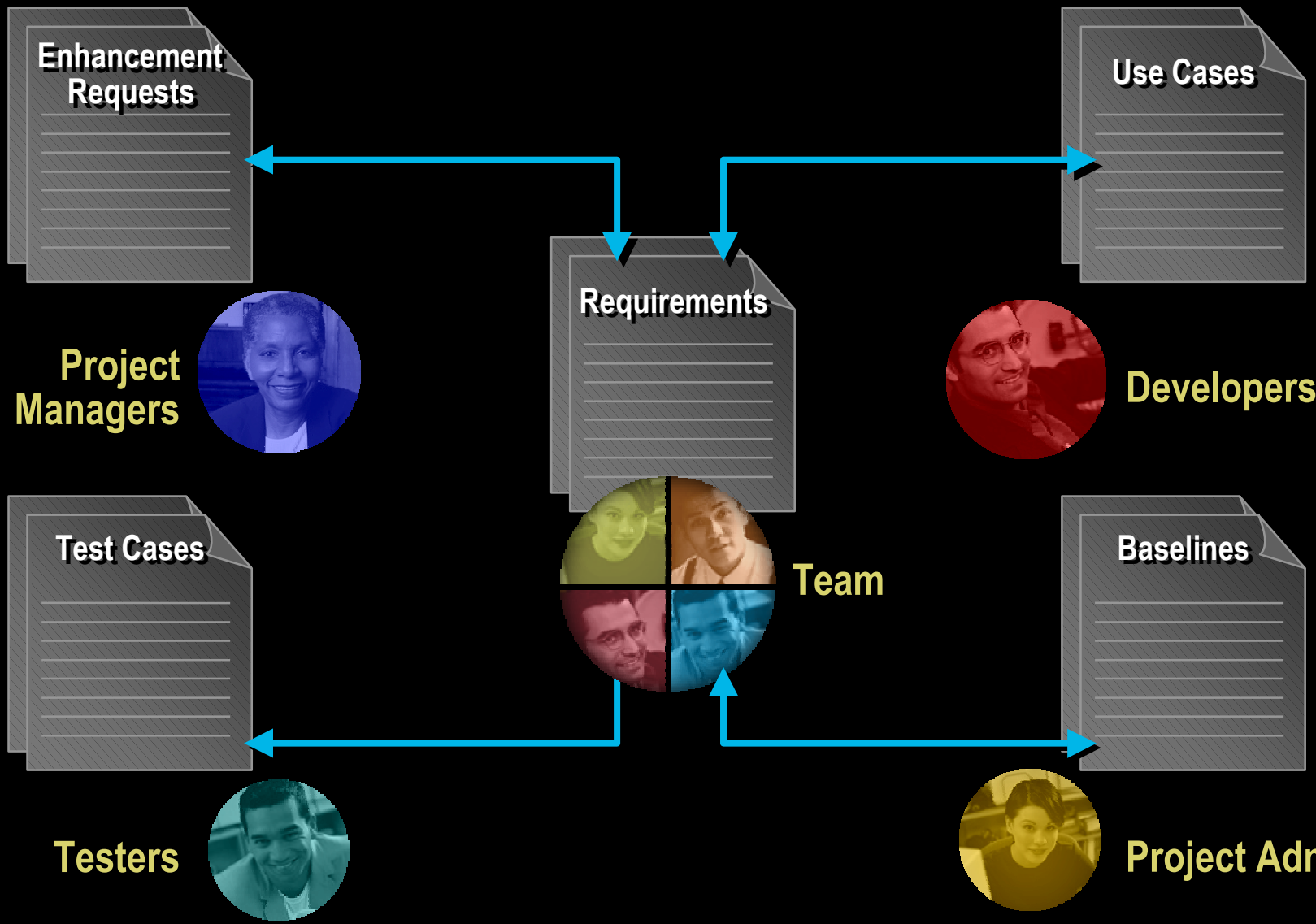
Ensuring your team identifies, builds, tests and documents the right system for your customer

Ineffective Communication

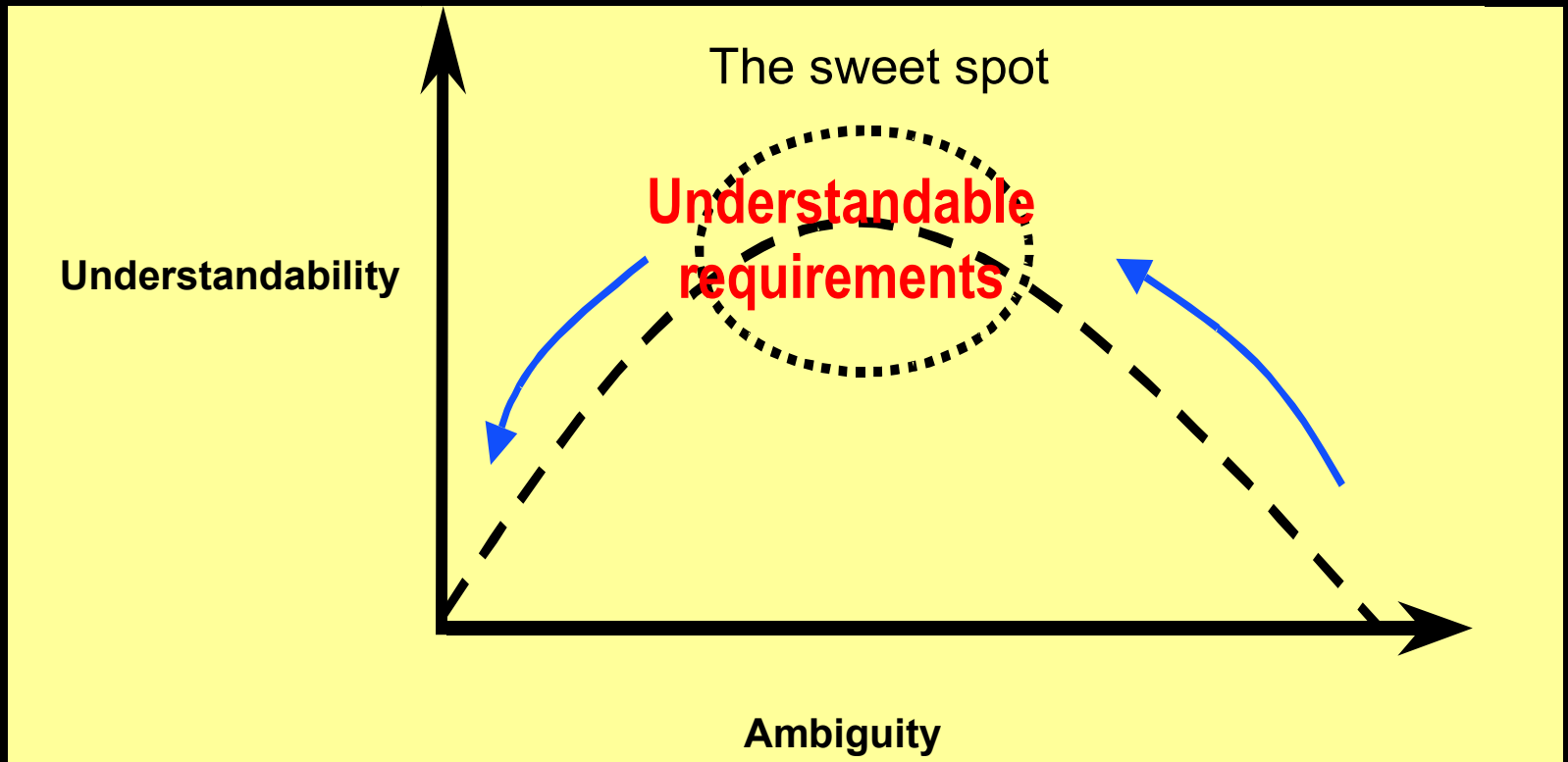
- Information is tossed over the wall between groups
- Tools don't work together or share information
- Too much translation, redundancy and guesswork



Integrate Requirements Across Tools and Teams



How much detail is enough?

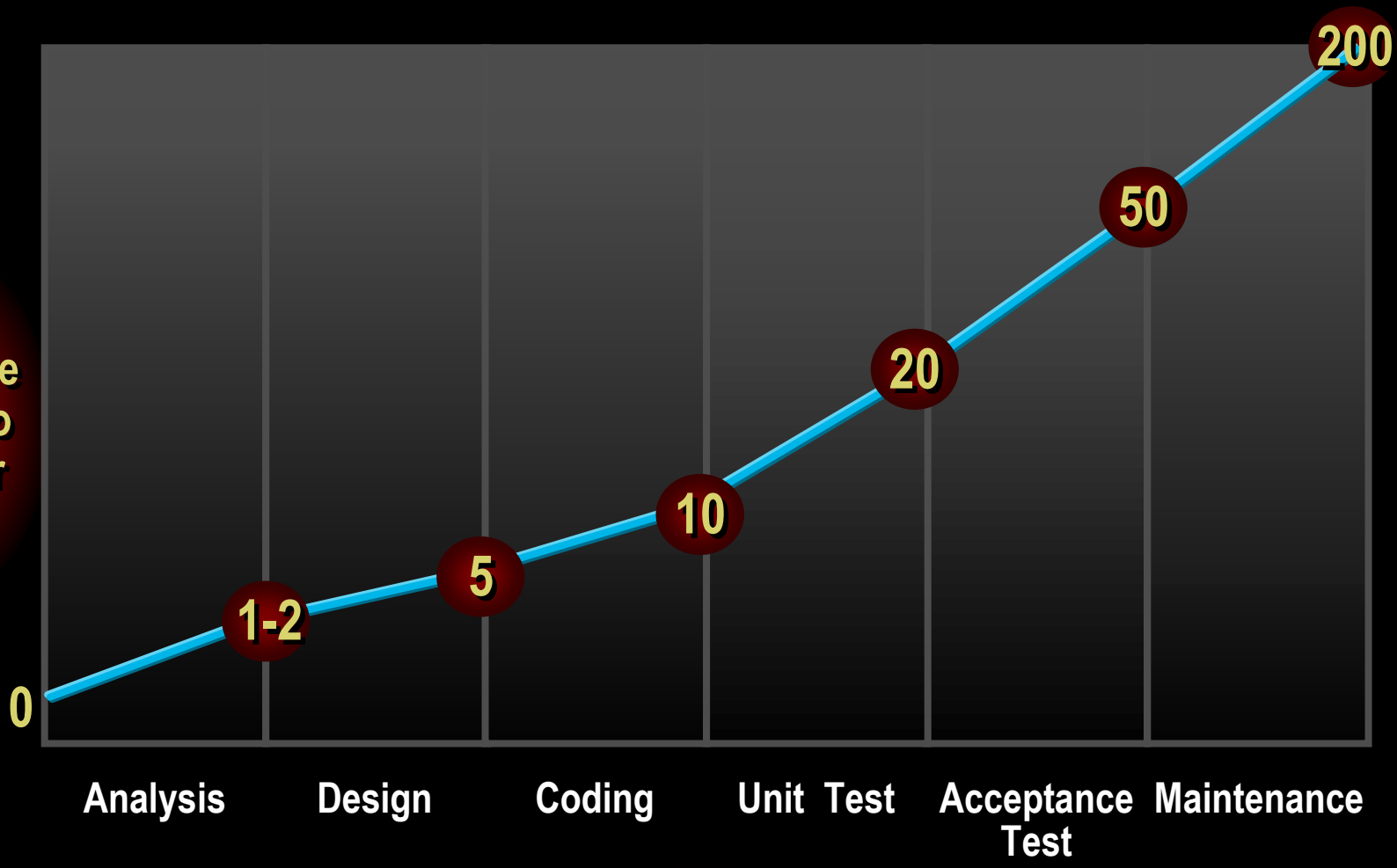


Techniques that reduce ambiguity in a requirement often decrease understandability and alienate customers and users. Our goal is to find the "sweet spot" at which we attain the greatest amount of understandability with the least amount of ambiguity:

Summary of a "GOOD" software requirement

- "Good" software requirement characteristics
  - Correct
  - Clearly understandable by all stakeholders
  - Consistent
  - Non-conflicting with other requirements
  - Verifiable or testable
  - Branchless (no conjunctions or exception statements)
  - Identifies the following:
    - Who is doing the task
    - The task being performed
    - The result of the task
    - The recipient of the results of the task

High Cost of Requirements Errors



Agenda

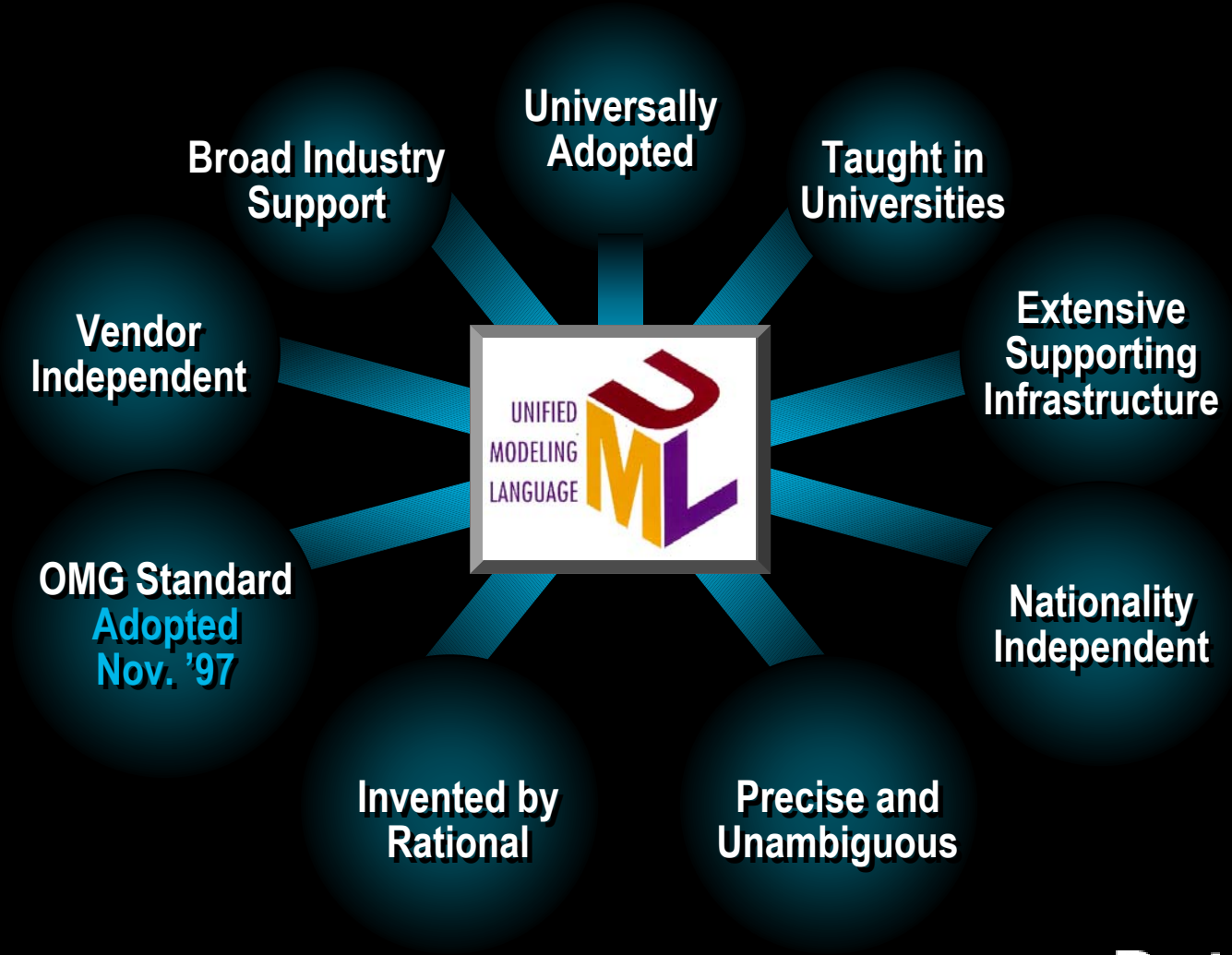
- Why we need to talk
- Process Integration
  - Requirements Management
  - Visual Modeling
  - Developer Testing
  - Automated System Testing

Another Construction Analogy

- Question:
  - How do all the members of a construction/engineering team know how to assemble their building?
- Answer:

UML: *A Universal Communication Tool*

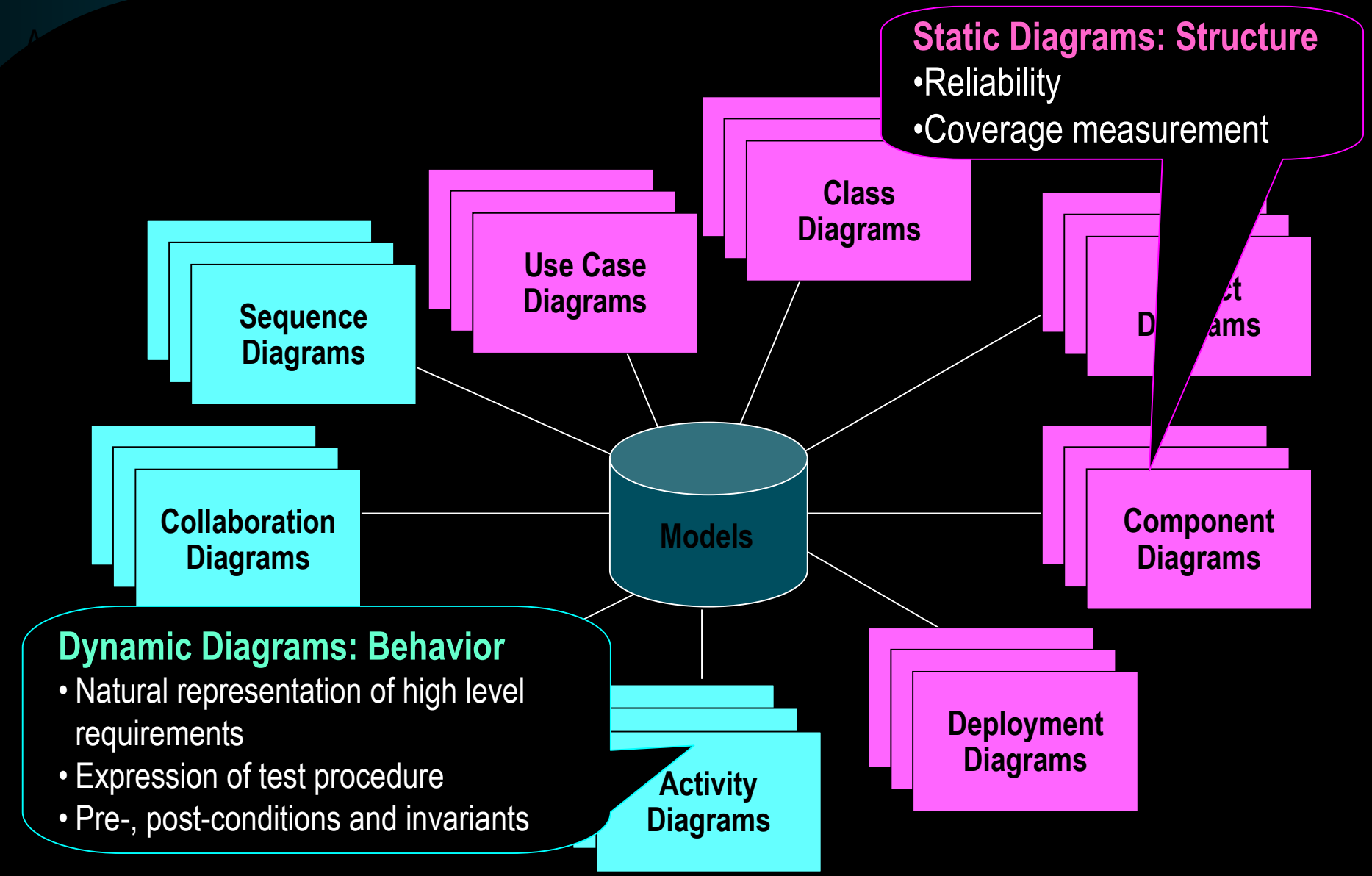
The OMG Standard for Visual Modeling



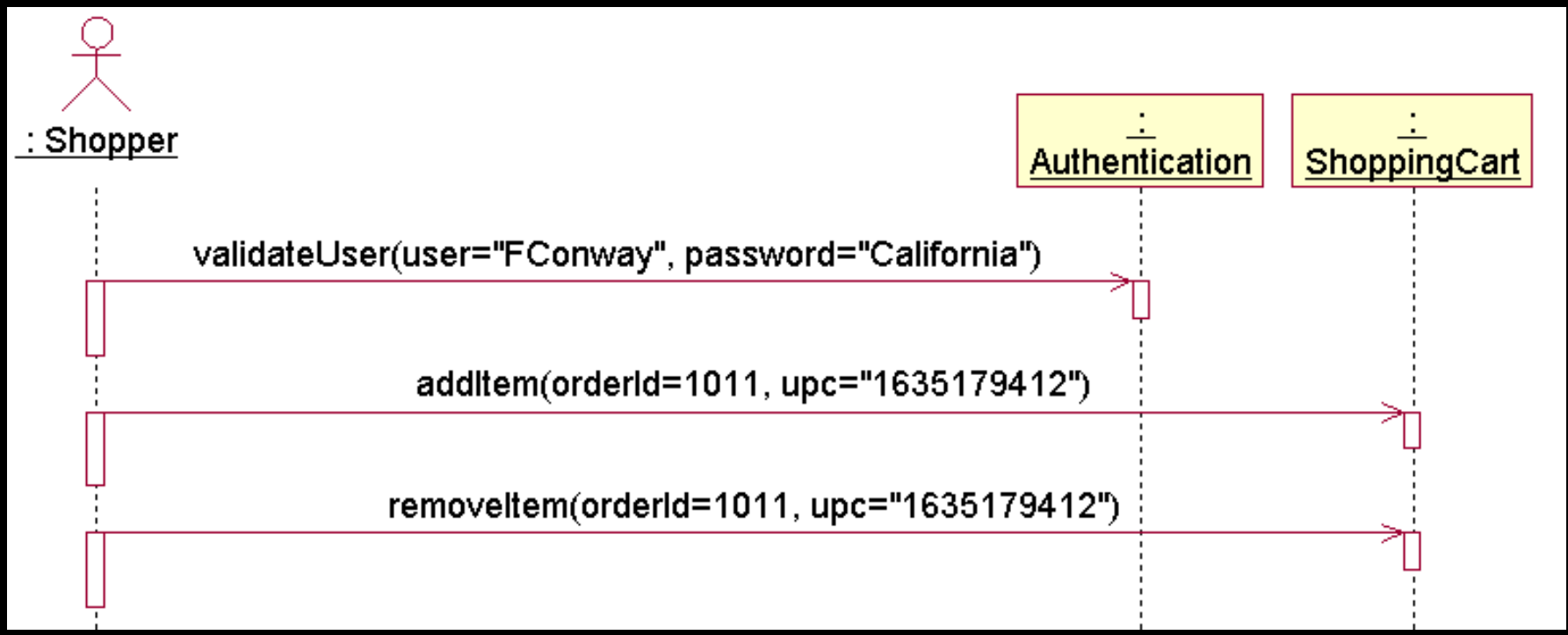
UML Concepts

- The UML may be used to visually model:
  - The interaction of your application with the outside world
  - The behavior of your application
  - The structure of your system
  - The architecture of your enterprise
  - The components in your system

What UML Can Offer

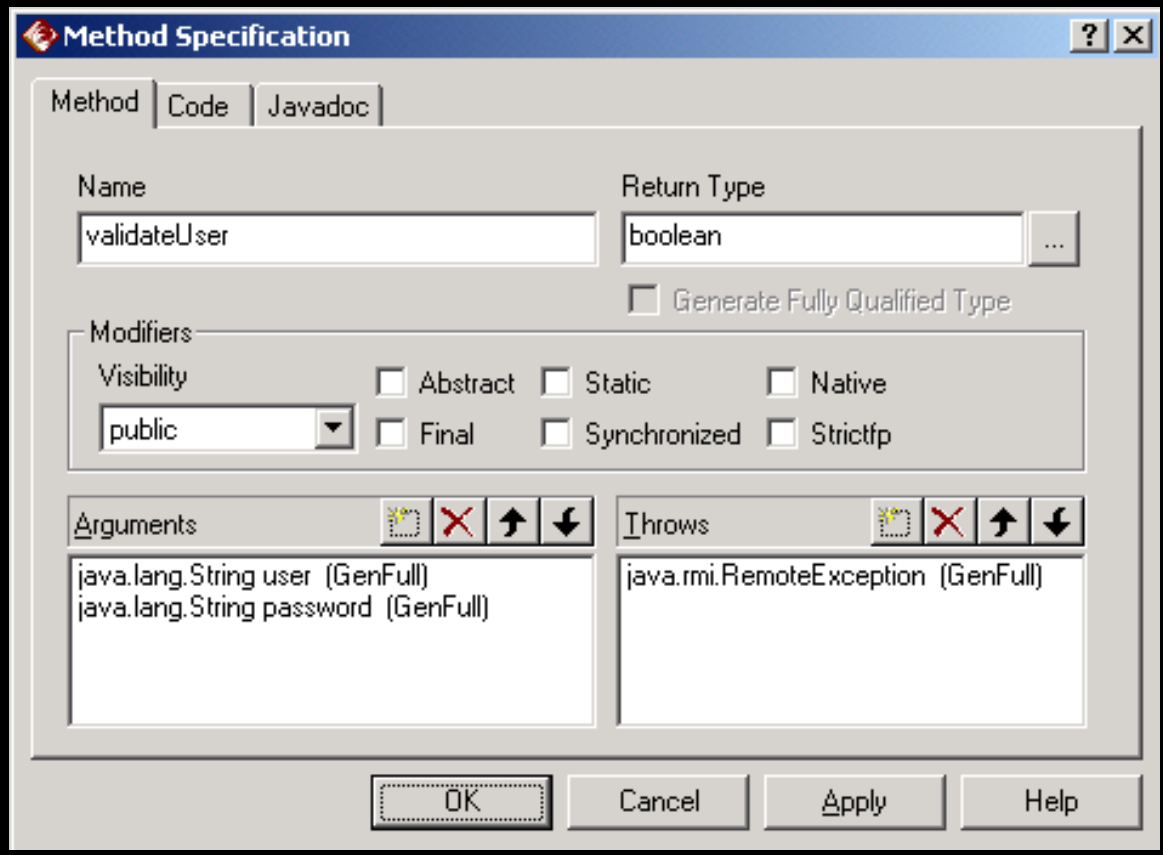


Example: Sequence Diagram



- Can you interpret this UML sequence diagram?

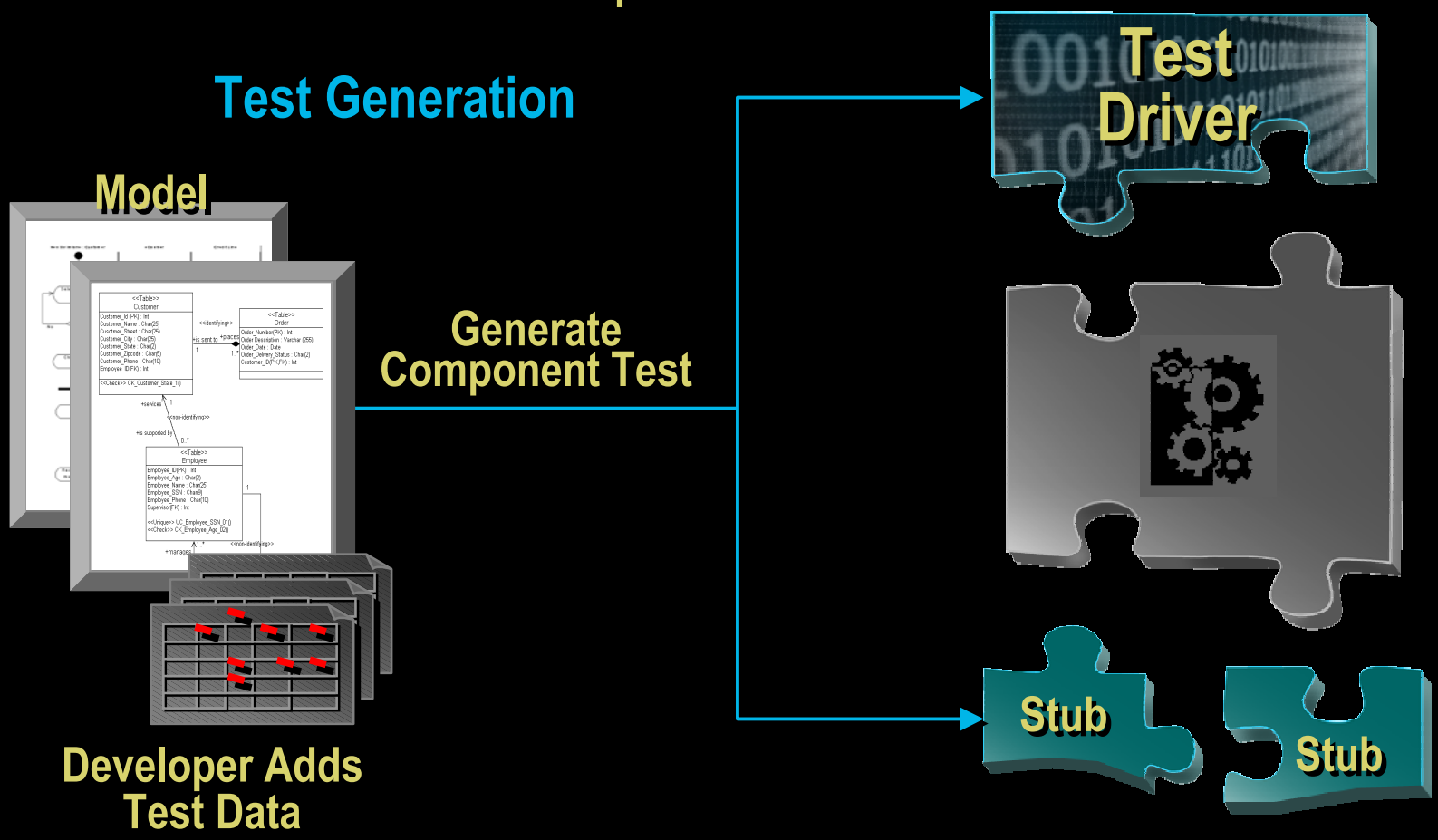
Example: Method Specification



- Is there enough information here to test?

Rational QualityArchitect

- Generate test code directly from model
- Provide test data and expected results



Agenda

- Why we need to talk
- Process Integration
  - Requirements Management
  - Visual Modeling
  - Developer Testing
  - Automated System Testing

Diagnostic Testing

- Black Box Testing
  - Common QA type testing
  - Output is Pass/Fail status
- White Box Testing
  - Traditional developer test
  - Output is detailed diagnostic information

What to Expect from Diagnostic Detail

- Code Coverage
- Memory Leak
- Run-time Error detection
- Performance Profiling

Sharing Diagnostic Data Collection

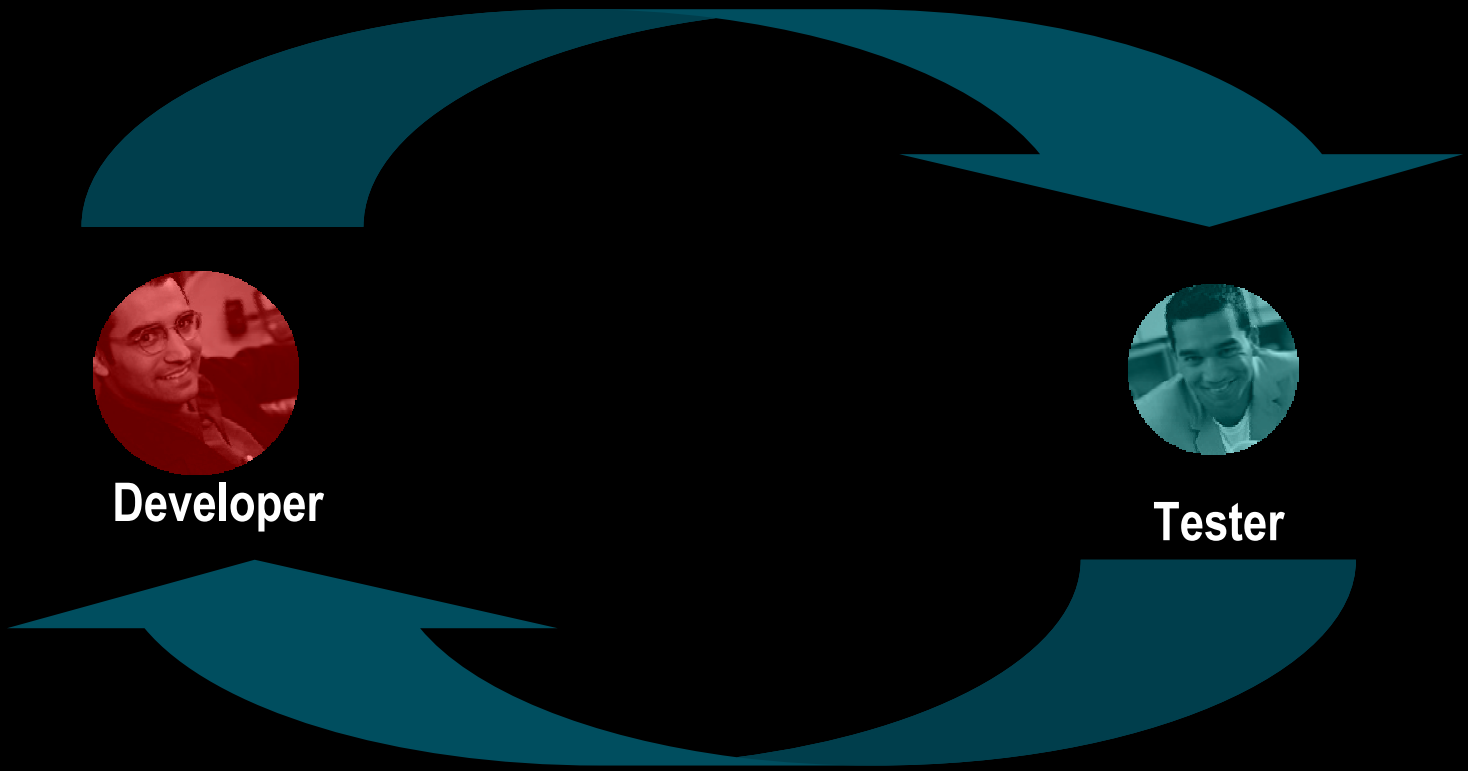
- Today's Theme: Shared Information
  - Do all parties need to know how to interpret information?

    No!

  - Enable testers to collect diagnostic information
    - Debug builds
    - Debug compiler information
    - Instrumented builds

Economics 101: Specialization of Labor

Developer provides tester with instrumented build



Testers return defect reports with diagnostic information attached

A Quick Plug for Unit Testing

- Unit Testing gets some traction

- Unit Test Evangelism!
  - Open source tool for unit testing java & other languages
  - Pierced the development shield
- A goal
  - Recycle developer unit tests for use at system test time!

Agenda

- Why we need to talk
- Process Integration
  - Requirements Management
  - Visual Modeling
  - Developer Testing
  - Automated System Testing

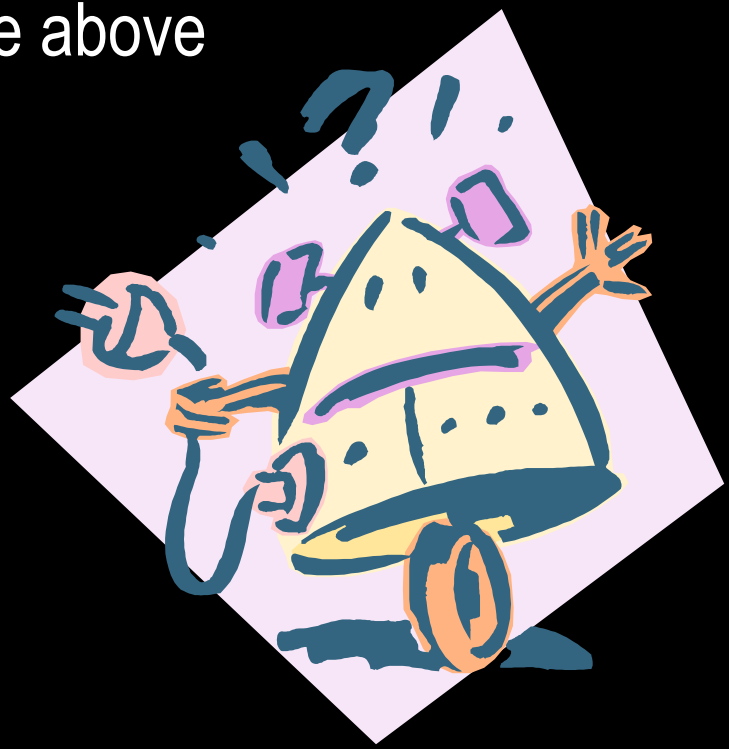
System Test Automation

- Multiple Choice:

A. Software Test Automation is a good thing?

B. Software Test Automation is a bad thing?

C. None of the above



What Does Matter

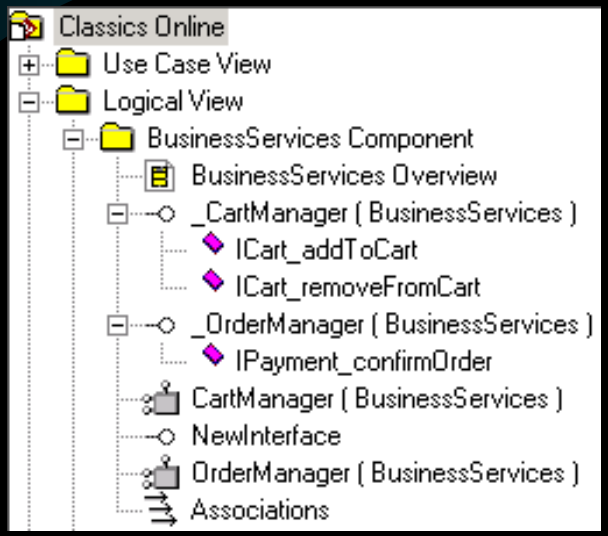
- That you test somehow
  - Risk Based
  - Context Driven
- That you measure
  - Requirements Coverage
  - Code Coverage
  - Model Coverage

What Is Not Measured Does Not Get Done

	Planned Test Cases	Implemented Test Ca...	% Test Cases Implem...
Order Processing	20	16	80
FEAT1 Point of Sale System	3	1	33
FEAT1.1 Cash register functions	1	1	100
Process New Sale	Yes	Yes	Yes
FEAT1.2 Maintaining the store´s inventory	2	0	0
Add new SKU	Yes	No	No
Replace SKU number	Yes	No	No
FEAT1.3 Supporting multiple cash registers per store	0	0	0
FEAT1.4 Initiating orders to replenish stock when necessary	0	0	0
FEAT2 Order Processing System	0	0	0
FEAT2.1 Provide for both automated and human-assisted order entry	0	0	0
FEAT3 Warehouse system	1	1	100
FEAT3.1 Manage receiving, warehousing, and shipping of merchandise	0	0	0
FEAT3.2 Provide real-time control over merchandise-handling equipment such as conveyor belts, barcode scanners, electronic scales, and the like	0	0	0
FEAT3.3 Interface with other Classics Inc. back-office systems such as Purchasing and Accounts Payable	0	0	0
FEAT3.4 Interface with external systems such as those of vendors and freight carriers	1	1	100
Verify Featured Selection	Yes	Yes	Yes
FEAT4 Home Shopping e-commerce system	2	2	100
FEAT4.1 An online catalog for web visitors to browse	2	2	100
Browse Mozart	Yes	Yes	Yes
Browse Sonatas	Yes	Yes	Yes
FEAT4.2 A customer account maintenance facility for opening and maintaining accounts for individual consumers	0	0	0
FEAT4.3 An order-entry capability supporting online sales and order fulfillment, interfacing with the Order Processing system	0	0	0
FEAT4.4 An order status inquiry system that allows customers to check on existing orders and send messages to the Classics Inc. Customer Service staff	0	0	0
FEAT5 Data and Database Integrity	0	0	0
FEAT5.1 Execute the database regression test suite V2.0.	0	0	0
FEAT6 Business Function	13	12	92
FEAT6.1 Cash Register	2	2	100
FEAT6.1.1 Clerk Functions	0	0	0
FEAT6.1.1.1 Process Sale	0	0	0
FEAT6.1.1.1.1 Cancel Sale	0	0	0
FEAT6.1.1.1.2 Request Subtotal	0	0	0
FEAT6.1.1.1.3 Delete Item from Order	0	0	0

- Requirements Based Coverage
- Measures effectiveness from stakeholder perspective

What Is Not Measured Does Not Get Done



- Measure Model Coverage
- Measures effectiveness from an architectural perspective

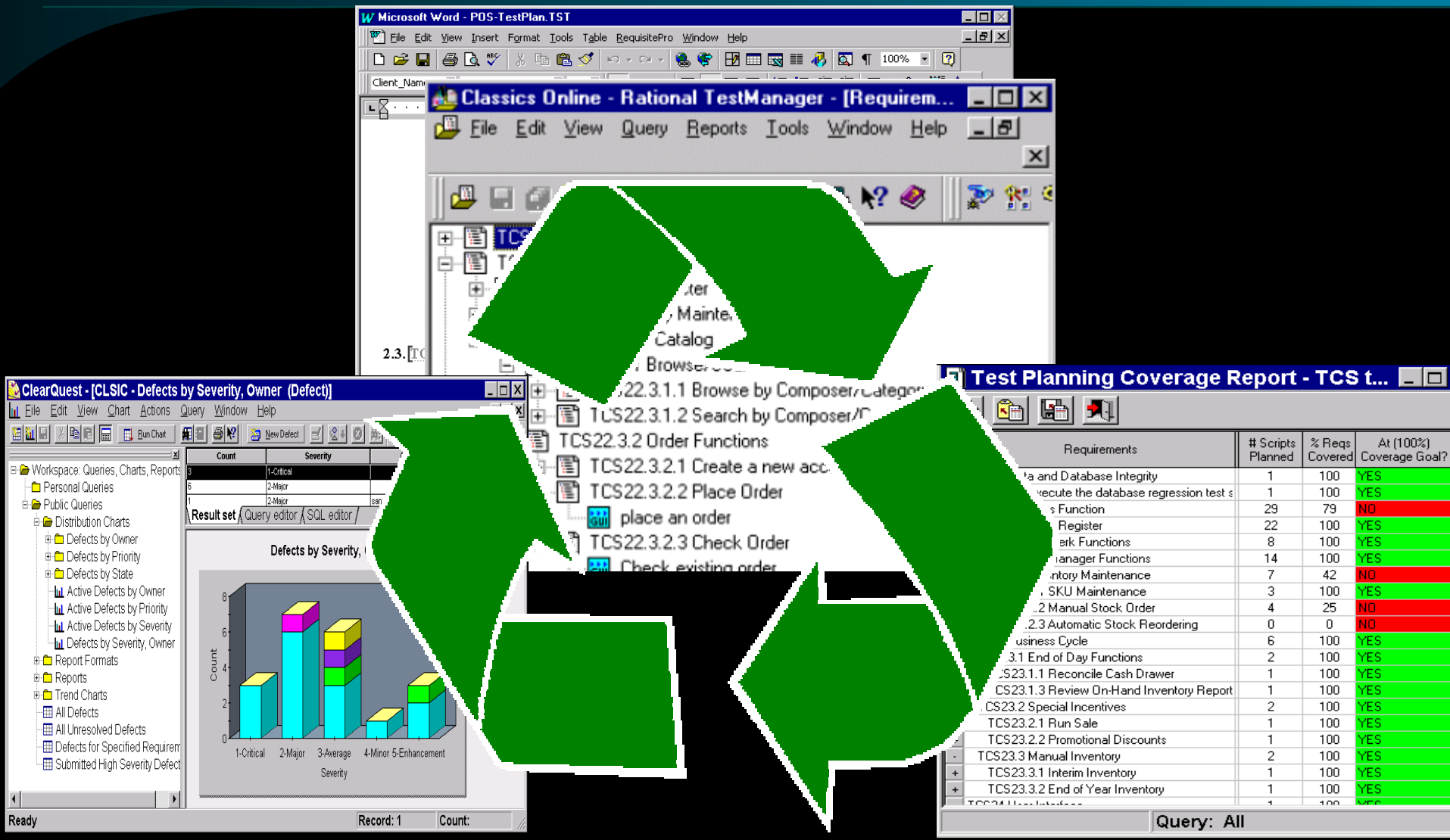
	Planned Test Cases	Implemented Test Ca...	% Test Cases Implem...
Classics Online	1	0	0
Use Case View	0	0	0
Main	0	0	0
Logical View	1	0	0
BusinessServices Component	1	0	0
BusinessServices Overview	0	0	0
_CartManager	0	0	0
ICart_addToCart	0	0	0
ICart_removeFromCart	0	0	0
_OrderManager	1	0	0
IPayment_confirmOrder	1	0	0
Order Confirmation	Yes	No	No
CartManager	0	0	0
NewInterface	0	0	0
OrderManager	0	0	0

What Is Not Measured Does Not Get Done

Run @ 09/08/1999 09:30:43 <no arguments>	64	38	33	46.48	254	251
C:\Program Files\Netscape\Communicator\Program\AIM\idler..	0	0	0	0.00		
C:\Program Files\Rational\Classics Demo\ClassicsApps\Atbu-..	64	38	33	46.48	254	251
C:\Program Files\Rational\Classics Demo\ClassicsApps\al..	9	10	8	44.44	35	30
frmAbout.frm	0	3	0	0.00	11	0
frmAbout.Form_Load(...)	0	missed			5	0
frmAbout.cmdOK_Click(...)	0	missed			3	0
frmAbout.picIcon_DblClick(...)	0	missed			3	0
frmMain.frm	9	7	8	53.33	24	30
frmMain.Form_Load(...)	1		hit		0	3
frmMain.Form_QueryUnload(...)	1		hit		0	3
frmMain.Form_Resize(...)	1		hit		0	6
frmMain.cmdOrder_Click(...)	1		hit		2	4
frmMain.mnuFileClose_Click(...)	1		hit		0	3
frmMain.mnuHelpAbout_Click(...)	0	missed			3	0
frmMain.mnuLogin_Click(...)	0	missed			3	0
frmMain.mnuOrderStatus_Click(...)	0	missed			3	0
frmMain.mnuPlaceOrder_Click(...)	0	missed			3	0
frmMain.mnuRestore_Click(...)	0	missed			3	0
frmMain.tabMain_Click(...)	0	missed			3	0
frmMain.treMain_Expand(...)	1		hit		0	3
frmMain.treMain_NodeClick(...)	1		hit		0	3
frmMain.txtFocusHolder_Change(...)	0	missed			2	0
frmMain.txtImageHolder_Change(...)	2		hit		2	5
C:\Program Files\Rational\Classics Demo\ClassicsApps\al..	55	28	25	47.17	219	221

- Measures code coverage
- Measures test breadth

Rational Suite Overview for Testing



- It's all about communication!

Parting Thoughts

- Key is shared data between team members
  - Easy access to data is mandatory
  - Automation is NOT a necessity
  - However as product team size and complexity increases, communication between parties becomes more and more critical
  - Communication happens when there are common tools, data, processes and goals
- Challenges
  - Upgrades to one data container may break integrations
  - Over documentation

Achieving Software Quality Through Process Integration

Brian Bryson
Technology Evangelist, Rational Software

bbryson@rational.com

HOW TO CREATE USEFUL SOFTWARE PROCESS DOCUMENTATION

Linda Westfall
The Westfall Team

lwestfall@westfallteam.com

3000 Custer Road, Suite 270, PMB 383
Plano, TX 75075

ABSTRACT

Whether our organization is using ISO 9001, the Software Engineering Institutes Capability Maturity Model - Integrated[SM], Total Quality Management, Six Sigma or some other quality framework, one of the cornerstones of any of these frameworks is to document our processes. Unfortunately efforts to document our process often end up in volumes of verbosity that sit on the shelf and gather dust. ***How to Create Useful Software Process Documentation*** introduces the reader to a simple, practical method for defining and documenting software processes that are easy to understand, easy to use and easy to maintain.

ABOUT THE AUTHOR

Linda Westfall is the President of The Westfall Team, which provides Software Quality Engineering and Software Metrics training and consulting services. Prior to starting her own business, Linda was the Senior Manager of Quality Metrics and Analysis at DSC Communications where her team designed and implemented a corporate wide metric program. Linda has more than twenty years of experience in real-time software engineering, quality and metrics. She has worked as a Software Engineer, Systems Analyst, Software Process Engineer and Manager of Production Software.

Very active professionally, Linda Westfall is the immediate Past Chair of the American Society for Quality (ASQ) Software Division. She has also served as the Software Division's Program Chair and Certification Chair and on the ASQ Certification Board.

Linda Westfall has an MBA from the University of Texas at Dallas and BS in Mathematics from Carnegie-Mellon University. She is an ASQ Certified Software Quality Engineer (CSQE), ASQ Certified Quality Auditor (CQA). She is also a Professional Engineer (PE) in Software Engineering in the state of Texas.

WHY IS PROCESS DOCUMENTATION IMPORTANT

One of the cornerstones to any quality program is documented processes. Processes are "codified good habits" [Down-94] that "define the sequence of steps performed for a given purpose" [IEEE-610]. By standarizing and documenting our software processes we can describe and communicate what works best in our organizations. This can help us:

- Ensure that important steps in our processes aren't forgotten
- Facilitate the propagation of lessons learned from one project to the next so we can repeat our successes and stop repeating actions that lead to problems
- Eliminate the need to "reinvent the wheel" with each new project while providing a foundation for tailoring our processes to the specific needs of that project

Documented processes provide the structured basis for creating metrics that can be used to understand our process capabilities and analyze our process results to identify areas for improvement. Standardized software processes are necessary for training, management, review and support.

Keep It Simple

There are many different methods for mapping processes. Examples include the IDEF0 modeling language [IEEE-1320] and entity process models [Humphrey-89]. There are also different proprietary tools that are specifically intended for use in defining processes. However, the main reason I use the method described in this paper is that it is very simple. While many larger companies have process experts, most of the small to mid-sized companies that I work with don't have that luxury. The people charged with defining and documenting their processes are the same people that are responsible for getting the product out the door. They don't have the time to learn special techniques or tools (or the money to purchase these tools).

The process documentation method I describe in this paper can be implemented using basic PC based word processing and graphics tools like Microsoft Word and PowerPoint. I have found that these tools are readily available to most people in even the smallest organizations. This allows everyone to easily document and maintain their own processes with minimal additional skills training and expense.

PROCESS DEFINITION OUTLINE

The method I use to document a process is based on the ETVX process defintion format which includes the definition of Entry criteria, Tasks, Verification and eXit criteria for a process. I expand this outline to include a statement of the purpose of the process, inputs to the process, a process flow diagram, the outputs from the process and the process metrics. Appendix A includes a template for documenting a process. Example process definitions are included in Appendices B and C:

- Appendix B - Software System Testing Process
- Appendix C - Execute System Tests & Report Anomalies Process

Purpose

The purpose is a statement of the value added reason for the existance of the process. It defines what we are attempting to accomplish by executing the steps in the process. For example, the purpose of a Software Testing process might be to validate the software system against the approved requirements and identify product defects before the product is released to the customer.

Inputs

Inputs are the tangible, physical objects that are input into and utilized during the process. These inputs may be work products create as part of other internal processes or they may be purchased items or other items supplied by sources external to the organization (e.g., customers, sub-contractors).

Entry Criteria

The entry criteria are specific, measurable conditions that must be met before the process can be started. This may include:

- Tasks and/or verification steps that must be satisfactorily completed
- Specific measured values that must be obtained
- Staff with appropriate levels of expertise that must be available to perform the process
- Other resources that must be available and/or ready for use during the process

Process Flow Diagram

A picture really is worth a thousand words. A simple flow diagram of a process can make that process easier to understand by showing the relationships between the various tasks, verification steps and deliverables and by showing who is responsible for each task or verification step.

Roles: The first step in creating a process flow diagram is to define the various roles of the process. These are the individuals or groups responsible for the steps that are part of the process. Their roles are listed in the "swim lanes" along the left side of the process flow diagram as illustrated in Figure 1.

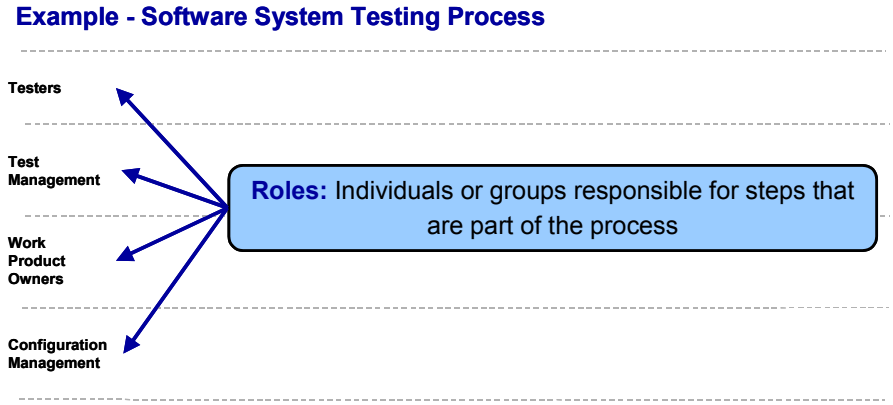


Figure 1 – Roles in a process flow diagram

In order to make our standardized processes as adaptable as possible, the roles should be labeled in generic terms rather than by using specific titles or the actual names of individuals or groups. This keeps us from being forced to update our process documentation every time there is a reorganization or staff turnover. It also allows us to easily tailor our processes to projects of various sizes. For example, on very large projects, an entire team might be assigned to one of the process roles but on very small projects a single individual might be assigned several roles.

Process Flowchart: A process flowchart can then be drawn across the "swim lanes" that illustrates the various tasks, verification steps, decisions and deliverables of the process, and how they are related to each other, as illustrated in Figure 2. This method makes it easy for an indivdual assigned to a role to read across their "swim lane" and identify their responsibilities.

If more than one role is responsible for a step, the box for that step spans multiple "swim lanes". For example, task T1: Requirements Elicitation in Figure 2 is the responsibility of the customer, marketing, product management and systems engineering. Occasionally, a task needs to span more than one "swim lane" but the order of the "swim lanes" does not allow for a single solid box to be used. This can be illustrated using a split box as illustrated in the verification step V1: FRS Technical Review & Approval in Figure 2. It should be noted that sometimes a simple reordering of the roles in the "swim lanes" can eliminate the need for this special notation. For example, in Figure 2, if the marketing and customer roles were swapped, a single solid box could be used.

Each task, verification step and deliverable in the flowchart is labeled sequentially (e.g., T1, T2, …,Tn, V1, V2, Vn and D1, D2, …, Dn) so that they can quickly referenced to their associated descriptive text.

Decisions that need to be made as part the process can be illustrated as a diamond (the standard flow chart symbol for a decision). These decisions be part of a step and be included inside step box (see example in step VI: Conduct Periodic Test Status Reviews the Software System Testing Process diagram in Appendix B) or these decisions can be separate tasks or verification steps in the process.

Along the bottom of the process flow diagram, I document the major deliverables of the process. I use the standard flow chart symbols for documents and data stores to distinguish between documents and electronic files or database items.

Example - Define System Requirements Process

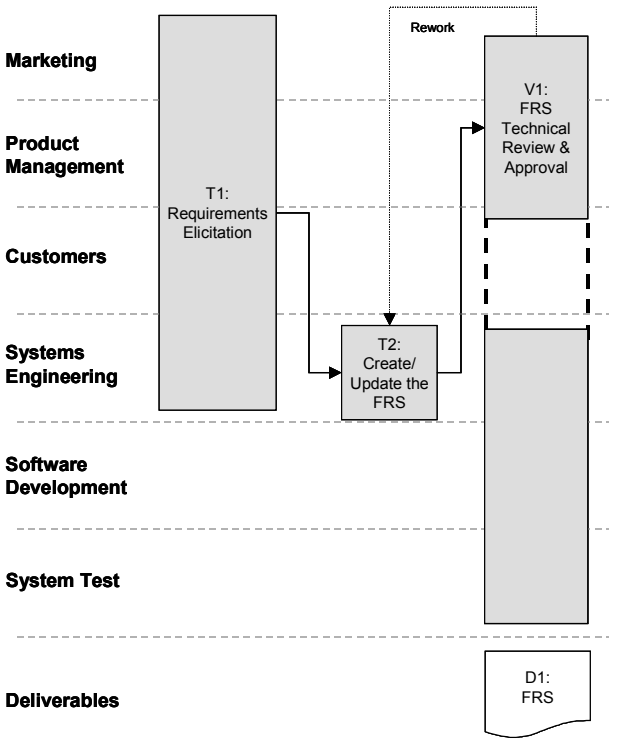


Figure 2 – Process Flow Chart

…, be of can a in flow

Tasks & Verification Steps

A task is a cohesive, individual unit of work that must be accomplished to complete a process. A verification step is a set of specific actions taken to evaluate the outcome(s) of one or more tasks to determine whether the requirements and/or specifications have been met and that the deliverables meet the required quality levels. For example, as illustrated in Figure 2, there are tasks for elicitiing the requirements and for creating the Functional Requirements Specification (FRS). The verification step V1 then evaluates the FRS document (the outcome of those tasks) through a technical review and approval step.For each task or verification defined in a Process Flow Diagram, a textual description should be included in the process definition. These descriptions may include:

- Detailed work instructions (or pointer to the work instructions) that describe how to accomplish the task or verification step including specific techniques to be used
- Additional descriptions of specific responsibilities including the specific assignment for the primary responsibility for the task and/or its deliverables (e.g., the System Engineer is responsible for eliciting the requirements with inputs from Marketing, Product Management and the Customers)
- Pointers to standards to be used when conducting the task or verification step or when creating the output (e.g., verification standards like formal document inspection standards, work product standards like coding standards)
- Pointers to standardized templates for creating the outputs of the task or verification step (e.g., document templates, verification checklists, meeting agenda templates)
- Required levels of expertise (or pointers to the descriptions of required levels of expertise) that must be possessed by those responsible for the task or verification step
- Other resources (e.g., tools, hardware) that must be available and/or used

Metrics

Descriptions of (or pointers to the descriptions of) specific metrics collected and/or used as part of the process.

Exit Criteria

The exit criteria are specific, measurable conditions that must be met before the process can be completed.

Outputs

Deliverables: Deliverables are the tangible, physical objects or specific measurable accomplishments that are the outcomes of the tasks or verification steps. I typically show only completed deliverables and not intermediate work products. For example, in Figure 2 I don't show the draft versions of the FRS but only show the final FRS under the verification step that results in its approval/completion.

Quality Records: While deliverables are the direct, intended outputs from the processes, quality records are secondary outputs that provide the evidence that the appropriate activities took place and that the execution of those activities met the required standards. Examples of

quality records include: test logs, minutes from meetings, audit reports, engineering notebooks, action item lists, and status reports. For each quality record, I list:

- Its custodian (the role responsible for collecting that record)
- Where that record will be maintained (e.g. project file, specific database)
- The minimum retention period for that record (the minimum length of time that the record will be kept available for access)

PROCESSES ARCHITECTURE

I have found that different levels of detail are required for defining processes at different times. For example, a very high-level view of the entire development process as illustrated in Figure 3 may be appropriate for planning, training, executive management review, or customer discussions. However, when the actual process is being executed, a detailed step-by-step set of work instructions is needed. There may also be times when intermediate level process defintions are appropriate.

This can be accomplished through the creation of a process definitions at different levels of detail (process architecture). If the process definitions are kept online, the linking of these various levels of process documentation can be accomplished easily using hyperlinks. For example, the System Test box in Figure 3 has been hyperlinked to the Software System Test Process definition in Appendix B (click on the System Test Box to jump to the Software System Test Process definition).

Each box in this process flow diagram could be linked to the associated lower-level descriptions as appropriate. If more detailed process definitions are required, hyperlinks to the next lower-level of process definitions could be created and so on, creating a hierarchy of process definitions. For example, the Execute Tests & Report Anomalies box in the Process Flow Diagram in Appendix B has been hyperlinked to the Execute Tests & Report Anomalies Process definition in Appendix C (click on the Execute Tests & Report Anomalies Box to jump to the Execute Tests & Report Anomalies Process definition). As appropriate, some of the boxes in the Execute Tests & Report Anomalies Process could also be hyperlinked to lower-level processes (e.g., CCB Analysis).

One of the advantages of creating a process architecture is the removal of redundancy. For example, the Execute Test & Report Anomalies Process defined in Appendix C has been generically defined. It could also be linked to from the Integration Test procedure or the Beta Test procedure. This eliminates the need to repeat these instructions and makes it much easier to maintain the documented procedures. The process arcitecture also defines:

- The ordering of process elements
- Interfaces among process elements
- Interfaces with external processes
- Interdependencies among process elements

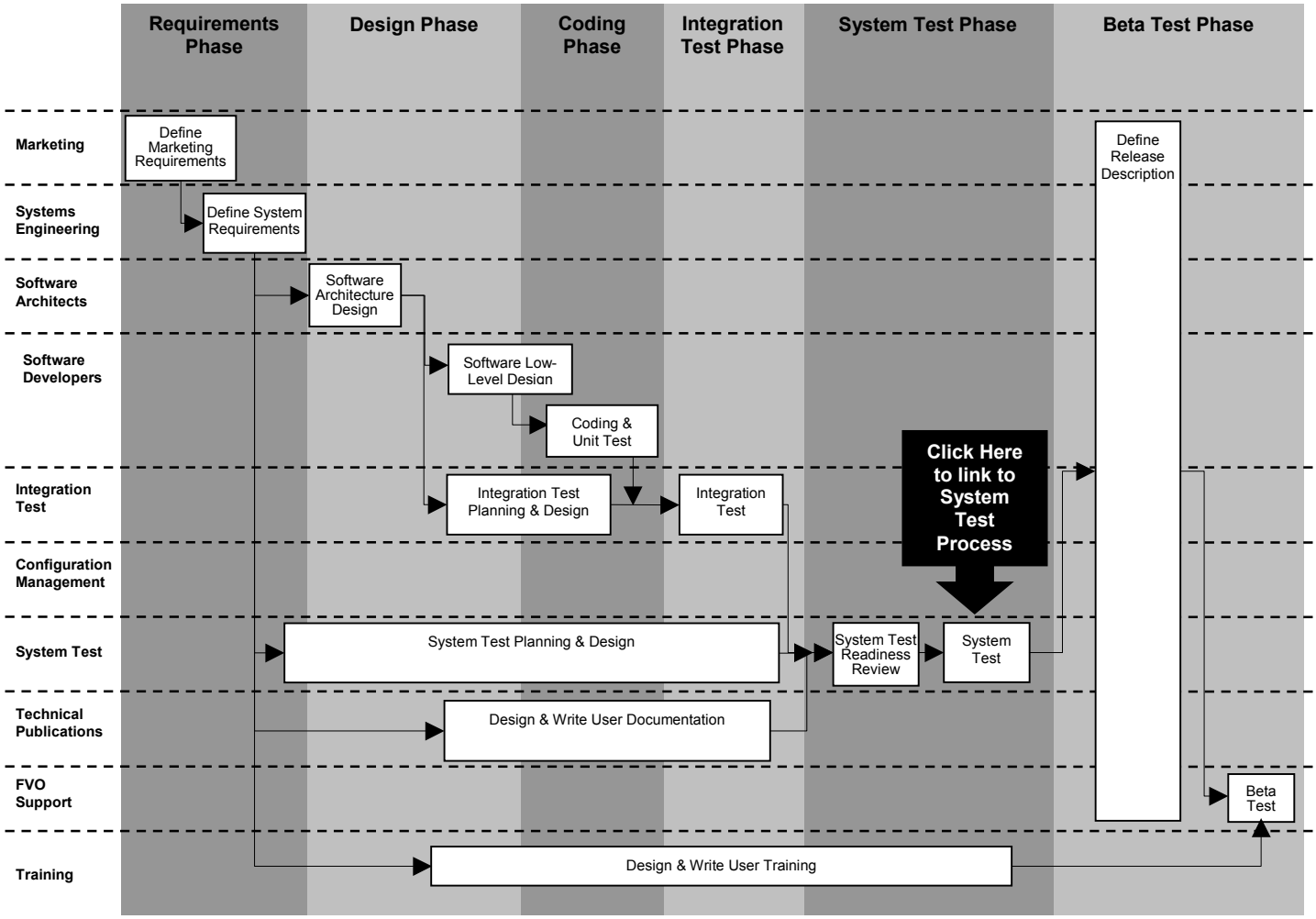


Figure 3 – Process Flow Diagram of Software Development Process

CONCLUSIONS

Having well defined and documented processes is an essential element of a quality system. However, this does not mean that specialized expertise and tools are required. Simple, useful process documentation can be created using the word processing and graphical tools available on almost any PC.

A heirarchical process definition structure can increase the usability of the process documentation while at the same time making them easier to maintain.

REFERENCES

Down-94: Alex Down, Michael Coleman, Peter Absolon, *Risk Management for Software Projects*, McGraw-Hill Book Company, London 1994.

Humphrey-89: Watts Humphrey and Marc Kellner, *Software Process Modeling: Principles of Entity Process Models*, http://www.sei.cmu.edu/publications/documents/89.reports/89.tr.002.html

IEEE-1320: IEEE Standards Software Engineering, Volume 4, IEEE Standard for Functional Modeling Language – Syntax and Semantics for IDEF0, IEEE Std. 1320.1-1998, The Institute of Electrical and Electronics Engineers, 1999, ISBN 0-7381-1562-2.

IEEE-610: IEEE Standards Software Engineering, Volume 1, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std. 610-1990, The Institute of Electrical and Electronics Engineers, 1999, ISBN 0-7381-1559-2.

Appendix A – Process Documentation Template

<Process Name>

PURPOSE

- <List of the key value added reasons for executing this process>
-
-

INPUTS

- <List of the inputs into this process (e.g. documents, work products or other items utilized during the process>
-
-

ENTRY CRITERIA

- <List of the specific measurable conditions that must be true before this process can be started>
-
-

PROCESS FLOW DIAGRAM

TASKS

T1. <A sequenced list of the specific steps and their descriptions that must be performed to execute the process>
T2.
T3.

VERIFICATION

V1. <A sequenced list of verification steps that are performed during the process as checkpoints to ensure that the process is being performed as required and that the products of the process meet the required quality levels>
V2.
V3.

METRICS

- <List of the specific metrics/measurement collected during and/or used during the process>
-
-

EXIT CRITERIA

- <List of the specific measurable conditions that must be true before this process can be completed>
-
-

OUTPUTS

The following deliverables are outputs of the test execution process:

D1. <A list of the major deliverables of the process and their descriptions>
D2.
D3.

The following quality records are outputs of the system test execution process:

Required Record	Custodian	Retention Period
<quality record name>	<role responsible for collecting/maintaining the quality record>	<retention period for the quality record>

Appendix B – Example Process Definition: Software System Test Process

Software System Testing Process

PURPOSE

- To validate the software system against the approved requirements
- To identify product defects before the product is released to the customer

INPUTS

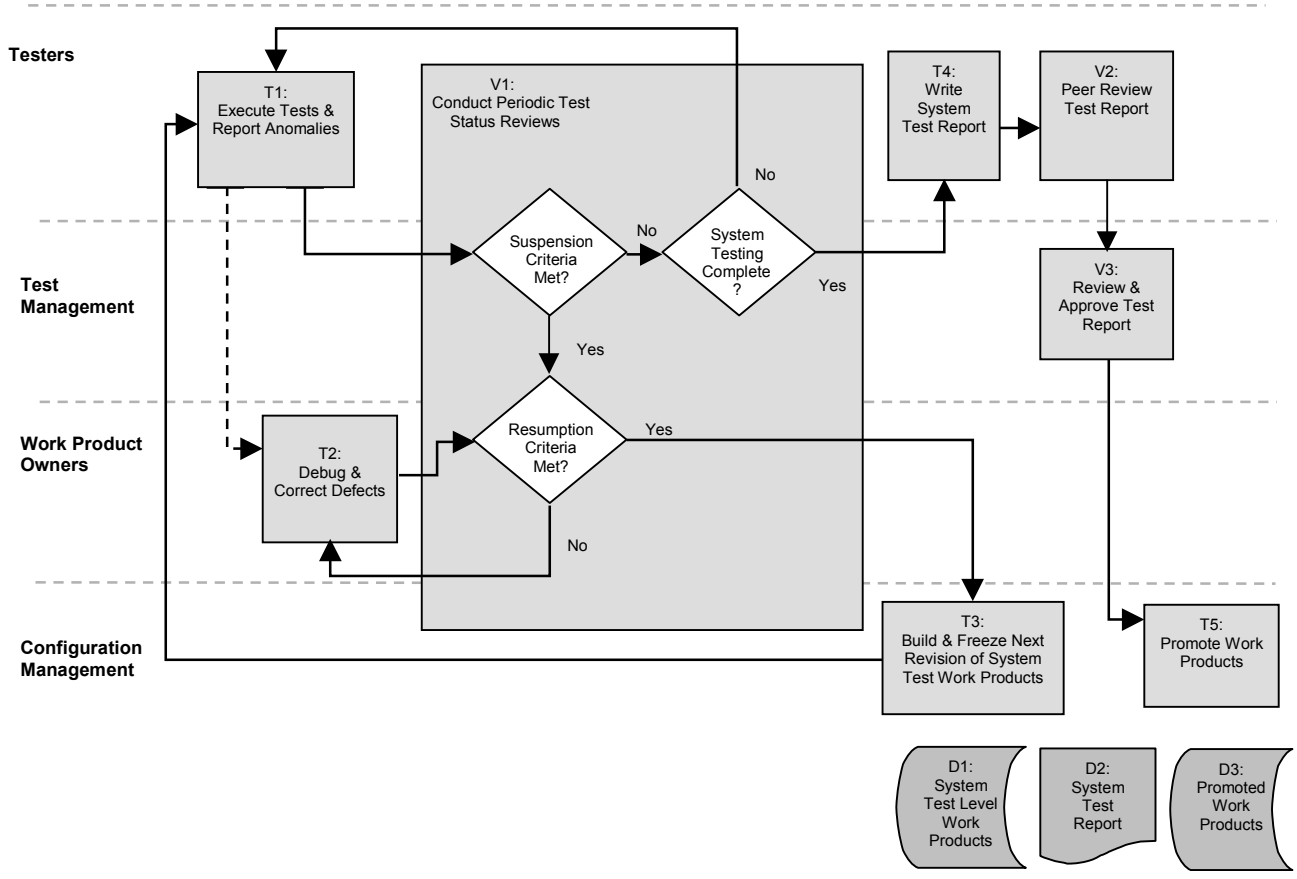
- System Test Plan approved and under CM control
- System Test Cases and System Test Procedures approved and under CM control
- User documentation and software installation procedures approved and under CM control
- Software build promoted to System Test state (note that the initial System Test level software work products are promoted from Integration Test).

ENTRY CRITERIA

The following criteria are verified during the System Test Readiness Review:

- System Test Lab ready and available for use
- Integration Test successfully complete
- Testing staff available with the appropriate levels of expertise

PROCESS FLOW DIAGRAM



Deliverables

TASKS

T1. Execute System Tests & Report Anomalies: The system tester executes a selected set of test cases for each system test load.

- If system test is suspended and restarted, these selected test cases will include cases to test all corrections and to regression test the software as appropriate.
- Any anomalies identified during system test are reported by the tester in accordance with the Anomaly Reporting process.

T2. Debug & Correct Defects: The owner(s) (e.g., software development, technical publications) of each work product that is suspected to have caused the anomaly debugs that work product and corrects any identified defect(s) in accordance with the Problem Resolution process.

T3. Build & Freeze Next Revision of System Test Work Products: Configuration management builds any updated revision of the software product(s) (e.g., software load, users manual, and installation instructions) that include the identified corrected components in accordance with the Software Build Process.

T4. Write System Test Report: At the end of the final cycle of System Test execution, the tester writes a System Test Report that includes a summary of the results from all of the System Test cycles.

T5. Promote Work Products: After the final test report is approved, all of the system test work products are promoted to the acceptance test status in accordance with the Configuration Management Promotion Process.

VERIFICATION

V1. Conduct Periodic Test Status Reviews: System test status review meetings are held on a periodic basis (as specified in the system test plan) during system test. If at any time it is determined that the suspension criteria (as specified in the system test plan) are met, system test execution is halted until the resumption criteria (as specified in the system test plan) are met and new revisions of the software work products are built and/or frozen.
These meetings are also used to determine when system test is complete based on the system test completion criteria (as specified in the system test plan).

V2. Peer Review Test Report: the testers peer review the system test report in accordance with the Peer Review process.

V3. Review & Approve Test Report: test management reviews and approves the final test report for distribution.

METRICS

- Number of test cases executed, passed, failed and blocked: This metric is reported at the weekly project team status meetings and utilized as input to determine the completion status of the system testing process.
- Anomaly report arrival rate by severity: This metric is reported at the weekly project team status meetings and utilized as input to determine the completion status of the system testing process and the quality level of the software deliverables.
- Cumulative anomaly reports by status: This metric is reported at the weekly project team status meetings and utilized as input to determine the completion status of the system testing and defect resolution processes, and the quality level of the software deliverables.

EXIT CRITERIA

- System Test completion criteria are met (as specified in the system test plan).

- Test Management approves System Test Report.
- Final System Test software work products are promoted to beta test status.

OUTPUTS

The following deliverables are outputs of the system test execution process:

D1. System Test Level Work Products: one or more intermediate System Test level software work products may be built to include the corrections to defects identified during System Test.

D2. System Test Report (see System Test Report Template).

D3. Promoted Software Work Products: at the completion of System Test, the final System Test level software work products are promoted to become the initial Beta Test level software work products.

The following quality records are outputs of the system test execution process:

Required Record	Custodian	Retention Period
System Test log	System Testers (project file)	1 year minimum
Minutes from all System Test Status Review meetings	System Test Manager (project file)	1 year minimum
Minutes from any Product Build Verification meetings held	Configuration Management (project file)	1 year minimum
Anomaly Reports of anomalies found during System Test	System Testers (Change Request Database)	1 year minimum
Pass/Fail/Blocked status of each test case	System Testers (Test Case Status Database)	1 year minimum
System Test Report peer review minutes	System Testers (project file)	1 year minimum

Appendix C – Example Process Definition: Execute Tests & Report Anomalies

Execute Tests & Report Anomalies

PURPOSE

- To execute a specific set of software tests
- To report and screen anomalies discovered during the execution of those tests

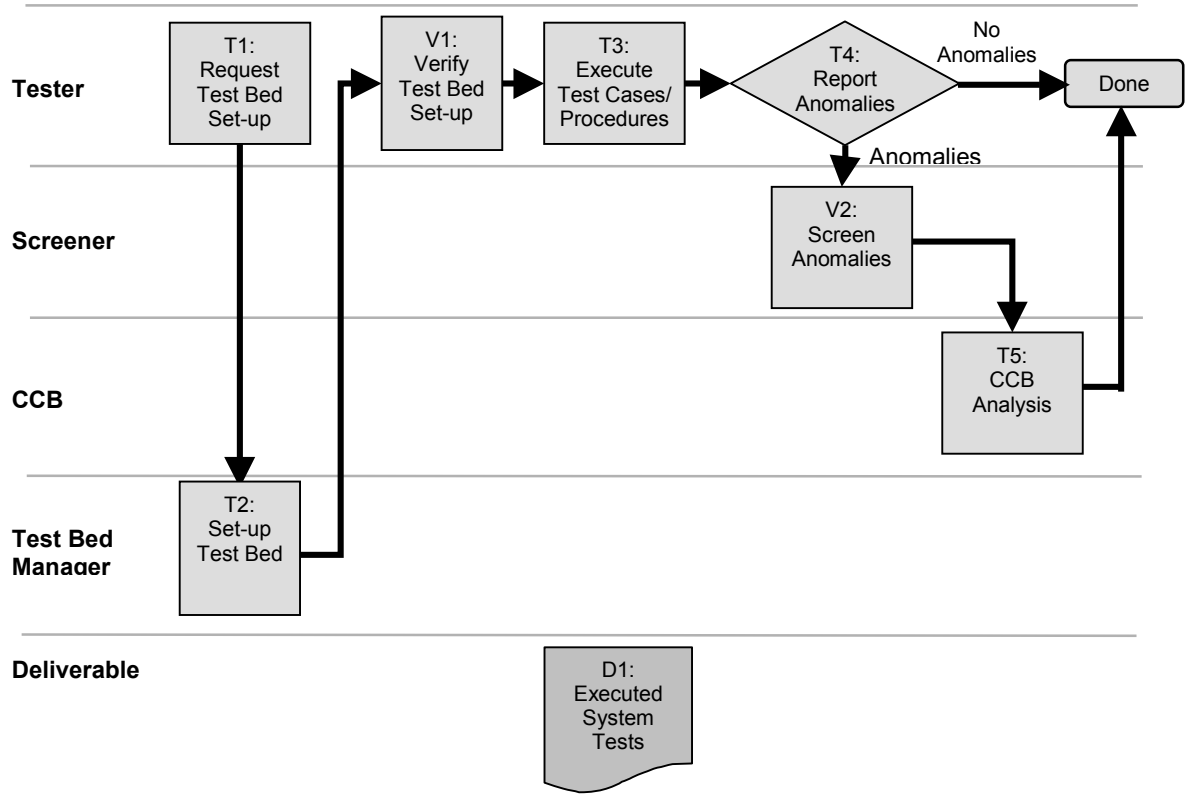
INPUT

- Test Cases and Test Procedures for the tests to be executed are approved and under CM control

ENTRY CRITERIA

- Test bed scheduled and available for use

PROCESS FLOW DIAGRAM



TASKS

T1. Request Test Bed Set-up: The tester completes a test bed configuration request form to reflect the test bed software, test database and hardware configuration required to conduct the test as specified in the test cases/procedures. This form is then submitted to the test bed manager.

T2. Set-up the Test Bed – The test bed manager configures the test bed as requested on the configuration request form.

T3. Execute Test Cases /Procedures: The tester executes the test cases/procedures and records the results of that execution in the test database (i.e., passed, failed, blocked). The tester records the test execution in the Test Log including the following information:

- Start/end time
- Tester(s) and observer(s)
- Environment (tools, test beds, simulators)
- Hardware configuration
- Software configuration (e.g., identification of which software work product versions/revisions are being tested)
- Cases/procedures executed
- Failures and anomalies observed
- Other notes and observations

T4. Report Anomalies: If there are any anomalies to report, the tester reports those anomalies in accordance with the Creating an Anomaly Report process. The tester should attempt to reproduce any anomalies discovered and report the results of those attempts as part of the description of the anomaly.

T5. CCB Analysis: The Change Control Board (CCB) then analyzes any reported anomalies in accordance with the CCB Analysis process. The tester reports any additional information as requested by the CCB.

VERIFICATION

V1. Verify the Test Bed Set-up: The tester uses the test bed configuration request form as a checklist and verifies that the test bed has been appropriately configured.

V2. Screen Anomalies: The screener evaluates the anomaly report for completeness and resolves any issues with the tester. This includes checking to ensure that the:

- Anomaly title is a clear, concise summary of the anomaly
- Description of the anomaly is complete and understandable and includes information about attempts to reproduce the anomaly
- Steps to reproduce the anomaly are described to an appropriate level of detail so that the developer can reproduce the anomaly during debugging
- Test bed configuration and hardware/software environment have been specified
- Test cases/procedures being run when the anomaly occurred are specified
- Severity of the anomaly has been appropriately specified

The tester reports any additional information as requested by the CCB.

METRICS

- None

EXIT CRITERIA

- All test cases/procedures have been executed or are blocked
- Test logs have been completed to record the test execution

- Anomaly reports have been created for all anomalies detected and those reports have been analyzed by the CCB

OUTPUTS

The following deliverables are outputs of the test execution process:

D1. Executed Tests: The set of test cases/procedures that are either executed or documented as blocked

The following quality records are outputs of the test execution process:

Required Record	Custodian	Retention Period
Completed Test Bed Configuration form	Testers (included as part of the Test Log)	1 year minimum
Test log	Testers (project file)	1 year minimum
Anomaly Reports of anomalies found during the execution of the tests	Testers (Change Request Database)	1 year minimum
Pass/Fail/Blocked status of each test case	Testers (Test Case Status Database)	1 year minimum

How to Create Useful Software Process Documentation

Presented by: Linda Westfall

The Westfall Team

3000 Custer Road, Suite 270, PMB 383
Plano, TX 75075-4499
phone: (972) 867-1172
fax: (972) 943-1484
lwestfall@westfallteam.com

www.westfallteam.com

Process are Codified Good Habits

A process is "a definable, repeatable, measurable sequence of tasks which produce a quality product for the customer" [Down-94]

Why Document Processes

Standardized software process documentation:

- Describes what usually works best
- Ensures that important steps aren't forgotten
- Eliminates need to "re-invent the wheel"
- Provides a foundation for tailoring to specific needs
- Facilitates propagation of lessons learned
- Provides a structured basis for measurement
- Required for training, management, review & tool support

Process Definition

Process definition outline:

- Purpose
- Entry criteria
- Tasks
- Verification
- Exit criteria
- Deliverables
- Quality records

ETVX Process Definitions

Purpose

Example - Software System Testing Process

Purpose:

- To validate the software system against the approved requirements
- To identify product defects before the product is released to the customer

Inputs

Example - Software System Testing Process

Inputs:

- System Test Plan approved & under CM control
- System Test Cases & System Test Procedures approved & under CM control
- User documentation & software installation procedures approved & under CM control
- Software build promoted to System Test state (note that the initial System Test level software work products are promoted from Integration Test).

Entry Criteria

Example - Software System Testing Process

Entry Criteria:

The following criteria are verified during the System Test Readiness Review:

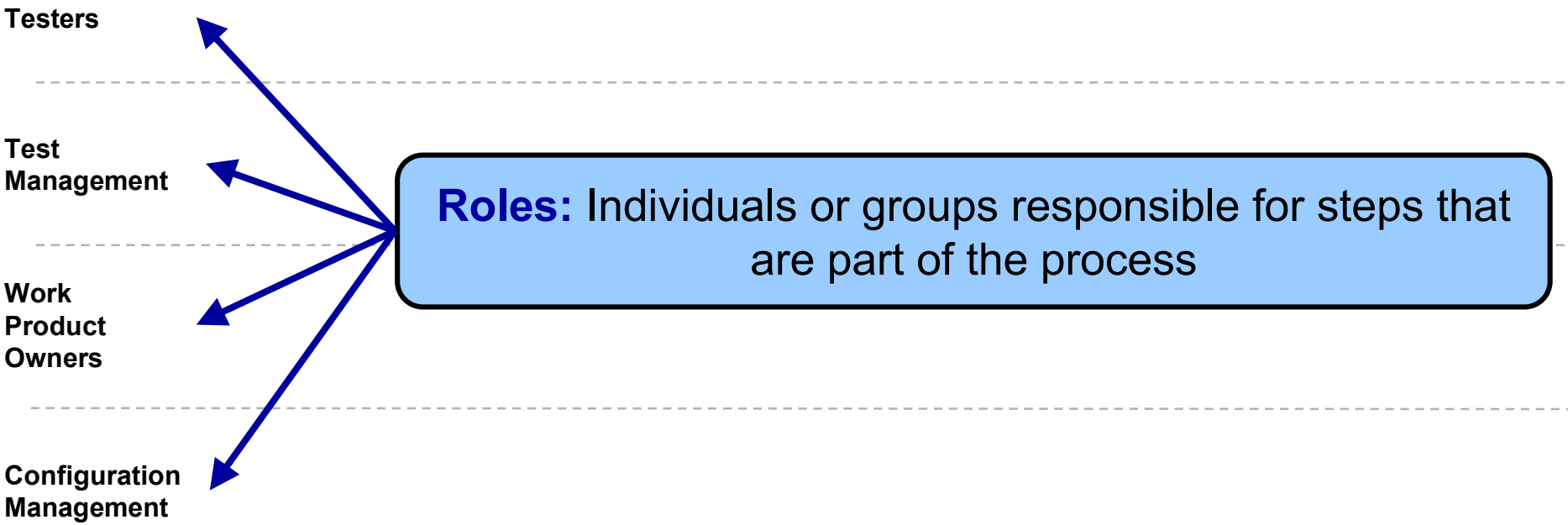
- System Test Lab ready & available for use
- Integration Test successfully complete
- Testing staff available with the appropriate levels of expertise

Process Flow Diagram

A pictures is worth a thousand words

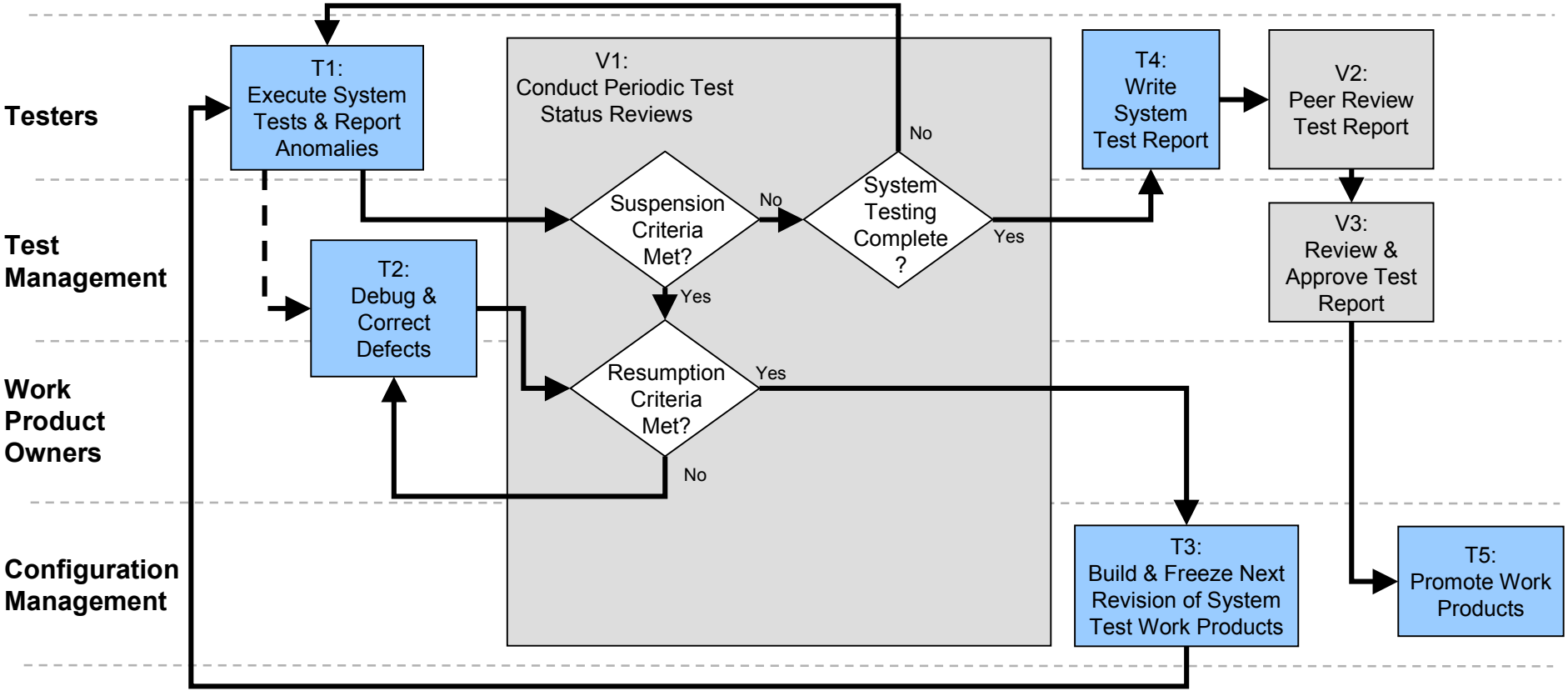
-- so create a Process Flow Diagram

Example - Software System Testing Process



Process Flow Diagram - Task Steps

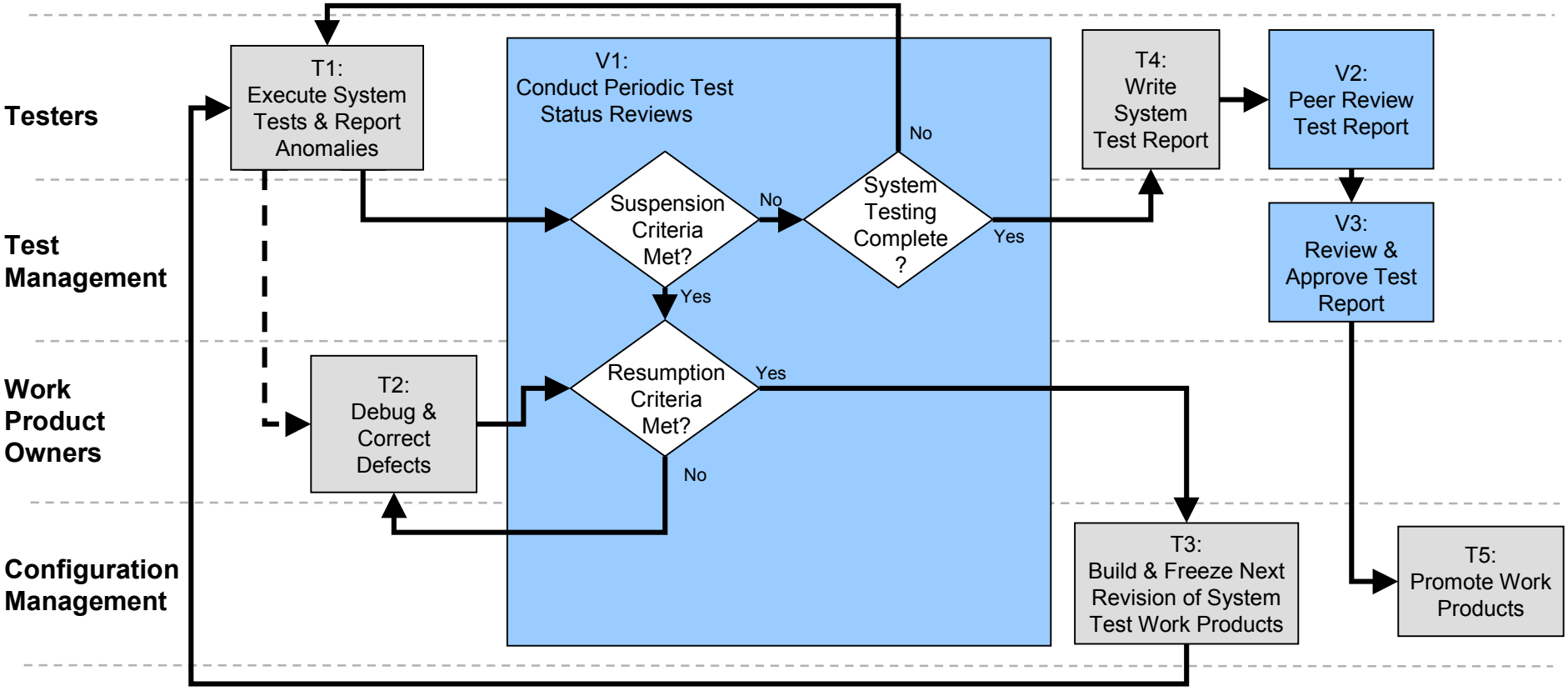
Example - Software System Testing Process



Task: A cohesive, individual unit of work that must be accomplished to complete a process.

Process Flow Chart - Verification Steps

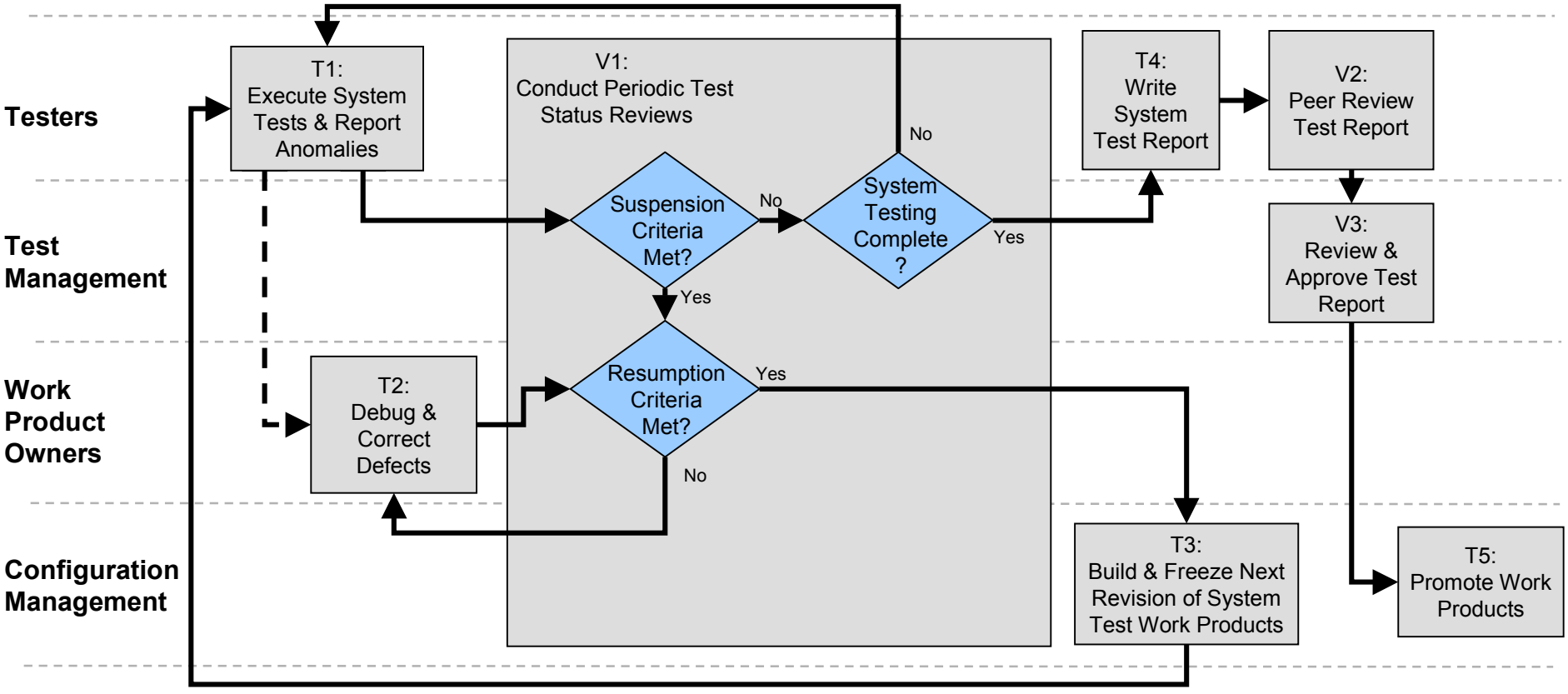
Example - Software System Testing Process



Verification: The evaluation of task outcomes to determine whether requirements and/or specifications have been met.

Process Flow Chart - Decisions

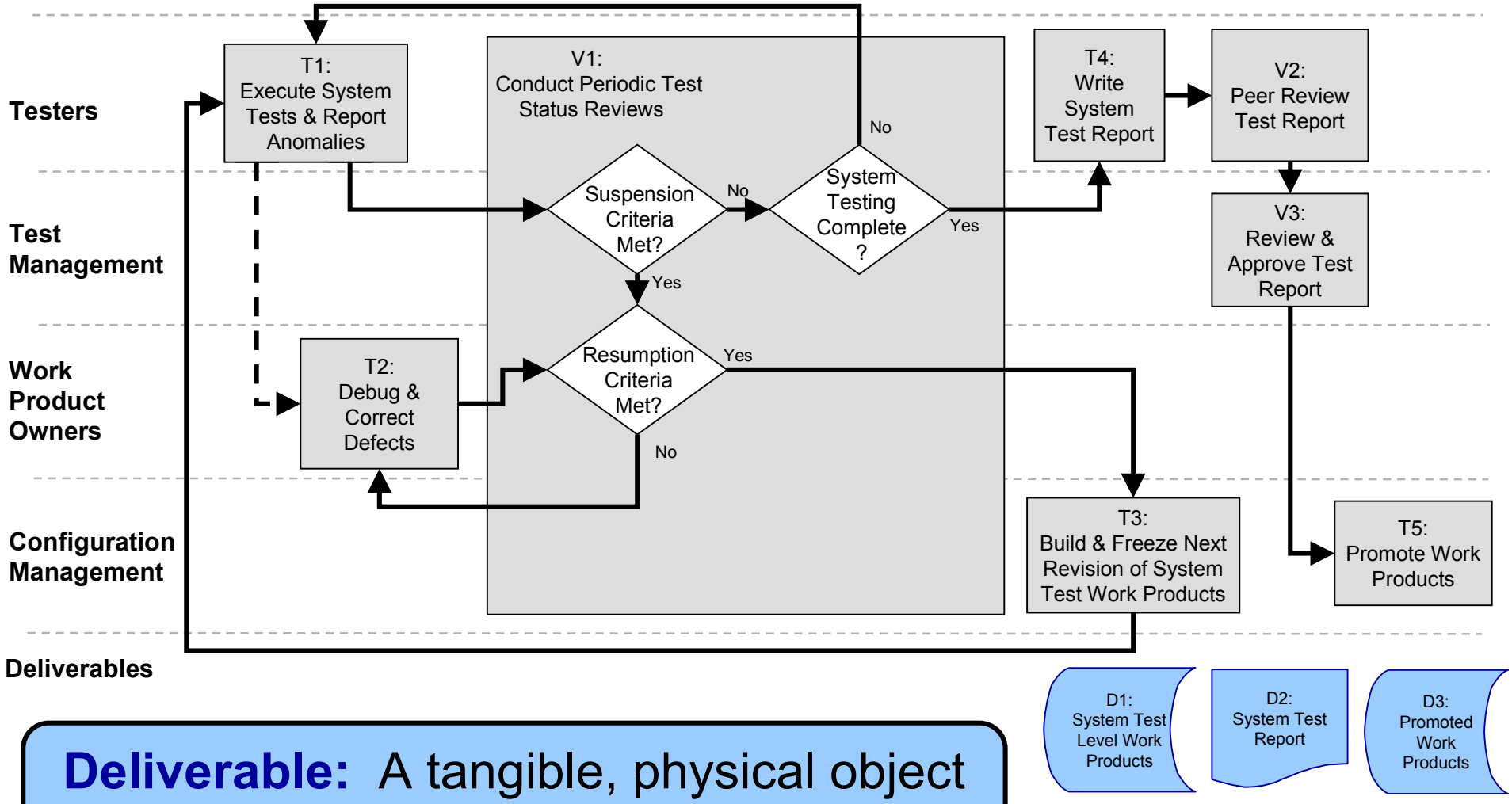
Example - Software System Testing Process



Decisions: Choices made based on predetermined criteria that control the flow of the processes.

Process Flow Chart - Deliverables

Example - Software System Testing Process



Deliverable: A tangible, physical object or specific measurable accomplishment

Task Definitions

Example - Software System Testing Process

Tasks:

T1: Execute System Tests & Report Anomalies: the system tester will execute a selected set of test cases for each system test load. If system test is suspended and restarted, these selected test cases will include cases to test all corrections and regression test the software. Any anomalies identified during system test are reported by the tester in accordance with the Problem Report process.

T2: Debug & Correct Defects: the owner (e.g., software development, technical publications) of each work product that is suspected to have caused the anomaly debugs that work product and corrects any identify defect(s) in accordance with the Problem Resolution process.

Task Definitions (cont.)

Example - Software System Testing Process

Tasks: (cont.)

T3: Build & Freeze Next Revision of System Test Work Products: configuration management builds any updated reversion of the software product(s) (e.g., software load, users manual, installation instructions) that includes the identified corrected components in accordance with the Software Build Process.

T4: Write System Test Report: At the end of the final cycle of System Test execution, the tester writes a System Test Report that includes a summary of the results from all of the System Test cycles.

T5: Promote the Work Products: configuration management promotes the system tested work products for use in beta testing.

Verification Definitions

Example - Software System Testing Process

Verification:

V1: Conduct Periodic Test Status Reviews: system test status review meetings are held on a periodic basis (as specified in the system test plan) during system test. If at any time it is determined that the suspension criteria (as specified in the system test plan) are met, system test execution is halted until the resumption criteria (as specified in the system test plan) are met and a new revision of the software load is build.

These meetings are also used to determine when system test is complete based on the system test completion criteria (as specified in the system test plan).

Verification Definitions (cont.)

Example - Software System Testing Process

Verification: (cont.)

V2: Peer Review Test Report: the testers peer review the system test report in accordance with the Peer Review process.

V3: Review & Approve Test Report: test management reviews and approves the final test report for distribution.

Metrics

Example - Software System Testing Process

Metrics:

- Number of test cases executed, passed, failed and blocked: This metric is reported at the weekly project team status meetings and utilized as input to determine the completion status of the system testing process.
- Anomaly report arrival rate by severity: This metric is reported at the weekly project team status meetings and utilized as input to determine the completion status of the system testing process and the quality level of the software deliverables.
- Cumulative anomaly reports by status: This metric is reported at the weekly project team status meetings and utilized as input to determine the completion status of the system testing and defect resolution processes, and the quality level of the software deliverable

Exit Criteria

Example - Software System Testing Process

Exit Criteria:

- System test completion criteria are met (as specified in the system test plan).
- System Test Report is approved by test management.
- Final System Test software work products are promoted to beta test status.

Outputs - Deliverables

Example - Software System Testing Process

Deliverables:

D1: System Test Level Software Loads: one or more intermediate System Test level software loads may be built to include the corrections to defects identified during System Test (note that the initial System Test level software load is the load promoted from Integration Test).

D2: System Test Report (see System Test Report Template)

D3: Promoted Software Work Products: at the completion of System Test, the final System Test level software work products are promoted to become the initial Beta Test level software work products.

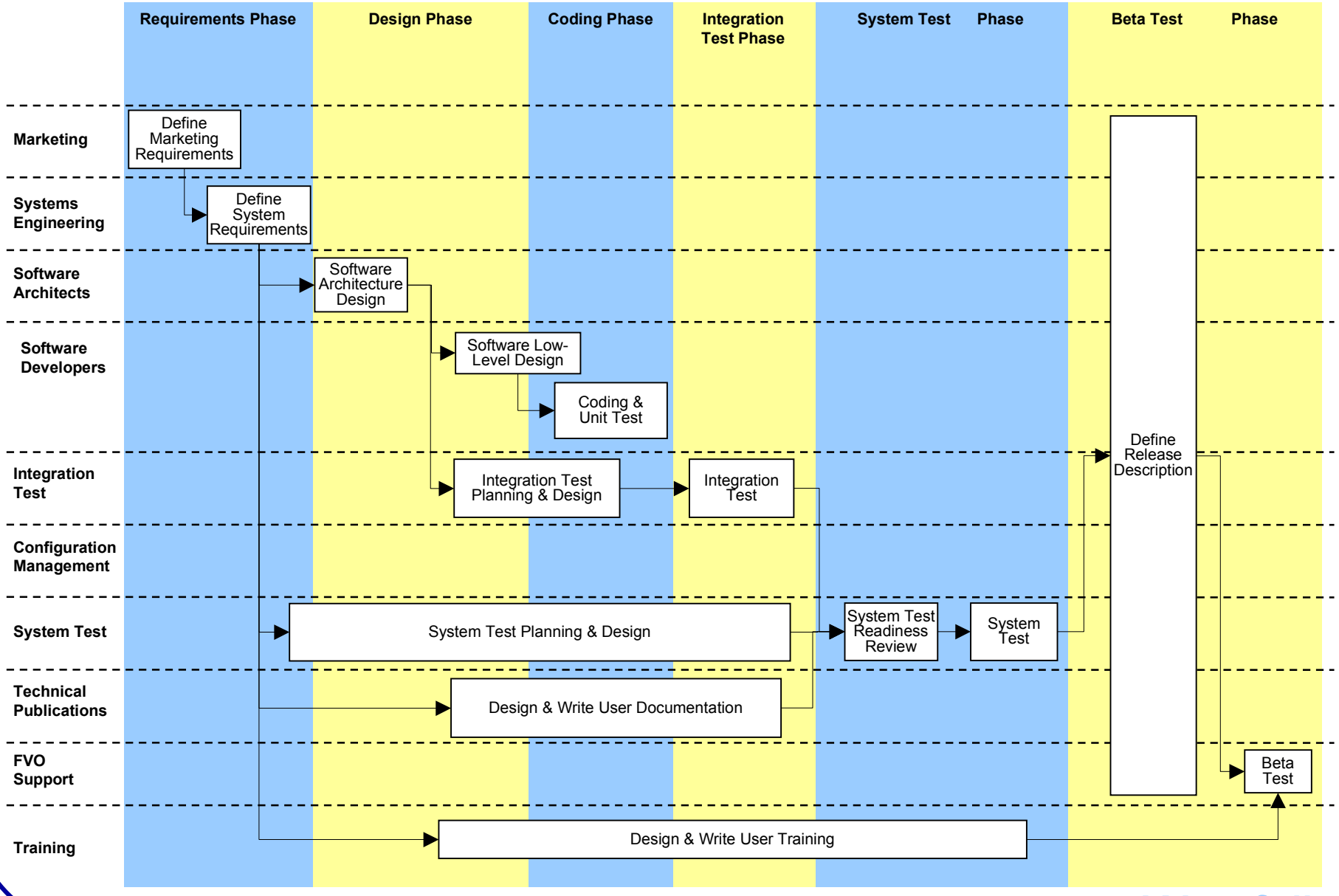
Outputs - Quality Records

Example - Software System Testing Process

Quality records:

Required Record	Custodian	Retention Period
System Test log	System Testers (project file)	1 year minimum
Minutes from all System Test Status Review meetings	System Test Manager (project file)	1 year minimum
Minutes from any Product Build Verification meetings held	Configuration Management (project file)	1 year minimum
Anomaly Reports of anomalies found during System Test	System Testers (Change Request Database)	1 year minimum
Pass/Fail/Blocked status of each test case	System Testers (Test Case Status Database)	1 year minimum
System Test Report peer review minutes	System Testers (project file)	1 year minimum

Processes Architecture



The Westfall Team

Exercise - Purpose

Example – Defining Software Requirements

Purpose:

-
-

Exercise - Inputs

Example – Defining Software Requirements

Inputs:

-
-

Exercise – Entry Criteria

Example – Defining Software Requirements

Entry Criteria:

-
-

Exercise - Process Flow Diagram

Example – Defining Software Requirements

Role

Role

Role

Role

Deliverables

Exercise – Metrics

Example – Defining Software Requirements

Metrics:

-
-

Exercise – Exit Criteria

Example – Defining Software Requirements

Exit Criteria:

-
-

Exercise – Outputs

Example – Defining Software Requirements

Deliverables:

-
-

Exercise – Outputs (cont.)

Example – Defining Software Requirements

Quality Records:

Required Record	Custodian	Retention Period

Are your diagrams human readable?

Dr. Roger Ferguson
Associate Professor
Department of Computer Science and Information Systems
Grand Valley State University
2227 Mackinac Hall
Allendale MI, 49401

Ralph Palmer
Burke E. Porter Machinery Company
Custom Machines
Director Of Engineering
730 Plymouth N. E.,
Grand Rapids,
Michigan 49505
www.bepco.com
e-mail: ralph.palmer@bepco.com

Abstract:

Throughout the software development process there are countless numbers of meetings where information is being presented to clients, developers and business professionals using diagrams annotated with "natural language elements" as a way to communicate this information (e.g., simple sets diagrams, block diagrams, work-flow diagrams, Entity/Relationship diagrams, Object Role Modeling, UML). How well do these diagrams communicate information to the intended audience? The following paper presents several principles that if followed in designing any diagram (or model) will help maximize the "communication" of the diagram; where communication is the comprehensibility, and learnability of that diagram as it relates to a person. These principles are based upon research performed by the linguist and philosopher Noam Chomsky and first theorized in 1953. Chomsky introduced a concept called, "Universal Grammar" which established the idea that humans are "hard-wired" to learn languages in a specific way. This "hard-wired" nature of humans can be exploited to maximize the "communication" of the diagram (or model). In other words, the principles established by Chomsky can (and ought to) be applied to creating, designing and developing your diagrams and modeling diagrams.

Acknowledgements: Karen Armantrout for her contribution to this paper.

Biographical:

Roger Ferguson is an Associate Professor in the department of Computer Science and Information Systems at Grand Valley State University in Michigan. He received his PhD in Software Engineering from Wayne State University and a Bachelors degree in Electrical Engineering from Michigan Technological University. He has worked with Ralph Palmer for about 1 year on this research, helping Ralph receive his master's degree.

Ralph Palmer is the Director of Engineering at Burke E. Porter Machinery in Grand Rapids Michigan. Burke E. Porter develops test equipment for the automotive industry specializing in Chassis Dynamometers and Roll/brake assembly verification machines. Ralph has a BSEE from Purdue University and a BA in German from Purdue University. Additionally, Ralph understands Hungarian and has an interest in languages. He will graduate with a MSCIS from Grand Valley State University in December 2002.

1.0 Introduction

There are countless numbers of meetings throughout the software development process. In these meetings, diagrams formally (and informally) annotated with natural language are used to present information to clients, developers and business professionals. How well do these diagrams communicate complete and unambiguous information to the intended audience?

The linguist and philosopher Noam Chomsky first theorized in 1953 that the mind could not produce the grammar required for human language from the data available in the environment alone [2]. The argument is known as the "poverty-of-the-stimulus". His conclusion was that the essence of language knowledge must already be in the mind [1]. In other words, language knowledge is biological. Just as man is biologically programmed to grow arms and not wings, so too is all mankind programmed to develop a "Language Facility" or a human grammar. He named this concept Universal Grammar (UG) and for him it is the essence of language knowledge [1,2,4,9]

Universal Grammar asserts that the apparent differences in the seemingly very different languages are in reality quite small and that "at some fundamental level all human languages conform to a particular pattern…" and that "syntactic structure plays a central role between physical form and abstract meaning". The principles of UG "lay down absolute requirements that a human language has to meet, the parameters of the Universal Grammar account for the variations between languages" [1,pg 55]. What we learn when we learn our mother tongue is a set of principles and parameters for a particular language. Thus, we learn the principles and parameter *settings* for a specific language. These principles and parameters that all languages must not violate UG is what endow us to learn human languages.

How does this relate to humans reading diagrams? Chomsky stated: if a language violated UG, we simply would not be able to learn that language the way we learn a human language. We would have to approach it slowly and laboriously the way scientists study physics, in which it takes generations after generations of labor to gain new understanding and make significant progress. Thus UG is directly linked to "learnability". Simply put, if the diagram does not follow UG, the diagram will require more specialized knowledge to read and thus be more difficult to understand.

2.0 The Problem stated more formally:

Diagrams are intended to illustrate a problem, communicate an idea, or present a solution to a problem. People put together these diagrams (e.g., simple sets diagrams, block diagrams, work-flow diagrams, Entity/Relationship diagrams, Object Role Modeling, UML) to represent their ideas. The problem is, they have no metric to guide the design of their diagram or model that indicates whether the intended audience will easily understand the diagram. This paper will demonstrate that following the basic principles of UG will produce diagrams or models that are more comprehensible and learnable and thus more readable by humans without specialized knowledge.

The paper has the following sections: Section 3.0, basic background of Universal Grammar (UG). Section 4.0, 5.0, is the analysis of some different modeling techniques using UG as a metric. Section 6.0 contains the experiments showing that using diagrams or modeling that follows UG standards do communicate better. The final section, section 7.0, contains the conclusion to our paper and further research thoughts.

3.0 Basic Principles and Parameters of Universal Grammar (UG)

The following section is based on *Chomsky's Universal Grammar* by Cook and Newson [1], i.e., this section demonstrates the basic concepts and principles of Universal Grammar (UG). Initially, it may seem confusing as to how to apply these concepts to diagrams (or models); however we demonstrate how this is done in section 4.0.

(Before starting this section, a quick note to the reader. This research and corresponding paper is early in the development process. We are trying to get feedback as to whether the idea has merit and is worth pursuing. We know there is much more to UG than what is presented. For example: Full Interpretation and Economy Principles are considered to supercede or are more or less "super" principles of Theta Criterion. Some, who know UG might point out that we do not recognize this. We are attempting to use only parts of UG for our purposes. Thank you for your consideration.)

There are degrees of **"markedness"** for the principles and parameters of Universal Grammar. The more a grammar feature departs from the core of Universal Grammar, the more **marked** it is. This implies that "markedness" is linked to learnability [1]. Simply put, if the diagram is highly marked, the diagram will require more specialized knowledge to read and thus be more difficult to understand.

Predicate: A predicate expresses the semantic and/or syntactic relationship between entities. For example the predicate "give": Entity A "gives" to entity B something C.

θ-Roles (theta-Roles): Is best described by an example, given the phrase **"Tom gave Scott a car."**. The predicate "give" has three θ-Roles, which are:

- An agent (Tom) to do the giving
- a theme (car) to be given
- A goal (Scott) to be the recipient of the giving.

Lexical Entry: is a predicate with all of its θ-Roles (e.g. the predicate give with an agent, theme and goal)

θ-Criterion Principle: To satisfy **θ-Criterion** a predicate "give" must have all θ-Roles present in the diagram (e.g., predicate give with an agent, theme, goal) to comply with UG. More formally, the θ-Roles described by a predicate have one interpretation. Each obligatory θ-role

selected by a predicate must be assigned to a referential expression. Each referential expression must be assigned a θ-role [1].

θ-Theory: θ-Theory is concerned with the way in which θ-roles are assigned. The θ-Theory states that θ-Roles tend to be assigned in a uniform direction. In English, this basically means there are two types of θ-roles assignments, internal and external [1]. Internal θ-roles are assigned to the right of the predicate, and external θ-roles are assigned to the left of the predicate.

Structure Dependency Principle: Word order is very important. The sentence "Mary likes John" does not mean the same, as "John likes Mary". Although both sentences are grammatically correct and each sentence contains exactly the same three words, they have different meanings and the correct interpretation is defined by the syntactic relationships of the words.

Full Interpretation Principle: Every word or symbol of a language must serve a meaningful purpose. For example: ∀ x 2 + 2 = 4, where ∀ x 2 means "for all x's" is superfluous to the meaning of 2 + 2 = 4. This is acceptable in mathematics but is not acceptable in human languages. Sentences such as "Every man like woman" is ungrammatical meaning "every" is superfluous to the meaning of the sentence. Full Interpretation means that every element that appears in a structure must have an interpretation.

Economy Principle: Superfluous elements of a language are not desirable. Both derivations and representations are subject to a certain form of "least effort" condition and are required to be minimal in a fairly well defined sense, with no superfluous symbols.

4.0 Analysis of Set Diagrams E/R, and UML Modeling Techniques

In this section we demonstrate how to apply UG to different design techniques (e.g., sets, UML, ER) where the design techniques will illustrate the following relationship.

Relationship:

"some instructors evaluate using a test of several students"
"some student is evaluated by this test by several instructors"

4.1 Set example:

Consider the following binary relationship represented as a set diagram below.

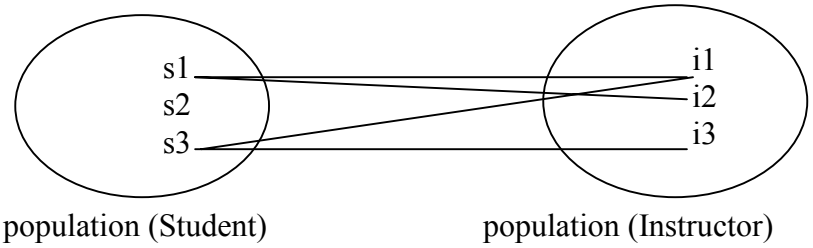


Figure 4.1 Using sets to show the relationship.

The set diagram has several problems; most notably, the diagram does not show the relationship between the entities (i.e., "evaluates" is not present in the diagram but is present in the relationship above). There are only lines, and nothing to indicate what these lines mean. This diagram completely relies on the client to understand the relationship prior to looking at the diagram. The relationship of "Instructors" to "Students" (and visa-versa) is not revealed by the diagram. There is a complete lack of information using set diagrams, and set diagrams typically do not satisfy θ-Theory, θ-Criterion and Structure Dependency Principle. This diagram is highly marked.

4.2 ER Model Binary example

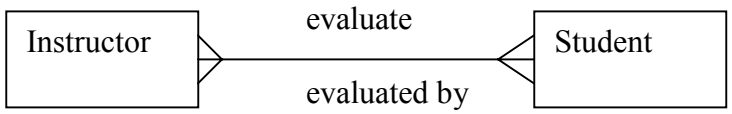


Figure 4.2 Using ER model to show the relationship.

As would be expected, the use of natural language for the association in the diagram conveys much better the relationship between the two entities. Using ER the model does show a relationship between the Student and the Instructor using the predicate "evaluate": The lexical entry for "evaluate" used in this manner is "evaluate<agent, possessor, patient>[1]". The Instructor is the external θ-role (which is assigned to the left of the predicate), but the student could be either the patient or the possessor (which are assigned to the right of the predicate). The diagram has left out a θ-role; as a result, the diagram is ambiguous

θ-Criterion: The θ-Criterion is not satisfied: Assuming "student" is the "patient", then the possessor θ-Role is left unassigned which leaves the criteria uncompleted; hence, the basis for understanding this diagram is subjective and dependent on the person who is looking at the diagram.

θ-Theory: The θ-theory is satisfied for the roles shown. However, due to the missing θ-Role, θ-theory can not assign the missing θ-role and the criteria uncompleted; hence, the basis for understanding this diagram is subjective and dependent on the person who is looking at the diagram.

Structure Dependency: The diagram can be read in both directions. This works because both readings do not violate the θ-criterion. That is, the agent and the patient roles are identically assigned with "instructor" taking the agent role and "student" taking the patient role in both readings. Although, the passive reading looks like it might violate θ-Theory, it does not since the Internal and External θ-Roles are assigned correctly.

Full Interpretation Principle: That every word or symbol of a language must serve a meaningful purpose has been satisfied.

Economy Principle: This diagram has no superfluous elements and therefore the economy principle is satisfied.

Finally, this diagram is significantly "marked".

4.3 UML Model Binary example

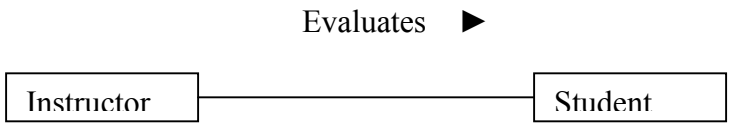


Figure 4.3 Using UML model to show the relationship.

Again, as would be expected, the use of natural language conveys a relationship between the two entities. However, this diagram is more "marked" than the previous ER diagram.

θ-Criterion: The θ-Criterion is not satisfied: Assuming "student" is the "patient", then the possessor θ-Role is left unassigned which leaves the criteria is incomplete; hence, the basis for understanding this diagram is subjective and dependent on the person who is looking at the diagram.

θ-Theory: The θ-theory is satisfied for the roles shown. However, due to the missing θ-Role, θ-theory can not assign the missing θ-role and the criteria is incomplete; hence, the basis for understanding this diagram is subjective and dependent on the person who is looking at the diagram.

Structure Dependency: Acknowledges the dependency by providing reading arrow. Unfortunately using this arrow causes a one-way relationship, i.e., the diagram does not show the relationship from Student to Instructor, and there does exist a relationship from Student to Instructor that is not revealed. Therefore, it is very difficult to determine if Structural Dependence has been satisfied since only one relationship is shown. Hence this solution is not declarative.

Full Interpretation Principle: That every word or symbol of a language must serve a meaningful purpose has been satisfied.

Economy Principle: Does not comply with the Economy Principle. The association direction-reading arrow is a superfluous element. There is only one possible interpretation for the association and therefore the use of the association direction-reading arrow is not needed.

5.0 Ternary Analysis

We now turn our attention to a more complex relationship, a ternary relationship. In this section we apply UG to the different design techniques as was done in section 4.0. Consider the following ternary relationship represented below.

Relationship:

"an instructor evaluates students with a test"
"a test is the basis of an instructor's evaluation"
"a student's evaluation is based on an instructor's test"

5.1 Set example: Consider the ternary relationships represented in Figure 5.1

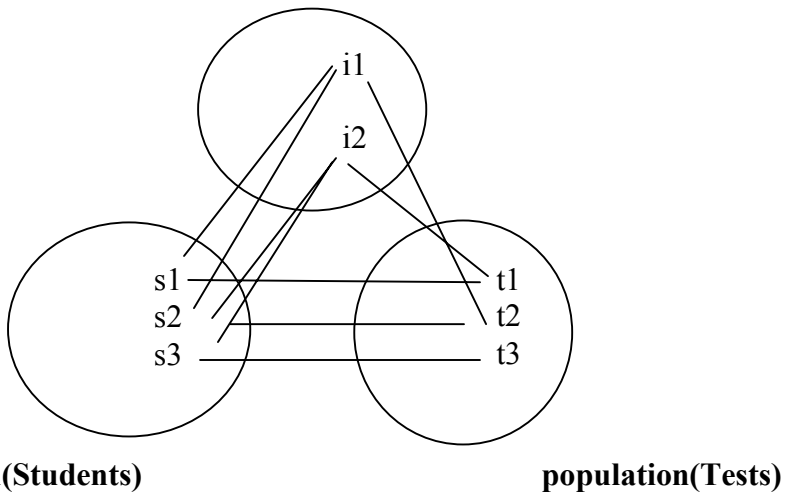


Figure 5.1 Using Sets diagrams to show the relationship.

The set diagram has several problems; most notably, the diagram does not show the relationship between the entities. That is, there are only lines, and nothing to indicate what these lines mean. This diagram completely relies on the client to understand the relationship prior to looking at the diagram. The relationships of "Instructors" to "Tests" to "Students" (and visa-versa) are not

revealed by the set diagram. There is a complete lack of information using set diagrams, and therefore, set diagrams typically do not satisfy θ-Theory and Structure Dependency Principle.

5.2 ER Model Ternary Example

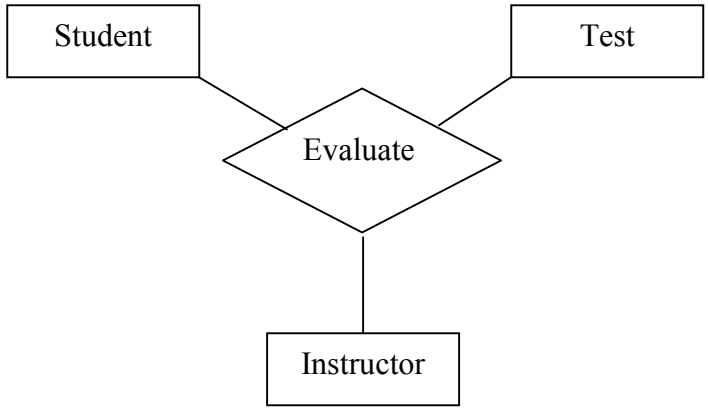


Figure 5.2 Using ER diagrams to show the relationship.

As before, the lexical entry for this semantic value of "evaluate" is evaluate<agent, possessor, patient>. The diagram is shown in Figure 5.2

θ-Criterion Principle: θ-Criterion is basically satisfied. All of the θ-Roles are present in the syntax described by the predicate and have one interpretation in the model; however, the diagram does not identify which is the agent, possessor or patient. More formally, the model does not provide the syntax required to assign the theta roles. It is ambiguous which entity is to receive which theta role.

θ-Theory: The θ-theory is not satisfied for the roles shown. θ-Theory cannot assign the θ-role and the criterion is incomplete; hence, the basis for understanding this diagram is subjective and dependent on the person who is looking at the diagram.

Structure Dependency: Completely ignored! The diagram does not provide any help to the reader on what order to read the diagram. In other words, students evaluate instructors using tests, or instructors evaluate students using tests.

Full Interpretation Principle: Since every word or symbol in the diagram has served a meaningful purpose, this principle has been satisfied.

Economy Principle: This diagram has no superfluous elements and therefore the economy principle is satisfied.

This ER diagram although marked, is less marked than the examples in Section 4.0.

5.3 Ternary UML Model example

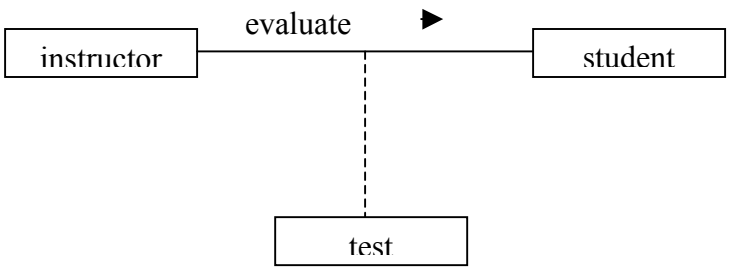


Figure 5.3 Using ER diagrams to show the relationship.

θ-Criterion: θ-Theory is satisfied. All of the θ-Roles are present in the syntax described by the predicate and have one interpretation in the model. UML improves on the ER ambiguity by employing the use of an "association type". In the above diagram, using a dashed line indicates the association type. The entity "test" is an attribute or a possessor attribute of "student" (i.e., external θ-Role). The model differentiates between the "agent" and "possessor" roles (i.e., internal θ-Role) by making the line that connects them to the predicate or association different. Hence, the θ-Criterion is more satisfied in this UML diagram than in the ER diagram but requires specialized knowledge of UML that many people may not have.

θ-Theory: The θ-theory is satisfied for the roles shown. UML recognizes the equivalent of UG θ-Theory and supplies the diagram with visual cues as to how to assign the roles properly. However, this again requires specialized knowledge that not all "experts" may have.

Structure Dependency: Acknowledges the dependency by providing reading arrow. Unfortunately using this arrow causes a one-way relationship.

Full Interpretation Principle: That every word or symbol of a language must serve a meaningful purpose has been satisfied.

Economy Principle: This diagram has no superfluous elements and therefore the economy principle is satisfied.

6.0 Experiments and Results

This paper demonstrates that following the basic principles of UG will produce diagrams or models that are more comprehensible and learnable and thus more readable by humans without specialized knowledge.

Experiments have been developed to demonstrate the following ideas:
1. Syntax assigns meaning to diagrams according to Universal Grammar principles. That is, the less marked a diagram the more easily the diagram can be understood.

2. Previous knowledge of the relationship can override what the diagram actually shows. This creates a situation where an expert in a particular area will rely on his or her own expertise to interpret the diagram rather than on the diagram itself, resulting in the possibility of very different interpretations that go undiscovered.

A series of six diagrams with questions for each diagram was given to 50 participants (more experiments slated for September 2002). Participants should have had the knowledge of how to interpret the diagrams; that is, if a diagram uses UML, the participant had knowledge of UML. Some diagrams gave no information about context and used symbols for all entities. These are called context-free diagrams. Some diagrams contained information about the context (e.g. students, test, instructors) and were called context-sensitive diagrams. Three diagrams had binary relationships and 3 had ternary relationships.

For example: Below is a simple binary context-free diagram with related questions, which requires UML understanding. This type of question was given to the 50 participants that have UML knowledge. More complex questions were also given to participants.

Question 1

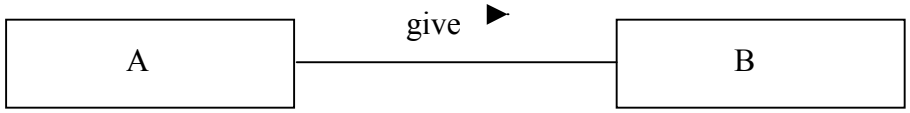


Figure: UML diagram:

There are two entities labeled "A", and "B" involved in the relationship described above by the association "give". Based on the above diagram, please circle the combination of answers that best describe the diagram; Note NA means: Non-answerable – that is, insufficient information to determine the answer.

1. Which entity is most likely to receive something?	A,	B,	neither,	both,	NA
2. Which entity is most likely do to the giving?	A,	B,	neither,	both,	NA
3. Which entity is most likely to be given?	A,	B,	neither,	both,	NA
4. What is being given?	A,	B,	neither,	both,	NA

Figure 6.1 An example question for the participants that is context-free.

6.1 Experiments

Experiments indicated that if UG standards are followed in preparation of a diagram, it is more understandable. That is, relationships between the different entities (e.g., see part 2 below) were understood faster and more "correctly" if UG was followed. The experiments were designed in the following way: Pairs of context free diagrams and diagrams with context were given for a limited period of time to developers, students and other business professionals. We then asked a series of questions relating to the diagrams. For the diagrams without context (e.g., entities labeled as A, B), the participants had to rely on the "syntax" of the diagram to answer the questions and it was predicted that this would be done according to UG. For the diagrams with context, or when domain knowledge (e.g., entities labeled as student, teacher) was supplied, it was predicted that the answers would be most correct if UG is followed. All indications so far have indicated the above assumption is true, and that following UG standards really makes diagrams human readable.

In the first part of the test, the following two diagrams (see figure 6.2) were presented, one with domain context and one without. Syntactically equivalent questions (see figure 6.3) were asked of each. The results and the interpretation are given below.

Part 1

Diagram 1 (without)

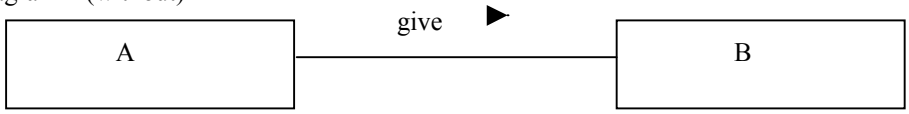


Diagram 2 (with)

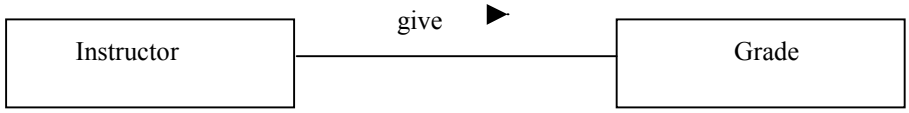


Figure 6.2 Binary UML Diagram with and without context

Figure 6.2 shows the diagram presented; it consisted of the association "give", which is a three θ-role predicate. That is, for the concept "give" to be complete, it requires three θ-roles. In the diagrams, "A" and "instructor" were the agent θ-roles (external θ-roles) and "B" and "Grade" were the theme theta-roles (internal θ-roles). Both diagrams purposely leave out the goal theta role (see lexical entry above, for agent, goal, theme). According to UG θ-Theory, the agent is assigned to the left of the predicate and the goal is assigned to the right of the predicate for English. There should be no ambiguity involving the agent, however, there will be ambiguity relating to the goal and theme θ-roles.

Questions

1. Which entity is most likely to receive something?
2. Which entity is most likely do to the giving?
3. Which entity is most likely to be given?
4. What is being given?

Figure 6.3 Questions asked of participants.

To find out if this prediction is true, the participants were asked several questions (see figure 6.4): The most relevant question that relates to the above prediction was: "Which entity is most likely to receive something?" This question relates to the goal theta-role. Over 90% of the participants answered "B" to the diagram without context. This was in accordance with UG since the goal theta role was an internal argument and position "B" was syntactically to the right of the predicate "give". Only 50% answered "Grade" for the diagram with context to the entity in the equivalent syntactic position. The "B" position was deemed perfectly capable by 90% of the participants to be the recipient of "give" but as soon as the domain context was added, this syntactic position occupied by the entity "Grade" was deemed less capable to be the recipient of "give" and fell to only 50%. This can be explained by UG as well. Since the question was asking for the internal "goal" theta role and the diagram showed only one syntactically correct position for an internal role but the supplied context of "Grade" did not semantically match the "goal" role but rather the "theme" role, ambiguity resulted. The UG principles at work are θ-Criteria, θ-Theory and the Structure Dependency Principle. None of these principles have been 100% satisfied and yet 90% of the respondents detected no problem when asked about the missing θ-role until context was given that conflicted with the syntax. Additionally, only 4% thought that the answer to this question was either "neither", "both" or "NA" for the diagram without context as compared 40% for the diagram with context. Thus, the influence of context and syntax that are at odds with each other introduces ambiguity or conflict between what the syntax of the diagram was inherently communicating based on UG and domain bias of the reader. Simply put, interpretation of a diagram will be based on participants' domain knowledge, and not only on what the diagram states. Resulting in multiple or different interpretations from different participants.

Questions	Context Free				
	A	B	Neither	Both	NA
1	6%	90%	2%	0%	2%
2	88%	6%	0%	2%	4%
3	38%	24%	10%	2%	26%
4	24%	12%	4%	0%	60%

Questions	With Context				
	Instructor	Grade	Neither	Both	NA
1	4%	54%	32%	0%	10%
2	94%	2%	2%	0%	2%
3	14%	52%	12%	4%	18%
4	2%	62%	12%	0%	24%

Figure 6.4 Results of the questions for part 1

As predicted, this ambiguity was not seen in the data for the agent theta-role. Eighty-eight (88%) percent answered "A" to the question "Which entity is most likely to do the giving?" for the diagram without context and 94% answered "Instructor" for the diagram with context to the entity in the equivalent syntactic position.

Part 2

Diagram 1

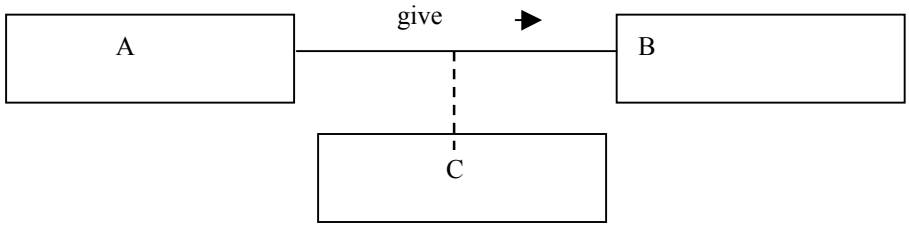


Diagram 2

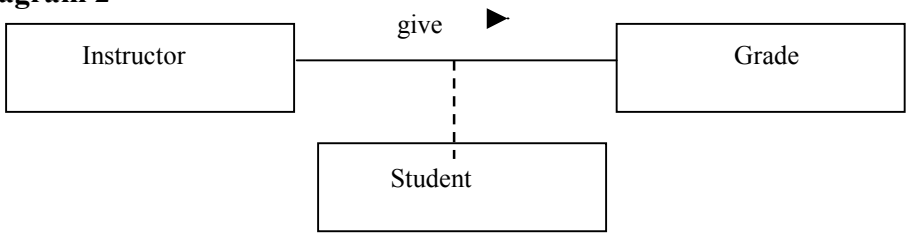


Figure 6.5 Ternary UML Diagram with and without context

The question of "Which entity is most likely to be given?" related to the theme theta role. This question produced a wide range of responses. However, a dominant response of 52% for "Grade" was measured in the diagram with context. This is up from 24% for the diagram without context for the same question. In this case, the syntax does not conflict with the requested and available theta role "theme" or grade. The improvement can be attributed to the given context only. For the question, "What is being given?" there were similar results (i.e., 62%). This too was expected since this basically is the same question worded differently.

The next set of experiments used a ternary relationship (see figure 6.5) and a related set of questions (see figure 6.6).

In these experiments, the association "give", again a three-theta role predicate, was used. That is, for the concept "give" to be complete, it requires three theta-roles. In the diagrams, "A" and "instructor" were the agent theta-roles and "B" and "Grade" and "C" and "Student" were the theme and goal theta-roles, respectively. Again, according to UG, agent theta-role is assigned to the left of the predicate (external θ-roles) and goal and theme theta roles are assigned to the right of the predicate for English (internal θ-roles). There should be no ambiguity involving the agent theta-roles and some ambiguity concerning the goal and theme theta-roles.

Questions

1. Which entity is most likely to receive something?
2. Which entity is most likely do to the giving?
3. Which entity is most likely to be given?
4. What is being given?

Figure 6.6 Questions asked of participants.

To confirm this prediction, participants were asked the question "Which entity is most likely to receive something?" which concerns the goal theta-role. Seventy-two (72) percent of the respondents answered "B" to the diagram without context and only 30% answered "Grade" for the diagram with context to the entity in the equivalent syntactic position. The "B" position was deemed perfectly capable by 72% of the respondents to be the recipient of "give" but as soon as the domain context was added, this syntactic position occupied the entity "Grade" was deemed less capable to be the recipient of "give" and fell to only 30%. Again, the drop can be explained as a result of the syntax pointing the reader in one direction and the context indicating another. However, 92% of the respondents were looking to the right of the predicate or according to the theta theory to answer this question concerning the internal theta role "goal".

The dominant response for this question for the diagram with context was 54% for the entity labeled "Student". In the diagram without context, the corresponding syntactic position only received 8%. Here too, the syntax of the diagram is overwritten by domain knowledge not inherent to the diagram.

Qs	Context Free					
	A	B	C	Neither	Both	NA
1	2%	72%	8%	2%	12%	4%
2	86%	4%	2%	0%	8%	0%
3	16%	32%	24%	8%	2%	18%
4	8%	12%	26%	2%	6%	46%

Qs	With Context					
	Instructor	Student	Grade	Neither	Both	NA
1	2%	54%	30%	6%	8%	0%
2	88%	4%	2%	2%	4%	0%
3	8%	16%	58%	6%	6%	6%
4	2%	8%	66%	4%	2%	18%

Figure 6.7 Results of the questions for part 2

As predicted, this ambiguity is not seen in the data for the agent theta-role. Eighty-six (86%) percent answered "A" to the question "Which entity is most likely to do the giving?" for the diagram without context and 88% answered "Instructor" for the diagram with context to the entity in the equivalent syntactic position.

The question of "Which entity is most likely to be given?" produced a wide range of responses for the context free diagram due not to a missing theta role in the syntax but due to the lack of ability of the syntax to assign the theta-roles correctly. This can be attributed to the diagram's complete lack of ability to assign case according to the UG Case Filter Principle [1]. However, a dominate response of 58% for "Grade" was measured in the diagram with context indicating that the improvement or decisiveness of the diagram could only be realized with domain knowledge not already inherent in the diagram. For the question, "What is being given?" we see similar results. This too was expected since this basically is the same question worded differently.

When the context free diagrams of Part One (see figure 6.4) and Part Two (see figure 6.7) were compared, there were interesting results concerning questions about the two theta-roles (goal and theme). The diagram in Part One was a two-entity diagram and purposely omitted one entity or theta-role. The diagram of Part Two was a three-entity diagram that correctly showed syntactic positions for the three theta-roles of the association or predicate "give". The three theta-roles of "give" are again the single external theta-role "agent" (the entity doing the giving) and two internal theta roles "goal" (the recipient of what was given) and "theme" (that which was given). All three of these roles are required to semantically complete the event described by the predicate "give".

Questions two entity	Context Free				
	A	B	Neither	Both	NA
1	6%	90%	2%	0%	2%
2	88%	6%	0%	2%	4%
3	38%	24%	10%	2%	26%
4	24%	12%	4%	0%	60%

Qs three entity	Context Free					
	A	B	C	Neither	Both	NA
1	2%	72%	8%	2%	12%	4%
2	86%	4%	2%	0%	8%	0%
3	16%	32%	24%	8%	2%	18%
4	8%	12%	26%	2%	6%	46%

Figure 6.8 Comparing context-free of Part one and Part two.

For the "goal" theta role, the difference in respondents dropped from 90% for the two-entity diagram to 72% for the three-entity diagram. It would appear that the addition to the diagram of the third entity has introduced more ambiguity and not less until one considers that the rate of respondents increased from 0% in the two-entity diagram for "Both" to 12% in the three-entity diagram and that 8% selected syntax position "C" for a total of 92%. In other words, this indicates that the respondents were indeed looking for this internal theta-role to be assigned according to UG, i.e. to the right of the predicate. The fact that the diagram could not assign the correct internal role does not violate this.

For the two-entity diagram, questions related to the theme internal theta role produced only a 24% response rate according to theta theory but the three-entity diagram increased the correct response rate to 56% (24% + 32%; also note, the Neither, Both, and NA responses fell off). Since the respondents had nothing else to go on but the syntax provided by the diagram, it could be argued that the improvement is due to the correct number of theta roles being present in the three-entity diagram or the theta criterion.

Qs 4 Three entity	With Context	C IO goal S	B DO theme D			
	Instructor	Student	Grade	Neither	Both	NA
1	2%	54%	30%	6%	8%	0%
2	88%	4%	2%	2%	4%	0%
3	8%	16%	58%	6%	6%	6%
4	2%	8%	66%	4%	2%	18%

Qs 5 Three entity	With Context	B IO goal S	C DO theme D			
	Instructor	Student	Grade	Neither	Both	NA
1	4%	82%	4%	2%	8%	0%
2	90%	4%	2%	2%	2%	0%
3	10%	8%	72%	4%	4%	2%
4	4%	0%	76%	4%	2%	14%

Figure 6.9 Results of the questions for part 3

The last question of the survey switched the two internal theta roles theme and goal or grade and student. This question was poorly worded and more work needs to be done. However, we still think the results are worth looking at briefly and are shown above in figure 6.8. The question concerning the internal theta role goal or "student" jumped from 54% to 82% and the response to the agent role or "instructor" rose from 88% to 90% and the correct responses to the theme theta role "grade" rose from 58% to 76%. Again, the only change to the diagram that could account for the improvement is the switching of the two internal theta roles. This result is very interesting to use, and UG theory on how internal theta roles are assigned needs to be further investigated. At this point, it is beyond the scope of this paper.

7.0 Conclusions and Future Research

This research is very early in the development process. However, the experiments performed on a small number of people (50 people) have indicated that if UG standards are followed in preparation of a diagram, there is less ambiguity and it is more understandable. A set of basic principles can be used to help determine whether a diagram is UG compliant. For example, when the θ-Criterion Principle and Structure Dependency Principle are violated, the diagram is ambiguous. This was illustrated in section 6; a predicate (e.g., give) was used in a ternary relationship that is syntactically correct, but semantically ambiguous diagrams with conflicting interpretations occurred. A series of larger experiments in a more formal setting will be preformed in the fall (>200 people) and the results will be available in September 2002. All

indications so far reveal that following UG standards really makes diagrams more human readable in that they have shown it possible to reduce ambiguity by relying less on context and more on the syntax of the diagram.

This is very new research and much more work is needed. The results of this research are preliminary and more experiments are needed. In addition, we are planning on generating a set of simple rules to apply to a diagram to help determine if it is UG compliant.

Works Cited/Further Reading

[1] Cook, V.J., and Newson, Mark. Chomsky's Universal Grammar. Cambridge, Massachusetts: Blackwell Publishers, 1996.

[2] Fromkin, Victoria, et al. Linguistics: An Introduction to Linguistics Theory. Malden, Massachusetts: Blackwell Publishers, 2000.

[3] Larmen, Craig. Applying UML And Patterns: An Introduction To Object-Orientated Analysis And Design. Upper Saddle River, New Jersey: Prentice Hall PTR, 1998.

[4] Levin, Beth. English Verb Classes and Alternations: A Preliminary Investigation. Chicago, The University of Chicago Press, 1993

[5] Sanders, G. Lawrence. Data Modeling. Danvers, Massachusetts: Boyd & Fraser Publishing Company, 1995.

[6] Wardhaugh, Ronald. An Introduction To Sociolinguistics. Cambridge, Massachusetts: Blackwell Publishers, 1992.

[7] Gliedman, J., "How Language Is Shaped: An Interview With Noam Chomsky." Making It Happen. Ed. Patricia A. Richard-Amato. White Plains: Longman, 1996. 381-389.

[8] Chen, P.P. "The entity-relationship model - towards a unified view of data". ACM Transactions on Database Systems, 1 , (1976): 9-36.

[9] Halpin, Terry. "Business Rules and Object Role Modeling." Database Programming and Design. October (1996) 1-7

[10] Halpin, Terry. "Augmenting UML with Fact-orientation." In workshop proceedings: UML: a critical evaluation and suggested future, HICCS-34 conference (Maui, January 2001)

[11] Jorgensen, Paul. "Data / Object Modeling – Class Notes" www.csis.gvsu.edu/~jorgensen

Charting the Course through Software Process Improvement

Les Grove

Tektronix, Inc.
PO Box 500; M/S 39-548
Beaverton, Oregon 97077
Les.Grove@tek.com

Abstract

There are dozens of topics and activities related to software engineering and development. Theoretically, each one could be continuously improved forever. But with limited resources, process improvement champions need be selective as to where they put their improvement efforts. An effective method of making that selection is to assess a group's or an organization's capabilities to determine which improvement activities should be considered first. Otherwise, big opportunities for improvement could be missed.

There are a variety of published assessment methods, from the time-consuming multi-day SEI/CMM appraisal to the small-scale assessment where developers merely fill out a short questionnaire. At Tektronix, a small-scale method was developed that pulled the best ideas from a variety of techniques to assess individual development groups as well as the company as a whole. This paper includes a short primer on software process assessments, reviews the small-scale method used (both at the group and company-wide levels), and highlights some of lessons learned after performing the assessment.

Author

Les Grove is a software engineer at Tektronix, Inc., in Beaverton, Oregon. Les has 17 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 at the University of Oregon through the Oregon Masters of Software Engineering program.

1. Introduction

The goal of software process improvement is to build an organization's software capability in the areas needed for strategic competitiveness, and to increase its productivity and quality to levels that exceed the competition. Improving software processes brings more control over projects, lowers costs, and improves time-to-market, product quality, and morale.

Where does one begin a software process improvement program? With so many activities related to developing software, it's impractical to improve them all simultaneously and difficult to select just a few to improve. Watts Humphrey begins his book, "Managing the Software Process" [5] with two proverbs:

> *If you don't know where you're going, any road will do.*
>
> *If you don't know where you are, a map won't help.*

If you don't know where your software organization is in terms of capabilities, then any process improvement initiative will do, but you risk not improving areas that need it the most. The most important part of any map is knowing where you are; that red dot that tells you, "You are here." In charting a course through improving software processes, the most important part is to know where your capabilities currently stand. This is accomplished through software process assessments. From the information generated from an assessment, specific areas of improvement can be identified and actions to improve these can be prioritized.

This paper describes how assessments were used at Tektronix to determine corporate-wide improvement initiatives. It provides a primer on assessments and how they fit into a continuous process improvement strategy. The assessment technique used is described, which covers both group-level and company-level assessments. Finally, the lessons learned are provided for those who wish to chart their own course.

For confidentiality, all assessment results shown in this report are fictional.

2. An Assessment Primer

A software process assessment is a tool that is used to learn how an organization works, identify process problems, and prioritize improvement actions. Assessments are part of an overall process improvement cycle that consists of four phases (see Figure 1). The whole cycle starts with the assessment phase, which lays the groundwork for improvement planning and actions. It establishes the current state of the organization and identifies areas in need of improvement. The results of an assessment can also be used as a comparison for future assessments. During the planning phase, the results of the assessment are evaluated and improvement activities are prioritized. A plan is established that aligns with the organization's vision, business goals, issues, and resources. During the acting phase, specific activities are conducted to make the improvements, which can include creating or modifying policies and procedures, acquiring tools, changing organization structures, or creating new training programs. During the learning phase, the organization looks back to see whether changes actually improved the development process: Did productivity improve? Were there fewer defects? Are products releasing sooner? A new assessment phase can then look for new opportunities for improvement and the cycle continues.

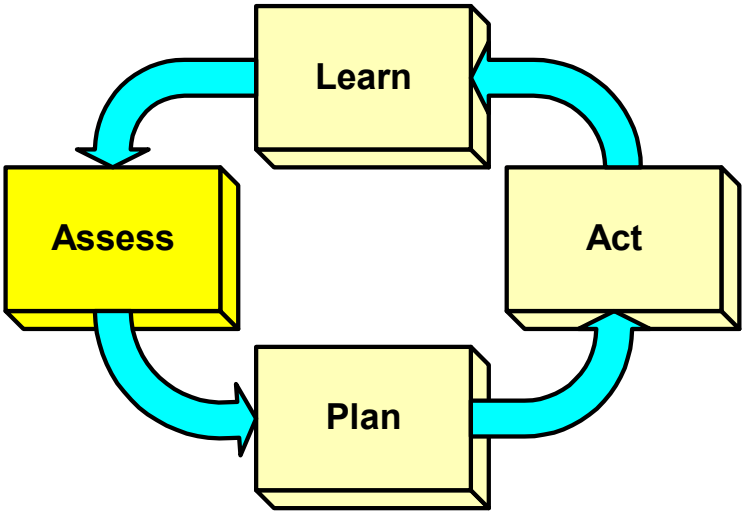


Figure 1 The Continuous Improvement Cycle

Several assessment techniques exist for various sizes of organizations (see references), but they all contain some basic elements and consist of three basic steps:

Step 1. Planning and preparing. The assessment method and scope are determined and the idea of doing an assessment needs management buy-in. Then, the activities that make up the assessment are planned and coordinated with the participants who will be assessed. Assessment materials are prepared: orientation materials, questionnaires/surveys, checklists, etc. If there will be a team of people conducting the assessment, they will need to be familiarized with the method.

Step 2. Conducting the assessment. The activities of the assessment are performed. This should begin with an orientation for the participants. If the assessment is based on the CMM, for example, the orientation could cover what the CMM is, how it enables process improvement, and why they should care. The orientation should also review the assessment method and process so they know what's about to take place and what's required of them. To get open and honest responses, the participants need to be reassured that they will not be identified in any report. Once the participants are ready, the actual assessment can take place. This can consist of administering questionnaires, conducting interviews, reviewing documents, and/or leading group discussions – depending on the method. Results are compiled, analyzed, and validated, which may require follow-ups with groups or individuals. Some sort of score (e.g., a CMM level) or a set of scores are determined.

Step 3. Reporting results. An assessment report is generated covering the scope, activities, findings, any ratings, and recommendations. Anonymity of participant responses must be maintained in the report. The report is delivered to all sponsors and the data from the assessment is preserved.

The next phase of the process improvement cycle is to develop a plan that sets out the desired results, a set of actions and milestones, risks, and responsibilities that will implement the improvements. The plan may include any or all of the recommendations in the assessment report, which then can be managed like a typical project.

3. The Tektronix Assessment

The assessment method used at Tektronix was derived from investigations of a variety of large-scale [1][2][6][9] and small-scale assessment methods [8][10][11]. There were elements from these different methods that worked nicely in the Tektronix culture:

- The Goal-Problem approach [8] worked well for aligning the scope of the assessment with managers' priorities.
- The Modular Mini-Assessment method [11] provided flexibility and allowed managers to customize their group assessment.
- Bootstrap [1] provided a method for assessing at two-levels (group/project level and organization/company level).
- The small-scale methods minimized the amount of time required from the participants. This increases participation.
- Some methods had simple scoring systems while others did not. Since the goal was only to determine areas for improvement, a simpler scoring system was preferred.

The overall goal of the Tektronix assessment project was to determine where to make company-wide improvements to software engineering practices. In order to accomplish this, eight group-level assessments were performed and used as representative of all software development groups in Beaverton. The results from the group assessments were combined to determine company-wide improvements.

3.1 Group Assessment Method

It was difficult to find teams willing to take days out of their schedules in order to participate in a large-scale assessment, so a small-scale method was developed requiring only a few hours from each person.

Step 1. Planning and preparing - First, a preliminary meeting was held with a software manager and their project lead(s). The manager's business goals and software development issues were discussed, and prioritized as crucial, high, medium, and low. The assessment process was reviewed and activities were scheduled. The manager decided how much orientation would be required for the group: either a short introduction to the assessment and questionnaire or a longer orientation that also promoted process improvement. This was the time to decide which project or projects the assessment would cover that best represented the group's practices.

Step 2. Conducting assessment - Before asking the participants to do anything, an orientation was given to the group just before giving them a questionnaire. Each question was answered with "Always", "Usually", "Sometimes", "Rarely", or "Never" depending on how well established and consistently the activity was performed on the group's projects. Each question had two additional options, "Don't Know" and "Not Applicable" and room for comments. The questions were scored with 1 point for responses of "Always" and "Usually", a half-point for "Sometimes". Responses of "Don't Know" and "Not Applicable" were ignored. Using all of the respondents' answers, a percentage was calculated from the total possible from each process area.

Each process area's set of questions contained different levels of questions. Basic-level questions attempted to determine whether the basic practices of a particular process area were carried out. Advanced-level questions covered advanced topics and whether practices were performed consistently. Optimal-level questions looked for best in class processes.

One of four grades was given to each score based on each area's level of performance and compared to the importance given to each during the planning meeting. An analysis matrix was generated showing the relationships between importance and grades (see Figure 2). Three or four process areas from the upper-right portion of the matrix would make up the focus of further investigation. Using the example, Testing, Project Tracking, and Reuse would be selected.

	Current status			
Importance to Organization	**Optimal**	**Advanced**	**Basic**	**Low**
Crucial	• Architecture		• Coding	• Configuration Management
High	• Defect Tracking	• Design	• Intergroup Coordination	• Product Integration
Medium	• Project Planning	• Project Tracking	• Requirements Development	• Requirements Management
Low	• Reuse	• SQA	• Testing	

Figure 2 Analysis Matrix*

A follow-up meeting was held with the participants where the questionnaire scores and analysis matrix were shown to the team. The assessment lead asked about discrepancies in the responses and other open-ended questions intended to promote discussions. Assessment team members took notes and made observations, listening for problems and possible root causes. After the discussion, the assessment team separately met to review notes and determine recommendations for the group. We've found that discussing key process areas in depth with the participants was the key to determining what was really going on in a group and gave us confidence in making recommendations. In almost every case, we found an underlying root cause or consistent theme to a group's lower scoring areas.

Step 3. Reporting results - An assessment report was written, reviewed by the assessment team, and delivered to the group's manager. The report covered the assessment process, scope, and findings and provided a vision of an ideal situation that the group could achieve and recommended actions the group should take in order to fulfill that vision. It was the manager's responsibility to plan and execute any improvement plans based on the recommendations in the report.

* For confidentiality, all assessment results are fictional.

3.2 Company Assessment Method

Having a method to assess a single group, the same basic process was used to paint a picture of the overall software develop capabilities at a company level.

Step 1. Planning and preparing - Before any questionnaire can be developed or any other activity can begin, the scope of the assessment needed to be determined.

The assessment scope was determined through interviews with 16 software engineering managers in Beaverton. Each manager was asked about their business goals and then was asked to classify each goal as *crucial*, *high*, *medium*, or *low*. From these goals, each manager was asked about the software development issues their group faced in attaining each goal. From these meetings, all of the software development issues were translated into "software process areas" from various process models and classified as *crucial*, *high*, *medium*, or *low* based on the originating goal. Issues that could not be easily classified were generalized and grouped where possible.

Each process area linked to a manager's *crucial* goals was given a score of 8 points, 4 points for *high*, 2 points for *medium*, and 1 point for *low*. Points for a process area were only counted once at its highest rank per manager and totaled (see Table 1). Process areas were classified by level of importance to Tektronix as *crucial*, *high*, *medium*, and *low* by looking for significant breaks in the point totals. The scope of the software process assessment was determined by taking the top 14 areas.

Process Area	Points	Importance
Architecture	108	Crucial
Coding	100	
Configuration Management	97	
Defect Tracking	72	High
Design	71	
Intergroup Coordination	66	
Product Integration	63	
Project Planning	48	Medium
Project Tracking/Oversight	44	
Requirements Development	44	
Requirements Management	42	
Reuse	32	Low
SQA	32	
Testing	26	

Table 1 Process Area Rankings[*]

With the scope determined, the assessment plan was developed and presented to the sponsors. With management buy-in, assessment materials were developed, which included an assessment questionnaire, materials for orientation presentations, report templates, and checklists. These were maintained during the entire project. An assessment team using volunteers was assembled and each team member was briefed prior to their involvement.

[*] For confidentiality, all assessment results are fictional.

Participating groups were selected based upon representation of the various product groups, managers who expressed interest, and schedule availability.

Step 2. Conducting assessments - This phase consisted of performing eight group assessments and combining the scores at the lowest questionnaire level that showed need. A chart similar to Figure 3 was created to show the ranges of scores for each process area (vertical bars). When dealing with multiple scores, the assessment team decided to look at the *potential* in each area by averaging the top two scores, producing a single score (horizontal marks). This showed the areas where Tektronix tools, culture, methods, policy, etc. affect the best teams. In other words, each potential-score represented the best any team can do in the Tektronix environment. Teams that scored low in an area where other teams excelled had internal issues to deal with in order to improve themselves in that area. Areas where the top teams had low scores represented the best opportunities for company-wide improvements. An analysis matrix at the company-level was developed (similar to Figure 2) and the areas with low potential and had the highest importance were targeted for improvement.

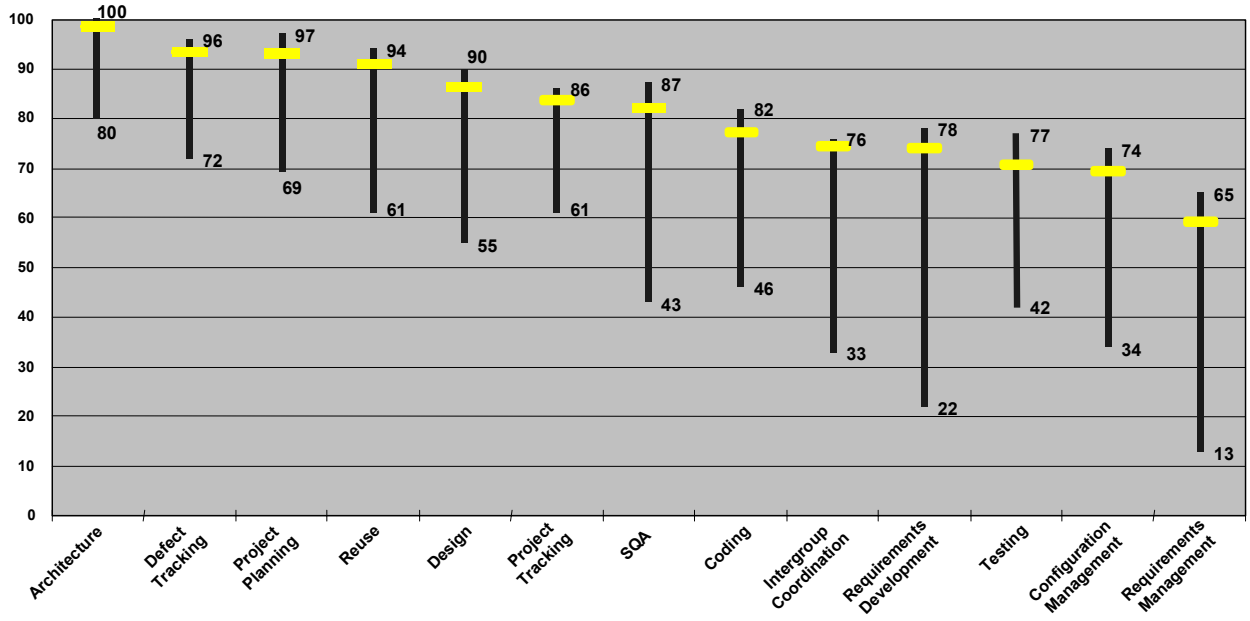


Figure 3 Spread of Scores (Potential)*

Step 3. Reporting results - A company software process assessment report was written, reviewed by the entire assessment team, and distributed to the sponsors of the initiative. The report included the scope, activities, overall findings, and company-wide recommendations. The report was delivered to the sponsors of the assessment. With the assessment completed, process improvement actions were targeted to the areas that provided the most benefit.

* For confidentiality, all assessment results are fictional.

4. Conclusion

An assessment shows where an organization currently functions and where there can be improvement. By developing an assessment strategy aligned with clear business goals, a course can be charted towards improving its processes. This paper showed one way of accomplishing this, and with any journey, we've learned some lessons:

- **Lead by example.** It's easier to sell the idea of assessments and process improvement if you can demonstrate your "capability level". This means planning, managing, and improving your processes as you go. Get feedback along the way to improve your orientation materials, questionnaire, report formats, etc.
- **Review, review, and then review the questionnaire.** You get poor data to base decisions on if people do not understand the questions.
- **Be ready to defend your idea.** Some people accept the idea of assessments and process improvement and some are not very familiar with it. Then there are those who are against it or have deep concerns. Be up front with why you are conducting an assessment and what you intend to do with the results. Be willing to work directly with individuals and give them some flexibility; often it is the lack of control over the process that people resist. Work *with* your participants, not *against* them.
- **This cannot be done alone.** Having an assessment team is vital to validate findings and provide different points of view.

Acknowledgements

The Tektronix assessment would not have been successful without the assessment team: John Burley, Jeremiah Burley, Aaron French, Al Zimmerman, Tim Lawson, and Mahroo Arsanjani. I would like to thank Creig Smith and Dave Brown for their support and all the software engineering managers who participated. I would also like to thank Dr. Richard Fairley at OGI/OHSU for his guidance during my research.

References

1. Bootstrap Institute web site, http://www.bootstrap-institute.com
2. D. Dunaway, S. Masters, *CMM-Based Appraisal for Internal Process Improvement (CBA IPA): Method Description*, Tech Report CMU/SEI-96-TR-007, SEI, 1996
3. K. Dymond, "Essence and Accidents in SEI-style Assessments or 'Maybe This Time the Voice of the Engineer will be Heard'", *Elements of Software Process & Improvement*, IEEE Computer Society, 1999
4. V. Haas, et al., "Bootstrap: Fine-Tuning Process Assessment" IEEE Software, vol 11, no. 4, July 1994
5. W. Humphrey, *Managing the Software Process*. Addison Wesley Longman, Inc., 1989
6. S. Masters, C. Bothwell, *CMM Appraisal Framework Version 1.0* Tech. Report CMU/SEI-95-TR-001, SEI, 1995.
7. McFeely, B., *IDEAL: A Users Guide for Software Process Improvement*, Handbook CMU/SEI-96-HB-001, SEI, 1996.
8. N. Potter, M. Sakry, *A Goal-Problem Approach for Scoping a Software Process Improvement Program*, PNSQC Proceedings, 2000
9. Software Process Improvement and Capability dEtermination web site, http://www.sqi.gu.edu.au/spice.
10. R. Whitney et al., *Interim Profile: Development and Trial of a Method to Rapidly Measure Software Engineering Maturity Status*, Tech. Report CMU/SEI-94-TR-4, SEI, 1994
11. K. Wiegers, D. Sturzenberger, "A Modular Software Process Mini-Assessment Method" IEEE Software, vol. 17, no. 1, January/February 2000
12. S. Zahran, *Software Process Improvement*. Addison-Wesley, 1998
13. D. Zubrow et al., *Maturity Questionnaire*, Tech. Report CMU/SEI-94-SR-07, SEI, 1994.

Intel SCM Assessment Instrument (SAI): A methodical approach to SCM process improvement

Dhinesh Manoharan
Platform Quality Methods
Corporate Quality Network
Intel Corporation
2111 NE 25th. Ave, MS: JF1-46,
Hillsboro, OR 97124-5961, USA
Dhinesh.Manoharan@intel.com

Abstract:

This paper provides a methodical approach to assess a software development organization's maturity level in the software configuration management (SCM) discipline. Traditionally, CMM or other software methodology assessments, while providing a broader picture of the SCM discipline, by design, do not provide enough tactical details about the strengths and weaknesses. This paper introduces the Intel SCM Assessment Instrument (SAI) that attempts to fill that gap by providing SCM practitioners an objective and comprehensive method to perform a current situation analysis of their SCM environment. The SAI is comprised of a questionnaire and a record and report tool. This paper assumes familiarity with SCM activities and terms.

Author Biography:

Dhinesh Manoharan is a staff software engineering process expert for Intel Corporation responsible for the configuration management methods in the Corporate Quality Network organization. He earned his BS in Computer Science and Computer Engineering from the Government College of Technology (India) and MS in Computer Science from Portland State University. He has a CMII certification from the Institute of Configuration Management. He is currently pursuing an MBA in Technology Management. He has over 8 years of experience in software engineering including programming languages, software configuration management, software quality assurance and project management in several software development projects. Other areas of interests include cryptography, network security and organizational behavior.

I. Introduction

Software Configuration management (SCM) is a critical and integral discipline that spans the product life cycle stages of software engineering projects. While several definitions exist, a generic working level definition for SCM is as follows: SCM is a discipline that establishes and maintains integrity, control, and communication of an evolving software product through out its life cycle. SCM has been recognized as a formal discipline in defense and government for several decades. However, software organizations across the commercial industry have taken note of this discipline only in the past decade or two. SCM is now routinely considered in the scope of all process improvement efforts.

As all change agents would agree, the first step in pursuing a process improvement effort is to understand the current situation in the project. Popular models such as the SEI Capability Maturity Model and the European BOOTSTRAP institute model, provide assessment capabilities to gauge the maturity level of the project team in several software engineering disciplines. However, these all-encompassing assessments stop short of generating tactical data for process improvement in a specific discipline. The Intel SCM Assessment Instrument (SAI) fills that gap by providing the ability to identify the tactical strengths and weaknesses in the eight practical basic SCM activities.

The objective of SAI is to measure, collect and track the basic SCM maturity levels of software project teams. The SAI helps to identify areas that need improvement so that helpful, instructive information (training, best methods, example collateral from other organizations, templates, and references) and/or hands-on assistance can be delivered to help the organization improve their basic SCM practices. It also helps organizations to track and document improvements in basic SCM competency through periodic assessments. The SAI consists of a detailed assessment questionnaire and a record and report tool.

This paper provides an overview of the SAI including an approach for planning, coordinating and conducting the assessment. The next section provides a list of terms and definitions used in the paper. The section following that describes the classification of the SCM practices at Intel. The subsequent section provides the rationale for the SAI, followed by benefits, roles & responsibilities, the assessment process and the record & report tool. The remainder of the paper briefs the reader on the tailoring guidelines, possible enhancements to the SAI and finally a few success stories at Intel Corporation.

II. Definitions

Term	Definition
Baseline	The specific versions of Configuration Items that define the product configuration at a specific point in the product life cycle. (See Formal Baseline).
Baseline Management	Baseline management is the definition, documentation, maintenance, communication and administration of a product's baselines
Build Management	Build management is the process of reliably constructing software product components and the software product.

Term	Definition
Change Control Board	A cross-functional team responsible for approving/rejecting requests for changes to software product configuration items that have been baselined.
Change Management	The identification, classification, control, and validation of changes to the software product. Change Management provides the process to manage the impact of change to your organization, projects and products.
Configuration	The identified product work items that will be controlled by the SCM processes. The product configuration must continually evolve during the life of the software product.
Configuration Item (CI).	Individual product work items that compose the product configuration. Configuration items can be assemblies of other configuration items.
Defect Management	The process of documenting (reporting) product defects, classifying them, moving them through a well understood resolution process, and studying why each defect came to be so that the overall process can be corrected in an attempt to ensure that like defects are discovered sooner, or not inserted at all
Formal Baseline	A baseline of configuration items that have been formally reviewed, validated, and/or agreed upon and thereafter serves as the basis for further development. Changes to items that have been included in a formal baseline must only be made through formal change control procedures. (See Baseline)
Interface Management	The process of identifying, documenting and controlling all functional and physical characteristics relevant to the interfacing of two or more items provided by one or more organizations. Functions of interface management can include subcontractor control and control of software produced by other organizations within a company.
Product Baseline	See Baseline.
Product Configuration	See Configuration.
Product Work Items	All software product components and related items necessary to develop, reproduce, and support those components. These include all documentation (e.g., both engineering and product documentation), software binaries, source code, tests, tools and data used throughout the product's life cycle. This is often referred to as the scope of SCM.
Release Management	The creation, control, archival and communication of a version of the product with the express purpose of making it available to external parties (i.e., vendors, beta sites or internal clients and external customers).
Revision Management	See Version Management.
SCM plan (SCMP)	The document defining how software configuration management will be implemented (including policies and procedures), who will perform the activities and the timing of such for a particular project, product, acquisition, program or organization
Software Configuration Management	A discipline that establishes and maintains the integrity, control and communication of an evolving software product throughout it's entire life cycle.

Term	Definition
Version Management	The process of storing, tracking and controlling the successive change or modification to product work items and components

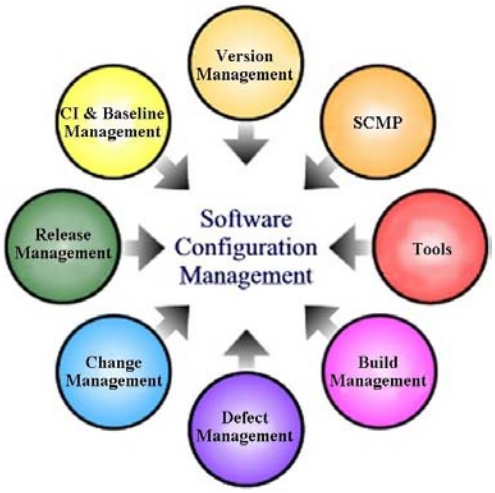
III. Classification of SCM practices

Traditionally, the SCM discipline has been described as being comprised of four key practices namely, Configuration Identification, Change Control, Status Accounting and Configuration Audit. These key practices are comprehensive when it comes to educating or understanding the SCM discipline. However, from the practitioner's perspective, the key practices need to be reorganized or broken down into several functional domains in order to allow for a systematic and detail-oriented approach in implementing and practicing the SCM infrastructure. The author has designated these domains as the key functional areas (KFA) of SCM. Note that the term is coined to look similar but different than CMM (KPA in particular).

From the perspective of an organization that is striving to implement sound SCM practices from ground up, all these key functional areas seem overwhelming. There are also certain SCM practices (KFA) that serve as the building block for certain other. For these two reasons, the author has classified the practices (KFAs) into two levels – basic and advanced. The basic practices need to be in place before the advanced practices are taken up for implementation and adherence.

Accordingly, basic SCM practices necessary for software development can be classified into eight practical activities: version management, configuration identification (CI) & baseline management, release management, change management, defect management, build management, SCM tools and SCM plan. The following diagram illustrates the key functional areas that comprise the SCM practices at the "basic" level. Note that two of the KFAs are actually SCM tools and SCM Plan. Although these two KFAs may not be considered practices per se, they are included for the following reasons: The SCM tools play a critical role in automating all practices, processes and procedures in place. The SCM Plan is a communication vehicle that ensures that the practices, processes and procedures are documented and disseminated to the stakeholders and users.

Basic SCM Practices



Basic

IV. Rationale for an SCM assessment

Before attempting process improvement in any organization, the process improvement owner needs to glean and assimilate 4 essential categories of data, called the 4Ps. The four Ps are as follows: People, Products, Processes and Pain-points. A good understanding of these 4 factors serves as a basis in understanding the current situation of the organization. This approach is certainly valid when it comes to SCM process improvement as well since any recommendations from the assessment must take these 4Ps into account.

People: This data can typically be gathered by obtaining the most recent organizational charts and also by talking to the human resources contact in the organization.

Products: It is important that a process improvement effort takes into account the type and complexity of the products that are currently under development. The process owner collects information about future product roadmaps so that the processes will support the flawless execution of the roadmap.

Processes: Before defining the new processes for any given organization, the process owner spends adequate time to understand the current process infrastructure, the strengths and weaknesses, the historical evolution of today's processes and retrospectives from past process deployments.

Pain-points: In many situations, the organization embarks on a process improvement effort when there are current or anticipated pain-points that are likely to impact the organization's efficiency and ability to provide quality products. These pain-points must be carefully gathered and root cause analysis must be performed to understand the underlying issues that have led the organization to this state.

While there are existing means to gather data that pertain to the People and Products factors described above, a systematic approach to gather objective and tactical information on the current Processes does not currently exist in one place. Also, the pain-points are usually the symptoms for which the root cause usually exists in the process adhered to by the organization. This relation between pain-points and processes is difficult to establish without a formal assessment process. These two reasons amongst several others necessitated the creation of the Intel SCM Assessment Instrument (SAI).

V. Benefits of SAI

This SCM assessment instrument will help to:

- Identify areas that need improvement so that helpful, instructive information (training, best methods, example collateral from other organizations, templates, and references) and/or hands on assistance can be delivered to help the organization improve their basic SCM practices.
- Track and document improvements in basic SCM competency through regular assessments.
- Educate the organizations on SCM methods required to obtain a basic level of SCM competency.

- Guide the organizations' SCM infrastructure (methods & tools) development through analysis of assessment results trends.

This SCM assessment instrument will not:

- Identify the effectiveness of the SCM practices that are being used by the organizations since it does not involve gathering existential proof for claims made by interviewees. A Software Quality Assurance team, if it exists in the organization, can also fulfill the needs of measuring effectiveness through audits and indicators.

VI. Roles and responsibilities for using SAI

Role	Responsibilities
SCM engineer / specialist for the firm or organization	• Conduct assessment for an organization(s) • Report results and make improvement recommendations to an organization(s) • Include results in Basic SCM Competency (BSC) compilation
Organizational SCM Owner	• Conduct assessment for an organization(s) • Report results and make improvement recommendations to an organization(s)
First level engineering manager(s)	• Complete or participate in assessment for his/her organization
Software Engineer(s)	• Complete or participate in assessment for his/her organization
SQA engineer(s)	• Complete or participate in assessment for his/her organization • Conduct assessment for his/her organization(s) • Report results and make improvement recommendations to an organization(s)

VII. The SCM assessment Methodology

The objective of this assessment is to measure, collect and track the basic Software Configuration Management (SCM) maturity levels of Intel organizations (groups, divisions and/or projects). The current version of the SAI covers only the basic SCM practices. Upon completion of this assessment, a baseline of the SCM competence in each of eight KFAs is determined.

This instrument is written for project level assessments. Since typically in a large company, the SCM practices and maturity often vary greatly among projects (within a division) and among divisions (within a group), assessments must be conducted on multiple projects to ascertain divisional and group SCM maturity levels.

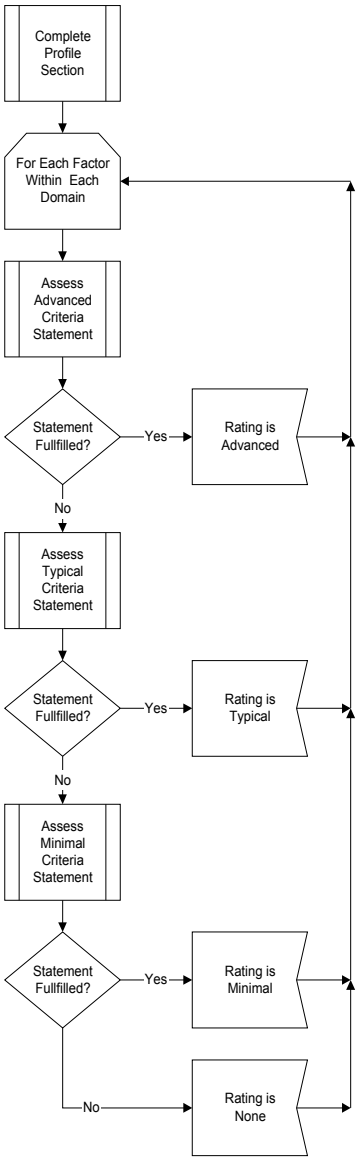
This assessment instrument is organized into nine sections, the Organization Profile and eight KFA Surveys. Each of the eight KFAs has a series of factors that are considered important to that practice. Each of the factors has three statements that decrease in maturity from Advanced through Typical to Minimal. The SCM Assessor

uses the accompanying Record and Report Tool to record the results of the assessment.

VIII. The Assessment Process

All of the organizational profile questions must be answered. All of the factors in each of the KFAs must be responded to. All nine sections must be finished for the assessment to be completed. To respond to a factor, begin by reading each of the statements (left to right or advanced to minimum). Select the statement that best describes the current situation in the project that is being assessed. If any aspect of a statement is not in practice it cannot be selected. If the project does not fulfill the minimal level, mark "*None"* on the answer sheet. The steps of this process are represented in a flow diagram below.

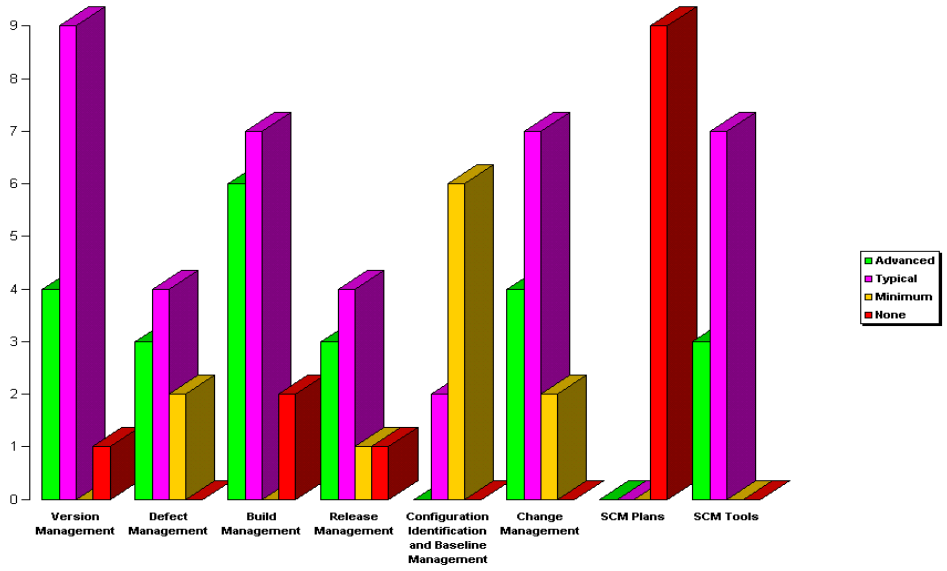
If you see a term or phrase that you do not recognize, refer to the *Definitions* section of this document. Descriptions for each of the domains are also in this section.
The following flowchart describes how an assessor uses the Basic SCM Practices assessment instrument.

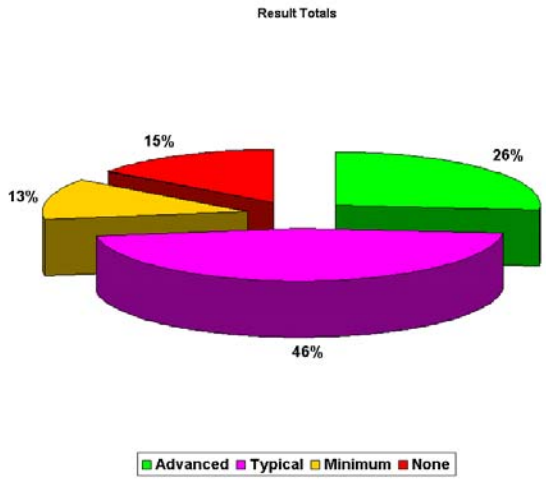


IX. The Record and Report tool

As the assessor conducts the assessment using the process described in the preceding section, the record and report tool can be used to record the gathered data. The record and report tool is an Excel spreadsheet with predefined templates and formulae. The template consists of a worksheet to record the maturity level (advanced, typical, minimal or none) of the assessed organization for each factor. The assessor can also record any comments and remarks made during the assessment that might prove significant in creating the assessment report.

The reporting worksheet provides histograms of the assessment results in various ways including KFA-specific charts, consolidated charts and summarized charts. Some examples of these charts are provided below. The reporting tool is limited in the sense that the charts only provide a numerical interpretation of the organization's maturity level. It must be underscored that it is the responsibility of the assessor to collate, correlate and interpret the assessment data to create an assessment report.





The report takes the organization's profile into account. In other words, an organization's maturity level does not in itself provide a complete picture without relating that level to the profile of the project. For instance, a small organization developing a small product may be found to be operating at the minimal maturity level. The fact that this small organization is operating at the maturity level does not automatically mean that moving towards a higher maturity level is that organization's goal. It may very well be the case that the minimal maturity level is most optimal level for that organization and any improvements beyond that will result in diminishing returns. On the other end of the spectrum, a large and complex product development organization operating at the minimal maturity level is at a great risk and hence needs to improve significantly to achieve a typical or advanced level of maturity.

X. Tailoring guidelines

The SAI is intended to be as comprehensive as it could be to meet the needs in the Intel product development organizations. However, any SCM specialist interested in tailoring the tool to his or her environment may do so by evaluating the gaps and adding the necessary factors for each assessment. The author has attempted to stay with the industry terminology in the SAI. If the adopters find a need to use alternate terms or terms that their context is familiar with, they may do so accordingly.

XI. Enhancements to the SAI

The current version of the SAI covers only the basic SCM practices. An area for future enhancement is to extend the SAI to include the advanced SCM practices (Change History & tracking, Audits, Status reporting).

The SAI record and report tool described in the document is implemented in Microsoft Excel. For greater ease of use, the tool could be implemented as a web-application thereby facilitating better and faster data collection.

XII. SAI success stories at Intel

The SAI was first created at Intel in 1999. For the first six months, the tool was used in pilot projects to study the usability and benefits of the tool. Once the feedback from the pilot was incorporated, the tool was made available to all Intel organizations. The tool has been extensively used in software development organizations of various sizes varying from a small team of 6 to large teams of 800 people. A basic rule of thumb followed has been as that the larger the size of the assessed organization, the larger the sample size (number of assessment participants) should be. The assessment reports have been used effectively in the process improvement efforts to reap business benefits.

Recently, the tool has been put to use in silicon product design organizations as well where the end product is not software. However, the design process involves extensive use of software, software methodologies and tools. Initial indications are that SAI will be instrumental in effecting SCM process improvements in these organizations as well.

The critical success factors for the SAI include the following:

- The assessors must be proficient in the SCM discipline and must be in a position to understand, interpret and communicate each factor under consideration. In

turn, they should be able to translate the responses into meaningful observations.

- As is true in all process improvement efforts, the management team in the organization must be fully supportive of conducting the assessment. After all, the assessment is one of the first steps in process improvement.
- The number of assessment participants must be 4-15 per project. The more the number of participants, the better the quality of data obtained.
- The assessment instrument must be used not only to obtain the baseline of the organization's maturity level but to periodically assess the improvements achieved. The assessment tool provides a quantifiable mechanism to measure SCM process improvement. It is recommended that the SAI be used on an annual basis to gauge the progress.
- The data collected during the assessments must not be used as a performance measurement method for the individuals involved. This ill-advised approach will result in suppression and sabotage of the real state of affairs in the organization.
- The organization must be committed to follow-up on the weakness identified during the assessment. Failing to do so will result in disappointment and loss of confidence amongst the participants. This also lessens the likelihood of using the process/tools again in the organization.

XIII. Conclusion

As today's software organizations are faced with increasing competitiveness, complexity and agility, it is prudent to pursue continuous process improvement activities in disciplines such as SCM to maximize effectiveness, efficiency and quality. But before venturing into such efforts to achieve the goal, the organization must complete an objective and comprehensive analysis of their current state. The SAI serves as a structured tool to collect, collate and interpret the maturity level of the organization and subsequently utilize the data as a baseline for the process improvement initiative. Periodic assessments will provide quantified graphical data about the SCM health in the organization. Many organizations across Intel have adopted this instrument and demonstrated success in their process improvement and the resultant business benefits. Other software organizations across the software industry could potentially also significantly benefit from SAI.

References

Berlack, H. (1992) Software Configuration Management New York: John Wiley & Sons, Inc.

Buckley, F. (1996) Implementing Configuration Management: Hardware, Software and Firmware Los Alamitos, California: IEEE Computer Society Press

Pressman, R. (1997) Software Engineering: A Practitioner's Approach. New York: McGraw Hill

Cagan, M. and Wright, A. (1992) Requirements for a modern software configuration management system, Continuous White Paper.

IEEE Standard 828-1990 (1990) IEEE Standard for Software Configuration Management Plans, American National Standards Institute.

Better Teams Produce Better Products
Pat Medvick and Becky Winant

Summary

In this paper we define team development stages; offer models which improve personal, and therefore team, effectiveness; offer tools which can be used as a framework to get started; apply models and tools to team stages; and examine why team quality affects product quality.

We once heard Gerald Weinberg say (1) that the quality of a software product cannot exceed the quality of the organization that builds it. Crosby (1979) and Weinberg (1994) show that higher quality correlates with better problem solving and clear processes. DeMarco(1995) points out that non- technological factors, embodied in the project sociology, have the greater impact on success. DeMarco and Lister (1999) observed that "jelled" teams work most effectively together. Jelling refers to bonding, which grows as people working together make progress towards a shared goal. High-functioning teams often are characterized by clear roles, healthy communication and a self-selected and self-directed process for decision-making while pursuing their goal.

Teams are made up of individuals and individual traits vary. When individuals merge to perform a task together they pass through stages of team formation, which are predictable. Each stage offers a clear opportunity, sometimes in the form of a hurdle. Once aware of this progression and the tests they'll face, people can make choices about how to proceed, how they might interpret information and how they might improve their chances of success in forming a functioning and cohesive unit. Our observation is that any team that successfully completes the first two stages is likely to be successful in developing useful products of good quality.

~

Most people have likely had at least one experience where they were part of a well-functioning team. Figuring out how to recreate such a team isn't obvious. What makes it so hard? The answer includes the diversity of individuals and the different expectations about teams**.** Techniques for improving team functioning are based on psychological/sociological tools and models for refining team communication and process.

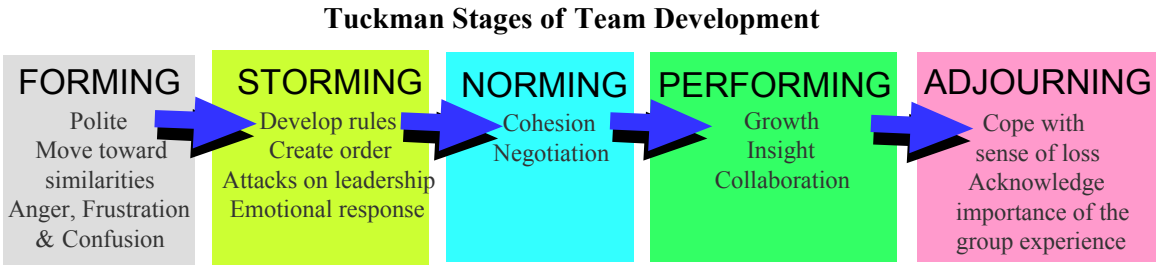
Each person in a team comes with strengths, weaknesses, skills and temperament. Teams are not so much a collection of people as they are a collection of interactions between individuals. Based on combinatorial logic, a formula to determine the number of unique human interactions is N*(N-1). This is relatively simple with 2 people: What Becky says to Pat, and what Pat says to Becky can result in totally different interpretations. Pat and Becky each have their own way of looking at the world and interpreting it. If you were to take a collection of 5 individuals, which could represent a project team, you would get 20 unique interactions. Being able to effectively deal with all this uniqueness can give any group of individuals a better chance of clearer communication.

Additionally, people differ in their basic definition of a team, and, thus, assumptions about ‘good’ team behavior. A common image is based on competitive, hard-driving athletes.

Sometimes "team" is a name for any collection of people assigned to work together. Yet another definition could be a collective of individuals who share a vision and set of values: the basis for our view of "better teams."

As we explored our team interactions while organizing a software conference, examined our experiences in companies and at U.S. national labs, and talked with others about their teams, we looked for patterns to help us understand why some experiences were great and others dismal. A model from psychologist Bruce Tuckman gave us a place to start organizing our information. Models from Virginia Satir have helped us better understand how to deal with individuals and interactions. Team charters and profiles of human personality, such as the Myers-Briggs Type Indicator, gave us insight into how we can make progress with less stumbling and more respect for each other.

Natural Stages of Team Development
Fig. 1



According to Bruce Tuckman teams have a natural life cycle. In 1965 he defined the stages as 1) Forming, 2) Storming, 3) Norming, and 4) Performing. In 1977 he added a final life cycle stage 5) Adjourning. This is model of succession and, as in a human life, not all groups who start will survive to the end or even the middle stage. Each stage provides opportunities and challenges and differs in the level of group cohesion and in the communication style of the collective. More predictably a group of people who successfully navigate the first three stages will jell into a team.

Life Cycle Definitions
Here are the characteristics of Tuckman's stages:

1) Forming.
In forming, a group is defined as working together. This stage establishes the group's awareness of their individual strengths, weakness, skills and temperament. People begin to learn about each other and the team goal. Initially, team members may be overly polite as they try to figure out why they are there, what value they bring, and whether others will accept them. During this period the team must clarify expectations and clearly define mission and task. The big question is "Why are we here?"

Peoples' mood during forming can shift between anger and frustration. As individuals learn about each other, people gravitate towards shared similarities as a basis for bonding. If information exchange is superficial, people may reject the similarities because trust is lacking.

To advance from forming the group must succeed in defining their task and identifying what they contribute as individuals.

2) Storming.
Storming tests the group's ability to work out how they will accomplish their task together by establishing their processes for working together. Storming addresses ground rules, roles and processes. Specifically the group must define their decision-making process and identify the necessary roles, resources, and procedures for performing their task.

Moods during storming can be turbulent. There may be personal attacks as people seek to define their place within the team – some jockeying for a leadership role – and people lobbying for rules that are important to them. Each individual's sense of leadership will be essential for calling fouls, maintaining focus, and keeping the team using its brand new processes. At this stage the group will find a healthy way to communicate; will thrash, resist and split apart; or will continue in a limbo of unresolved rules and expectations.

If navigated successfully, the team figures out how to work through problems and their basic contract together, or at least gets something on the table. Specific results are not as crucial as the success of the process. This latter element determines the team's level of trust and their commitment to accepting their own decisions, which they may choose to refine later. Completing this stage gives the team an early success, which strengthens the bonds that they have begun to form.

Sometimes groups remain in storming for a long time, and may be unable to get out. This can be similar to an adult, grumbling over the difficulties faced as a teenager or as a small child. A frozen-in-time condition overshadows their ability to progress to a more appropriate and healthier state. Occasionally internal coalitions form in an attempt to overcome disharmony and accomplish the task.

3) Norming.
This stage exercises what the team has established as rules, roles, processes and procedures. At this point the team accepts each member as a contributor and works toward accomplishing their task. They need to renegotiate their decisions from the last phase. Norming tests the team's ability to navigate inevitable changes and challenges that come with time. Team members continue strengthening the connections needed for communication within the team and with other organizations.

The mood of norming is seeking harmony. At this point people have an investment in the team and the momentum is to preserve that cohesiveness. The only disruption may be a return to frustration and conflict during renegotiation. The level of team trust required to get to norming generally carries people through any difficulties at this point.

Norming concludes when the team realizes that their successfully negotiated roles, rules and process permit proceeding. They no longer feel obligated to focus on the mechanics of team functioning and now are free to focus fully on their task.

4) Performing.
At this point the team concentrates on task accomplishment. Through performing their roles and progressing toward completion, the team shares insights and grows closer and more effective in coordinating interconnected responsibilities. This is the stage where jelling is in full swing. Zen masters use the term "flow" to describe the almost unconscious and always joyous movement through the passages of work.

The mood of the team is excitement and liveliness. Individuals gain insight into their work, their role in the task, and their role as part of a team.

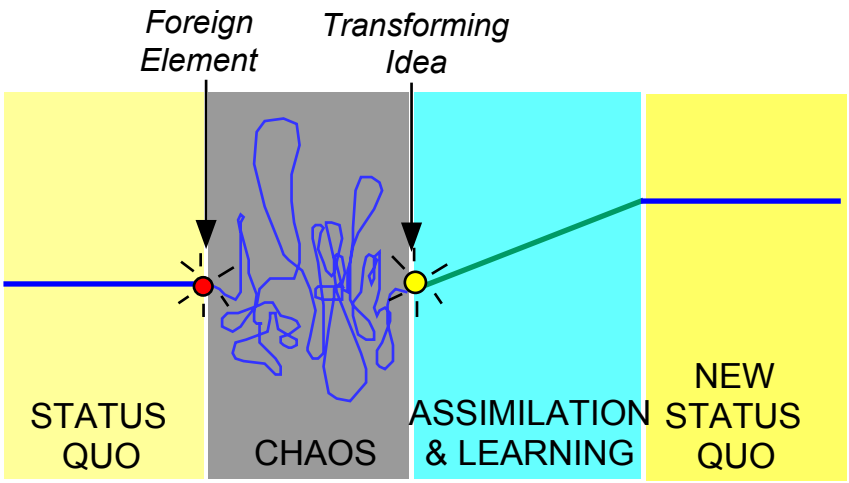
The end of the task marks the end of this stage.

5) Adjourning.
This stage serves to acknowledge the importance of the group experience. It also is a point at which people may express or come to terms with the sense of loss as a team disbands. The mood may be both somber and celebratory. The ability to express feelings at this time reinforces the team's achievements and acknowledges its triumph.

Exiting this last stage each person has had the opportunity to review the bigger picture of their experience, to grasp a better understanding of the results of choices they made, and to carry these experiences forward with new awareness.

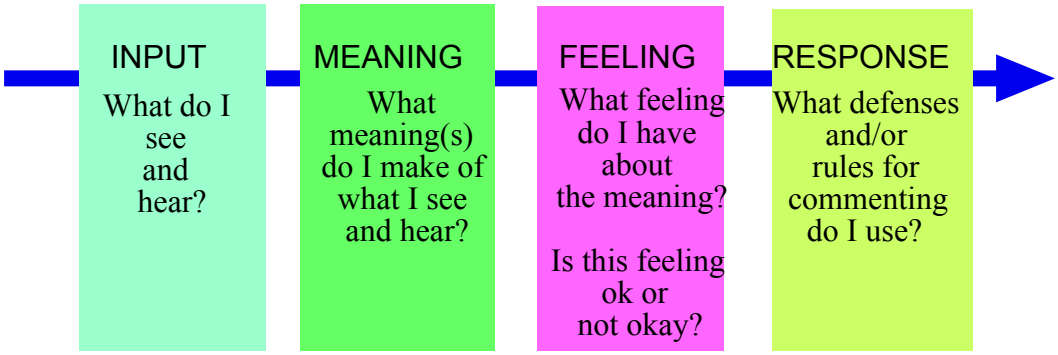
Models That Help
Psychologist and a family therapist, Virginia Satir, captured models of human behavior that offer insight into why dysfunction may happen and what characterizes functional behavior. Three of her models that we will describe are: The Change Model, The Interaction Model, and Coping Stances.

The Change Model
(Fig. 2)



Virginia Satir observed that all groups, family or otherwise, go through the same process of change, shown in figure 2, which has four main phases: Status Quo, Chaos, Assimilation and Learning, and New Status Quo. In this process people are jolted out of status quo when a foreign element introduces change into their world. The foreign element may be imposed from the outside – for example, the stock market falls and investors close down your company. It may come from the inside – your role on the team is eliminated. At this point you will enter the next phase – chaos. In chaos, people are not sure how to behave or react to immediate situations. The length of chaos is not predetermined. It may be long or may be very short. At some point a transforming idea, the realization of how you can move forward through change, occurs If a team is experiencing change together, everyone shares the transforming idea. Following this shared realization, people develop new ways of working that support and reinforce this new idea. This phase is called 'assimilation and learning'; change is being incorporated. Eventually the change cycle is complete upon entering to a new status quo; everything is routine and calm.

The Interaction Model
(Fig. 3)

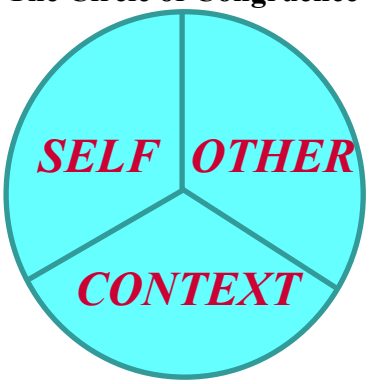


Satir noted distinct steps in any communication. These include: Input, Meaning, Feeling, and Response. Sometimes, consciously or not, your communication with others considers each step; sometimes you may rush through a conversation without much consideration and then wonder why you are in an argument.

Each step can be associated with questions:
• At input, "What do my senses detect? What observations do I have? What do I see or hear?"
• At meaning, "What meaning do I make of my observations?"
• At feelings, "What feelings do I have about the meaning(s) I have made? What feelings do I have about these feelings?"
• At response, "What defenses do I need, if any? What rules do I use for commenting in forming my response?"

Coping Stances
(Fig 4)

The Circle of Congruence



The elements of a whole and healthy stance, called "congruence" are:
• *Self*. You acknowledge your own wishes and needs when interacting with others.
• *Other*. You acknowledge the wishes and needs of others.
• *Context*. You remain cognizant of the current situation in which the interaction takes place.

When someone responds to something you say and you perceive a "mixed message" you are experiencing what Satir called "incongruence". The phrase "mixed message" aptly describes behavior that lacks consideration of any element of the interaction. Such behavior may include:
• *Placating*, which is the consequence of not including *Self*.
• *Blaming*, which is the consequence of not including *Other*.
• *Super-reasonable*, which is the consequence of ignoring *Self* and *Other*.
• *Irrelevant*, which is the consequence of ignoring *Self, Other* and *Context*.

The Myers-Briggs Type Indicator (MBTI):
Isabel Briggs Myers is known for her study of human personality, which led to the development of the Myers-Briggs Type Indicator (MBTI). MBTI is an instrument that measures individual preferences along four dimensions. Very roughly they are:
1. How you process information,
2. How you verify information,
3. How you make decisions, and
4. How you take action.

This paper will not present MBTI to any depth. We introduce it because in our experience it provides insight into behaviors that may stem from a preference different than your own. For example, on the first dimension Becky has tested "Extravert" (the MBTI spelling), and Pat, Introvert. In working together we recognize that Becky prefers to talk about things to process information. Becky realizes that Pat won't want to spend lots of time on a phone and would prefer time and silence for her internal processing. Years ago, before Becky learned about MBTI she was much less patient with people who seemed unresponsive to her invitations to talk things out. Now she knows to offer space and more choices so that Introverts can work in their most effective mode of processing. The result improved communication and increased harmony.

Frameworks Help
A framework can focus people's attention on issues and questions that can help them in their work as a team and in becoming a team. Sometimes the suggested framework is used as is and sometimes a group adapts it to their needs. Examples of frameworks useful during team development are: team charters, temperature readings, and retrospectives.

A Team Charter.
A team charter makes the values, processes, procedures and mission of a team explicit. The process of creating a charter can make assumptions visible, head off potential conflict, and establish team choices for work, negotiation and communication. In some cases people write down their decisions in a formal charter document. Other teams, less formal or smaller, may decide that they do not need a document after they have gone through the process.

The elements of a team charter provide agenda items that give people a place to explore what they might become as a team. The core of a team charter includes:

• What is our stated goal?
This lets the group revisit what they believe their shared task or mission to be. In forming a statement, ambiguities and variation can be addressed openly and a sense of purpose can emerge.

• What do we (this particular set of individuals) value as qualities of a team? The list of qualities the team deems important shapes a picture of who they all are and what is important in how they work together.

• What decision-making process will see us through any problems or conflicts? A decision-making process is necessary for the team to resolve differences that may arise, accept or reject work products, and, even, change the processes they may agree upon later in the charter. This acknowledges that a charter is a set of ground rules, which must flex to handle what is unknown at this point.

• What expectations do we have about our task/mission?
Along with the statement of their goal, this exposes any expectations people have about pursuing that goal. Sometimes people identify an initial list of individual tasks and roles.

• What expectations and procedures do we have for our communication?
Teams need to consider if there are preferences for various communication forms: meetings, email, memos, reviews, phone calls, web sites, and so forth. People may choose one form to associate with "official" communication. They might ask: How do we declare a work product done? What do we do to keep each other informed without being intrusive? Additionally any number of related questions about information flow, opinion solicitation, and status may be included.

• What resources do we have or might request?
As people discuss all of the above points, they need to tally resources and might discover that they need a resource they don't have – for example, easy access to a meeting room. Tools, space, people, and time may all be considered. This discussion may suggest how easy or difficult it might be to successfully accomplish their task.

After discussing all of these points, people may have additional lists – open issues, risks, and questions for their sponsor and research items. Team members may discover they have a virtually impossible task or may feel more confident about working together.

A final process in team chartering is "sealing the deal". The group may conduct a vote or shake hands or both to accept the charter as their working agreement. If there is a document, all participants might sign.

The team charter provides a framework for discussion. The discussion helps create initial team ties and methods of communication that can pay off during later stages.

Temperature Readings

The temperature reading is a tool from Virginia Satir, introduced to change "the temperature" within people, between people and among people" (Satir, Banmen, Gerber, Gomori, 1991). This formatted session solicits the exchange of information in the following order:

1) Offering appreciations to another person in the group;
2) Providing new information to others that you hadn't offered before;
3) Stating puzzles about something that you would like help solving;
4) Submitting a complaint with a recommendation on how it might be resolved; and
5) Articulating your hopes and wishes for the group, yourself or the project.

In these sessions people get to learn about the human beings that are their teammates and this eases the tensions when the going gets tough for the group.

> *"Technical people often feel uncomfortable sharing their personal feelings toward a project. But there are recognized levels of information beneath the surface of what we hear and see*." (Strider, 2002*)*

Not everyone must to say something in every temperature reading. No offers may be forthcoming for a given category. You might get a lot of information or little. The information may be broad in nature or very focused. Naomi Karten further explains the 5-step temperature reading (Karten 2001).

Retrospectives

The framework of a retrospective could be used at any point where people need to answer the question: How did we get here? This can help to resolve outstanding issues that might get in the way in the future. In fact, a retrospective might use any number of frameworks as mechanisms to help people focus and take a new perspective. While it is not in the scope of this paper to explore retrospectives, we will discuss them where helpful. We suggest Norm Kerth's book listed in the references as a great place to learn more (Kerth, 2001)

Specific Applications In Different Stages

Stage 1: Forming

As you enter a group you want to know who the others are and why you are here. Being aware that personal preferences, such as MBTI, exist and knowing something about differences broadens your perspective. Being aware of your own preferences can allow you to be tolerant and open to learning about differences in others.

Simple frameworks can work to ease the tension of first time introductions. Team meetings in non-technical settings, such as pizza lunches, coffee breaks, or off-site retreats, establish non-threatening venues. Smaller, frequent meetings might work where people can easily meet face-to-face. Brief introductions in informal settings can remove a burden of feeling a need to prove oneself or feeling exposed.

The group will be exploring and asking themselves questions: What is this team going to do? Is there something interesting for me? Will it be fun? And, they will be asking others: Does anyone know about our intended goal?

Today some teams don't have the luxury of face-to-face meetings. While technology, such as video conferencing, can be useful for simple introductions and placing a face with a name, it won't address the heart of interaction – what does he or she mean by that? With remote interactions, such as those by phone, email or video, Satir's Interaction Model reminds you of choices. Often we filter information through the meaning and feelings of past interactions (Emery 2001). When working with people where there is no mutual history, we might ask ourselves questions with open possibility: What might that mean?

A tough situation is finding yourself on a team with a "sworn enemy" or someone with whom you have had past disagreements and tensions. Consider the congruence Coping Stance. If you remain with the context, and not the past, and consider both your needs and the other person's, in some cases past difficulties can ease. However, sometimes they won't.

Opening up communication lets you learn about each other and make congruent choices for yourself and the team. Introducing the framework of temperature readings lets people say things that don't fit in business, project-oriented, or free form sessions, but are necessary for building confident and flexible interactions. In this first stage, temperature readings are more successful with an outside facilitator.

The freedom to disagree and to air potential problems supports team trust and increases the likelihood of selection of the ‘best’ course as the team moves through later stages. This freedom includes letting an individual chose to leave the team if they do not fit.

Stage 2: Storming

The storming stage provides the opportunity for the team to develop conflict management skills as they detail their task and define how they will accomplish it.

The model that explains why storming is painful is The Change Model. Discussing matters of importance – your work and your role and responsibility – can be uncomfortable, particularly with unfamiliar people. This may introduce a "foreign element" that jolts you out of your status quo expectations. Even if you are not experiencing chaos, someone else may be. Chaos is a phase where people slip into the defenses of incongruent stances.

Because storming is a stage that can make or break a team, developing a Team Charter can guide the individuals to the topics that are important to collaboration. This framework sets agenda items for discussion and reduces extensive debates on where to start. Following a plan can eliminate free-for-alls, which can increase the turbulence of storming, and keep people focused on the work.

The team can use information from MBTI, a tool for discovering differences in individual preferences for communication and work habits, while determining specific work products, tasks, roles and responsibilities.

Remembering Coping Stances and congruent behavior can improve your effectiveness when offering suggestions and comments on proposals. When people discuss their charter there may be disagreement. As conflict builds it can be easy for behavior to slip into placating, blaming or being super-reasonable. The circle of congruence is a reminder that your interaction should acknowledge your concerns, address the concerns of others and be in context of the discussion.

The Interaction Model is important for resolving disagreements. When you sense confusion or misunderstandings, you explore the communication steps: what was heard, saw or observed, what was made of that, how it affected feelings and how a response came to be.

Should tension mount or communication get really tangled, the framework(s) of a retrospective or team review might get everyone refocused. One example of a retrospective framework is called a time line. During storming people might create a time line of team charter development to expose points in time where they got stuck. Each person adds his or her personal experience on the time line chart. The chart will exhibit information, such as energy level and specific events or decisions. As the group looks at the entire picture of individual experiences, often people recognize patterns or see something totally new. These sorts of observations can lead to very fruitful discussions and productive resolution.

As we mentioned earlier, there are groups for whom storming is a continual state. This is most likely to occur when trust didn't happen in Forming and also when managers or sponsoring organizations mandate participation in the "team". Sometimes healthier sub-teams form in order to complete their work products. People might quit or go around the manager to get reassigned if possible. Certainly this scenario can benefit by lots of practice using the Satir models and tool frameworks.

Stage 3: Norming

Moving into norming, the team accepts each member as a contributor and works effectively together. The models and tools that were critical in the earlier stages need not be as prominent here. The key tool for the team is practicing their congruent interactions while following their chosen process and focusing on the task.

MBTI can be used to examine the balance of diversity on the team. Certain types of errors or omissions may occur when many people share the same preference. For example, some types like to take immediate action and are driven by time and schedule. Other types defer action and are driven by a need to have sufficient and appropriate information. Both are important. A team that is aware that a preference is missing can find a solution that keeps them in balance.

Retrospectives can continue to be tools used to strengthening connections within the team and communication with the sponsoring organization.

The team's interaction with the outside, such as managers, sponsors and customers, is where the models come back to the forefront.

Stage 4: Performing
At this stage a team is most productive. A good team will spend most of its time performing. In a well-meshed team the members work together honestly with respect for each other. For example, software and team reviews will be conducted in a manner that promotes team learning while improving product quality.

Teams tend to have integrated models, tools, and their own process into a nearly unconscious response.

Stage 5: Adjourning
All too often a team may be dispersed before they have the opportunity to move through adjourning. Like moving to a new place where you must leave friends behind, allowing time for the mourning and celebration of this stage promotes and honors the teams learning experience.

A retrospective offers framework(s) for an adjourning process and lets the team examine their lessons together. Discussing factors that contributed to successes and failures provides an opportunity to gain more insight into team effectiveness.

A temperature reading is often used during retrospectives. At this stage the team members can make full use of the opportunity to share what they feel "within them, between them and among them".

Observations About Teams

1) Good teams can be real, but how they got there isn't obvious.
In writing about non-technological issues (DeMarco, 1995), Tom DeMarco recalled something Fred Brooks said. Fred Brooks taught a software engineering lab at the University of North Carolina and had students do the same project in the same schedule in the same size group for many years. Students could use different techniques and methods.

> *"But, he [Brooks] reflected, methods and techniques didn't seem to matter as much as he expected. They didn't matter nearly as much as certain non-technological factors like team harmony, clever role allocation and a group sense of purpose"*

Although we have not had a lab where experimentation could be repeated, this is what we have experienced as well. Most people have experienced a well-functioning team, but the factors that contribute to such a team are not obvious.

2) Movement through life cycles stages isn't the same for all teams.
Different teams will progress through the stages of development at different rates. The rate depends on past history, individuals involved, mission risks, and environment imposed by organizational culture. Becky and Pat have both seen teams that zipped through the first two stages and cruised through the next two. That isn't typical.

3) Teams can get permanently stuck in storming.
Sometimes people can't agree on a process or ground rules, let alone discuss values. This often results in little, if any, progress towards the team's goal. Teams may make progress while stuck

in storming. In this latter case, people have figured out creative ways to cope with the dysfunction, while getting their job done.

Pat once had the joy of becoming the new member of a team that had been in the storming stage for over a year. The good news was that gaining a new team member did not deteriorate the level of functioning. Pat enabled progress by running interference, thus minimizing the team lead's tendency to micromanage and provoke team members. This band-aid worked for a while. Upper management provided the true fix by replacing the team lead.

4) Storming is the most crucial stage for team definition.
By the end of this stage the charter and processes are fully debated, clearly defined, and understood. Teams that survive storming successfully are aligned and have the strength to withstand future disruptive events.

5) Norming can be a trap, too.
While the team may be productive at this point, they may be more focused on conformity and group harmony than product quality. This can result in eager agreement that reduces the quality of decisions.

Sometimes insufficient diversity can result in a team that fails to ask certain questions or challenge a choice, which may reduce \effectiveness in improving product quality.

6) People outside the team can disrupt the team.
Managers, customers, and sponsors are three examples of people who have the ability to throw "foreign elements" into the team experience. Examples are changing the schedule, changing specifications, firing someone, adding a new person, or interrupting needlessly. As they develop their process, teams need to provide for important communication outside their group.

7) A team is not a constant unit.
A team may be disrupted at any point. Someone may leave and someone new may join. But the new collection of people is not the same team. This principle is easy to understand if you think about Brooke's Law:

Brooke's Law: Adding manpower to a late software project makes it later.

The reason that this is true is because the new person needs to be brought up to speed. That often requires that one of the team members must be sidelined to do that job, resulting in two people on the team who aren't functioning at full capacity.

8) Anytime the membership of a team changes, the team starts the life cycle all over.
A change of membership means the group is a new group.

If there are new members, there will need to be a period of getting to know new members, and they you. The process, ground rules, roles and responsibilities may need to be renegotiated. A team with a strong process whose members are readjusted will likely progress through the early stages quickly to regain productivity. This sort of team may be very fluid in how they take on roles and responsibilities. In that case and where the addition is only one or two people, an established team member can work with the new members to bring them up to speed on the process, task and team history with minimal disturbance.

If a team has lost a person with a key skill, traversing the stages over again may be more difficult.

9) If the team membership changes, the team may be thrown into chaos.
The loss of a key team member may leave the others at a loss as to how to replace that person's skill. The addition of a new member with competing skills or a forceful nature can throw a team into chaos.

A strong foundation anchors a team's center and maintains values and focus. A weak foundation promotes chaos and storming and reduces time for the specific team task.

Jelled Teams and Better Products
Let's revisit the observation we started with from Gerald Weinberg:

> *The quality of a software product cannot exceed the quality of the organization that builds it.*

Perhaps it makes more sense now that we can:
• Recognize the natural team development life cycle, and the challenges people face in forming and storming. A team that becomes stuck with their own process is less likely to create work products that integrate properly.

• Understand the tacit models that govern human interactions. We can see that quality communication is not a simple thing.

• Allow that individual awareness level is unknown. Any behavior we witness could easily be misinterpreted and foster vengeful reaction instead of a responsible reaction.

• Assume control of our own actions and reactions through practice of Satir models; by learning about personal differences, such as that suggested by MBTI; and by using frameworks that can help us move forward and help us take a step back and reflect.

Gerald Weinberg (Weinberg, 2001) reminds us that

> *The team is the basic design unit for software engineering processes. ... When there are multiple eyes, there are many more chances to see a fault.*

Imagine if all organizations recognized the asset that is called a "team". Those organizations that have invested in functioning teams gained skilled teams, knowledge about growing good teams, and products that fully reflected the quality that a peak-operating unit produces. Not to mention the cost savings from having an existing team or teams that share values, processes and the experience of performing their best.

Team synergy creates a combined pool of knowledge and maximizes the probability of optimum choices. The potential pitfall of an ill-conceived mission, which might derail success or quality, is most likely to be exposed by a team with a strong center, mutual trust, and a sense of responsibility to return the best decision to their governing organization.

Notes:
(1) A communication during a SEM98 meeting in August 1998 at which both Pat and Becky were present.

References:

Crosby, P., 1979, Quality Is Free, New York, New York: Penguin Books

DeMarco, T. and Lister, T. 1999, Peopleware, New York, New York: Dorset House Publishing

Emery, D.H. 2001 "Untangling Communication", STQE magazine July/August. Also http://www.stickyminds.com/, #2975.

Karten, N. 2001. "Conducting a Temperature Reading". http://www.stickyminds.com/, #2535

Kerth, N. 2001, Project Retrospectives: A Handbook for Team Reviews, New York, New York: Dorset House Publishing

MBTI web sites: http://www.KnowYourType.com/ ; http://www.typelogic.com; http://www.ibiblio.org/personality/faq-MBTI.html

Perceptions of Engineering Teams at NASA (LaRC): Findings from a Survey of Engineers & Scientists. http://www.icase.edu/newresearch/teamwork/factors5.html

Robbins, H. The NEW Why Teams Don't Work: What goes Wrong and How to Make it Right. Berrett-Koehler Publishers.

Satir, V., 1988. The New Peoplemaking. Palo Alto, CA: Science and Behavior Books.

Satir, V., Banmen, J., Gerber, J., and Gomeri, M. 1991, The Satir Model, PaloAlto, CA: Science and Behavior Books

Strider, E., 2002. "Improving Projects by Communicating What's Below the Surface" http://www.stickyminds.com

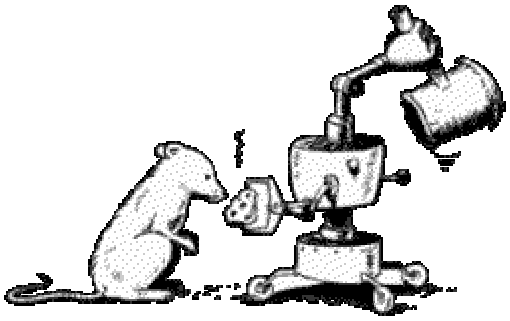
Weinberg, G. M. 1994, Quality Software Management, Volume 3: Congruent Action, New York, New York: Dorset House Books.

Weinberg,G. M., 2001. "Weinberg on the Essential Team". http://stickyminds.com #2937

Zoglio, S.W. 2001. 7 Keys to Building Great Work Teams. Teambuildinginc.com, Articles section. Or http://www.stickyminds.com, #2769

Building a Better Defect Tracking System

Bill Beck, Test Engineer, Hewlett Packard DeskJet Printers
18110 SE 34th Street, Vancouver, Washington 98683
360-212-4541 (voice), 360-212-4073 (fax), bill_beck@hp.com



Abstract

If you wanted to build a better mousetrap, you might start by researching a little, defining the essential elements, then designing and building the mousetrap, all the while keeping your eye on the goal: catching mice. What about building a better Defect Tracking System (DTS)? The same steps should be followed, with a similar focus on the goal. And, just as three different people will come up with three different ways to build a better mousetrap and quite different ideas about the essential components, so in defect tracking systems will each person have a different vision of the ideal system.

This paper doesn't attempt to describe the perfect system; it simply tells the story of one effort to build a better defect tracking system that others might learn from our experience.

Biography

Bill Beck is the senior Test Engineer in the Firmware Test Lab in the HP Vancouver Personal Printing organization, with 9 years' experience. His group provides testing and quality engineering services to the firmware development teams for all DeskJet printers. His areas of interest and experience include: the PCL (Printer Control Language) language and associated language conformance testing; Defect Tracking management and training; test planning and test development and process improvement.

Bill Beck brings an interesting background to his position at HP. He has a B.A. in Latin and a Master of Divinity, which he has supplemented with courses in programming, testing, and related areas. He served as an officer in the U. S. Navy for 4 years, and as a pastor in the Lutheran Church for 22 years.

1 INTRODUCTION

This paper describes a major, long-term effort to build a better DTS: to replace multiple defect tracking systems with one business-wide system, and do it in the context of a broader, collaborative environment. In addition, the solution had to integrate the defect reporting and fixing activities of outsource test and development vendors.

Our business (HP imaging and printing group) involves at least 8 separate HP locations on four continents. Each location has unique needs based on the widely differing products being developed, as well as on the different cultures, development and test lifecycles, and defect tracking histories used at the various locations. Even at a single site there are disparate needs between hardware, firmware, and software developers and testers. This paper will have a Vancouver Site flavor (the author's location), but will reflect the overall effort and the various perspectives of all the participants.

Further complicating the picture is that this defect-tracking project was only one part of a broader effort to create a collaborative development environment encompassing Source Control, Defect Tracking, and Knowledge Management. The "better mousetrap" was only one element in the overall project, which involved a combination mousetrap, stapler, CD player, and baby swing! A complete description of the whole project is beyond the scope of this paper, but it will be briefly described below in order to provide the context for the development of the Defect Tracking System, and the challenges faced by the teams working on it.

This paper will outline, briefly, the history that led to the effort, the process that has been followed in the creation of the Defect Tracking System, the progress to this point, the prospects for completion of the project, and the client/vendor relationship. Finally, the paper will discuss lessons learned through this process.

2 HISTORY

This section will focus on the Vancouver Site, as a microcosm representing the whole. Vancouver Site has two main divisions, each focused on a different segment of the overall printer market. Vancouver Personal Printing is the design center for DeskJet printers, intended mainly for home and small office printing. InkJet Publishing Division focuses on the high performance printing market. Each division has its own R&D groups for Hardware, Firmware, and Software, as well as it own quality and testing organizations.

2.1 Defect Tracking Patchwork

Over the past several years at this single site, there usually have been three or more defect tracking systems active at any given time. On the hardware side, defect tracking started as sticky notes on a cabinet or scribbles on a whiteboard. This was replaced by a flat-file database created locally, using FileMaker Pro. In January 1999 the hardware teams joined with the firmware teams and moved to a homegrown, web-based system, with an Oracle back end. This system has been widely adopted and well accepted. The firmware teams, prior to this new system, had been using an HP corporate system that was also used by the driver teams. The Macintosh driver team used different systems, one for OS9 and one for OSX. This kind of scenario was duplicated at each of the other HP sites.

There had been two previous attempts at the Vancouver Site to come up with a comprehensive system that would be used by all teams. The primary goal was to allow defects to be easily transferred from one team to another. The first attempt ended when the system failed to meet the minimum requirements established at the beginning of the program and showed little prospect of being able to do so without massive additional investment of energy, people, and money. In addition, the new system was plagued by frequent system crashes of users' systems. I, for example, was a regular user and found myself rebooting about every half hour. A stringent list of required features and the inflexibility of the various disciplines contributed significantly to the problems encountered. Following a major rebellion among those using the new system, there was a quick and painful retreat to the old system. A second attempt to develop a new DTS died through lack of funding and commitment during a time of rapid change at the Vancouver Site.

2.2 Collaborative Development Program

Meanwhile, changes in the industry and in HP were requiring development to move out of individual division and site silos. Collaborative development was becoming a necessity. Early efforts in this direction derived from cooperative efforts between San Diego's All-in-One and PhotoSmart divisions and the Vancouver divisions (Vancouver designed the basic printers on which the San Diego products were based). The first effort was a firmware architecture that minimized rework for each new product line. Over time, additional efforts were started, with increased cooperation among the divisions and the rising use of third-party vendors for software and firmware development and testing. These efforts were hampered by differences in development environments at the various sites, different development models, widely different and incompatible defect tracking systems, as well as by the fragmentary nature of these efforts.

About two years ago, a major effort was undertaken to establish a Collaborative Development Program (CDP) that would unify all the different divisions under one model. This involved a common development environment and source control system, a common defect tracking system, and a single knowledge management effort. The vision was to have this all running in the framework of a hosted web environment that would be accessible to all HP divisions and to any third party vendors involved in these collaborative efforts. The knowledge management piece was intended to provide not only document management, but also file sharing, communication forums, and mailing lists. We are not fully there, yet, but have made considerable progress. The pieces are coming together and being used by increasing numbers for a growing list of projects, even while we continue to work on it. However, that the system evolved into an integrated, seamless package has been both good news and bad news.

Having an integrated tool has theoretical advantages, but it also creates a tremendously complex project when you try to bring all these pieces together at once. Further complicating the effort was the "remodel the house while living in it" approach. About 18 months ago the CDP framework went live for the first projects. There were an interim Source Control tool (CVS) and an interim defect-tracking tool (Issuezilla) in place from the start, though neither was intended to be more than a temporary placeholder. Both these tools and the CDP framework itself were being worked on even as a growing number of teams began using them. The framework is project-based, and individuals began creating projects for very different purposes without clear guidance on how projects should be related. Furthermore, the framework in these early stages

was limited to two levels, category and project, while the intent of the final version is to have a multi-level hierarchical pattern.

A major crisis point came this past February, when an "upgrade" to the CDP framework brought the whole system to its knees. Wait times went from seconds to minutes. This framework problem had a major impact on the DTS side of things as resources were redeployed to solve the problems caused by the update, since the same vendor was providing both the CDP framework and the DTS tool. To add insult to injury, the delays required renegotiating the contract with the vendor to refine and extend it, adding further delays.

3 PROCESS

As CDP started taking shape, core teams were formed for the overall framework and for the individual components. This section will describe the process followed by the Defect Tracking System (DTS) core team.

3.1 DTS Core Team

The DTS core team had representatives from six sites representing at least 8 different divisions. It is important to note that almost all of the core team members were "volunteers", for whom this was only one of several responsibilities. This imposed some limitations, but it also meant that most of the team members were actively involved in using one or more of the current defect tracking tools in their primary job and had both a strong interest in making the new tool work and an excellent firsthand knowledge of the needs of a defect tracking tool. The one person assigned specifically to the team was responsible for organizing and leading the DTS Core Team meetings, representing DTS at the CDP Core Team meetings, serving as liaison with the vendor, maintaining the project website and other documentation. Even he had other responsibilities.

One of the key elements in the early discussions, and something that had to be revisited at various times in the project, was the development of a common vision that enabled each individual group to let go of some things in order to define a common process and set of expectations that would enable all of the groups to work well in a collaborative environment. This required a balancing of our individual needs, preferences, and desires with the goal of a common tool that would enhance our productivity when working collaboratively.

We also formed separate subteams to work on a state model, the user interface (mainly the common set of fields to be used in the tool, both required and optional), a glossary, data migration, and training. These teams formed and disbanded as necessary, with a lot of overlap in membership because of the size of the core team.

Weekly meetings kept team members up-to-date on progress and enabled prompt response to changes and issues that came up. Even with the weekly meetings, there was a lot of reviewing and recalling what had happened in the past as the project progressed. This was due, in large measure, to the fact that most of us were working on this only part time, and each had many other responsibilities.

3.2 Developing a Specification

As the team began its work, we recognized that a clear understanding of our needs and wants would be important, so we surveyed all the divisions for their understanding of the essential elements in a defect tracking system. Using this information, we developed an External Reference Specification (ERS) that detailed and prioritized our list of requirements.

Here is a list of the "essentials" distilled from the 15-page document we created. We required a single tool that (1) supported various operating systems in (2) a unified environment that also provided source control and knowledge management, while providing continuing access to legacy data, and (3) enabled collaborative development both within HP and with third-party partners. This (4) web-based tool would be (5) controlled by a flexible state model, (6) easy to use, and (7) provide robust, reliable service to (8) an expanding group of users. Also required were (9) customizability and (10) versatile reporting capabilities. The following paragraphs discuss these requirements in some detail, along with a chart (Figure 1) showing how the legacy systems and the tools under development might be scored in relationship to these requirements.

Requirements 1, 2, 3, 4, 7, and 8 were all basic requirements of the CDP as a whole, and were not negotiable. All taken together are intended to provide a flexible, efficient environment that would enable collaborative development by anyone authorized by HP, anywhere in the world with access to the Internet. This requires a satisfactory level of security that is flexible enough to allow each person logging in access to only the appropriate and necessary areas. Web access means that the system will work on UNIX, Windows, or Macintosh systems; this was an essential requirement that narrowed the choices in the previous efforts to acquire a unified DTS in the days before the Internet became an option. The requirement (8) relating to the ability to handle an expanding base of users was crucial when you consider that we would begin with 3,000+ HP users and 1,000+ non-HP users and expand to 15,000 or more, spread over 300 projects to begin with and expanding to 600 or more.

The state model requirement (5) was included because this was a feature common to virtually all the prior defect tracking systems. Also, it was felt to be necessary to guide users toward the right behaviors. Smaller systems can get by with a less-structured solution, but it was seen as essential in our environment. The state model we developed was a very simple one featuring only four states: submitted, assigned, resolved, and closed. While we agreed to hold to these four states, the tool had to be customizable with respect to the actions triggered by state changes. This has not yet been implemented in the system under development, but has been promised for the near future.

Easy to use (6) was an obvious, non-negotiable requirement, because it is a fact of life that users will rebel against a complicated, burdensome system. We didn't want to lose important defect reports because of user frustration or confusion.

Customizability (9) (with respect to the fields present in the tool) was identified as essential for two primary reasons. First, we knew from the beginning that there would be very different needs because of the great differences in the products we develop. For example, consider the component field. There would be one set of possible values that a printer division would use, while the values used by a storage division or a network device division would be totally different. The core team accepted this need from the beginning. The second need was recognized

later and was related to our realization that some flexibility would be required to encourage adoption of the new system. We started out wanting to enforce a single, uniform set of fields so that every group would be using the same fields. The core team managed to come up with a set that everyone on the team could live with, but it was harder to convince others. So, we accepted the fact that some groups would need a few special fields and that a little flexibility would help ease acceptance of the new system. Thus far, the changes requested have been minimal.

Finally, flexible reporting capability (10) was identified as another very important need. There would be some "canned" reports available from the beginning, but each team wanted more, and there were questions about whether all the fields would be available for creating customized reports. The use of metrics varies among the different product managers. For some, the metrics they are used to gathering and the reports they are used to receiving are central to their management of their projects. In addition, they want to be able to compare metrics from project to project, so importing the legacy data and including it in their reports was important as well. The concerns have been alleviated by assurances that all fields will be available for searching and reporting, and the data can be exported to other tools for creating any reports not available, or not able to be generated within the tool. Efforts are underway to create the variety of reports used by different teams.

In Figure 1, I have summarized the extent to which different defect tracking systems meet the requirements for this project. The legacy systems are all currently in use, along with three or four smaller systems, and this is just at the Vancouver Site. There are probably dozens of different systems still being used throughout our various divisions.

Figure 1:

	1	2	3	4	5	6	7	8	9	10
	Single Tool	Unified Envir.	Enable Collab. Devel.	Web-based Tool	State Model	Easy To Use	Robust, Reliable Service	High Capacity	Custom.	Flex. Reports
Legacy System 1 (ATM)	NO	NO	NO	NO	YES	YES	YES	NO	SOME	YES
Legacy System 2 (ATM+Web)	NO	NO	NO	YES	YES	YES	YES	NO	SOME	YES
Legacy System 3 (DTS)	NO	NO	NO	YES	YES	YES	YES	NO	SOME	YES
Interim Tool (Issuezilla)	NO	YES	YES	YES	YES	YES	SOME	SOME	NO	NO
Pilot	YES	YES*	YES	YES	NO	YES	SOME	SOME	SOME	SOME
Production Release	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES

Dark gray shading indicates features that preclude continuing use of a legacy tool for future needs.
Light gray shading indicates areas of concern, but that wouldn't prohibit current use of a tool.

*The pilot projects will be using the tool in a Beta version of the CDP framework, running on a test server. Full integration will not be possible until the updated framework is officially released.

3.3 Choosing a Tool

The next step was to use the ERS requirements to evaluate available defect tracking tools and other options. We considered two paths forward. The first was to choose an off-the-shelf product that could be integrated into our framework, while the other involved creating the tool from scratch. After a period of evaluation, it was determined that none of the off-the-shelf tools would satisfy our needs. There were several reasons, among them the following: high cost of tools and licenses, unwillingness of tool vendors to work with us to customize their tools for our needs, and the difficulty of integrating the tool into our overall framework.

After a careful evaluation of the various possibilities, the core team for the overall Collaborative Development Program accepted the proposal of the vendor who was earlier selected to develop the framework and to provide Application Service Provider (ASP) services for the whole program. Their proposal involved customizing a defect-tracking tool from a project they were already planning. They were willing to work closely with us as the primary client to ensure the tool met our needs. Since they were providing the framework as well, we could also count on the defect-tracking portion being well integrated into the framework. The cost of the project would also be much more reasonable.

Once the vendor was selected, the team began the long process of working with the vendor to ensure that progress was being made and that the developing tool would meet the requirements.

4 PROGRESS

Currently, we have our user interface (UI) in place and are testing and evaluating the tool, though it is not yet integrated into the CDP framework. The first use of the new tool was the pilot program, which began in August with two small volunteer teams. Once the support team has these teams set up and running, additional teams will be added. The DTS tool has been integrated into an unreleased version of the CDP framework operating on a separate server. There is direct access to the tool from the main framework, but it isn't fully integrated. Because all the pieces are not integrated in one tool, we expect to have a somewhat higher level of user dissatisfaction. As we proceed and the integration improves, we can expect the user satisfaction to rise. Anticipated rollout date for the full production version is now tentatively scheduled for December 2002 or January 2003. (Original rollout was scheduled for June 2002.)

One of the major areas of concern at this time is the state model or workflow engine. One of our prime requirements was a robust, yet flexible, state model controlling movement through the defect life cycle. The vendor, originally, did not want to provide this in the initial production version, but has now signed on to provide it. Though it was not in by the start of the pilot projects, it is expected to be added within the next month. The delay has seriously slowed our progress in testing and evaluating the tool.

Another area of concern is related to "selling" the tool to the decision makers. There is high-level support for moving in this direction, but individual program managers have to be convinced that this new tool is right for their current project. I am in the process of developing a presentation for one manager that will make the case for his team switching when the production version is released. His attitude is, to quote: "Unless you can show me that the new tool will do everything I am doing with my current tool or better, I will stay with the current tool."

Those of us on the core team who have been using the tool are satisfied that the tool will meet the needs of its users, if the vendor follows through on its commitments to provide the remaining functionality on schedule. The two main pieces remaining are the state model engine and the integration into the framework.

5 PROSPECTS

We are confident that this new tool will assist our disparate teams in working together collaboratively and will be a significant improvement over what we currently have. Nevertheless, there will be a months-long effort to migrate new projects to the new tool.

Since there are so many different, virtually independent groups involved in the project, migration plans have been, or are being, created by each team as they prepare to move to the new tool. This has been complicated by the delays and uncertainty in recent months, as well as by the reaction to late availability of the other pieces in the overall solution, especially the Source Control piece. Given that this project is supported at the highest levels in our organization and will eventually have to be adopted by all groups, each team has had to address the following questions:

1. Does the tool have the features we need to handle defect tracking for our projects?
2. Does the availability of the tool, or its pilot version, fit our window of opportunity for switching tools for the current or next project?
3. What other objections or concerns do team members have that need to be addressed before our migration?
4. What plans do our partner organizations have, i.e., will their migration/non-migration dictate our plans?
5. What legacy information do we have to import into the new tool to allow us to do comparative reports of past and current projects?
6. What are the essential steps in making the transition, and when do they have to happen?
7. Will the planned, standard training be sufficient, or do we have to plan for additional training specific to our organization?

6 WORKING WITH A VENDOR

While I had worked with different third-party vendors on various test projects, this development project was a new experience for me, as well as for most of those on the core team. Our original focus was on the tool and its specifications, but we soon came to learn that our relationship with the vendor, both as a company and with its individual members, was a major component of the total picture. In the paragraphs that follow, I will outline some of the significant aspects of that relationship.

6.1 Contract Matters

One thing we certainly learned through this experience is that **contracts matter**! Most of us on the core team had very little to do with drafting and negotiating the contract, but that didn't mean we were not affected. We were reminded regularly how much our progress, our relationships with the vendor, and our schedule were tied to the contract. There was a business team that took care of the original negotiations, but it became clear to all of us that changes would be beneficial. The compensation structure was not appropriately tied to the deliverables, and incentives for on-time delivery were not strong enough. When delays required a renegotiation of the contract, intense effort was required to structure the contract properly. In addition, the negotiations dragged on to the point that further work was halted because the initial contract had expired.

One of the things that became problematic was that, because the core team was not a part of the negotiations, we were not sure of our relationship with the vendor team. Over time we learned what we could ask and expect, as well as what was out of our hands, but it took a long time and affected a number of our efforts. It would have been better to have a clear understanding from the start.

6.2 Requirements Management

This lack of clarity affected the way we dealt with our list of requirements, too. After we developed our list of requirements, we had a telephone conference call to discuss them with the vendor representatives. When they responded to some of the requirements by saying that they would not be a part of the first production release, we didn't know how to respond. The conversation that followed did not lead to a clear statement of our expectations on some important areas. It was only when these areas were revisited later that we were able to firmly state the necessity of these requirements being met. The fact that virtually all communication was done via conference call was a complicating factor, and this will be addressed below.

6.3 Progress Tracking

The uncertainties in our relationship with the vendor mentioned above, as well as the limitations of the original contract affected our ability to track progress and our ability to hold the vendor to the promised schedule. We had monthly and quarterly schedules, but deliverables kept slipping. Because those of us having regular contact with the vendor did not have responsibility for or awareness of the details of the contract, the slippages continued. We gradually became more assertive, especially as we started getting answers to our questions about our rights and responsibilities according to the contract. The new contract is enabling us to manage this aspect in a much more satisfactory manner.

6.4 Relationships

Our relationships with the vendor were positive overall, but we were handicapped by the lack of face-to-face relationships. This was true not only for our relationship with the vendor, but also for our own relationships on the core team. This frequently affected our ability to communicate well. Midway through the project, we were really struggling over several important points, and extended conference calls did not lead to significant progress. At this point, we recognized that a face-to-face meeting was necessary to break the logjam and move forward. Four of us made a two-day trip to meet with the vendor. On the first day the four HP representatives met to prepare,

and we benefited greatly from meeting together face-to-face. We came to a much better mutual understanding and were able to present a united front in our meeting with the vendor the next day. The meeting with the vendor was also very productive. We were meeting them face-to-face and benefited from the opportunity for an extended meeting without the limitations of telephone conference calls and net meetings. The progress made in a four-hour meeting was greater than what we had accomplished in two months of conference calls that spanned more than 12 hours by my reckoning. We finally understood one another in ways that had never been true before.

Looking back, I would say that an early face-to-face meeting of the HP participants and one or two with the vendor representatives would be have been immensely beneficial, and I would encourage anyone involved in a similar project to plan such meetings.

7 LESSONS LEARNED

1. It was at times frustrating to hear the vendor use the excuse that they were developing this tool for other clients as well as for us in order to avoid making changes to satisfy our requirements. They had assured us that we were their primary client, and we had made it very clear that our requirements had to be met. NEXT TIME: Establish clear understanding of the relationship. Preferably, the project would be handled as if HP were the sole customer, and our requirements would be seen from the start as just that, requirements.

2. The core team had one person for whom this was a main responsibility throughout the project, while all others were either on the project for shorter periods or were basically "volunteers." That is, they had other primary responsibilities and were able to commit time and effort to this only as a secondary responsibility. The thing that enabled the team to complete the project was the individual and collective commitment to the importance of having this kind of common tool, and one that would serve the needs of the various groups using it. NEXT TIME: Establish realistic expectations for the amount of commitment required, and make this assignment a part of each participant's primary responsibility. In other words, not just a volunteer project.

3. Since this was a part-time responsibility for most of the team and spread over many months, frequent reviews of the ERS and of decisions made were needed. Various members took the lead in bringing the team back on track when it strayed from earlier decisions and directions. NEXT TIME: Maintain better, more accessible records as a project evolves. A Decisions Made list with rationale could be valuable over time, reducing the need to revisit earlier decisions.

4. It was complicated enough to be designing a defect tracking tool for multiple divisions; to do so under the umbrella of a much more comprehensive project raised the effort to a new level. NEXT TIME: Hopefully, there won't be a next time! Seriously, though, recognize how complicated such a process is and sequence the introduction of the various parts. I would do the defect-tracking piece as a self-contained project, then integrate it into the framework piece once it has been widely adopted.

5. The tool was an evolving thing, starting out being used by small numbers of people with interim tools in place. As the project progressed, more and more users became involved, and the uses became more and more varied. NEXT TIME: Maintain tighter control over who can join the system while it is under development. Keep it small until the feedback from the original users has been used to refine the project. Doing a formal risk analysis might also be helpful to plan for the kinds of things that could hinder progress.

6. Face-to-face meetings were very helpful and more should have been planned. NEXT TIME: Begin with a face-to-face meeting, and schedule additional ones, either to address blocking issues or as a regular part of the schedule.

7. Schedule delays required contract renegotiations this spring, a complicated and drawn out process. This has further delayed the project, since we reached a stage where no work could be done until the new contract was approved. This was our first effort at this kind of project, and the initial contract suffered from our lack of experience; appropriate incentives and penalties tied to specific deliverables might have smoothed our troubled path. NEXT TIME: Use the specific lessons learned from this project to guide future, similar projects. Structure the contract to provide both clear, specific deliverables and appropriate incentives to assure they are achieved on schedule.

The Metrics of Refactoring

Title: The Metrics of Refactoring
Author: Al Lake
e-mail: alake53@attbi.com
Phone: 503-390-7576
Address: PO Box 18203
Salem, OR 97305

Paper Overview:

Refactoring is supposed to improve the readability and testability of a code segment without changing the inherent functionality.
Which software complexity metrics best describe the changes that take place in refactoring?
Can existing software complexity metrics show a difference where refactoring was done?
What are the causes of refactoring?
What are the predominate reasons for refactoring?
Can 'code smells' be quantified?

Keywords:

Refactoring
Software metrics
Factor analysis

Biographical:

Al Lake has more than 25 years of software development experience. In high school and college he worked as a computer operator and wrote FORTRAN programs. In the 'early years', he worked as a mainframe programmer/analyst using COBOL, FORTRAN, EZTrieve ™, Assembler, and OS/JCL. He then worked as a mini-computer programmer/analyst using COBOL, FORTRAN, BASIC, and RPG. After that he did development on both Macs and PCs, using several languages like BASIC, Pascal, C/C++, FORTRAN, and several database languages. He has separate Bachelors of Science degrees in Math and Computer Science and a Masters degree in Computer Science. He has development experience with object-oriented development, database systems, visual languages, software metrics, various programming languages, and software quality processes. He has worked as a computer consultant for over 10 years and as a project leader on several projects. Currently he is a software development/quality assurance engineer for Mentor Graphics.

The Metrics of Refactoring

Background

Research Purpose

The purpose of this research was to determine if readily available software complexity metrics detect a significant difference in code segments where refactoring was done. Measuring refactoring provides a means to evaluate the types of changes made during the refactoring process and provide automated methods of determining code where refactoring could be utilized.

While using the term 'code smells' [Fowler99], to determine areas of code that need refactoring, may make sense to a software developer, it is hardly quantifiable. A more reliable and quantifiable method of detection needs to be found.

Process

The process for this research was the following:

- Select refactorings – Establish a criteria for selecting refactorings, select the refactoring, and analyze them
- Select software metrics – Software complexity metrics were analyzed and an appropriate set of software complexity metrics selected
- Run metrics before and after refactoring – Analyze the refactorings with the selected set of software metrics; compare the source code before and after refactoring using existing software complexity metrics
- Analyze the metrics – Analyze the results of the software metrics
- Perform factor analysis – Use factor analysis to reduce the number of software metrics
- Analyze the factor analysis results

Refactoring

Background

"Refactoring is a technique to improve the quality of existing code" [Fowler99]. The process of refactoring is to improve the readability and testability of a component without changing the inherent functionality and to add tests, e.g., unit tests, to the code. Refactoring is done in small steps: modify the code, test the code, and repeat. It is not a radically new and different concept. Historically, most maintenance programmers and their managers have had the dilemma of whether to rewrite existing code or modify it one more time. The process of refactoring does provide a well-ordered procedure to make these changes and emphasizes the need for testing.

The Metrics of Refactoring

As code is maintained and modified over time, it has the tendency to become 'brittle'; error-prone and difficult to maintain. The developer has the choice to rebuild the code or replace the code. Refactoring is another option that allows the rebuilding process to include insertion of small-localized tests called unit tests. An important part of refactoring is the addition of these tests to the code. Unit tests are generally small in-line tests that are written before the source code is developed or modified. When the test runs correctly the development or modification is complete. The emphasis here is to clarify and define the expected output early before coding takes place.

Refactoring can be done when the existing code is cumbersome to work with or is difficult to understand. The purpose of refactoring is to increase the comprehensibility of the source code, to make code easier to understand and modify without changing the functionality

Rules of Refactoring

Some rules for refactoring:

- Do not make refactoring corrections and add functionality at the same time.
- Have good tests before beginning to refactor. Run the tests as frequently as possible, at least after each change.
- Make all changes in short deliberate steps. The process of refactoring involves making many small-localized changes that result in a higher level change.

Selection of Refactorings

The refactorings used in this research were taken from Martin Fowler's book, Refactoring: Improving the Design of Existing Code [Fowler99].

The criteria for selection of refactorings was the following:

- Choose representative refactoring code segments that contain sufficient code size.
- When more than one example was given in the book, the larger or more complex example was chosen for this project.
- Choose refactoring code segments that are independent and not intermediary, e.g., refactoring code segments that do not require the use of additional refactoring to complete or to start. For example, the *Extract Class* refactoring starts with a simple person class, creates a new class to accept a portion of the responsibility, makes a link between the new and old classes, uses the refactoring *Move Field*, and then uses the refactoring *Move Method*. If multiple refactorings are used in the refactoring process, the determination of the cause of the changes or the changes made will be questionable.

The Metrics of Refactoring

- Only one pair of source code, before and after, was used for each refactoring.
- The intermediary changes in the refactoring process were not considered in this project. The source code was not modified from what is shown in the refactoring book [Fowler99], i.e., no changes were made to make the code segments complete or correct.
- A complete set of tests is a requirement for refactoring. The tests necessary to test the source code were not analyzed and were not included in this research.

Of the 72 refactorings in the refactoring book [Fowler99], 32 refactorings were selected. The refactorings chosen are shown below:

- Change Unidirectional Association to Bi-directional
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Decompose Conditional
- Encapsulate Downcast
- Encapsulate Field
- Extract Interface
- Extract Method
- Hide Delegate
- Inline Method
- Inline Temp
- Introduce Assertion
- Introduce Explaining Variable
- Introduce Foreign Method
- Move Method
- Parameterize Method
- Preserve Whole Object
- Pull Up Constructor Body
- Remove Assignments to Parameters
- Remove Control Flag
- Replace Array with Object
- Replace Constructor with Factory Method

The Metrics of Refactoring

- Replace Error Code with Exception
- Replace Exception with Test
- Replace Magic Number with Symbolic Constant
- Replace Nested Conditional with Guard Clauses
- Replace Parameter with Explicit Methods
- Replace Parameter with Method
- Replace Type Code with State/Strategy
- Self Encapsulate Field
- Split Temporary Variable
- Substitute Algorithm

Software Metrics

Goal of Software Metrics

The goal of software metrics is to provide a measure of the complexity of a program. Code that is more complex is likely to be more error-prone, both to develop and to maintain. For example, when driving, the speed of the car is a gauge or metric as to how safely the car is being driven. A 'safe and prudent speed' is an expert judgment because of the many factors for safe driving: visibility, road conditions, precipitation, condition of the car, the driver's physical condition, the driver's experience, and many more conditions and characteristics. As such, 'safe and prudent speed' is a summation or combination of many conditions. In the same way software metrics measure different attributes of the complexity of the source code. No single software complexity metric defines the complexity of a program.

A detailed reference of software complexity metrics can be found in [Conte86, Chidamber94, Lake94, Lorenz94, and Lake99].

Software Metrics Selected

Traditional software metrics are widely used and well understood by researchers and developers, and are available from earlier procedural programming languages. These metrics have been validated for use with object-oriented languages [Chidamber94, Lake94, Lorenz94, and Lake99].

The metrics used in this project include a selection of the Object-oriented Programming (OOP) complexity metrics and traditional procedural complexity metrics. These metrics were selected from more than 125 metrics.

The Metrics of Refactoring

Measure	Description
# Data variables	Number of data variables
Class internals	Instance variable usage Function-oriented code Comment lines per method Percentages of commented methods
Class size	Number of public instance methods Number of instance methods Number of instance variables Number of class methods Number of class variables Lines of code in a class Number of classes
LOC	Number of lines of code Number of source lines of code (SLOC) Number of comment LOC
Method size	Lines of code in each method
Operators/operands	Halstead's counts (η_1, η_2, N_1, N_2, N, V, E)
v(G)	Cyclomatic complexity number

Table 1: Selected Software Metrics

Reducing The Number Of Metrics

The large number of software metrics used in this research facilitated the research by providing a broad range of areas of analysis, but also complicated the task of determining what aspects of the software can be evaluated. With such a large number of software metrics to choose from, it is difficult to discover which metrics are effective measures of software complexity for the sample source code. The number of metrics must be reduced to a manageable and useful number. Reducing the number of software metrics makes collecting the information easier and improves the understanding the importance of specific measures.

Grouping Metrics

One method of reducing the number of software metrics is to group them. This is done by defining some property and grouping all of the metrics that share that property in the same group. However, this grouping is subjective since it

depends on what properties seem significant to the person defining the taxonomy, and grouping may also cause important factors to be overlooked. This type of grouping depends upon a number of factors, such as the expertise and experience of the developer, the development language, and the development environment.

Factor Analysis

A more objective method of grouping metrics is the statistical technique of factor analysis. Factor Analysis is a generic name given to a class of multivariate statistical methods whose primary purpose is to reduce and summarize data. Factor Analysis is a technique particularly suitable for analyzing complex, multidimensional problems. Factor analysis is a statistical technique for grouping together items that measure the same factor. Items in the same group are highly correlative, but have relatively small correlation with items in different groups. Using this method each derived group of metrics can be considered as a correlated complexity factor that provides the same complexity information as all of the other metrics.

Results

The most significant factors for all of the systems are shown below:

Data input:	observations
Number of complete cases:	32
Missing value treatment:	listwise
Standardized:	yes
Type of factoring:	principal components
Number of factors extracted:	3

Factor Number	Factor	Eigenvalue	Percent of Variance	Cumulative Percentage
1	LOC	9.06709	64.765	64.765
2	# methods	1.44395	10.314	75.079
3	# variables	1.39391	9.956	85.035
4	# classes	0.97383	6.956	91.991
5	Class size (LOC)	0.48499	3.464	95.456

Table 2: Results of Top 5 Factors

The Metrics of Refactoring

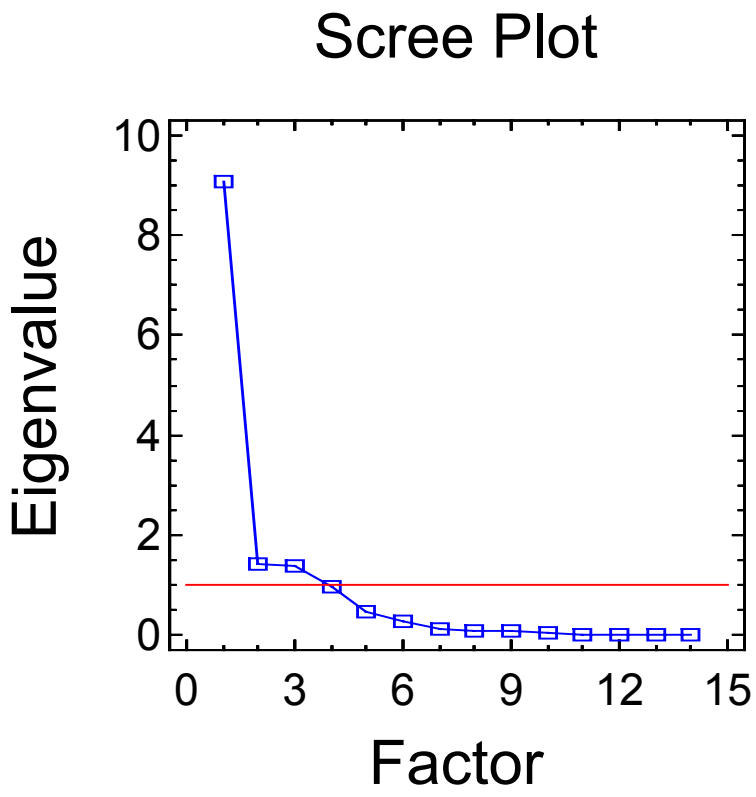


Figure 1: Scree Plot

The Scree Plot displays the number of factors that will be extracted from an analysis. The Minimum Eigenvalue option shown on the Scree plot is a horizontal line drawn on the graph. This plot shows the eigenvalues for the first 14 factors. The Minimum Eigenvalue for this plot is 1.0, which was the value used to decide on extracting 3 factors. The eigenvalues are proportional to the percent of the variability in the data attributable to the factors.

The purpose of the factor analysis is to obtain a small number of factors, which account for most of the variability in these cases of refactoring. In this case, 3 factors have been extracted, which account for 85.035% of the variability in the original data. The principal components method was selected, so the initial communality estimates have been set to assume that all of the variability in the data is due to common factors.

The results shown above demonstrate that 3 metrics can account for 85 percent of the variability of the original data from 32 cases and 24 metrics.

The Metrics of Refactoring

Conclusion

Refactoring is supposed to make the code easier to work with. Generally, these types of refactoring are necessary because of the following reasons:

- the design of the application was incomplete or was modified,
- the developer implemented the code in a manner which would require change, either sloppy coding or a very complex implementation, or
- the code has been highly modified and become brittle.

Actually some of these reasons could be combined in the same source code file.

In the cases shown the refactoring performed one of the following functions:

- Hide information within the inheritance tree, i.e., fixed an inheritance problem
- Extracted or replaced a common element so that it could be used easier
- Combined or re-built an expression, such as modifying an expression so that the expression is easier for the developer to use.

Previously there has been no relationship shown between software complexity metrics and refactoring. This research has shown that the changes made in refactoring can be represented by a small number of metrics.

References

[Chidamber94] Chidamber, Shyam and Chris Kemerer, "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, June 1994, pp. 476-492.

[Conte86] Conte, Samuel, Hubert Dunsmore, and Vincent Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings, Menlo Park, CA., 1986.

[Fowler99] Fowler, Martin. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Co., Inc, Reading, MA, 3rd printing November 1999.

[Halstead77] Halstead, Maurice, *Elements of Software Science*, Elseview North-Holland, 1977.

[Lake92] Lake, Al, and Curtis Cook, “A Software Complexity Metric for C++”, Proceedings of the 4th Annual Oregon Workshop on Software Metrics, March 1992.

[Lake94] Lake, Al, "Use of Factor Analysis to Develop OOP Software Complexity Metrics", Proceedings of the 6th Annual Oregon Workshop on Software Metrics, April 1994.

[Lake99] Lake, Al, “Factor Analysis of Metrics”, Trends in Software Engineering Process Management (TSEPM), November 1999. http://www.costxpert.com/tsepm/nov99/art3.htm

[Lorenz94] Lorenz, Mark, and Jeff Kidd, *Object-Oriented Metrics: A Practical Guide*, Prentice-Hall Object-Oriented Series, 1994.

[McCabe76] McCabe, Thomas, *A Complexity Measure*, IEEE Transactions on Software Engineering, December 1976.

Mighty Morphing Benchmark Power:

Ensuring Reliability, Performance, and Scalability

Gregory Baran
Metrica Labs, Inc.
Portland, Oregon 97225

Abstract

Metrica Labs has developed a strategy to increase significantly the degree of assurance that a computer software and hardware system will stand up to the stressful conditions confronted in the field. In the real world, software and hardware from perhaps dozens of vendors is often combined with custom interfaces and applications to form an enterprise system. The possibilities for subsystem interaction are huge, with potentially expensive consequences.

Metrica proposes an approach to the problem of meaningfully sampling the space of software and hardware configuration and operational interactions. A scientific viewpoint is taken to defining the structure, content, and use of specific micro-benchmarks. With this structure guiding the design of the tests, the results of command timing can be very effectively analyzed statistically to determine the influence of each factor, both individually and in combination. This is the heart of the ability to discern both pronounced and subtle interactions between factors, such as choice of applications, pattern of use, or level of utilization.

The paper concludes with a detailed report on the application of this approach to file system and storage performance.

Author

Gregory Baran is President of Metrica Labs, Inc., a consulting company that for seven years has specialized in producing models and simulations to guide computer system architecture, and in developing methodologies to optimize the performance and reliability of existing systems.

1 Anticipating Customers' Needs

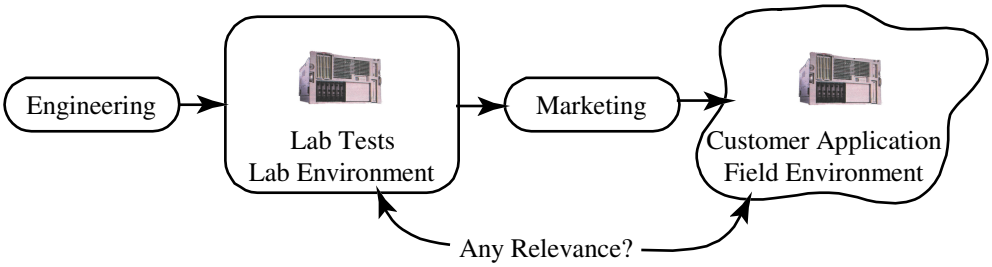
1.1 MORPHING?

No, it is not good enough to use one benchmark, however strenuous, to test a system and call it done. Popular benchmarks like TPC-C[1] or WebBench[2] present a workload to the target system that imitates a business scenario. By design, they exercise only those operational patterns excited by the imitated application. While useful as marketing tools, single-scenario benchmarks do not assure a vendor or a customer that the system can and will perform real-world tasks well.

Figure 1 shows the heart of the problem. Vendors test products in controlled lab environments. Yet when customers use a product, their applications and environment will differ from the lab setup in unpredictable ways. The problem is worst for general purpose systems like servers and least for embedded systems. Engineering holds the responsibility to minimize unpleasant surprises for customers when they do the unexpected.

How, then, can designers prepare for the unexpected? Shareholders and customers, and therefore marketing, engineering, and sales, will settle only for a totally successful program that ensures reliability, performance, and scalability.

Figure 1: The Players



Bending test programs to cover a much more meaningful set of conditions and request patterns than standard benchmarks cover is called *morphing*. Through effective testing, measurement, analysis, and modelling, this can be done, though it sounds unrealistic and expensive.

How can a staff cover enough ground to be confident in the robustness of a system's quality, yet finish in a reasonable amount of time for reasonable expense?

1.2 WHAT DO CUSTOMERS NEED?

Every successful company must meet the needs and satisfy the desires of customers to make sales. What do customers want to hear about the computer products that you want them to buy? Customers need a variety of information both in for evaluating a product before purchase and after purchase. The first column of Table 2 lists four types of information that customers need. The following sections review the meaning of that information, with comments on the advantages and disadvantages of methods that generate it.

---

[1] TPC-C is produced by the Transaction Processing Performance Council (*www.tpc.org*)

[2] Ziff-Davis Benchmark Operation produces WebBench (*www.zdnet.com*)

1.2.1 A-B Comparison on Standard Applications

A *Standard Business Operation Benchmark* has one primary purpose: to provide a yardstick by which to measure systems or components. Industry magazines or trade organizations generate benchmarks for a small group of prototypical applications. Examples include TPC-C, SPEC[3], or WebBench. The outcomes are often promoted by the marketing departments of the winners.

Standard benchmark results interest customers who feel the benchmark reasonably represents their requirements. The result is typically one number, such as "transactions per second" or "floating-point operations per second". Few companies use these benchmarks as stimuli for deeper understanding of their product's capacities. Because of its perceived value to marketing, many companies work hard to tune their systems for the standard benchmarks. The relevance of the results to any particular customer remains in doubt, however. *Caveat emptor!*

In contrast, the approach to custom benchmarking presented here lays the foundation for the producer to anticipate a wide variety of behavior in the field. Anticipation is important since the range of uses required by customers has to be assumed to be very broad. The custom benchmarks exercise the system in bite-size pieces that customers can more easily relate to their own situations. Quality results are obtained through the systematic analysis of the measured results.

1.2.2 Performance on Customer's Own Application

A number of commercially available systems simulate hundreds or thousands of users for standard client-server or Web environments. These tools exercise a real application with realistic requests. The relevance of such direct manipulation for system assessment is clear (answering the question posed in Figure 1).

Unfortunately, like the standard benchmarks, typical reported output consists only of a single graph of throughput against the number of simulated users. Without an accompanying analysis of the stress and response of the environment to the applied loads, the simple curve has little value to guide insight into system behavior. Something else has to be done, which is where the approach described here comes in.

These methods include the custom crafting of benchmarks or test programs that stimulate a system in complex but controlled ways. Along with quantitative measurements of the stresses on the supporting system, a wide range of customers can estimate their own application's performance prior to installation using these modelling techniques. The producer benefits from having a better and more complete story to present to customers quantitatively showing the capabilities of the system.

Live system measurements of system throughput and response time show what has actually been achieved in the field. By the time a system is deployed, though, it is too late to ask whether the system will work adequately since full commitment is already done. To be useful, assurance must precede commitment.

1.2.3 Reliability on Customer's own Applications

How reliably will a product run the customer's own applications? Is the product adaptable to a wide range of circumstances, including the customer's? This requires robust behavior, enhanced by uncovering and repairing vulnerabilities, of the system under the broadest possible range of conditions. Normal benchmark programs emulating business practices cannot hope to test a range of behaviors outside of their particular regimes.

---

[3] The Standard Performance Evaluation Corporation produces the SPEC benchmarks (*www.spec.org*)

The benchmark creation methodology described here casts a much wider net than normal benchmarks. In testing, close observation of the results of one iteration of Metrica's custom benchmarks can guide the generation of a second generation of benchmarks. Refinement of the benchmarks relentlessly homes in on just the right combination of conditions that can cause instability or failure. A system surviving this stressful process has thus been demonstrated to be robust under a broad range of *documented* conditions.

Indeed, a live system can in principle be tracked in operation to ensure that it stays within the performance parameters that have actually been tested. Should the system wander outside of the high confidence regime of behavior, the excursion can be automatically recognized and noted with alarm, if desired.

1.2.4 Capacity Planning and Scaling

What configuration will deliver the desired performance levels for the customer's operational application? How does the product cope with increasing service demands? What paths for expansion are available to support growth? How much headroom is available to serve peak demand?

Operating system and middleware counters form the foundation of most of the quantitative analysis that can be performed on a system. These counters and rate reports monitor both the stress induced on the system being studied, and in some cases, monitor the amount of useful work being done.

For example, the operating system counters might report disk and network activity along with CPU usage, while middleware software reports the number of eCommerce transactions completed over the same time interval. If necessary, software-based diagnostics can be supplemented with hardware monitoring of network or bus activity.

The modelling technique proposed here uses a combination of the custom benchmark results and the counters and stress indicators to compute the scaling capabilities of a design. With these techniques, it is also possible to estimate the benefits that will be realized by optimization or architectural changes, enabling them to be weighed against the development and testing costs.

Table 2: How this approach supplies better quality critical information needed by the customer

Customer Need	*Conventional Testing Methods and Diagnostics*			*This Method*
	Standard Business Operation Benchmark	Drive Customer's Applications in Lab	Live System Monitoring of Customer's Application	*Custom Benchmarks + Counters + Queuing Model*
A-B Comparison on Standard Applications	basic purpose of standard benchmarks			*make custom or (new) standard benchmark*
Performance on Customer's Apps		ok but only over the range of actual load that was run	measure actual performance achieved	*predict, and validate, using this method, extrapolate to range beyond that tested*
Reliability on Customer's Apps		learn something only if system crashes during testing	live system directly reveals reliability, but when it's too late (e.g. NYSE)	*use clues from stresses to expose vulnerabilities*
Capacity Planning and Scaling			live system measurements validate predictions	*queuing model essential for this*

2 The Method

2.1 Meeting Customer Needs

Naturally, customers see things from their own point of view. The problem for the vendor is to provide for all customers' needs without knowing for sure what they are.

2.1.1 Importance of Broad Coverage

The pace of technological development forces many seasoned computer product customers to employ whatever means are at their disposal to accomplish their goals. New applications of necessity are introduced on existing systems and components. Usually, the existing systems were not designed specifically to run the new applications. New applications, databases, communications technologies and all the rest have to run right here, right now.

Under these uncompromising conditions, many computer product customers, reasonably enough, are a little like hungry eleven-year-olds that cradle stuck peanut butter jars while searching for a nut cracker or maybe even a baseball bat. The pressure to perform creates tremendous problems for both the producers and consumers of products.

This report presents a way to make sure that any product will work for the most desperate and creative users. The right understanding of the problem and its solution can ensure maximum product acceptance. Morphologically driven experimental design produces right understanding.

2.1.2 The Morphological Approach

Morphology is a branch of biology that deals with the form and structure of organisms. How do you make sense of nature in all of its variety? The result of a morphological analysis is the categorization of biological organisms into a coherent structure with family, structural, and evolutionary relationships laid out clearly. It is a diagnostic approach.

System analysis combines diagnostic with synthetic reasoning. It is diagnostic because a system's design dictates the external interface and internal state. Imagine that there is a set of parameters known that describes the interface and the internal state. This set of parameters can be exploited to drive the design, in turn, of a broad range of effective, efficient test programs or benchmarks. The process of analyzing the interface, devising benchmarks, and analyzing results is the essence of the method.

Suppose the system being analyzed is a software library, consisting of a finite number of functions with a finite number of parameters each. These are the controls available to a programmer using the library, through the application programming interface, or API. Internal to the system exercised by this library will be some state that arises from the history of operations performed along with the state of the environment. The *morphological space* of that interface is the set of all operations, arguments, and state that can occur, in all combinations.

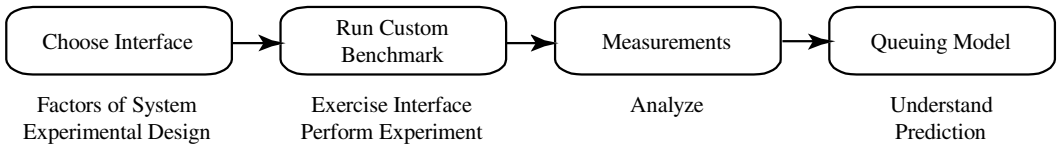
Later in this paper, a short report of a target system study is presented. The lab work exercised a limited API. It used only the basic file operations of open, close, read, write, seek, create, and remove. The underlying state consisted of the file system, buffer caches, and operational environment of the system. Even for this interface the combinatorial possibilities of operations and state are vast.

Using this efficient approach, you can produce a top quality product, don't need settle for less than near-complete coverage, and can keep within budget.

2.2 THE COORDINATED TOOLBOX

How do you navigate an infinite space of possibilities without getting lost?

Figure 3: Comprehensive Morphological Analysis -
Thorough tests, precise measurement, complete analysis, and accurate predictions



2.2.1 Overview

Figure 3 shows the highest level of organization of the approach described here. As discussed above, the first step is to choose an interface to study. A single interface can be handled easily, as well as multiple interfaces. Once the interface has been chosen, a set of key factors are listed, and the levels for those defined. A factor is a variable in the experiment. The factor's levels are the values the factor assumes the experiment. In some experiments, all combinations of factors and levels are run independently. In others, a subset is run. In the report presented below, the benchmark exercised the file I/O interface using the factors listed in Table 6.

The *Custom Benchmark* exercises the interface to the system according to the patterns and specifications given by the experimental design. The chosen factors directly dictate requirements of the custom benchmark program. To obtain maximum benefit, the method requires the recording of detailed *Measurements* of the system under test, and the benchmark's progress.

A wide range of statistical procedures then bear down on the collected data as part of the analysis. Some of the results obtained at this point are the certain identification of what does and what does not affect the results of the benchmark. When these results are compared between competing systems, differences in approach, and strengths and weaknesses are almost always identified.

Finally, as a check on the analyst's understanding of how the important pieces in a system work together, a *Queuing Model* is generated based on the measured data and analysis. The output can be used for capacity planning, assessment of the impact of changes, and identification of bottlenecks.

The same methodology also can systematically expose fatal design or implementation errors. The statistical techniques of analysis are very sensitive to small changes excited by experimental factors. By reorganizing the experiment in a second or third round of benchmarking as guided by the results of preceding rounds, bad behavior can often be amplified causing crashes. Scripting and extensive measurements renders these results repeatable and understandable.

The morphing method makes sure that everyone involved in a product keeps their attention focused on system issues. How often have either hardware or software developers claimed great things for their contributions, or warned of dire happenings should an unfavored proposal be implemented? This method makes sure that the contributions are measurable and accountable: No more science fiction.

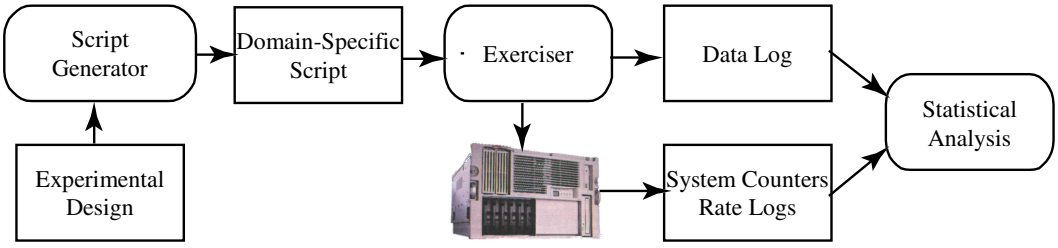
2.2.2 Detailed Walk-Through

To understand the key steps in the method, it will be necessary to expand the components summarized in Figure 3 in more detail, as presented in Figure 4.

The first step is in lower left: the *Experimental Design* box.

The *Experimental Design* box represents the particular choices of factors (the variables) and chosen levels (the settings) handled by the *Script Generator* program specific to the domain being studied. Each application domain requires its own script generator and its own definition of factors and levels suitable to it. Indeed, in the disk I/O tools, many more factors exist that can be handled by the *Script Generator* (listed in Table 6*)*. Their levels were held constant here to avoid complications in presentation. A real study tackles the effects of all factors.

Figure 4: Systematic Benchmarking Method



The *Exerciser* executes the script or scripts. The experimental design can optionally demand that multiple scripts run in parallel. In the present local server I/O study, for example, thirty-two parallel threads executed simultaneously during the most complicated experimental runs. The Exerciser program itself consists of a common part that drives the overall script execution, and a core domain-specific part appropriate to the targeted API. The *Data Log,* as maintained by the *Exerciser,* records initiation time and response time for every command.

Statistical analysis of logged timing data reveals immediately the strengths and weaknesses of a system. If the same scripts were run without the knowledge that they represent one case from the design of experiments, the meaning would be obscure. There would only be the isolated fact that a particular I/O at a certain address completed in *z* milliseconds. But analyzed in the context of the experimental design's pattern, the trends of a system's response as a function of the experimental factors proclaims itself clearly. This enhances the system level understanding while respecting the effect of low-level mechanisms.

2.2.3 Mechanisms: Discovering How? and Why?

In the background, while the scripts are executing, the operating system reports its vital signs in a separate log of system diagnostics. The Perfmon library provides performance information for Windows 2000. On Linux, the traditional UNIX vmstat tool was used. In more complex situations, service-specific metrics can also be recorded.

The system counters critically support two extremely important results.

System behavior as shown by counter and rate data is very eloquent in speaking about how a system responds to the stresses placed on it. Given enough counter types, such as that provided by Windows, much is discovered about the mechanisms employed by the system for each experimental run. These clues are critical to competitive analysis.

For example, perhaps a competitor's system needs fewer I/O interrupts or context switches than yours. Does that explain how they accomplish better performance, say? Does context switching always correlate negatively with performance? Cause and affect can frequently be inferred. This can be established, usually, *without* any special instrumentation of the system under test, given good enough experimental design and good enough system counters. For example, a CORBA layer might record its own vital signs. The internal CPU metrics reported for

Intel processors (say using Vtune) may be crucial. All such information is welcome provided its collection does not interfere too much with benchmark operation.

This is a huge benefit. You can learn the costs of each high level operation without complex instrumentation of application software. Measure the amount of useful work done along with the system stress level. In most cases, detailed accounting of the amount of work needed to support progress can be estimated without direct instrumentation of middle-level systems, as demanded by some competing methods.

This is very different from running a benchmark and reporting a single numerical result alone such as transactions per second. As shown below, a system may exhibit acceptable average performance, yet be performing extremely badly for a significant fraction of cases. The outlying cases are important clues to the presence of systemic problems. If left untreated, a vulnerability of the system will be overlooked that may have very bad consequences in the field.

2.3 MODELLING

2.3.1 Scaling the System

Some form of performance counters is also key to driving a scaling study. Typically, for relatively high-level studies, a small group of critical resources determine a system's responsiveness: CPU, memory, I/O buses, storage devices, I/O ports, etc. The counters or rate indicators measure the amount of load placed on the system by each request. Queuing model techniques can extrapolate these behaviors to levels that were not tested directly. Configurations can be examined in a what-if study, say to estimate what results when adding paths, or increasing the number of key components.

The model results stand on the measurements. The result is a very accurate estimate of the limits and capabilities of a design.

2.3.2 Model Making and the Scientific Method

The essence of a scientific investigation is to be able to predict what will happen given a system and a set of input parameter values. The theory prescribes how to transform input values into output values, thus making a prediction. A theory is a model of reality. Therefore, a useful theory makes a prediction that can be checked against an experimental result. When the measured outputs agree with the predicted outputs, the theory has been successful in describing reality. When the results and predictions are inconsistent, something is wrong. Then, to successfully reproduce the measured results, either revise the theory or the measurements.

For example, the flight path of a baseball, once it leaves a bat, can be computed given its initial position and velocity using Newton's laws. One testable result may be a prediction of the place of impact, say Section C29, Row W, Seat 7. If an observed ball hits ten rows short and one section to the left of the predicted location, however, it suggests that something has not been taken into account. Perhaps the initial velocity was measured inaccurately. Or a fancier, and probably more accurate, model would include the effects of wind resistance, spin on the ball, and atmospheric density and humidity.

The models are important for closing the loop on system understanding. They are the only way to perform meaningful capacity planning and to locate bottlenecks.

2.4 HOW MORPHING DELIVERS WHAT THE VENDOR NEEDS

In this section, three key techniques are introduced that directly enhance the quality and competitiveness of a product. Each method will be discussed separately and its value established. The approach described can be delivered by a select range of tools and methods (Table 5).

Table 5: Support for Ensuring Product Quality

What Vendors Need to Do to Supply Customers the Right Data	The Morphing Method: The Toolbox		
	Custom Benchmark Design and Execution	*Statistical Analysis of Measurements*	*Queuing Models*
Deep analysis of competition in A-B comparisons	chooses the playing field for the comparison	discover causes for differences	immediately expose bottlenecks and estimate limiting behavior, with causes
Marketing Message: Custom Benchmark with meaning	use a custom benchmark as standard for all!	knowledge of causes could drive marketing message	extrapolate results to future hardware, software
Minimize subsystem interaction vulnerability	broad, deep, efficient coverage of multiple interfaces, minimize vulnerability	statistics can tell you how near the edge of reliability a test has come	
Minimize bottlenecks		service times for critical resources aid analysis of bottlenecks	with good instrumentation and analysis, identify bottlenecks
Capacity Planning and Scaling tool	good unit tests with interactions forms a basis for extrapolation to any application	system throughput and service times forms basis for capacity planning	only way to perform reliable capacity planning and scaling analysis, not point benchmarks!

2.4.1 Custom Benchmark Design and Execution

The choice of interfaces to exercise and their combination defines the ground for system comparison. Since it is so inexpensive to define new benchmarks, and the value of variety is great, why not produce a set for any occasion? Benchmarks for stress testing will exercise more combinations of patterns in less obvious ways than other benchmarks will. It may even be desirable to promote a suite of custom benchmarks as a standard of comparison within an industry segment.

Take the example of Network Appliance Corporation's *Postmark* benchmark. Postmark simulates a mail distribution environment. Its method is the creation, writing, closing, opening, reading, and deletion of numerous files in rapid succession. Such a benchmark is trivial to generate for any environment using the experimental design and script generation techniques discussed above.

By combining multiple interfaces and patterns of access, it is easy to execute broad, deep, and efficient coverage of multiple interfaces. Careful statistical analysis of the results minimizes vulnerability to unintended interactions and high-order effects, long before the customer ever obtains a product.

And, since even large applications are made up of a large set of smaller operations, good unit tests with interactions form a basis for extrapolation to any application. All you need to know is the mix of these operations that occur in the larger application and instantly estimates of its performance are available. An application whose performance can be estimated this way is TPC-C.

2.4.2 Statistical Analysis of Measurements

Statistical analysis of the relationship of responsiveness to experimental factors shows what's important. You know what causes observed differences: hastily convened panels of experts are

usually unnecessary. Also, a deep, reliable knowledge of the competition's limitations could become the basis of an informed marketing message, or open an approach for seizing a competitive advantage.

Statistics can also tell you when a system is becoming unstable. Perhaps an average response is still being achieved, but some underlying resource is becoming exhausted, or extreme response time values begin to appear. These are causes for some level of alarm, and need to be investigated.

Additionally, measuring the service times for critical resources quite directly identifies the existence of bottlenecks and pegs the ultimate system throughput limitations. System throughput and service times, in turn, forms the basis for any believable capacity planning computations.

2.4.3 Queuing Models

The combination of a simple queuing model and service time measurements immediately exposes bottlenecks. It also provides estimates for limiting behavior, along with the reasons behind it. Since no special proprietary knowledge is required in most cases, it can be performed both for your own system and for the competition's.

Models are the only way to perform reliable capacity planning and scaling analysis. Normal scaling curves from most commercial test software suites are totally inadequate.

As a bonus, the model extrapolates results to future hardware and software of unbuilt systems.

2.5 Publicizing Quality and Performance

Every competitive computer product, from the largest enterprise installation to the most humble embedded controller, must perform well and reliably. Ideally, customers, management, and engineering share access to hard evidence of these qualities.

The ideal has not often been achieved. An extensive survey of computer storage, server, and networking product manufacturers has revealed a lack of performance or robustness data that is available to customers (Metrica Labs commissioned study). Potential reasons for this include a lack of appreciation of its importance, a feeling that it is too expensive and time consuming to generate, or lack of awareness of helpful resources.

3 Sample Investigation: Server Local Storage Access

The sample investigation studied a disk I/O benchmark suite running on a dual-processor Pentium III server with one IDE system disk and four SCSI 160 hard drives. Within this setup, four factors were varied having two to four levels each:

- *Operating system*: Linux kernel version 2.2.18, Linux kernel version 2.4, Windows 2000
- *Access organization*: Random, Sequential
- *Percentage reads vs. writes*: 0%, 100% (Seq) or 20%, 80% (Random)
- *I/O request size:* 8 KB, 64 KB
- *Threads competing for each SCSI disk*: 1, 8

for a total of sixteen individual runs on each of three systems.

3.1 Linux vs. Windows 2000 on an Intel-powered PC Server

The trade press reports that Linux is a growing presence in Web servers. This paper introduces one type of work that can be performed within the methodology presented.

The sample application described here was an in-depth study of the disk storage subsystems of Linux and Windows, specially performed for this report. The results of an extensive lab

investigation into the quality of storage I/O subsystems on a medium size departmental server running Windows 2000, Linux kernel 2.2.18, and Linux kernel 2.4, respectively, are presented there.

How do the three systems compare in performance? Which is more efficient? Which has more headroom? Since two versions of the Linux kernel were studied, the differences introduced by the new Linux kernel will be measured.

Did you know that:

- Windows 2000 uses less average CPU power to perform file system I/Os than either Linux kernel version tested (2.2.18 or 2.4) (Table 13)
- Yet, Linux completes almost all file system writes *much* faster than Windows 2000 (Figure 10)
- Linux's CPU time for file system I/O is primarily spent performing context switches, based on statistical correlations.
- Performing certain common patterns of file system I/O, all three systems frequently spent over 1 second completing a single I/O. Linux 2.4 had numerous commands taking as much as 6.6 seconds to complete! While queuing delays can explain the 1 second response times, anything much longer is certainly a symptom of system software meltdown. This behavior should be investigated and eliminated (Figure 12).
- Do not ever request 8 KB reads or writes under Windows if you care about performance (Tables 8 and 11).
- To perform an effective tuning of a Linux system, however, requires much better information than the few tidbits that Linux provides. Linux provides perhaps 10% of the useful information reported by Windows 2000.
- The bottom line: Both Windows 2000 and Linux 2.2.18 are reasonable systems to use as file servers. Linux 2.4 is not, yet, on this system.

The main reason for discussing these items in technical detail is to show what can be obtained without detailed design information through the morphing methodology. Since an understanding of the mechanisms that produce these results is not required to establish the value of the method, the underlying mechanics will not be presented. Yet, an architect or developer of these OSs may well be troubled by these findings. Evaluate these conclusions in the context of the hardware configuration used for testing.

The methodology described above describes the steps taken in any investigation. Here, three families of tests were run:

- One requesting thread, one disk, run through the file system of the host O/S
- One requesting thread per disk, four disks run in parallel, access through the file system, to test the robustness of the SCSI subsystem primarily, and the OS's ability to handle light multitasking
- Eight requesting threads per disk, four disks run in parallel, access through the file system, to measure the extent to which increased multitasking degrades CPU, memory, and bus performance, beyond simple queuing effects

The factors exercised in this work are just a small subset of those supported by the experimental design / exerciser combination. Here is the complete set currently supported:

Table 6: Supported file access experimental design factors

Quantity	*Description*
I/O Block Size	I/O block size requested, not necessarily constant during a run
Seek Address Delta	Formula for adding noise to the starting address of an I/O, say, off by 1 or 2 bytes plus or minus
I/O Type	Can be mixed for reads and writes, read, or write
I/O Mix Reads	Proportion of reads in the mix
Direction	For sequential access, forward or backwards
Seek Pattern	Currently random or sequential, though
Distribution	Sample from this distribution for generating seek addresses
Concurrent Target Count	How many target files / devices being accessed concurrently within a script
Total File Size	Size of the target file
Seek Offset Origin	Where to start sequential access
Target Type	File, or raw disk partition, etc.
In Cache	Should the contents of a file be pre-loaded in buffer memory, or not
File System Block	Size of file system block when formatted
Volume Manager	Is there a disk volume manager in use?
Pre-existing	Does the file(s) exist before running the benchmark, or not
Delete-On-Close	Remove the file(s) when a file is closed, or not

The system used to run the experiment was chosen to be a representative contemporary server of average capability for a small department or company:

- Dual 800 MHz Intel Pentium III processors
- 512 MB of 133 MHz SDRAM
- Intel STL2 motherboard with two built-in Adaptec SCSI 160 channels
- Four Seagate 10 000 RPM 9.1 GB SCSI drives with 4MB cache RAM, on a single SCSI chain
- Single IDE hard drive for system hosting

The systems were run with both processors enabled at all times. Complete with a SuperMicro case, extra fans, etc., the hardware cost only $2,700 when it was acquired in February, 2001, about the price of a midrange laptop!

The software versions being run were:

- SuSe Linux Release 7.1 with 2.2.18 kernel
- SuSe Linux Release 7.1 with 2.4 kernel
- Windows 2000 without service packs applied

3.2 Performance Timing Results: Who Does What Well?

3.2.1 Single Disk, Single Thread

A single SCSI disk was hit with twenty-four different patterns of access: random, sequential forward, sequential backward, large and small block, reads, writes, reads and writes, etc. Each of the 24 patterns attacked a different 64 MB file, so caching effects only came into play when caused by the benchmark itself.

While this data can form the foundation of a detailed study of operating system - driver - I/O bus - SCSI - disk drive interaction, brevity requires that only a quick summary of the highlights be presented. The multi threaded data presented below is more relevant to evaluating servers.

The main points are:

- Windows 2000 is much worse at writes than Linux: It is a factor of over 210 times slower for 8K block sequential writes than either Linux version. It takes about 1.5 disk revolutions per write, while Linux (45 μsec per I/O) just caches the data and writes it back later. Linux believes it's all right to write to a disk buffer without forcing it to disk, even in volatile RAM!
- All systems have large intervals of inactivity, seemingly at random times
- Disk revolutions are missed very frequently (any response time over 6 ms) on all three systems

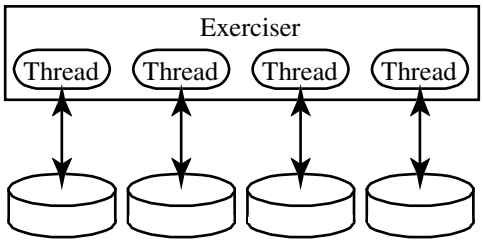
A further statistical analysis of the read command results of the single disk access shows that for all three systems, the factors apart from read vs. write that have significance are the direction of access, whether forward, backward, or random, and the percentage of reads (more reads means higher performance). I/O block size does not have a statistically significant effect on read performance, probably because the performance is dominated by disk seek time.

On writes, only direction makes a difference for Linux (backward is best while forward is worst!). Windows 2000 behaves completely different for writes: Much better performance for 64 KB blocks than 8 KB, and best performance for 20% reads, worst for 100% writes and worse for 80% reads.

3.2.2 Four Disks, Single Thread

The setup for the multi-threaded disk access experiments was as follows. Each thread executed a script that accessed a single disk, with four operating in parallel (Figure 7).

Figure 7: Configuration of four disks, one thread accessing each



The first result to look at is the response times for the individual commands. Table 8 summarizes the results for the eight different script types run. Timings are provided separately for reads and writes on successive lines where appropriate. The data in this table summarizes 2000 individual I/O calls per thread, or 8 000 total for all four threads. Times under each OS column, the last three, are in milliseconds, averaged over the run for the command. In a full study, it would be important to show error bars for these quantities. In the interests of minimizing distractions, they have been omitted here. Basically, Windows 2000 is slowest in all cases.

Table 8: Single-thread IO response times in milliseconds (smaller is better)

Experimental Conditions				Response Time by System		
Org	Read %	IO Block (KB)	Cmd	Linux 2.2.18	Linux 2.4	Windows 2000
random	20	64	R	10.0 ms	10.4 ms	12.3 ms
			W	4.2	4.3	5.9
random	20	8	R	1.9	2.0	6.6
			W	0.8	0.8	3.1
random	80	64	R	8.3	8.4	8.6
			W	5.9	5.9	7.1
random	80	8	R	4.3	4.0	4.9
			W	3.1	2.8	4.1
seq	100	64	R	1.9	1.8	2.4
seq	100	8	R	0.1	0.1	0.1
seq	0	64	W	1.2	1.5	2.4
seq	0	8	W	0.1	0.1	0.4

A good representation of the performance differences is the Box Plot, shown in Figure 9, for the case of 80% random reads with an 8 KB block size.

There are six representations of statistical distributions shown, one for each system for each of read (R) and write (W). The boxes represent the heart of the distributions. They include the 25th to 75th percentile. The T-shaped extensions are called “whiskers”, and they represent the tails of the distribution out to 95%. Beyond that, individual plot points show statistical outliers and extreme values. These are aberrations in the sample times, and are identified as such by statistical analysis (the program *Statistica* version 5.5 was used in this work).

For this experiment, all three systems and both commands ran at much the same average rate. Linux 2.4 (green) has more fast reads than the other two systems.

Figure 9: Distribution of I/O response times for random 80% reads, 8 KB block, one thread per disk

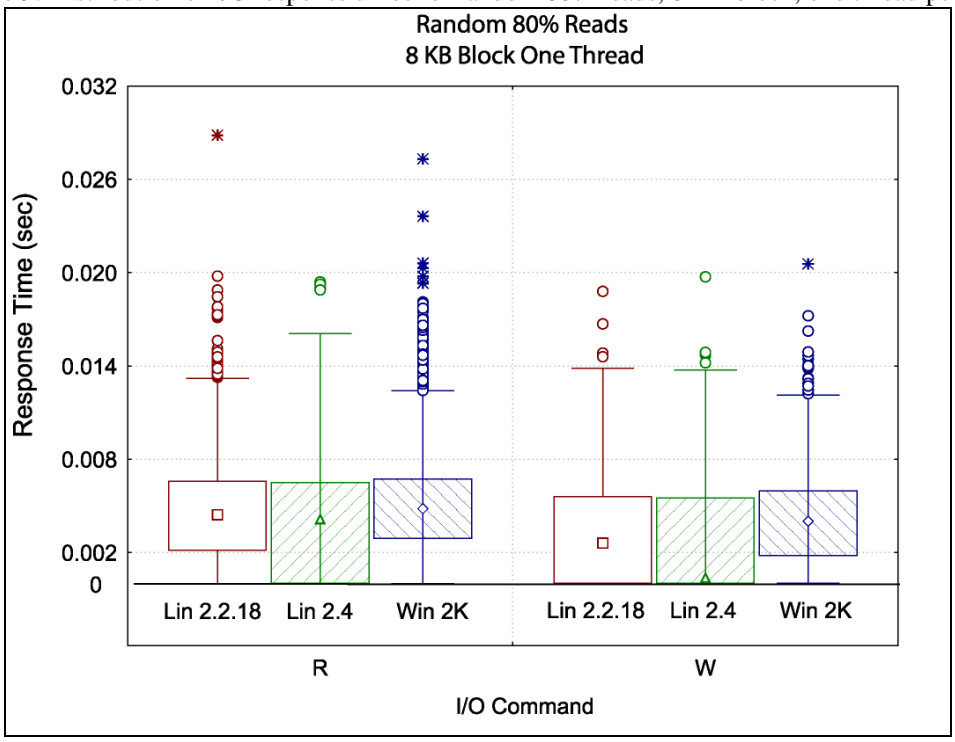
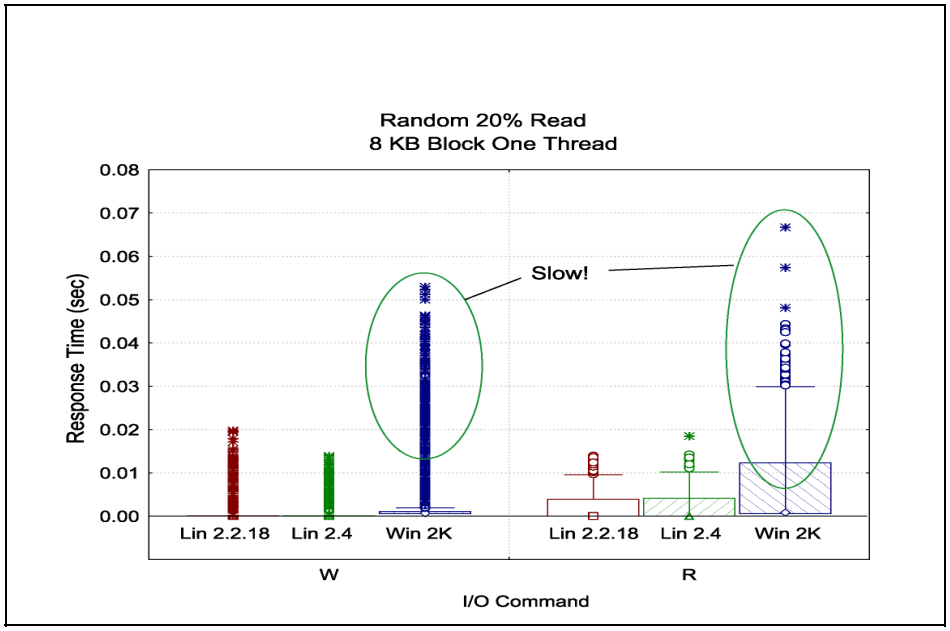


Figure 10: Distribution of I/O response times for random 20% reads, 8 KB blocks, one thread per disk



The results for Windows 2000 look much worse for the next case, of random 20% reads of 8 KB blocks, as shown in Figure 10. Both versions of Linux completed their I/Os in almost no time for almost all writes, with some individual cases extending to 20 ms. Windows 2000 looks very different, with much higher average behavior and extreme values out to 55 ms writes and 70 ms reads. These results should be studied more deeply by all developers, but especially by Microsoft in this case. One of the strengths of the morphing approach is the reproducibility of the benchmark results, and the short time needed for execution.

3.2.3 Four Disks, Eight Threads per Disk

The final set of runs made involved eight execution threads competing for access to a single file on a single disk. Four groups of eight threads were run in parallel, one per disk, for a total of 32 competing threads on four disks. The average I/O response time is given in Table 11.

Table 11: Eight-thread IO response times in milliseconds (smaller is better)

Experimental Conditions				*Response time by System*		
Org	Read %	IO Block (KB)	Cmd	Linux 2.2.18	Linux 2.4	Windows 2000
random	20	64	R	46.2 ms	55.6 ms	47.2 ms
			W	44.5	50.3	45.4
random	20	8	R	2.8	1.0	20.1
			W	2.2	0.8	20.0
random	80	64	R	57.9	44.8	45.6
			W	55.6	43.0	45.7
random	80	8	R	6.6	4.6	12.9
			W	6.1	3.9	12.9
seq	100	64	R	2.5	2.3	3.9
seq	100	8	R	0.1	0.1	1.9
seq	0	64	W	32.0	19.9	40.0
seq	0	8	W	0.1	0.1	28.9

The main intention behind choosing to drive eight threads was to stress the multitasking capabilities of the operating systems. The single thread work already showed that the disks will be the bottleneck here long before eight threads have their way. So it is not so much of an I/O test, but a system test.

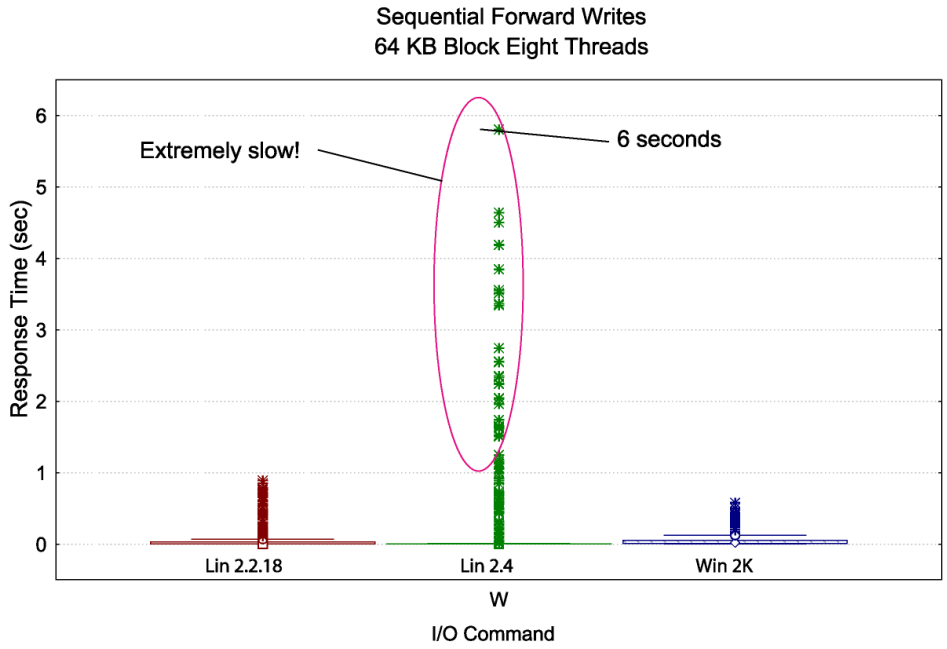
Several general statements can be made. Windows 2000 takes a back seat in performance once again, for most cases. See especially the last entry, sequential 8 KB writes, which take an average of 28.9 ms on Windows and only 0.1 ms on either Linux. Similar results are seen in the 20% random read case with 8 KB blocks.

Such a large discrepancy suggests different behavior on the two systems. Apparently, Linux just writes the data to memory and doesn't worry about flushing to disk, while Windows forces the blocks out. Implications of this difference in policy for reliability in the face of power failure etc. would be prudent to investigate by someone interested in using Linux for server applications. On the other hand, it is possible that Windows suffers from a performance bug in this case.

In some respects, Linux 2.4 is a step backwards from the 2.2.18 kernel. In Figure 12, the distribution of response times for the sequential 64 KB block write case is shown. The results for Linux 2.2.18 and Windows are pretty comparable, and not too unreasonable. The average response time was won by Linux 2.4 at 19.9 ms, followed by Linux 2.2.18 at 32 ms, and trailed by Windows 2000 at 40 ms.

The tails of the distribution, however, are extremely bad in the case of Linux 2.4. The response time of a single I/O request took up to 6.6 *seconds*. This illustrates the importance of paying close attention to all aspects of benchmark performance, including the timings of individual operations. This run probably stumbles on a problem in the Linux 2.4 kernel that may otherwise be hard to recognize and diagnose. With this methodology, the behavior is readily apparent, and calls attention to itself immediately, begging for explanation.

Figure 12: Distribution of I/O response times, sequential write, 64 KB block, 8 threads per disk - Linux 2.4 kernel does a pretty bad job here



3.3 System Resource Stress: OS Counters

The use of operating system counters to diagnose is a large and interesting topic. Only one aspect of their use will be presented here in the interests of brevity.

3.3.1 CPU Utilization per I/O

The CPU is the central resource shared by all active processes, and there is a finite capacity for doing work. Table 13 presents the average total CPU time (user plus system over both CPUs) per I/O (both reads and writes) for each type of run, on each OS. Times are given in □sec.

There are two entries for each experimental run, defined by the organization ("Org"), the read percentage ("Read%"), the I/O block size ("I/O Block"), and the number of threads ("Threads"). Lower numbers are better. Entries are paired on successive lines with the 1 thread case first, then the 8 thread case. The lowest time in any category is depicted in italics. Most lowest entries are from the Windows 2000 system.

The average CPU time for Linux 2.2.18 was 367 μsec, for Linux 2.4 was 435 μsec, and for Windows 2000 it was 276 μsec. Overall, that's a clear win for Windows over Linux in the category of CPU efficiency.

Several outstandingly bad CPU hog cases exist, like Linux 2.4 running random 20% reads with eight threads. In all cases, the amount of CPU used increases in the 8 thread cases compared to the 1 thread cases. This happens due to the excess overhead of context switching, stack management, I/O buffer management, and other operating system tasks necessary to run the benchmark. Some of these operations are chronicled in the Perfmon data of Windows 2000, and are therefore available for further analysis. Not so for Linux which keeps its secrets owing to the poor information content of *vmstat*.

This overhead can increase by a large proportion, like the Linux 2.4 case of 8KB 20% random reads which increases from 72 μsec to 600 μsec per I/O. Windows only increased from 93 μsec to 189 μsec in comparison.

Table 13: Average CPU time per I/O in μsec over the course of a run

Experimental Conditions				*CPU Time per I/O by System*		
Org	Read %	I/O Block (KB)	Threads	Linux 2.2.18	Linux 2.4	Windows 2000
random	20	64	1	366 μsec	302 μsec	*279* μsec
			8	624	1059	*461*
random	20	8	1	80	*72*	93
			8	460	600	*189*
random	80	64	1	292	*261*	296
			8	*340*	531	381
random	80	8	1	*72*	93	105
			8	386	534	*173*
seq	100	64	1	*264*	277	424
			8	684	838	*470*
seq	100	8	1	252	188	*113*
			8	434	571	*167*
seq	0	64	1	321	*250*	421
			8	605	*556*	565
seq	0	8	1	243	236	*98*
			8	444	588	*182*

There is a lot to explain here on the part of the operating system producers. What causes the increase in the amount of CPU time for each sequential read of 64 KB blocks from 277 μsec to 838 μsec (Linux 2.4) for one thread and eight threads per disk, respectively? Similarly, for Linux

2.2.18 say, in the random 80% read case with 8 KB block size. A large amount of CPU time is used to perform context switches, it appears. Alert Consumer Reports!

3.4 Computer Server Queuing Model

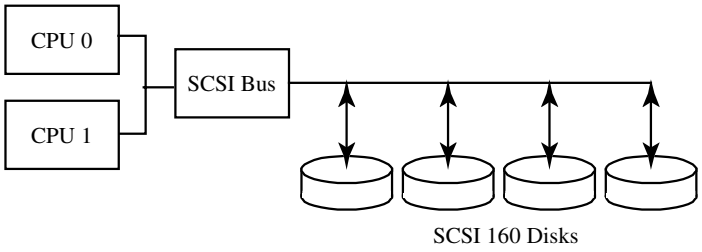
In performance investigations, a computer system is abstracted to a set of service centers, the key places where useful work is performed. Examples of service centers in a computer system are disks, I/O controller cards, or CPUs. Shared centers normally have queues of waiting customers or users. Depending on the mathematical technique used to compute the properties of the model, the connections between components may or may not be important.

The model of the system used to describe the server system includes these three types of components with multiple CPUs and SCSI disks:

- CPUs (2)
- SCSI Bus and Controller (1)
- SCSI Disks (4)

The logical arrangement of these parts is shown in Figure 14. Each service center takes some amount of time to do its job for each customer request. All of these are shared items, so each has a queue of waiting threads or processes demanding service.

Figure 14: Service Centers Representing the Server System



For the type of analysis used here, only two sets of information are needed. One set defines the classes of customers and the number in each class. The other set defines the amount of service each customer type needs from each service center in the system. The method of analysis is called Mean Value Analysis, or *MVA* for short. It is simple to set up, solves the problem very quickly, and produces results that are of accuracy comparable to the input data driving the model.

MVA does not handle some relationships among components well or at all, but those are not encountered in the server considered here. For example, a request that seizes control of more than one service center at a time cannot be represented. This occurs in many computer buses, for instance. The textbook *Quantitative System Performance: Computer System Analysis Using Queuing Network Models* by Lazowska, Zahorjan, Graham, and Sevcik (Prentice-Hall, Englewood Cliffs, NJ: 1984) is a good reference on the subject.

3.4.1 Queuing Model Input

In the benchmarks run here, there are four classes of customers, one for each thread of execution that addresses a given disk. Only three performance metrics are needed as inputs:

- Average CPU time needed to support one I/O request
- Average disk service time needed to perform one I/O request, measured at the disk
- Amount of time spent on the SCSI bus transferring data

3.4.2 Queuing Model Output

The two main outputs of the model is the utilization of the resource or service center, and the queue length at that service center. These are named U and Q, respectively. Utilization is expressed as the fraction of the time a request is being processed at a service center. Q is the average number of requests either being executed or waiting for service. Two quantities derived from the per class, per service center Us and Qs are the throughputs and response times. Throughput is the number of requests completed per unit time in each customer class. Response time is the amount of time needed to complete a request per customer. It is the sum of the service times at each service center for a given customer class, plus any time spent waiting in a queue for service.

3.5 EXAMPLE: SEQUENTIAL FILE READ ACCESS, 20% READS, 8 KB BLOCKS, ON WINDOWS 2000

Windows 2000 is used as the test case for queuing model analysis because the Perfmon diagnostics present physical disk metrics directly. Linux does not presently offer that information.

The measured parameters that were derived from the benchmark runs used in the model are presented in Table 15. These quantities were computed from the recorded *Perfmon* data. The computation is straightforward, but not totally trivial, though, since a mixture of file reads and writes were run, and the disks execute two queues in parallel, the reads and the writes. Approximately the same service times were seen on each of the four disks, so the same service time is used for the four simulated disks.

The results are the important part. Agreement between the predicted utilization, queue lengths, and response times means that the model is a fair representation of the organization and operational behavior of the system. Disagreement means that something has been understood mistakenly. Typical causes might be errors in the service time measurements, a parallelism in the system not represented in the model, or additional service centers in the real system that have an effect but are not represented in the model.

Table 15: Computed Service Times for Benchmark, the only model inputs

Quantity	1 Thread	8 Threads
CPU time per CPU	0.049 ms	0.123 ms
SCSI Bus time	0.050 ms	0.050 ms
SCSI Disk time	4.98 ms	1.5 ms

Model results for the one thread case are given in Table 16. The measured CPU utilization was a good bit higher in the real system than the model predicts. Experimentation with the model shows that the CPU result is very sensitive to the exact service time used for the disks. Otherwise, the SCSI disk results and the overall response time came out quite closely. Clearly, this is a benchmark that is completely dependent on the performance of the disks.

Table 16: Single thread Queuing Model results

Quantity	Measured	Predicted by MVA
SCSI Ctlr U	NA	0.04
SCSI Ctlr Q	NA	0.04
CPU U	8.5%	4%
CPU Q	NA	0.04
SCSI Disk U	96.4%	97%
SCSI Disk Q	0.96	0.97
Response time	4.9 ms	5.1 ms

Model results for the eight thread case are given in Table 17 where the CPU utilization agrees much more closely with the measured value. The utilization of the disks is again nearly 100%, so this case is still disk limited for the most part. The queue length on each disk is only 5 in reality, meaning that at any given time three threads are occupied in the CPU, while the model's prediction is that 7.7 threads are waiting for the disk. Once again, the response time per request came out very close to the measured value

The average service time at the disk is 4.98 ms in the single thread case, and just 1.5 ms in the eight thread case. The benchmark has a random file access pattern, with no commonality across thread scripts as to disk block addresses. This suggests that in a short run such as this, that memory caching effects do not come into play. This is indirect evidence for a form of optimization within the disk controller with respect to command chaining. Since there were five pending requests on the average at all times on the SCSI controller, it is likely that an optimal order was imposed on those requests to minimize rotational latency. If executed naively, the service time would have been 4.98 ms still.

Table 17: Eight thread Queuing Model results

Quantity	Measured	Predicted by MVA
SCSI Ctlr U	NA	13.3%
SCSI Ctlr Q	NA	0.15
CPU U	35.6%	32.6%
CPU Q	NA	0.48
SCSI Disk U	98.3%	99.5%
SCSI Disk Q	5.0	7.7
Response time	12.9 ms	12.1 ms

3.6 SCALING EXTRAPOLATION

Since these benchmarks are disk limited, it is interesting to see what effect faster disks would have on the overall behavior of the system. This can be done with the queuing model, and without spending another penny on disks before knowing what you'll get. Suppose some new disks are used with higher densities or faster rotational speeds so that the service times on the disks are lowered by 40%. What happens?

The answer is: Much better response time and throughput. The disks are still nearly saturated (93% utilization) though CPU utilization went up to 54% from 32.6% in the eight thread case. Overall response time becomes 2.6 ms (was 5.1 ms) for one thread and 7.7 ms (was 12.1 ms) for eight threads. Throughput goes from 194 to 391 I/Os per second, and from 663 to 1 099 for one and eight threads respectively.

Alternatively, what if two disks are added having the same performance characteristics as the real disks? Let two of the four customer classes now balance their I/O requests among one old and one new disk.

The result: It helped the two customer classes using two disks a lot, but only for the eight thread case. The one thread case cannot take advantage of parallel disk access. The new eight thread response time dropped to 6.8 ms from 12.1, while the other two classes stayed at 12.1 ms. Throughput also jumped for these two classes of customers to 1 178 I/Os per second each. CPU utilization dropped to 18% from 32%.

Add more disks for performance!

4 Questions and Answers

4.1 Can this method be applied to other areas of interest?

The example presented here evaluates multiple disk access on a single system. the morphing methodology can also be applied to any of the following areas. The method is so powerful that it does not have to change even when the types of logs and stimuli differ, always providing the same quality of insights.

Systems
- Web server - client architectures
- Database and SQL processing
- Network-attached storage servers
- RAID systems
- Network components: routers, concentrators, bridges
- Modelling environments: eCAD, numerical simulation, real-time training

Software
- General purpose operating systems
- Middleware
- Embedded operating systems
- Application software
- Device drivers
- Libraries / DLLs / shared libraries

Low level components
- Embedded systems (context switches, memory management, interrupt response, etc.)
- Disk drives
- Graphics controllers
- I/O controller optimization (disk, communications, network, server cluster, etc.)
- I/O bus bridge chip, memory controllers, I/O controller ASIC

4.2 What about live systems?

Look on the live system as a whole set of benchmarks. The experimental design controls are not there, but you have two important ingredients for successful performance modeling and understanding:
- a measure per unit time, over time, of the number of transactions of different types that are run (in queuing model terms this is the request arrival rate λ(Transaction Class)), and
- the system stress averaged over the same interval of real time

These two can be combined with statistical analysis to yield service times at the key subsystems. This exposes bottlenecks and warns about impending problems in capacity, long before trouble shows up to users. With enough independent time intervals, correlation and later cause and effect will be discovered.

4.3 How reliable are extrapolations and model results?

The predictions of the queuing models are typically good to better than 5%. Confidence in extrapolations increases when the model accurately predicts what was actually measured given

the inputs. See the Computer Server Queuing Model section above.

4.4 ARE THICK BOOKS ON TUNING VARIOUS COMMERCIAL SYSTEMS VERY USEFUL?

There are many books like *Tuning and Sizing Windows 2000* by Curt Aubley (Prentice-Hall 2001) or *Sun Performance and Tuning* by Adrian Cockcroft and Richard Pettit (Prentice-Hall 1998). They give numerous details of how specific systems operate and what the diagnostic numbers mean. The reader is really being trained to be a sleuth in the vendor's OS world.

The problem is that the world has no good map: Hunt around everywhere, you *might* find out what's going on if you follow the guidelines and have some luck.

The morphing approach, applied to exactly the same operating-system provided data leveraged by these authors, unerringly guides the investigator directly to the source of the problem. The statistical methods make the difference between working hard and working smart. That's what computers were supposed to do for us!

With the morphing methods, there is no need to become an expert in details to reap the benefits of deep understanding of cause and effect at the system level.

5 Glossary

Some terms used in this description of this approach to system testing may be unfamiliar to readers. Here are brief definitions of a few potentially troublesome terms.

Context switch - The step of going from one task to another. This means suspending execution of the currently running program or process because it has called a system-supplied function that cannot finish immediately. Most commonly, this happens with I/O operations, or with voluntary suspensions of execution like at semaphores.

Command chaining - A feature of high-performance disk controller hardware whose purpose is to increase performance. A new command may be executed before already-entered commands if no logical problem is caused (e.g. write preceding a read of the same data when the commands were entered read then write) when the I/O commands can be completed faster in the new order. This is determined by a combination of the device driver software and the disk controller hardware.

Morphing - Here used as a short-hand term for the process of applying the exhaustive morphological methods to the mapping of the possibilities. In mechanics, the coordinates required to uniquely specify the state of a system generate phase space. In computer science, a system is specified by its configuration and state, both of which are very complex.

Morphological - The term, as used here, means "all possible situations or states". Fritz Zwicky introduced the term in his book *Morphological Astronomy* (Springer-Verlag 1957). He used morphological methods to explore the alternatives for vehicles in space and technologies for exploration of the planets, for example. Various newer methods have been introduced over the years for generating the set of possibilities. Many alternative contributions have been reported in the literature on enhancing creativity.

Confirming the Effectiveness of the Requirements Generation Model: An Industry-Based Empirical Study

Markus K. Gröner & James D. Arthur
Department of Computer Science; Virginia Tech; Blacksburg, VA 24061-0106
{groener|arthur}@vt.edu

Abstract

Product quality is directly related to how well that product meets the customer's needs and intents. It is paramount, therefore, to capture customer requirements correctly and succinctly. Unfortunately, most development models tend to avoid, or only vaguely define the process by which requirements are generated. Other models rely on formalistic characterizations that require specialized training to understand. To address such drawbacks we introduce the Requirements Generation Model (RGM) that (a) decomposes the conventional "requirements analysis" phase into sub-phases which focus and refine requirements generation activities, (b) constrains and structures those activities, and (c) incorporates a monitoring methodology to assist in detecting and resolving deviations from process activities defined by the RGM. The results of an industry-based study are also presented and substantiate the effectiveness of the RGM in producing a better set of requirements.

Keywords

Requirements Identification, Requirements Generation, Customer Intent, Requirements Engineering, Software Engineering, Software Methodology

1. Introduction

This paper presents an industry-based study to help assess the effectiveness of the Requirements Generation Model (RGM) [4] in software development efforts. The RGM is designed to minimize the difficult task faced by every requirements engineer, i.e., to capture customer requirements accurately and succinctly. Every software development effort depends on procedures and methods such as those found in the Waterfall Model [6], the Spiral Model [2], and prototyping approaches [7] to name a few. Yet, no matter how much rigor is applied to the process, the resulting product is only as good as the set of requirements by which it is defined. The RGM consists of a framework and a monitoring methodology that constrains and guides a requirements engineer and customer through the requirements definition process. The objective of the RGM is to provide a structure within which requirements are captured meeting customer intent. This paper provides a brief overview of the RGM and presents the findings of an empirical study conducted to determine its effectiveness. A complementary subset of the RGM components was selected for the study. The positive contributions the RGM makes to the requirements definition process is substantiated by the empirical data derived from the study.

The remainder of this paper is structured as follows: Section 2 provides a brief overview of the Requirements Generation Model; Section 3 describes the selection process for two groups in this study; Section 4 outlines how the empirical study was structured; Section 5 presents the data and conjectures supported by that data. Finally, Section 6 provides some concluding remarks.

2. An Overview of the Requirements Generation Model

The Requirements Generation Model (RGM) refines the conventional "requirements analysis" phase by imposing a framework and methodology that structures the requirements elicitation, recording and evaluation processes. Similar to existing models like Participatory Design (PD) [3] and Joint Application Design (JAD) [5], the RGM seeks to reduce the disparity in domain knowledge between the requirements engineer and customer. The RGM differs from those models, however, in that it focuses extensively on the requirements phase, and within that phase, prescribes activities that address a more specific and well-defined set of objectives.

2.1 The Framework

As illustrated in Figure 1, the RGM framework partitions the conventional requirements analysis phase into an initial indoctrination phase, which is then followed by an iterative requirements capturing phase. The objectives of the indoctrination phase are to (a) introduce the customer to the requirements definition process, (b) provide the requirements engineer with an overview of the customer's problem domain and needs, and (c) describe the participants' tasks and responsibilities in the requirements

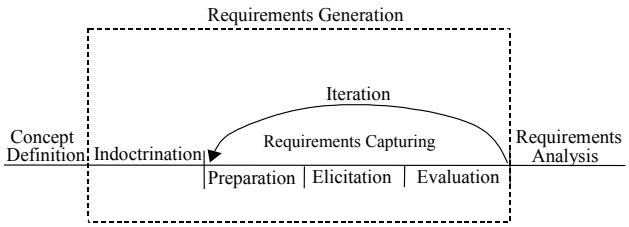


Figure 1: The Requirements Generation Framework

definition process. The objectives of the requirements capturing phase are directly related to the three sub-phases that comprise it – preparation, elicitation and evaluation. Respectively, the primary objectives of those sub-phases are to (a) define the scope of the up-coming elicitation meeting and ensure that all participants have completed their pre-meeting assignments, (b) enable the requirements engineer to accurately identify and record software requirements as expressed by the customer, and (c) evaluate the contributions of the preceding elicitation meetings, identify unresolved (or new) issues, and determine if an additional refinement iteration is needed. All three sub-phases are embedded within an iterative framework that, in turn, encourages the progressive identification, refinement and elaboration of individual requirements. Collectively, the three sub-phases and the iterative framework define the requirements capturing process. Through phase-specific guidelines and protocols, and through its iterative framework, this process promotes a structured approach to the interaction format and information exchange between the customer and the requirements engineer.

2.2 The Monitoring Methodology

Even within a structured requirements elicitation process problems can still surface, and before being recognized as such, have an adverse impact on that process. Hence, identifying and addressing such problems *as they occur* is an important component of the RGM. The RGM provides this detection/correction component through its *monitoring methodology*. That is, the RGM's monitoring methodology detects deviations in the prescribed elicitation process and suggests appropriate remedies. In more specific terms the monitoring methodology is composed (a) automated and manual procedures that constantly "monitor" the elicitation process for defined problems, and (b) methods that assist in their resolution. An example of one simple procedure is setting a timer to provide an audible "alarm" when a meeting has exceeded its stipulated time duration.

In effect, the framework and methodology of the RGM have been purposefully designed to (a) add structure and control to the requirements generation activities, (b) support the accurate capturing of requirements, and we conjecture, and (c) promote an effective requirements generation process.

3. Selection Process

The selection of participants for this empirical study calls for two groups of comparable size and experience. One group, designated as the "Non-RGM" group, serves as the control group for the study. The Non-RGM defines requirements and implements software using currently accepted practices. The second group, called the "RGM" group, is the experimental group. The RGM group receives training in the use of the RGM and in the application of its activities to the software development process. Both the RGM and Non-RGM groups work in the same industrial setting, supporting financial data processing systems and software. Prior to the study, the development efforts and track records for each group were similar.

The details of the empirical study and a discussion of how we attempted to avoid introducing bias is discussed in the following section.

4. Empirical Study Setup

This section describes the details of the empirical study. We focus on 1) those parts of the RGM that are deployed in the empirical study, 2) the environment in which the empirical study is conducted, 3) characteristics of the projects developed within the empirical study, 4) the people involved in the empirical study, and 5) the approach used to insert the RGM into the development process.

4.1. A Description of the Empirical Study

An empirical study was conducted to assess the benefits of using the RGM within an industrial development environment. In this study, we show that using the RGM instead of the currently existing requirements generation process improves the effectiveness of the requirements generation process, and has an additional positive impact on the remaining development phases. Through this study, we establish three benefits of employing the RGM, although many more are believed to exist:

1. By using the RGM to more effectively capture requirements as stated by the customer, we expect to see less (if any) schedule slippage for major project milestones.
2. By using the RGM we expect fewer adverse impacts in later development phases due to requirements changes.
3. By using the RGM we expect those directly involved in the project development, as well as the customer, to have a higher level of satisfaction with the project.

Changing a development environment is a difficult task at best. We desire, therefore, to implement minimal, but effective changes. Hence, we have selected elements of the RGM that we expect will contribute to the requirements generation process, but require minimal implementation overhead. These elements are:

- *Issues/Participant Notifications* – Participants in upcoming requirements elicitation meetings are notified in advance of issues they are expected to present.
- *Meeting Notifications* – Resolutions and problems are distributed in a timely manner before meetings, as well as meeting place/date/time and alternative ways to attend the meeting are announced.
- *Silent Parking Lot* – Each participant brings a note pad and records tangential (off-topic) thoughts and issues. At the end of the meeting the leader collects those additional notes and relays them to the appropriate person(s) for proper consideration.
- *Limited Unstructured Requirements List Reviews* – A thorough review of "just-captured" requirements by the requirements engineer and the customer.
- *Controlled Interrupts* – A method embraced by the RGM and intended to reduce interruptions during a requirements elicitation meeting. This method complements the use of the Silent Park Lot.
- *Avoiding the "Guessing Game"* – This method is used to detect and minimize the introduction of vague requirements stemming from the stakeholder "guessing" at the correct requirement. Responses containing terms or phrases like "possibly", "maybe", and "I think so" indicate questionable understanding.
- *Seeking Requirements, Not Solutions* – The RGM is specific in stipulating that the requirements engineer must avoid the introduction of requirements that suggest *a priori* solutions, and thereby, constrain requirements formulation.

The above activities and methods were employed as dictated by iterative *Requirements Capturing* component of the RGM (i.e., within the preparation, elicitation and evaluation sub-phases).

4.2. Environment

The study environment is comprised of the physical surroundings, the hardware, and the operating systems being used, as well as the atmosphere established through the organization of the study. This section details those components and discusses the efforts made to avoid the introduction of bias.

4.2.1. The Physical Environment.

The empirical study is performed in a large financial institution. The department within which the study takes place is comprised of five separate groups. Four of these groups work within the same building but are physically separated. The fifth group works in a separate building. The physical environment is one of cubicles for each employee, with either a single PC running MS Windows 95, or two PCs, one with MS Windows and a second with Unix. Developers use either MS Visual Basic or C/C++ on the Unix operating system.

4.2.2. Avoiding Bias

One concern when conducting a study is the introduction of bias – bias compromises otherwise valid results. A first step in avoiding the introduction of bias is to set up an "experimental" group and a "control" group and *minimize their interaction*. For example, those elements to be studied, i.e. changes to the normal environment, are introduced through the experimental group, while the control group continues to function the same as before.

Our empirical study employs an experimental group and a control group. The experimental group (referred to as the RGM group) uses those previously described components of the RGM. The control group (referred to as the Non-RGM group) continues to employ the currently established development process. For both groups we use several techniques to minimize the introduction of bias:

- Both groups work in the same department, but work in physically separate locations.
- Both groups work on the same type of applications but which have no overlapping components. Each group has a completely different set of customers.
- Only the experimental (RGM) group is trained in the RGM process. No one else is aware of this special training.
- Necessary artifacts are given only to the RGM group.
- We monitor both groups throughout the empirical study.
- Neither group is aware that they are part of a study. The data to be collected is a standard component of both group's submission requirement for projects.

The six items noted above serve to minimize the introduction of bias into the empirical study.

4.3. Project Characteristics

This section presents the characteristics of the projects used in the empirical study. Those characteristics include the elements that determine selection, such as project size, application type, and development and target environments. Also included are the roles of those involved in the projects.

4.3.1. Project Selection

The following set of characteristics were used in selecting the projects for the groups in the study:

- Project Time Length – must be estimated at no more than six months to completion.
- Project Size – each group must have two large projects and two small projects. Project size in the given development environment is determined by estimated project cost. A project of greater than $250K is large, and below $250K small. This cost includes the purchase of hardware as well as employee time. Size is not an indicator for the expected project length, as small projects may be developed in parallel with less than 100% human resource dedicated to a single project at a time. In this environment, it is common for small and large projects to start and finish at the same time due to this factor.
- Project Complexity – all projects must be a financial processing application with database access. The large projects were critical and essential to the daily operations of the business. The small projects were system functionality enhancements.
- Personnel – between the two groups corresponding participants must have equal qualifications, experience, and length of time within their groups.

Additional project characteristics and division between the two development groups are as follows:

RGM (Experimental) Group:

- Two large projects (Projects 1 & 2).
- Two small projects (Projects 4 & 5).
- All projects are financial applications with simple data manipulation (calculations), database look-ups, and database stores.
- All projects are to be developed in either MS Visual Basic or C++ on Windows 95 or Unix based operating systems, respectively.
- All projects are deployed on either Windows 95 or Unix based operating system machines.
- *Roles:*
  - Large projects each have 1 project manager, 1 systems analyst, and 2 developers, as well as a single customer.
  - Small projects each have 1 project manager, 1 systems analyst, and 1 developer, as well as a single customer.

Non-RGM (Control) Group:

- Two large size projects initially. One project is canceled early during requirements definition. (Project 3 is one that remained).
- Two small size projects (Projects 6 & 7).
- All projects are financial applications with simple data manipulation (calculations), database look-ups and database stores.
- All projects are developed in either MS Visual Basic or C++ on Windows 95 or Unix based operating systems respectively.
- All projects are deployed on either Windows 95 or Unix based operating system machines.
- *Roles*:
  - Large project has 1 project manager, 1 systems analyst, and 2 developers, as well as a single customer.
  - Small projects each have 1 project manager, 1 systems analyst, and 1 developer, as well as a single customer.

For each group, we selected two large projects and two small projects using the criteria stipulated above, any other similarities or differences were coincidental. The selection process for the RGM projects, however, did include the exception that project start dates must be after the anticipated completion date for the training of the RGM group.

4.3.2. Personnel

In this section we provide an overview of the people who are involved in the study projects. It is important for the study that all have comparable skill sets, education and roles.

- *Project Managers:* There are a total of four project managers – two in each group. All project managers have a minimum of ten years of experience in managing projects for various organizations, and all have been within the same department for a minimum of one and a half years. Three of the project managers have an MBA degree, and one has an MIS degree.
- *Systems Analysts*: Two systems analysts (one in each group) are part of the empirical study. Both analysts have a minimum of three years of work experience as both developers and analysts. Each has a Masters Degree in Computer Science.
- *Developers:* Each group in the empirical study has four developers working on the projects. Each developer has a minimum of two years of work experience programming in both MS Visual Basic as well as Unix C/C++. All developers have a Bachelors Degree in Computer Science.

4.4. Empirical Study Process

To maximize the probability of a successful study and to support the collection of valid data, several issues had to be resolved before the study is conducted. This section focuses on those particular issues.

4.4.1. Training

The RGM group had to be trained in the selected activities and methods of the RGM to be used during the study. Only project managers and systems analysts are trained because they alone have direct contact with the customer and will be using the RGM. Training of the group includes an overview of the RGM, as well as a discussion and rationale for using the selected RGM methods and artifacts. To help avoid the introduction of bias or other behaviors that might impact the validity of the study, we did not reveal to the group that a study was being conducted.

4.4.2. Oversight

To ensure adherence to the RGM process and to assist the group in prescribed activities thereof, we attended several of the earlier requirements elicitation meetings. Assistance was required only a few times, and those were primarily related to *Controlled Interrupts* and to the use of the *Silent Parking Lots*. Following the requirements elicitation process we met with the project managers and systems analysts to provide them feedback as to their adherence to the RGM, and to receive their thoughts on how we might further enhance the RGM.

5. Data Collection, Presentation, and Interpretation

The data described in this section was collected from a company-wide status-reporting database. In addition to reporting phase initiation and completion dates, project managers are required to update their project status when major milestones are reached (such as finishing the requirements phase and moving into design), and when project status change. Project status colors are established to indicate how the project is progressing – those colors, and what they imply, are as follows:

- *Green* – project is proceeding on schedule and within budge. Overall the project is encountering no current risks.
- *Yellow* – project has run into a possible risk that could impact either schedule or budget. The project needs to be closely managed with an emphasis on minimizing perceived risk.
- *Red* – project has encountered a setback that adversely impacts either schedule or budget. At this time the setback needs to be evaluated and necessary project changes negotiated with the customers to resolve the red status.

During the study we also collected observational data during meetings with, and among, the customers, project managers and analysts. This data was used to make adjustments to the RGM and helped explain certain unexpected findings.

Together, the subjective and objective data collected through the status reporting database serve to confirm three conjectures concerning the beneficial impacts of using the RGM. We examine each conjecture and related data items in the following subsections.

5.1. Conjecture 1

> *Projects using the RGM to capture customer intent should experience less slippage in project phase completion dates.*

Stated in other terms, we expect that projects developed under the auspice of the RGM will meet their projected phase completion dates more often than similar projects which do not employ the RGM. To confirm this conjecture, we need to know the original date a phase is supposed to be completed, and the actual date the phase is completed.

Although the RGM has its highest impact on the requirements generation phase, its benefits are also realized in succeeding development phases. For the purpose of this study, therefore, we include an examination of the requirements definition, design, and implementation phases.

The data presented in Tables 1 and 2 represent phase-by-phase slippage days for the RGM and Non-RGM projects respectively. Slippage days are computed as the difference between the originally specified ending date for each phase and the actual ending date for that phase. (Adjustments were made to account for the compound impact of slippage on originally specified ending dates.) A negative number indicates the project ended earlier than expected. The projected duration is the number of days originally projected. The projected duration allows one to place the slippage days in perspective.

Table 1: RGM Group Projected Duration And Slippage Days

Project 1 *(large)*	**Projected Duration**	**Slippage Days**
Definition	31	0
Design	16	7
Implementation	54	-7
Total	101	0
Project 2 *(large)*	**Projected Duration**	**Slippage Days**
Definition	31	0
Design	246	-9
Implementation	2	9
Total	79	0

Project 4 *(small)*	**Projected Duration**	**Slippage Days**
Definition	25	0
Design	59	-3
Implementation	5	15
Total	89	12
Project 5 *(small)*		**Slippage Days**
Definition	26	0
Design	52	-1
Implementation	14	25
Total	92	24

Table 2: Non-RGM Group Slippage Days

Project 3 *(large)*	**Projected Duration**	**Slippage Days**
Definition	27	27
Design	0	215
Implementation	44	-8
Total	71	234
Project 6 *(small)*	**Projected Duration**	**Slippage Days**
Definition	19	9
Design	30	-2
Implementation	151	-31
Total	200	-24
Project 7 *(small)*	**Projected Duration**	**Slippage Days**
Definition	15	52
Design	14	1
Implementation	18	5
Total	47	58

5.1.1. Analyzing Slippage Days

To confirm (or refute) our first conjecture, we examine the data presented in Tables 1 and 2 from two different perspectives.

First, we view the data as a simple phase-by-phase comparison of slippage for both the large and small projects. To help simplify the process and to help remove any potential bias that could be introduced by looking at the relative and absolute magnitude of the numbers, we use two indicators, "<" and ">", to capture "slippage." That is, "<" means that the RGM group had less slippage than the Non-RGM group for that particular phase on a same size project. Conversely, ">"indicates less slippage for the Non-RGM group. Using the above slippage indicators, Table 3 depicts a cross comparison of the large projects for the experimental (RGM) and control (Non_RGM) groups. Similarly, Table 5 provides the cross-comparison for the small projects.

Our second perspective is based on a magnitude indicator. The magnitude indicator is intended to reflect the *collective* extent to which the slippage days differ between the RGM and Non-RGM development efforts. The magnitude indicator is computed as a function of the number of slippage indicators ("<" and ">") given in Tables 3 and 5. That is, for a given set of cross-comparisons if all of the slippage indicators within a phase are "<" then we set the magnitude indicator for that phase to "++", indicating a significant advantage for the RGM development effort. If more than half but not all slippage indicators within a phase are "<", then the magnitude indicator for that phase is set to "+", indicating a simple advantage for the RGM group. If the number of slippage indicators is evenly split among the cross-comparisons for a given phase, then we set the magnitude indicator for that phase to a "0", indicating that no advantage exists for either the RGM or Non-RGM development effort. Similarly, for the ">" slippage indicators we compute phase-wise magnitude indicators of "--", "-", and "0" denoting the extent of the advantage for the Non-RGM development effort. Tables 4 and 6 provide the magnitude computations for the large and small project cross-comparisons, respectively.

We now provide and analysis of project slippage using slippage days (Tables 1 and 2), slippage indicators (Tables 3 and 5) and magnitude indicators (Tables 4 and 6).

Table 3: Large Project Slippage Comparison

	RGM Project 1	**Comp.**	**Non-RGM Project 3**	**Diff.**
Definition	0	<	27	-27
Design	7	<	215	-208
Implementation	-7	>	-8	1
Total Days	0	<	234	-234
% Slippage	0	<	330%	-330%
	RGM Project 2	**Comp.**	**Non-RGM Project 3**	**Diff.**
Definition	0	<	27	-27
Design	-9	<	215	-224
Implementation	9	>	-8	17
Total Days	0	<	234	-234
% Slippage	0%	<	330%	-330%

Table 4: Large Projects Magnitude Indicators

	Comp. Dir.	Magn.	Total Diff.	Avg. Diff.
Definition	2 < and 0 >	++	-54	-27
Design	2 < and 0 >	++	-432	-216
Implementation	0 < and 2 >	--	18	9
Total	2 < and 0 >	++	-468	-234
% Slippage	2 < and 0 >	++	-659%	-330%
Total Magnitude	8 < and 2 >	++		

5.1.2. A Comparison Among the Large Projects

Table 3 provides a cross-comparison of slippage days for the large RGM and Non-RGM projects. The comparisons show that both RGM group projects (1 & 2) performed better than the Non-RGM group project (3) in the requirements definition and design phases. The Non-RGM group, however, faired better in its implementation phase for Project 3 by finishing eight days ahead of schedule. Nonetheless, we also note that Project 3 completed its implementation phase only one day ahead of the RGM Project 1, which completed its implementation seven days ahead of schedule.

Looking at the total number of slippage days, as well as the percent of slippage, the two RGM group projects (1 & 2) finished the combined definition, design, and implementation phases on the time. The Non-RGM project did not fair as well. Due to the large setback in the design phase, the project came in 234 days late, or 330% slippage over the initially planned schedule.

Examining the magnitude indicators shown in Table 4, we also observe that the large RGM projects fair much better than the Non-RGM counterpart – the one exception is in the implementation phase. The overall (or "total") magnitude indicator also significantly favors the RGM development effort.

Therefore, based on the data provided in Tables 1 – 4, we are confident in stating that the larger projects developed within the RGM framework tend to meet their projected phase completion dates more often than their counterpart projects which did not employ the RGM.

Anecdotal note: We noticed on several occasions that the Non-RGM group developers started implementation well before the requirements or design documentation was completed. Often, implementation can proceed based on "knowledge" about the overall project's goals – and therefore independent of requirement and design details. This might provide an insight as to why they finished their implementation phase ahead of schedule. We also noticed that the "base" code from similar applications was later modified to meet requirements.

Table 5: Small Project Slippage Comparison

	RGM Project 4	Comp.	Non-RGM Project 6	Diff.
Definition	0	<	9	-9
Design	-3	<	-2	-1
Implementation	15	>	-31	46
Total Days	12	>	-24	36
% Slippage	13%	>	-12%	25%

	RGM Project 4	Comp.	Non-RGM Project 7	Diff.
Definition	0	<	52	-52
Design	-3	<	1	-4
Implementation	15	>	5	10
Total Days	12	<	58	-46
% Slippage	13%	<	123%	-110%

	RGM Project 5	Comp.	Non-RGM Project 6	Diff.
Definition	0	<	9	-9
Design	-1	>	-2	1
Implementation	25	>	-31	56
Total Days	24	>	-24	48
% Slippage	26%	>	-12%	38%

	RGM Project 5	Comp.	Non-RGM Project 7	Diff.
Definition	0	<	52	-52
Design	-1	<	1	-2
Implementation	25	>	5	20
Total Days	24	<	58	-34
% Slippage	26%	<	123%	-97%

Table 6: Small Projects Magnitude Indicators

	Comp. Dir.	Magn.	Total Diff.	Avg. Diff.
Definition	4 < and 0 >	++	-122	-30.5
Design	3 < and 1 >	+	-6	-1.5
Implementation	0 < and 4 >	--	132	33
Total	2 < and 2 >	0	4	1
% Slippage	2 < and 2 >	0	-144%	-36%
Total Magnitude	11 < and 9 >	+		

5.1.3. A Comparison Among Small Projects

Table 5 provides a comparison among the smaller projects in the study. More specifically, it depicts four cross-comparisons using two RGM group projects (4 & 5) and the two Non-RGM group projects (6 & 7). The data

provided in Tables 1 and 5 confirms that the RGM group experienced no slippage during the requirements definition phases and actually ended the design phases for their projects ahead of schedule. The RGM group did experience some slippage in both projects during implementation – twelve and twenty-four days of total slippage (13% and 26% slippage) for Projects 4 and 5, respectively.

Yet, when compared to the Non-RGM group the RGM group still fairs better. The Non-RGM group experienced slippage for both projects during their definition phases (as compared to no slippage for the RGM projects). In comparing the design phase, the RGM projects faired better in 3 out of 4 of the cross comparisons, losing out only in a comparison between RGM Project 5 and Non-RGM Project 6 – and even in this comparison, both projects finished design ahead of schedule. Table 5 does not show any definite advantage either way when comparing slippage during implementation and overall total – they appear to split evenly. A pertinent observation, however, is that the Non-RGM Project 6 finished almost one month early. As we stated in the previous anecdotal note, we impute such success to a tendency for Non-RGM developers to start implementation before definition and design are complete.

Because the magnitude indicators reflect a more collective view of project "success", an examination of the data presented in Table 6 reveals a clearer picture of how the small RGM projects compared to their Non-RGM counterparts. Relative to project slippage, the magnitude indicators show that the RGM projects performed significantly better during the requirements definition phases than did the Non-RGM projects. The RGM group also performed better during the design phase (although not as impressive when compared to the requirements definition phase). During the implementation phases the Non-RGM group did significantly better than the RGM group. From an overall perspective, however, the magnitude indicator favors the RGM projects. On a final note, when we compare slippage percentage, the RGM projects experienced 36% less slippage than did the Non-RGM projects.

Once again, and similar to the conclusions drawn for large projects, there does appear to be a beneficial contribution in using the RGM for small project development efforts. Hence, we assert that the data presented in Tables 1 – 6, and the comparisons provided in the previous sections confirm our first conjecture, that is,

> *Projects using the RGM to capture customer intent should experience less slippage in project phase completion dates.*

Anecdotal note: The data provided in the previous tables suggest that with an increase in project size there is a commensurate increase in the benefit of using the RGM. We also expected to find that the RGM would result in an increased communication overhead during the definition phase. Based on our observations and discussions, however, no such increase was apparent.

5.2. Conjecture 2

> *Projects using the RGM should exhibit less adverse impact stemming from late discovery of requirements.*

We contend that given similar projects development efforts, if one project uses the RGM during the requirements definition phase, it will have a better success at identifying and recording customer requirements earlier than a project not employing the RGM. Consequently, the project employing the RGM will experience less adverse impact stemming from the *late* discovery of needed requirements. Our experience has shown that scope and requirements changes are to be expected for many projects – not that this is desirable, but simply a fact of our development process. When identified earlier in the software development process, however, these necessary changes can be corrected for a fraction of the time and cost it would take to fix them during the later phases of the development cycle [1]. By emphasizing a more complete and accurate identification of customer requirements during the requirements phase, the RGM helps minimize the adverse impact stemming from the late discovery of requirements.

During the study the project status database was also used to record changes stemming from late requirements discovery and to note their potential impact (if any) on the development effort. The project database supports the entry of three project status indicators throughout a development effort. These status indicators are associated with the project's schedule, budget and overall project status (or "health"). Project status changes are indicated by shifts in status colors. For example, the schedule status can shift from "green" to "red", indicating substantial problems ahead. There are three types of "bad" shifts: (G)reen to (Y)ellow, Green to (R)ed, and Yellow to Red. To help reason about Conjecture 2, we count the number of "bad" status changes *stemming from the late discovery of requirements*. Tables 7 and 8 capture those counts for the RGM and Non-RGM development efforts, respectively.

Table 7: RGM Group Project Status Change

Project 1	**Schedule**	**Budget**	**Overall Status**
G->Y	0	0	0
G->R	0	0	0
Y->R	0	0	0
Project 2	**Schedule**	**Budget**	**Overall Status**
G->Y	0	0	0
G->R	0	0	0

Y->R	0	0	0
Project 4	**Schedule**	**Budget**	**Overall Status**
G->Y	0	0	0
G->R	0	0	0
Y->R	0	0	0
Project 5	**Schedule**	**Budget**	**Overall Status**
G->Y	0	0	0
G->R	0	0	0
Y->R	0	0	0

Table 8: Non-RGM Group Project Status Change

Project 3	**Schedule**	**Budget**	**Overall Status**
G->Y	2	2	2
G->R	2	0	0
Y->R	1	0	0
Project 6	**Schedule**	**Budget**	**Overall Status**
G->Y	1	0	0
G->R	0	0	0
Y->R	0	0	0
Project 7	**Schedule**	**Budget**	**Overall Status**
G->Y	3	1	2
G->R	0	0	0
Y->R	1	0	1

5.2.1. Interpreting Status Changes

The benefits of incorporating the RGM in a "traditional" development process become apparent when one examines the data presented in Table 7 and Table 8. For all three status categories (schedule, budget and overall), the two large projects and the two small projects of the RGM group experienced *no status changes due to late discovery of requirements throughout the development effort*. On the other hand, the single large project and the two small projects for the Non-RGM group all experienced less than desirable project status changes. Project 6, the best of the three for the Non-RGM group had only one status change stemming from the discovery of late requirements, and that was from green to yellow. (Again, consistent with observations supporting Conjecture 1, this project performed better than the other Non-RGM projects.) The other two Non-RGM projects shifted to yellow several times, and each also had shifts to red.

As stated earlier, the RGM is designed to promote the identification of a more complete and accurate set of requirements early in the software development lifecycle. Consequently, we should expect (and conjecture) that

> *projects using the RGM should exhibit less adverse impact stemming from late discovery of requirements.*

The difference between the number of status changes for the RGM group projects as compared to those for the Non-RGM group projects, i.e. 0 – 18, confirms our second conjecture.

5.3. Conjecture 3

> *Projects using the RGM result in a higher satisfaction level for the project manager and business customer a greater percentage of time throughout the project to closure.*

Unlike the previous two conjectures, which are substantiated through the examination of objective data, the confirming data for Conjecture 3 are more subjective in nature. In particular, this third set of data tracks the level of satisfaction (or frustration) as directly expressed by the project manager and customer. The project manager and customer independently record their current satisfaction (or frustration) in the project status database whenever the project status changes or milestones are reached. We compare the satisfaction indicators, "yes" or "no", to determine the percentage of satisfaction versus frustration over the project development time for the Project Manager (PM) and the Business Customer (BC). Table 9 and Table 10 show the satisfaction data for the RGM group and the Non-RGM group, respectively. That is, for each project the tables indicate the total number of times the PM and BC indicate satisfaction (yes) or frustration (no). The percentage of "yes" and "no" satisfaction responses relative to the total number is also given for the PM and BC (PM % and BC %).

Table 9: RGM Group Project Satisfaction

Project 1	**PM**	**PM %**	**BC**	**BC %**
Yes	5	100%	5	100%
No	0	0%	0	0%
Total	5		5	
Project 2	**PM**	**PM %**	**BC**	**BC %**
Yes	5	100%	5	100%
No	0	0%	0	0%
Total	5		5	
Project 4	**PM**	**PM %**	**BC**	**BC %**
Yes	6	100%	6	100%
No	0	0%	0	0%
Total	6		6	

Project 5	PM	PM %	BC	BC %
Yes	3	100%	3	100%
No	0	0%	0	0%
Total	3		3	

Table 10: Non-RGM Group Project Satisfaction

Project 3	PM	PM %	BC	BC %
Yes	8	57%	9	64%
No	6	43%	5	36%
Total	14		14	
Project 6	**PM**	**PM %**	**BC**	**BC %**
Yes	9	100%	9	100%
No	0	0%	0	0%
Total	9		9	
Project 7	**PM**	**PM %**	**BC**	**BC %**
Yes	4	44%	6	67%
No	5	56%	3	33%
Total	9		9	

5.3.1. Interpreting Satisfaction Levels

Table 9 shows that the PM and the BC were satisfied with the RGM projects 100% of the time. This is not surprising since the four projects (a) met or were close to their expected delivery dates (see Table 1), and (b) had no "bad" status changes throughout their development lifecycle (see Table 7). Overall, this shows that the projects using the RGM succeeded in meeting customer expectations. The satisfaction data for the Non-RGM group projects is provided in Table 10 and conveys a picture similar to the one formed when analyzing the data for Conjecture 2. That is, Project 6 faired better than the other two projects by garnering 100% satisfaction from both the PM and BC. For the other two Non-RGM projects, however, the BC was frustrated at least 1/3 of the time. An even higher level of frustration is indicated by the PMs for these two projects – they reported being frustrated with the project progress almost 50% of the time.

Hence, based on the data provided in Tables 9 and 10 we can state that (at least for this study)

The RGM promotes an increased level of satisfaction among project managers and business customers.

Anecdotal Note: During post-project reviews for the two RGM group projects (4 & 5), the business customer and other stakeholders stated that their expectations was were always met or exceeded. With respect to RGM methods, the *Silent Parking Lot* approach was judged by those involved as one of the best tools used during requirements elicitation and other meetings. The feeling of "not forgetting" necessary items, as well as "being heard without speaking" was articulated as a major benefit.

6. Conclusion

The data presented in this study contrasting RGM and Non-RGM development efforts has led to a confirmation of three conjectures. That is, the RGM

- helps in decreasing project slippage,
- reduces the detrimental impact stemming from late requirements discovery, and
- reduce the frustration levels for project managers and their business customers.

We further note that the elements of the RGM employed during this study was only a subset of those available – yet even this subset was effective in promoting better requirements definition. We expect that employing the full set of RGM activities and procedures will add additional strengthen and structure the all-too-often *ad hoc* requirements definition processes.

Structuring, monitoring, and controlling the requirements definition process are all goals of the RGM. Although still in its infancy, the RGM represents one additional step towards defining a comprehensive and integrated approach that provides effective support for the requirements generation process.

7. References

[1] J. D. Arthur, M. K. Gröner, K. J. Hayhurst, and C. M. Holloway, "Evaluating the Effectiveness of Independent Verification and Validation," *IEEE Computer*, vol. 32, pp. 79-83, 1999.

[2] B. Boehm, "A Spiral Model for Software Development and Enhancement," *Computer*, vol. 21, pp. 61-72, 1988.

[3] E. Bravo, "The hazards of leaving out the users," in *Participatory Design: Principles and Practices*, D. Schuler and A. Namioka, Eds. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc., Publishers, 1993, pp. 3-11.

[4] M. K. Gröner and J. D. Arthur, "An Operational Model Supporting The Generation of Requirements That Capture Customer Intent," 17th Annual Pacific Northwest Software Quality Conference, Portland OR, Oct. 1999, pp. 286-302.

[5] P. Mambrey and B. Schmidt-Belz, "Systems Designers and Users: Fictions and Facts," in *Systems Design for, with, and by the user*, U. Briefs, C. Ciborra, and L. Schneider, Eds.: North-Holland Publishing Company, 1983, pp. 61-69.

[6] W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," presented at WESCON, 1970.

[7] I. Sommerville, *Software Engineering*, 5th ed. Reading, Mass.: Addison-Wesley Publishing Co., 1996.

Platform Strategy Views and Associated Risks in Product Lines

Dr. Peter Hantos
Xerox Corporation
1300 Midvale Avenue, Suite 606
Los Angeles, CA 90024 USA
+1 310 473 6939
phantos@ieee.org

ABSTRACT

Developing and maintaining a product line of multiple products that are sharing a common, managed set of features one product at a time is no longer economically viable if a multi-project business case exists. The implementation of a strategic reuse process becomes the key factor in achieving fast, efficient, predictable, low-cost and high quality products. In the context of reuse, a product line will represent a product family, with a set of related systems that are built from and leveraging off a common set of core assets. The core set of common, reusable components is called the Platform of the product line.

In this paper we introduce the 5+1 View Model of Platform Strategy to present the key elements of successful Platform-based Product Line Development. Many times platform strategies are initiated and developed strictly on the basis of an asset-reuse and parallel development business case. The 5+1 View Model analysis shows that multiple views and well-defined scenarios are necessary to have a full control over the chosen strategy. The analysis also shows that total risk avoidance as a risk mitigation strategy is not feasible, and part of the success of a strategy model stems from its flexibility, and the tools provided to the decision makers to handle changing circumstances. The included case study, based on the experiences during the development of a black and white digital printer product line, further demonstrates some possible blind spots for managers of product line development.

Keywords: Product Line, Platform, 4+1 View Model, 5+1 View Model, AR (Action Request), Defect Tracking, Risk Mitigation.

AUTHOR'S BIO:

Peter Hantos is Manager, Process Technology and member of the Software Engineering Process Group at the Xerox Office Systems Group. Prior positions included Department Manager for Software Quality Assurance, Software Process Improvement, and System and Reliability Testing. Earlier, as Principal Scientist at the Xerox Corporate Engineering Center, he authored numerous corporate software process standards. He started his professional career in the US as Visiting Professor at the Computer Science Department of University of California, Santa Barbara.

He is currently a member of the P1074 IEEE Workgroup, helping to develop the IEEE Standard for Software Life Cycle Processes. Dr. Hantos is a Member of ACM, Senior Member of the IEEE, and holds M.S. and Ph.D. degrees in Electrical Engineering from the Budapest Institute of Technology, Hungary.

Platform Strategy Views and Associated Risks in Product Lines

Dr. Peter Hantos
Xerox Corporation
1300 Midvale Avenue, Suite 606
Los Angeles, CA 90024 USA
+1 310 473 6939
phantos@ieee.org

INTRODUCTION

Developing and maintaining a product line of multiple products that share a common, managed set of features one product at a time does not seem economically viable anymore if a multi-project business case exists. Under these conditions, the implementation of a strategic reuse process becomes the key factor in achieving fast, efficient, and predictable delivery of low-cost and high quality products. In the context of reuse, a product line will represent a product family, with a set of related systems that are built from and leverage off a common set of core assets. Specifically, the products within a product line share an architecture and common components, or other artifacts such as concepts, plans and processes, to name a few. Traditionally, the word "platform" invokes the image of a foundation. In this paper, I refer to the core set of common, reusable components as the platform of the product line[1]. Also, without emphasizing the fact, the paper is addressing the development of product lines of software, or software-intensive products.

Strategy is the art of devising and employing a careful plan to accomplish a goal[2]. Unfortunately, it is a common mistake to assume that a reasonable definition of the business drivers and documentation of a high-level architecture is enough to yield a viable strategy. Two of the most dangerous assumptions that can plague a product line strategy are (1) that "product line development is really not that different from traditional product development", and (2) that "software is software, regardless of how it is developed…" I hope that this paper provides enough insight to sensitize the reader to these caveats through the discussion of a theoretical model and a specific case study.

Platform Business Drivers

Companies must be able to deliver new products that have real value to customers in different markets in a timely and profitable way. In traditional product development, product releases are mainly serial and independent. Reuse between releases is not designed. Usually, the reuse is opportunistic, and takes place only on the code level. Product variants are introduced in a serial fashion, and so it is difficult to systematically reduce time-to-market. In contrast, during platform development the common components are factored out and developed once. Using the platform as a foundation, multiple products or product

[1] There are many perspectives on how the term "platform" is used. To some extent the meaning attributed is "in the eye of the beholder". Suppliers of solutions might view platforms as an integration framework and a set of components and services. Suppliers of end products might consider a platform to be a modular design concept with standard subsystems. For suppliers of subsystems or components the platform may be a modular design with standard components or parts. Still, other variations of platform terminology exist in the specific hardware, software, and software-intensive system development domains.

[2] The word strategy comes from the Greek "strategos" that literally means "Generalship". The roots are clearly pointing to military use, and in fact even nowadays it is quite popular in management consulting circles to discuss business strategies using military metaphors and quoting famous military strategists/authors like Sun-Tzu or Karl von Clausewitz.

families can be concurrently developed for different markets, and with reduced time-to-market. A product line roadmap is developed as a refinement of the product roadmap in the business plan, outlining each product's value proposition, new features, functions and technologies, as well as both common and differentiating requirements for each product.

Architectural Views
Based on the product line roadmap a large-grained Software Architecture is defined as a basis for software design. The recommended practice for modeling architecture during definition and design is called the "4+1" View Model of Architecture [1]. A detailed discussion of the approach is beyond the scope of this paper, but some fundamental aspects are as follows. During the use of traditional, less systematic architecture definition processes, the applied diagrams are sometimes non-standard, overloaded, and difficult to understand. As a common problem, those attempts at product design do not reflect the specific concerns of the "customers" or "stakeholders" of the process, creating further confusion and digressions during the software implementation phase. The "4+1" View Model consists of the Logical View for describing end-user functionality, the Development View to accommodate programmers and software management, the Physical View to accommodate system engineers, and, finally, the Process View to be used by integrators. The 5th view is called Scenarios, and its purpose is to bring together the other views and assure their integrity. Scenarios are also considered to be abstractions of the most important functional requirements, and in Object Oriented methodology they are called Use Cases [2].

The main idea of this paper to introduce a "5+1" View Model for Platform-based Product Line Development Strategies. The proposed model implements the benefits of the 4+1 View Model of Architecture, but also offers a solution to some shortcomings of the well-known architecture model. The use of these views will highlight the potential risks due to underestimation of certain factors or the unintentional omission of some views. Please note that some of the Views or their aspects are not new and are well documented in the literature. The purpose of this paper is to structure and augment the Views to help organizations that have chosen the platform-based product line approach to development.

PLATFORM TERMINOLOGY

For sake of clarity, a few definitions and interpretations of the most important terms used in this paper are provided below. As indicated previously, the terminology is somewhat subjective, and the overloading or re-interpretation of words is a result of the fact that the number of applicable words is limited.

Platform for Product Lines or Product Families
The platform of a product line is sometimes called a *Product Platform*, and the members of the product line are considered derivatives or variants (*Variant Products*) of the platform. Architecturally, a Product Platform concentrates the common elements of the entire product line (including the expected elements of the non-existent, "to-be-designed" product variants) facilitating an very effective reuse of those assets.

Technology Platforms and Platform Elements
These designations are used to further classify the reusable core components of a Product Platform. Conceptually, the backbone of a Product Platform is a collection of Technology Platforms, which can be further decomposed into Technology Platform Elements. During the evolution of the product line, the introduction, reuse or extension of those Technology Platforms are tightly managed. Please note that in the context of new product development, the term "technology" tends to focus on those technologies in which the organization professes to have a core competency. The elements of these technologies serve as vectors of differentiation from competing companies' products. Other technologies are usually acquired in the form of COTS (Commercial Off The Shelf) components, and their use during the evolution of the product line is managed asynchronously according to different considerations, such as vendor-driven upgrades, licensing conditions, etc. In the case of Xerox, for example, scanning, imaging, and marking represent critical platforms for all multi-function office product lines, while operating systems, networking protocols and stacks, or database systems are also important technologies but are treated as COTS.

PLATFORM STRATEGY VIEWS

Market/Business View
While the factoring of the architecture for reusability is driven by strictly technical considerations, the market segment analysis, market requirements and business objectives play a critical role in the development of the overall product line strategy and delivery processes. As indicated earlier, this View is the most understood. For a good introduction to the business drivers see McGrath's book [3].

Architectural View
The Architectural View is the most straightforward, "original" view of Platform Strategy. It is primarily driven by the implicit potential of code reuse in reducing time-to-market. There is another dimension of Platform-based architectures, valid not only in Product Line, but in single product delivery environments as well. Using platforms is an excellent tool in managing product complexity by attempting to define and implement an optimal coupling structure for system components. The platform approach can help in avoiding multiple, independent implementations of similar functions, and in reducing the system cohesiveness to the necessary minimum. The "4+1" View Model does not include platform-specific practices, just the means to highlight and document software components, their properties, and their interrelationships. Nevertheless, the model is very effective when it is used in conjunction with the earlier mentioned product line roadmap.

Process View
Due to the increased dependencies and parallel development threads, traditional processes are not adequate for delivering product families concurrently rather than sequentially. Product Line development usually can not be described with a single, conventional life cycle model. The recommended approach, an application of Barry Boehm's Anchor Point [3] model, integrates concurrent processes, represented by one or more life cycle models. For further details on Anchor Points see Barry Boehm's seminal article on the subject [4].

Organizational View
Developing and delivering product lines require special and agile organizations. The teams in such organizations are able to quickly "change hats" and work on either platform development, or specific, platform-based product development for the product line.

Political View
Due to the increased number of stakeholders and most likely conflicting objectives, the understanding of the political context in every dimension (platform teams vs. product teams, product line's positioning vs. other products/product lines in the Enterprise Product Portfolio and Marketing Plans, etc.) is critical to the successful implementation of a Platform Strategy.

The "+1" view
This view – similarly to Kruchten's model – is represented by Scenarios. The purpose of this view is to provide a mechanism to verify the integrity of the overall View model of a particular platform strategy.

Due to space limitations, in the rest of the paper we will only focus on the Process, Organizational, Political and Scenario Views of platform-based product line development.

---

[3] Anchor Points are a set of milestones used to plan, organize, monitor and control a project. Anchor Points are positioned at the end of the development phases, and enable a set of interlocking controls, specifications, and organizational standards. Extensive completion criteria is established for each milestone. The criteria at minimum includes completeness, consistency, traceability, testability, and feasibility.

PROCESS VIEW

Development of a Platform-based Product Line

As previously indicated, the source of the time-to-market gain in the case of platform-based product line development is the concurrent development of the various products. In fact, the core assets or "Platform" are being developed simultaneously. The only limitation is that the platform must reach a certain maturity level before it can actually be used in a product. Every team – platform development or product development – will choose the most appropriate life cycle model, and there are no guarantees that they will be the same. The product line management challenge is to synchronize the otherwise asynchronous, and different, parallel development streams, as demonstrated in Figure 1. Please note: for the sake of simplicity all streams are using the same software development life cycle model.

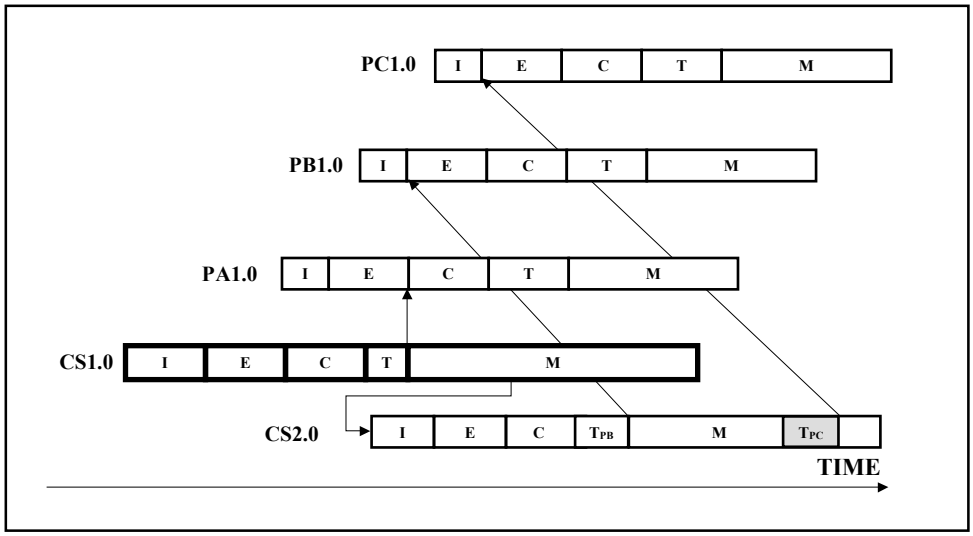


Fig. 1. Platform-based Product Line Development

In this example, the product line consists of three software products (PA, PB, PC,) built on the same platform (CS – Core Software). PA release 1 (PA1.0) is built on the first release of the platform (CS1.0), while the first releases of PB and PC (PB1.0 and PC1.0 respectively) are using the second release of the core software, CS2.0, as their platform. All software is developed using the Rational Unified Process (RUP) [6], an iterative/incremental life cycle model. The development phases are Inception (I), Elaboration (E), Construction (C), Transition (T), and Maintenance (M). The end of the development phases are marked by the earlier mentioned Anchor Points. According to Boehm, Anchor Points represent a stable state of development, and process performance can be uniformly specified and tracked around these milestones regardless of the details and produced artifacts of the underlying life cycle model. Unfortunately, a detailed description of the phases and Anchor Point definitions is beyond the scope of this paper.

During the Inception phase of the Core Software development, the design is centered around the development of a product line, addressing the particular needs of a market segment in a given industry. Based on the understanding and availability of core capability technology platforms and necessary COTS components, an initial version of the product platform architecture is defined. PA1.0 design assumes a certain level of understanding and stability of the platform, but due to market pressures cannot wait until the platform is completely developed and delivered. The conceptual development and elaboration of PA1.0 starts on the basis of the general product line design agreements, without the actual platform code. CS1.0 is not launched to any customer, only released to the PA1.0 development team at the end of the Transition phase. During the Maintenance phase of the platform, based on the PA1.0 feedback from the

field [4], a second release of the platform is initiated. In addition to field results, the recognition of changing market conditions or the emergence of new technologies can also trigger a new platform release. As the diagram shows, Product Teams, developing PB1.0 and PC1.0 pick up this new, enhanced version of the platform at different times.

Maintenance of a Platform-based Product Line

The maintenance of a product line also involves the use of concurrent process streams. In Figure 2 other dimensions of the complexity of the product line processes are depicted. After launch, the products' maintenance only ends when the organization decides about the End of Life (EOL) of the particular product. The Product Teams (independently from each other) decide when they wish to incorporate the new releases of the platform into their products. Double arrows represent the exchange of defect data and related information between the platform team and the product teams.

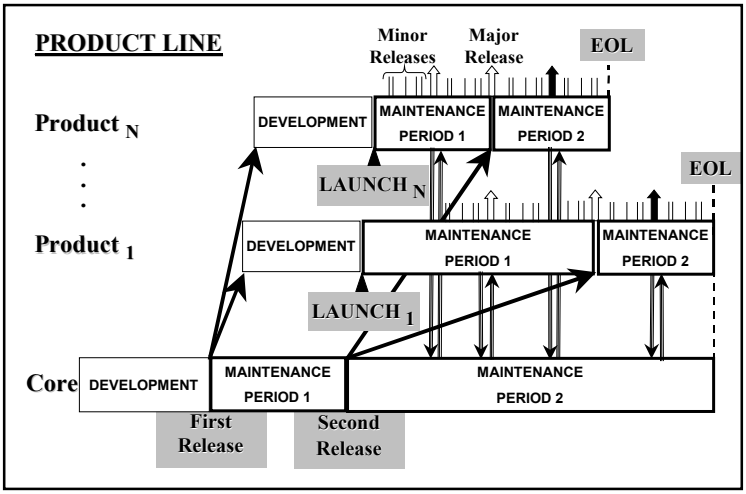


Fig. 2. Maintenance of a Platform-based Product Line

Major Process Challenges

The primary temptation is to achieve the "absolute" time-to-market reduction by developing the core and the variant products totally simultaneously. This level of coupling is highly disadvantageous and risky, due to the immaturity of the core at this early stage. As the example shows, the earliest recommended engagement time for product programs is at the end of the Elaboration Phase of the platform, but later Anchor Points are even more desirable.

The second error that program managers may make is not using Anchor Points for process synchronization, just bolting together the processes at arbitrarily selected times. As indicated earlier, software components developed in an incremental/iterative fashion are by definition unstable between the Anchor Points, and their integration is undesirable.

ORGANIZATIONAL VIEW

Managing platform-based product line development requires a highly flexible, matrixed organization to achieve optimal manpower utilization. The developers are grouped together according to specific competencies, in a functional structure. Platform development is managed by a CDT (Core Delivery Team,) pulling in developers from the appropriate functional areas, such as imaging, networking, operating system drivers, databases, etc., to create Platform Teams around the Technology Platforms of the product line. Other developers within the functional areas support the various PDTs (Product Delivery

[4] Field information may come from marketing or sales entities, focus groups, end-users, field-technicians, independent test-sites, etc.

Teams) in delivering the finished products, and making the augmentations to the core software necessary to satisfy the requirements of their respective customer base.

Major Organizational Challenges
In any process stream, the workload is not uniform, and the actual activity distribution is dependent on the particular life cycle model used. Since the parallel product development process streams are launched independently, proper staffing of platform development and product delivery teams becomes a difficult task. In early stages, the design and development of the platform is a major undertaking, while PDTs are still in the state of conceptual design/inception of their software. When the platform matures, less effort is required to maintain it, and the majority of developers are redirected to support product launch activities.

POLITICAL VIEW

Kruchten has stated in his paper, that the need for multiple architectural views is driven by the fact that a one-dimensional depiction of the architecture will not address the concerns of all stakeholders. Nevertheless, the "4+1" View Model only reflects the differences between selected stakeholders' explicit *work objectives*, and does not address the potential of either overt or covert *clashes* amongst them. In other words, it is clear that end-users, programmers, managers, integrators, and system engineers need different Views and different tools to carry out their tasks, but the perspectives offered to describe the software architecture are not enough to successfully encompass the platform strategy. For complex programs such as product lines, the inclusion of a political angle in the development of strategy can offer a significant reduction of risk exposure. A special challenge associated with organizational, political issues is that they can change during the product line life cycle. Some political considerations are as follows:

Terminology
Just noticing the difficulties I had introducing new, product line and platform concepts in this paper, one can extrapolate the level of difficulty senior managers will face when implementing a platform-based product line delivery strategy. Due to repeated misunderstandings regarding platforms, technology, technology platforms and such it is difficult to convey the strategy, and its corresponding organizational and process changes. Besides lack of understanding on behalf of developers, senior managers also have to face the fact that some people do not want to give up the status quo (Their current role, scope of responsibilities, titles, meetings, etc.). Another source of clashes the different interpretation of what reuse means. Many people have a hard time comprehending for example, that to be appropriate for reuse, software must be developed with the explicit intent of future reuse, tested, and documented according to higher standards. If we move out of the area of small-scale reuse (like for example the application of Rogue-Wave C++ libraries) to the domain-focused reuse ultimately what platform-based development is intended for, the rules change, and opportunistic reuse is simply not acceptable.

Fads and Initiatives
All fads have a kernel of usefulness, but when a fad is over, people tend to throw the baby out with the bathwater, so to speak. The development community might have already tired of other unsuccessful productivity initiatives, and may have to be convinced of the merits of the platform process. A sequence of fads (OOP-Object Oriented Programming, Reuse Initiatives, O-O for Reuse, COTS, combined with quality programs like CMM™ [5], CMMI℠ [6], Six Sigma, and ISO 9000) can totally wear down a development community. Senior management has to gauge the state of the organization, and also understand the dynamics of fads as it relates to their own organization.

[5] The Capability Maturity Model (CMM) is a trademark of the Software Engineering Institute (SEI) at Carnegie Mellon University (CMU), Pittsburgh, Pennsylvania

[6] The Capability Maturity Model Integration (CMMI) is a service mark of the SEI /CMU, Pittsburgh, Pennsylvania

Politics in Marketing
The product line approach from a marketing/business perspective is generally accepted, because of the appeal of developing products simultaneously for several markets. Technical and implementation issues aside, from a marketing/business perspective having product lines is good, and having platforms and pursuing reuse are also good. Nevertheless, from an engineering management perspective, one of the most difficult tasks is to determine if in fact the platform approach is better than the single product approach to deliver a product line.

Organizational Politics
There is no need to detail all the well-known political issues an organization can face, since these issues are not platform strategy-specific. Nevertheless, based on my experience, I would like to highlight an interesting issue. As I discussed earlier, developers have to be moved on demand from platform development to product delivery. It turns out that being part of a successful launch is viewed as an important personal experience, on the other hand successful delivery of a platform release happens relatively quietly and without any visible recognition. This motivational issue created substantial difficulties in managing the developer pool.

THE "+1" VIEW – SCENARIOS

Scenario analysis is one of the oldest tools of strategists, particularly in the military domain [6]. It has also been successfully applied in strategy formulation for emerging technologies in emerging markets [7]. The use of scenarios, or so called use cases in software development was first proposed by Ivar Jacobson in 1992 [2]. Finally I have provided a brief introduction to Kruchten's approach of using scenarios/use cases in defining architecture views [1]. Are these approaches really the same, just used in slightly different contexts? If not, which one(s) would be applicable for platform strategy development? Closer analysis reveals that the common thread is much weaker than we might think. I am providing a comparison on Table 1:

DOMAIN	KEY FOCUS
Military	War games
Business Planning	Mental model of decision makers
OO Design	Requirements gathering and system design
Software Architecture Modeling	Architectural View Model Integrity

Table 1. Application of scenarios in different domains

For platform strategy development, I propose a combination of the applicable focus areas, by introducing the following three guiding concepts to generate and use scenarios.

Requirements Gathering:
Requirements gathering is the very first step in the development of any product. To collect requirements – based on the positive experiences with use cases – ***normal operational scenarios*** should be used. A non-exclusive list of normal operational scenarios is as follows:

- First product release for the product family
- Introduction of new product(s) of the established product family
- Handling defects during development
- Handling defects after launch
- Discontinuing product(s) of the product family
- Discontinuing the product family

Handling Extreme Conditions:
While the War Games approach is appealing, in a highly pressured business environment it is not feasible. There will never be enough time to develop complete contingencies for every possible extreme

scenario, even though that is what an ideal risk management strategy would dictate. The viable solution is to identify and prepare the decision makers to operate during miscellaneous process perturbances. The following sample scenarios are taken from John A. Bers' paper [8]:

- What if product launch is delayed because we cannot perfect the technology in the time frame that we think?
- What if requisite enabling or complementary technologies are delayed?
- What if competitors enter early and aggressively?
- What new technologies could displace ours?

Finally, depending on one's risk aversiveness, optimizing the Platform Strategy can be classified as a normal or extreme operational scenario. For most organizations, defining a Platform Strategy is such an overwhelming task that nobody wants to consider any future changes to the strategy. On the other hand, experiences with platform-based products show that even if a complete overhaul of the strategy is unlikely, planning for periodic evaluation and fine-tuning is a prudent approach.

CASE STUDY

In the rest of this paper specific scenarios related to Action Requests in platform-based product lines are discussed. "Action Request" (AR) is a widely used term for defect reports and non-defect related improvement requests. The case study is based on the experiences during the development of a black and white digital printer product line, which due to special hardware elements such as the print engine, is classified as a software-intensive system. All printers of the product line share a common controller, and the difference between the products of the product line is manifested through different print engines with different speed and finishing options. With respect to action request resolution, the CDT works closely with the PDTs, and periodically delivers a major release of the Platform software, which the PDTs might or might not pick up for immediate upgrade of their products. This sharing and reuse of common components creates some unique problems for product line developers.

The Disposition Dilemma

Deciding how to dispose an AR submitted by any of the PDTs poses special challenges, and requires a close interaction between the various product teams and the core team. Figure 3 shows a simple matrix depicting the decision maker's dilemma. P1-P3 are the products in a product line, and Core represents the corresponding core assets or product platform. "X" is used to indicate that an AR has been submitted against the corresponding product. Please note that product teams can also submit requests to the core team, and in that case the core team's responsibility is to consolidate the request in their own tracking system if appropriate.)

Product Line Elements	Combinations of Related Action Requests					
P1	**X**			**X**	**X**	**X**
P2		**X**		**X**		**X**
P3			**X**		**X**	**X**
Core	*Least Likely*	*Least Likely*	*Least Likely*	*50%*	*50%*	*Most Likely*

Fig. 3. Likelihood of AR Redirection to CDT

If a request impacts only one product in the product line, then it is highly unlikely that the request would be immediately redirected to the Core Team. If similar requests show up in all three products' customer circle, then the solution most likely will involve changes to the core system elements, and the request will be redirected to the core team. If there are some products in the product line that are not affected by the

request, then further analysis might be warranted. The point is that depending on the results of this triage, related requests might be tracked at up to 4 different places, even if are ultimately resolved in only one place. The implications of this disposition are serious. Any change deemed to be resolved in the core assets, could create a regression problem, and in the process of trying to advance one product ahead, we might put all the other products of the product line into jeopardy with an inappropriate fix.

The Reporting and Analysis Dilemma

Usually, teams track their own ARs and provide status of problem resolution during periodic Operations Reviews. In the case of a single product the status is straightforward: "Currently there are 123 Open AR's logged against our product, and the AR trend is as follows…" On the other hand, status reporting in product line environments is not easy. Figure 4 presents a sample aggregate AR table that would be the basis for monthly status reporting. W-1, W-2, W-3, and W-4 represent the four weeks of the reporting period. The numbers in the product rows show the number of ARs logged against the particular product in the reporting period. The numbers in parenthesis represent the number of ARs that were redirected to the core team for resolution.

Product Line Elements	W-1	W-2	W-3	W-4	# of Logged ARs by Teams	# of Actively Worked ARs by Teams
P1	10(2)	10(8)		10	30	20
P2	10(3)		10(5)	10	30	22
P3		10(5)	10(5)	10	30	20
Core	5	13	5	0	23	23
Totals					113	85

Fig. 4. Sample Aggregate AR Table

Some further discussion of the hypothetical disposition assumptions is still needed to fully understand the numbers in the table. In weeks 1 and 2 – while respectively 5 and 13 requests were redirected to the core team – all requests were deemed to be independent, and were registered as such by the core team in their tracking system. In week 3, P2 and P3 both redirected 5 issues to the core team. During further triage, it was determined that the problems were related, and consequently only 5 separate, independent ARs were logged against the core (Gray area in the table.) In week 4 none of the product requests had core implications, hence the 0 in the appropriate row.

It is clear from the example, that in the case of a product line, three different numbers need to be reported to get a clear picture of the status:

(1) The users submitted **90** ARs against the three products of the product line
(2) A total of **113** AR instances are being tracked at any given time by **4** organizations, regardless of the fact that some of them were deemed to be duplicates and redirected to the core team for resolution
(3) In reality, actual work is done only on **85** ARs.

If engineers working on the defects submitted against the core decide to further combine AR's during the fix process, then the total number of actively worked ARs will further decline, increasing the gap between the tracked and worked on ARs.

The reason why these considerations are important is because without understanding these details people simply expect that adding the independently reported numbers from P1-P3 and Core will represent the total number of ARs for the product line. The clarity of these records is also critical from defect analysis' point of view. For example if any kind of statistical analysis is used, it is essential to make sure that analysis is only made on the data-set of the actively worked ARs.

After fixing a problem in the core software, regression tests have to be conducted with all the affected products and the appropriate product AR records have to be updated.

The main lesson from the example is that in a robust AR tracking system that is appropriate for product line development we have to provide full traceability, and the ability to record the AR state-changes during the disposition and resolution of the request.

SUMMARY

By developing the 5+1 View Model of Platform Strategy we demonstrated the key elements of successful Platform-based Product Line Development. Many times platform strategies are initiated and developed strictly on the basis of an asset-reuse and parallel development business case. The 5+1 View Model analysis shows that multiple views and well-defined scenarios are necessary to have a full control over the strategy. The analysis also shows that total risk avoidance as a risk mitigation strategy is not feasible, and part of the success of a strategy model stems from its flexibility and the tools provided to the decision makers to handle changing circumstances. The included case study further demonstrated some possible blind spots for managers of product line development who defined their processes only on the earlier mentioned, limited asset-reuse concepts.

REFERENCES

[1] Kruchten, P., "Architectural Blueprints – The '4+1' View Model of Software Architecture", IEEE Software, November 1995

[2] Jacobson, I. & Christerson, M. & Jonsson, P. & Overgaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992

[3] McGrath, M. E., *Product Strategy for High-Technology Companies*, McGraw-Hill, 1995

[4] Boehm, B. W., "Anchoring the Software Process", IEEE Software, July 1996

[5] Kruchten, P., *The Rational Unified Process, An Introduction*, Second Edition, Addison-Wesley, 2000

[6] Deweerd, H.A., "Political-Military Scenarios," RAND Corporation, Document No. P-3535, 1967

[7] Lynn, G. S., Morone, J., Paulson, A., "Emerging Technologies in Emerging Markets: Challenges for New Product Professionals," Engineering Management Journal, vol. 8, no. 3, September, 1996

[8] Bers, J. A., Lynn, G. S., Spurling, C. "Scenario Analysis: A Venerable Tool In a New Application: Strategy Formulation for Emerging Technologies in Emerging Markets", EMJ 97 report, <<< http://mot.vuse.vanderbilt.edu/mt242fl/Final/emjpaper.doc>>>

Design for Testability

Bret Pettichord
bret@pettichord.com
www.pettichord.com

Abstract

The paper provides practical suggestions that will inspire teams to make their software products more testable. It cites examples of testability features that have been used in testing various software products, including verbose output, event logging, assertions, diagnostics, resource monitoring, test points, fault injection hooks and features to support software installation and configuration. It also hopes to change some common prejudices about testability. It argues that all automation requires some instrumentation and that all automated testing must differ, to some degree, from actual customer usage. Wise automators will seek to manage these differences, rather than avoid them. It also provides a detailed exploration of how testability issues affect GUI test automation and what kinds of provisions testers must make when testability is lacking. It argues, therefore, that many of the objections to creating test interfaces are often overstated. The paper concludes with a discussion of how testers can work with product developers to get important testability features built into software products.

Bret Pettichord helps software teams with their testing and test automation. He is co-author of *Lessons Learned in Software Testing*, a columnist for Stickyminds.com, and the editor of TestingHotlist.com. Based in Austin, Texas, he is the founder of Pettichord Consulting and provides consulting and training throughout North America. He is currently developing methods for testing with Extreme Programming and other agile methods.

Design for Testability [*]

Bret Pettichord
bret@pettichord.com
www.pettichord.com

To be presented at the
Pacific Northwest Software Quality Conference, Portland, Oregon, October 2002

1. Introduction

This paper provides practical suggestions that will inspire teams to make their software products more testable. It also hopes to change some common prejudices about testability.

My interest in testability stems from my experience automating software tests for over a decade. Time after time, the success of test automation projects has hinged on testability issues. Automation sometimes has required subtle software changes. In other circumstances, the entire testing strategy depended on interfaces or facilities fortuitously provided by the product architecture. On the other hand, when the software wasn't well adapted for testing, automation was disappointing, forcing the team to depend on manual testing. Other authors have also noted the importance of testability for test automation.

I was slow to realize the importance of testability for test automation (Pettichord 1999). I'd already identified three keys to test automation: (1) a constructive relationship between testers and developers, (2) a commitment by the whole team for successful automation and (3) a chance for testers to engage early in the product development cycle (Pettichord 2000). These turn out to be exactly what is needed to get testability designed into a product. You need (1) cooperation from developers to add testability features, (2) a realization that testability issues blocking automation warrant attention from the whole team, and (3) a chance to uncover these challenges early, when the product is still open for design changes. Indeed, teams often find that it is more efficient to build testability support into a product than to construct the external scaffolding that automation would otherwise require.

Testability goes by different definitions. This paper defines testability 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

[*] I thank the following for comments on early drafts: Noel Nyman, Johanna Rothman, Marge Farrell, Ross Collard, Brian Marick, Sam Guckenheimer, Phil Joung, Melora Svoboda, Jamie Mitchell, Nancy Landau, Dawn Hayes, Ken Pugh and anonymous reviewers provided by PNSQC. This paper benefits from prior presentations at the Test Automation Conferences in San Jose (March 2001) and Boston (August 2001) and from discussions at the 2nd Austin Workshop on Test Automation (January 2001; participants: Alan Jorgensen, Allen Johnson, Al Lowenstein, Barton Layne, Bret Pettichord, Brian Tervo, Harvey Deutsch, Jamie Mitchell, Cem Kaner, Keith Zambelich, Linda Hayes, Noel Nyman, Ross Collard, Sam Guckenheimer and Stan Taylor) and the 6th Los Altos Workshop on Software Testing (February 1999; participants: Ned Young, Jack Falk, James Bach, Jeff Payne, Doug Hoffman, Noel Nyman, Melora Svoboda, Chris Agruss, Brian Marick, Payson Hall, Bret Pettichord, Johanna Rothman, Dave Gelperin, Bob Johnson, Sandra Love, Cem Kaner, Brian Lawrence and III).

or place it in specified states. These are the aspects of testability that are affected by the software design and that have greatest impact on the feasibility of automated testing. Ultimately, testability means having reliable and convenient interfaces to drive the execution and verification of tests. The appendix discusses other definitions of software testability in the literature.

Many teams are frustrated trying to get the testability features they want. Testers don't expect their requests to be met. Programmers are often unaware of what features would help testers the most. This paper concludes by suggesting strategies for bringing testers, programmers and managers together in supporting testability.

A convenient distinction would be to distinguish between testability—test support included in the software product code—and test automation—test support external to the product. It would be convenient, but as this paper will illustrate, it is often a difficult line to draw. Traditional test automation approaches actually rely on hidden testability implants. Monitors may be inserted into either the product code or the operating system; either way they provide testability. Indeed, as you move away from the notion of an independent, arms-length testing team—as this paper encourages—it becomes harder to draw a clear line between automation and testability. This paper argues that close cooperation between testers and programmers is a key for testability. But it also hopes to facilitate that cooperation by providing a catalog of specific features that product teams have used to make software more controllable by or more visible to testing. These include test interfaces that facilitate automation and diagnostic features that facilitate verification and analysis. It's a long list, though not exhaustive. Use it to generate ideas your team can use for your product.

After discussing some basics, this paper visits various software features and modifications that have improved testability. These include logging events, providing diagnostics, monitoring resources, facilitating configuration, and test interfaces. I've been collecting examples of testability features included in software products for some time. These examples come from various sources: conference talks, magazine articles, workshop reports and my own experiences. One goal of this paper is to simply collect and organize these notes in one place.

There's been some controversy regarding the merit of using test interfaces as opposed to testing via the user interface. This paper presents several arguments for seriously considering creating test interfaces. It then provides a detailed discussion of graphical user interface (GUI) test automation and the testability issues that often affect it, and the alternatives that are available when testability isn't available. This discussion should be helpful to GUI automators, but it is also provided to dismiss some of the misconceptions regarding the ease and veracity of automated GUI testing.

2. Visibility Basics

Visibility means the ability to observe the outputs, side effects and internal states of the software under test. It's a behavioral attribute: it refers to the software when it runs dynamically.

But a more fundamental aspect of visibility is simply having access to the source code, design documents and the change records. This is clearly a prerequisite for most testability improvements. The change records are stored in the configuration management system. They

indicate what changes were made to what files by whom and when. Conscientious programmers will attach additional comments indicating the purpose of the change. Many configuration management systems can be set up to automatically send out email when changes are checked in.

Not only do testers need access to this information; they need to know what to make of it. At least some of the testers on your team need to know how to read code and how to interpret the design modeling languages used. Nowadays these often include sequence diagrams, activity diagrams (née flowcharts), class diagrams, and entity relationship diagrams. (Guckenheimer 2000, Adams 2001) Before testers can ask for test interfaces, fault injection hooks or other testability features, they need to have a basic understanding of system design, what interfaces are already available and where testing hooks might be placed. Testability requires collaboration between testers and developers using a common language.

3. Verbose Output

The verbose modes found in many programs are good examples of testability features that allow a view into the software operation. These are especially common with Unix commands and programs.

Consider the Unix mail command. It's a command line program that takes an email address for an argument, and then prompts for a subject line and the body of the message. If you're not using the verbose mode, that's all you'll see. But when you use the "-v" switch with the command, the mail program will display its communication with the local mail server. An example is shown in Figure 1.

This example shows a short email message being sent by wazmo@io.com to bret@pettichord.com. The first four lines show the interaction between the mail client program running on the machine named "eris" and the user known as "wazmo".

The remainder is the verbose output. The local system (eris.io.com) is sending the email to the local mail server, which in turn will take care of delivering it to pettichord.com. The verbose output shows the communication between eris and the local mail server (deliverator.io.com). The lines prefixed by ">>>" are the commands sent by eris to deliverator. The lines prefixed by three-digit numbers are deliverator's replies. (This example is based on DiMaggio 2000)

There are several ways that you might take advantage of this verbose mode. For one, this information could help you if you were testing the mail client. It could also be useful if you were testing a mail server (in this case, an SMTP server). The verbose mode might tip you off that the server is not responding correctly. It could also be used to help debug mail configuration problems. If mail isn't getting through, the verbose mode could help expose where things are going wrong.

Verbose mode also helps you learn about how this communication between mail client and server (i.e. the SMTP protocol) is supposed to work. The communication protocol is actually just text being passed back and forth. That's what the verbose mode is exposing. After using the verbose mode to learn how it usually goes, you might want to use the telnet program to communicate directly with the mail server and see how it handles command variations. This would allow you to test the error handling of the mail server.

Figure 1. Unix Mail's Verbose Mode

```
mail -v bret@pettichord.com
Subject:   testability example
           Sample text.
.
Cc:
bret@pettichord.com... Connecting to mx.io.com. via relay...  220-deliverator.io.com
           ESMTP Sendmail 8.9.3/8.9.3; Fri, 12 Jan 2001 15:34:36 -00 220 Welcome to
           Illuminati Online, Fnord!
>>> EHLO eris.io.com
250-deliverator.io.com Hello IDENT:wazmo@eris.io.com [199.170.88.11], pleased tu
250-8BITMIME
250-SIZE 5000000
250-DSN
250-ONEX
250-ETRN
250-XUSR
250 HELP
>>> MAIL From:      <wazmo@eris.io.com> SIZE=67
250 <wazmo@eris.io.com>... Sender ok
>>> RCPT To:        <bret@pettichord.com>
250 <bret@pettichord.com>... Recipient ok
>>> DATA
354 Enter mail, end with "." on a line by itself
>>> .
250 PAA07752 Message accepted for delivery bret@pettichord.com... Sent (PAA07752
           Message accepted for delivery) Closing connection to mx.io.com.
>>> QUIT
221 deliverator.io.com closing connection
```

4. Logging Events

A verbose mode is one technique for logging software events. Logs help testing by making the software operations easier to understand. They also detect bugs that might be difficult to notice otherwise. When bugs are found, logs can help focus on the faulty code and help with debugging.

However, you can easily log too much. Overburdened with detail, logs may become hard to decipher. And they may get very long—too long to read and possibly taking up too much storage, causing problems themselves. Logging excessive detail can also burden system performance.

Errors are probably the most important things to place in the logs. But one of the purposes of logs is not just to record that things went wrong, but also what happened beforehand. Like a flight recorder, software logs can uncover the events that lead up to the failure. Too often

programmers are unable to fix bugs because they don't know how to replicate them. Good logs contain the information needed to replicate.

To address these concerns, Marick (2000) has developed several heuristics for determining what should be logged and how log storage can be managed. One recommendation is to log user events:

> For purposes of most programmers, a UI event is something that provokes messages to other subsystems. Something that happens solely within the UI layer is not of enough general interest to justify its space in the log. For example, most programmers do not want to know that a particular mouse button had been clicked somewhere in a dialog box, but rather that the changes of the whole dialog had been applied. (Marick 2000: 8)

Other events to consider logging include the following:

- Significant processing in a subsystem.
- Major system milestones. These include the initialization or shutdown of a subsystem or the establishment of a persistent connection.
- Internal interface crossings.
- Internal errors.
- Unusual events. These may be logged as warnings and a sign of trouble if they happen too often.
- Completion of transactions.

A ring buffer is a data structure that is filled sequentially. When the data written reaches the end of the buffer, additional entries overwrite whatever was written at the beginning. Ring buffers are often used to store logs. Because they have a fixed size, you don't have to worry about them filling all available storage. Marick recommends saving a copy of the ring buffer whenever an error occurs.

Log messages should include timestamps and identify the subsystem that produced them.

Using variable levels of logging helps keep logs manageable. When a debug or verbose mode is enabled, more detailed messages are logged. Because of the potential volume, it is often desirable to be able to turn on debugging on a subsystem-by-subsystem level.

When making use of standard components, such as web servers, application servers or databases, you can probably take advantage of the logging mechanisms already built into such components. For these, you often must enable or configure the logging feature. Indeed most operating systems have logging services that may already be logging information you need.

Commercial tools that capture execution history when errors occur provide another approach to logging. These tools, including RootCause by OC Systems and Halo by InCert, work by instrumenting the executables when they are loaded. (Rubenstein 2002)

Once you have some sort of logging in place, how do you best make use of it? When I'm testing a new system, one of the first things I do is review any logs that it generates. I'll try to understand how the logs correlate to the actions I'm executing on the software. I do this to learn how the different parts of the system interact and also to understand what a normal log looks

like. This can then serve as a baseline for later testing, when I'm seeing errors or unusual messages appearing in the logs.

What else can you do with logs? Johnson (2001) and others provide several suggestions.

- Detect internal errors before they appear at the system level.
- Find out how to reproduce bugs.
- Learn about your test coverage. What messages haven't your tests yet triggered?
- Learn about customer usage. What features do they use the most? What kinds of mistakes do they make? Focus your testing on what customers actually do.
- Determine which customer configurations are most common and focus testing on them.
- Uncover usability problems. Are there certain pages that lead to a high level of user abandonment?
- Understand load patterns and use these as a basis of setting realistic load test objectives.

5. Diagnostics, Monitoring, and Fault Injection

Diagnostics are techniques for identifying bugs when they occur. *Monitors* provide access to the internal workings of the code. Event logs are an example of a type of monitor. Diagnostic and monitoring features make it easier to detect bugs when tests encounter them. Some bugs don't immediately lead to obvious failures. Diagnostics and monitoring help uncover them. They also help testers understand software behavior and therefore think of more good tests. *Fault injection* is a technique for testing how software handles unusual situations, such as running out of memory. Effective fault testing often requires testability hooks for simulating faults.

A common diagnostic is an assertion. Code often makes assumptions about the data that it works with. *Assertions* are additional lines of code that make these assumptions explicit. When assertions are violated (i.e. assumptions prove untrue), errors or exceptions are automatically raised. Assertion violations should not normally occur; when they do, they signal a bug. For example, code for computing a square root could include an assertion to check that the input was non-negative. If not, it would abort.

Without assertions you might not notice when a bug occurs. Internal data may now be corrupt, but not result in a noticeable failure until further testing accesses the data. Assertions also help isolate the locations of bugs. Assertion violations typically report the location of the assertion code.

Recent interest has focused on a special kind of assertion, known as a probe. A *probe* is like an ordinary assertion, checking a precondition of a function, except that it handles failures differently. Ordinary assertions typically abort when an assertion fails, usually with a little error message. Probes continue running, placing a warning in a log. Or at least they give you the option of continuing. From a testing perspective, either works for finding bugs.

Classic practice includes assertions in debug code that is tested internally, but not deployed to the customer. However, more and more, developers are including assertions (or probes) in the delivered executables. A common objection to this is that assertions may impair

performance. Although, this could happen, this isn't the best way to address performance. Various studies have shown that even good programmers are poor at predicting which parts of their software take the most time. Performance improvements need to be made with the use of a profiler, a tool that measures the amount of time spent executing different parts of the code. No one should assume that assertions in general will hurt performance. Useful references for how and when to use assertions include Fowler (2000: 267 - 270), Payne et al (1998) and Maguire (1993).

Design by contract is a design technique developed by Bertrand Meyer. In effect, it specifies how to use assertions to define a class interface. A helpful overview is contained in Fowler (2000a: 62-65) and a detailed exposition is contained in Meyer (1997: Ch. 11).

Another strategy for detecting invalid data is to write a program that walks through a database looking for referential integrity violations. It is, of course, possible to configure databases to automatically enforce integrity rules. But this is sometimes avoided because of performance issues. Right or wrong, it opens up the likelihood for data integrity bugs. A diagnostic program that searches for such problems can help isolate these bugs and detect them earlier. Such a diagnostic program may be included with the product, to help users isolate problems in the field. Or it might simply be used as an in house testing tool.

A technique used to help find memory errors in Microsoft Windows NT 4.0 and Windows 2000 is to modify the memory allocation algorithm to put buffers at the end of the allocated heap and then allocate backwards into the heap. This makes it more likely that buffer overrun errors will cross application memory boundaries and trigger a memory error from the operating system. A similar technique can be used with the recent Visual C++ .NET compiler. Its /GS switch adds a code to the end of each buffer allocated by your application. Any code that overwrites the buffer will also overwrite the /GS code, resulting in a new stack layout that can be easily detected (Bray 2002).

Effective methods for finding memory leaks require monitoring memory usage. Otherwise you won't notice them until you run out. Some applications have embedded memory monitors that give a more detailed view of the application memory usage (Kaner 2000). In other cases, teams have used commercial tools such as Purify or BoundsChecker.

Testing is often easier when you can monitor internal memory settings for inspection. A good example of this can be seen by entering "about:config" in Netscape. It will dump out all the configuration settings. It's a great tool for tracking down the source of configuration problems. (I used it while writing this paper.) This kind of output is particularly helpful for problems that only occur on particular machines.

Applications running on Windows 2000 or XP can use a special test monitor called AppVerifier. While you are testing your application, AppVerifier is monitoring the software for a number of targeted errors, including: heap corruptions and overruns, lock usage errors, invalid handle usage, insufficient initial stack allocation, hard-coded file paths, and inappropriate or dangerous registry calls. The AppVerifier includes monitoring, fault injection and heuristics for error detection (Phillips 2002, Nyman 2002.

Sometimes testers need access to internal data. Test points are a testability feature that allows data to be examined and/or inserted at various points in a system. As such they allow for

monitoring as well as fault injection. They're particularly useful for dataflow applications (Cohen 2000). Mullins (2000) describes a product that could have benefited from test points. He tested a system composed of multiple components communicating over a bus. One of the components was sending bad data over the bus. This went unnoticed because other components screened out bad inputs. All but one, that is. The faulty data lead to a failure in the field when this component failed due to a particular bad input. Observing the data stream during testing through test points would have detected this fault before it resulted in a component failure.

Fault injection features assist with testing error-handling code. In many cases, environmental errors are exceptional conditions that are difficult to replicate in the lab, especially in a repeated and predictable way (which is necessary for automated testing). Example errors include disk-full errors, bad-media errors or loss of network connectivity. One technique is to add hooks for injecting these faults and triggering the product's error-handling code. These hooks simulate the errors and facilitate testing how the program reacts. For example, in testing a tape back-up product, it was important to check how it handled bad media. Crumpling up sections of back-up tape was one way of testing this. But we also had a hook for indicating that the software should act *as if* it found bad media on a specified sector of the tape. The low-level tape-reading code acted on the hook, allowing us to test the higher-level code and ensure there was no data loss in the process. (It would skip the bad section of tape, or ask for another.) Houlihan (2002) reports using a similar technique in testing distributed file system software.

Another technique for fault injection is to use a tool, such as HEAT or Holodeck, that acts as an intermediary between the application and the operating system. Because of its position, it can be configured to starve the application of various system services. It can spoof the application into acting as if the system were out of memory, the disk were full…remove network connectivity, available file system storage and the like. (HEAT and Holodeck are included in a CD accompanying Whittaker 2003.). Note that fault injection is a testing technique that depends on either adding code to your product (testability) or using an external tool (automation). If you don't have a tool for the faults you want to inject, you may find it easier to build in the necessary support than build an external tool.

6. Installation and Configuration

This paper reviews several examples of how software can be made easier to test: sometimes by building and customizing external tools; sometimes by building support for testing into the product. Sometimes it's a blurry distinction, with external code later incorporated into the product. The same holds for testing software installation and configuration. Testers often spend a significant amount of time installing software. Tools and features that simplify installation often provide effective means for automating testing.

Making the installation process scriptable can yield significant advantages. Install scripts can automatically install the software on multiple machines with different configurations. Many software products use InstallShield to create their installers. InstallShield installers feature a built-in recorder that stores installation options in a *response* file. Testers can use response files to drive unattended installations. These response files have an easy-to-read format, making them easy to modify with different location paths or installation options. (InstallShield 1998). Microsoft Windows also comes with support for scripting unattended installations. These

scriptable interfaces amount to test APIs for the installers. They are useful regardless of the specific installer technology used.

Installers should reliably and clearly report whether any errors occurred during installation. Some products have installers that generate detailed installation logs but no single indicator of success, suggesting that users search through the log to see if it lists any errors. Of course testers must check this as well. While I was testing such a project, other testers wrote scripts to parse through the logs, collecting lines that began with the word "error". I worried that we might miss errors that were labeled differently in the log. The product used multiple installation modules; it was risky to assume they were all using the same error reporting conventions. Far better would have been to expect the product to define its own mechanism for determining whether an error had occurred. Including this logic in the product (rather than in the test automation code) would likely result in a more reliable design, would lead to it being tested more thoroughly and seriously, and would also provide value to the users.

Many software products store persistent data in a database. Testers need to be able to control and reset this data. Whether reproducing bugs or running regression tests, testers must be able to revert the database to a known state. Realistic testing often requires preloading a test bed into the database. These testing requirements define the need for products to support configuring and initializing test databases and automatically loading test data in them. Even products that don't use databases can benefit from routines to automatically pre-load data.

Another feature that facilitates testing is the ability to install multiple versions of a software product on a single machine. This helps testers compare versions, isolating when a bug was introduced. Networked application testing is particularly aided by the ability to install multiple product instances on a single machine. This aids testing deployment configurations and products scalability by allowing the full use of the available lab resources. Typically this requires configurable communication ports and provisions for avoiding collisions over resources like the registry. Because installed products are nowadays more likely to wired into the operating system, supporting multiple installations is sometimes difficult. Another approach is to use products like vmWare that allow multiple virtual machines to run on a single physical machine.

7. Test Interfaces

Software testing occurs through an interface. It might be a graphical user interface (GUI). It might be an application programming interface (API), a component interface (e.g. COM, ActiveX, or J2EE), a protocol interface (e.g. XMI, HTTP, SOAP), or a private debugging or control interface. Some interfaces are meant for humans, others for programs.

Manual testing is often easiest using a human interface. Automated testing, as this paper argues, is often easiest using programming interfaces. In particular, programming interfaces to core business logic and functionality are significant testability features, providing control and visibility.

The likelihood and willingness of development teams to provide test interfaces varies considerably. Some see it as a necessity; others don't see the point. Some will even create them without notifying testers, using them for their own debugging and integration testing. Programming interfaces often support automated testing earlier and with less effort than required

when testing via a GUI. Because of this, I've often recommended the use and development of programming interfaces for testing. This is admittedly a controversial suggestion. Some testers have said that programming interfaces they've had access to were less stable than the user interfaces and therefore less suitable for testing (see Szymkowiak's comments in Kaner et al, 2001: p 119, note 5). Clearly test interfaces need to be stable or else tests that use them will be worthless.

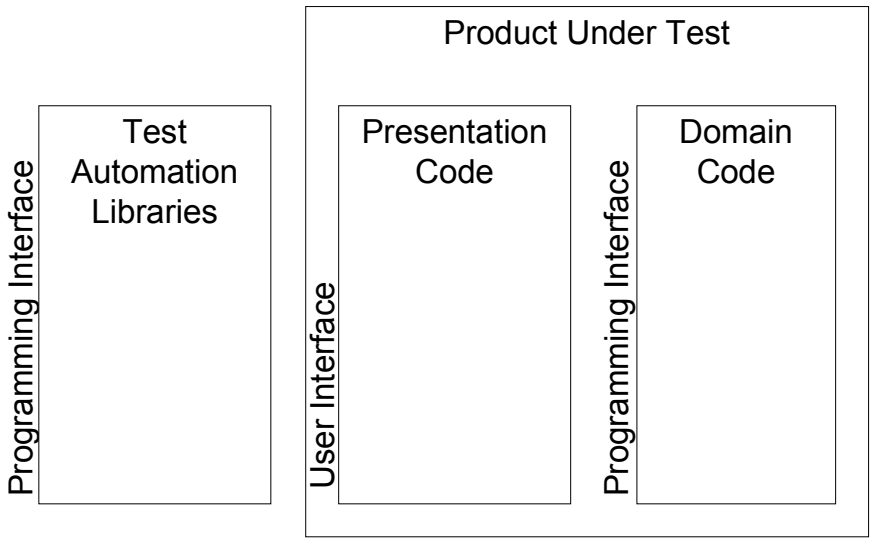
Unconfirmed reports indicate that the original version of Excel included a test interface. Because mathematical accuracy is a key requirement for spreadsheet software, the ability to run lots of automated tests frequently was essential. The test interface that was provided to Excel was later exposed to users and is now accessible through its Visual Basic interface. Thus did a testability feature eventually develop into a popular customer feature.

The value of test interfaces has even been clear to public-domain game developers. Xconq is a public-domain game. It was developed with a game engine and separate interfaces for different platforms. One of its interfaces is a command-line interface specifically provided to facilitate testing (Xconq). If public-domain games can have their own test interfaces, then commercial business software should be able to get them as well.

Many business software products have included customer-focused APIs which testers have made use of. Examples include AutoCAD, which has interfaces in several languages, Interleaf, which includes a Lisp interface, and Tivoli, which has a command-line interface as well as CORBA bindings. In these cases, testers exercised significant business functionality through the APIs.

Some have questioned the idea of testing via a different interface from what customers will use. They question the *veracity* of the tests: will these tests find the same bugs that users can be expected to encounter?

Figure 2. Programming Interfaces Available for Testing



A software product can often been divided into the core business logic or key functional—we'll call this the domain code—and the code that interacts with the users—we'll call this the presentation code. Sometimes the boundary between these two layers is clearly defined. In other cases it's rather murky. Ideally, the presentation code calls a clearly defined programming interface to the domain code. As Figure 2 makes clear, if you test via the internal programming interface, you won't be covering the presentation code.

GUI test automation requires a means to describe the tests. At some level, this amounts to an interface that allows testers to describe what buttons to press and what text to type. Automated test scripts are programs, and the language they use to specify the tests amounts to a programming interface. This programming interface is supported by the test tool, its libraries, and customization code that the automators may have provided. This automation support code often also includes maintainability support (to make it easier to revise the tests when the inevitable changes are made to the user interface) (Kaner 1998).

The figure illustrates that testing via the test automation libraries (an external programming interface) covers the presentation code. It also "covers" the automation libraries themselves. This isn't code you care to have covered; problems in it can lead to false alarms. *False alarms* are tests that fail because of a problem with the test apparatus or environment, and not a problem with the software under test. (Some people call these *false positives*, but I find this term confusing, as they feel rather negative.)

Products that will be tested via an internal programming interface are sometimes designed with a thin GUI. This is a minimal layer of presentation code. This approach also requires assurances that that presentation layer calls the same internal interface as the automated tests. (Fowler 2001)

My view is that either approach—automated GUI testing or API testing—exercises the product somewhat differently from the way an actual user would. Because of these differences, and because potential errors in either test approach could cause bugs to go undetected, either approach should be supplemented with manual testing. Both are valid strategies. The API approach, however, is often simpler and more productive. Concerns regarding its effectiveness are often over-stated.

8. User Interface Testability

GUI test automators know that testability issues often bedevil automation. This section provides an overview of GUI test automation and the testability issues that automators often encounter.

This paper devotes considerable space to these issues in order to demonstrate two points. One claimed benefit of GUI automation is that it tests the way a user does. Obviously this isn't true for API-based testing (of products that are primarily used through a GUI). However, automated GUI testing also often requires deviations from actual user behavior. This does not indict GUI automation; rather it demonstrates that any automation strategy—GUI or not—must contend with issues of veracity.

The second point is that GUI automation, like API testing, often requires testability modifications. Some think that mixing test code with product code undermines the integrity of the testing. I don't believe this. In any case, this paper will demonstrate that GUI test automation

is only possible with some amount of instrumentation. Sometimes it happens rather unobtrusively. In other cases the demands are fairly large. Once you accept the necessity of changing the product to support testing, I hope you will consider the many testability ideas cataloged in this paper.

Modern graphical user interfaces are composed of windows that contain multiple controls. Windows, also known as dialogs, typically can be moved or resized and have title bars across their tops. Controls, also known as widgets, present information and accept inputs. Common controls include text fields, push buttons, menus, and list boxes. These interface elements are common whether the product runs on Microsoft Windows, Macintosh, Linux or Unix.

Web-based applications have similar elements. The various pages of a web application correspond to windows. Pages allow all the standard controls to be embedded in them. Some web applications use Java or ActiveX to give them the feel of standard user interfaces.

GUI test tools are scripting frameworks with GUI drivers. Their GUI drivers allow them to identify controls, record user events affecting controls, and trigger events targeted at specified controls. User events include entering text in a text field, clicking a button or selecting an item from a menu.

Product developers use GUI toolkits and libraries to build product interfaces. These contain the code that creates the windows and controls in the interface. Some will use the libraries that are provided with the operating system or browser libraries. Others will use libraries provided with their development tools or obtained from outside vendors. Still others build their own.

A common problem that GUI test automators face is custom controls. *Custom controls* are controls that are not recognized by a given GUI test tool. The following sections outline several strategies that automators have used to make custom controls testable. Sometimes this requires tool customization. Sometimes it requires testability changes to the code.

Some examples of custom controls that have been reported by students in my seminars include:

- Grids, or tables, with other controls embedded in them, such as drop-down list boxes.
- Tree views, also known as explorer views. These are lists that allow some items to be expanded to show more items “under” them.
- Lists in Delphi, a development environment.
- 3270 emulators used to access mainframe applications.
- Powerbuilder, another development environment.
- Icons that appear in an interface.
- List boxes with specialized content, e.g. graphics.
- Flow graphs used to display dependencies between application objects.

The process of assessing and ensuring testability of your product with a given GUI test tool is thus:

1. Test compatibility of your test tool and development tools early.
2. Define standards for naming user interface elements.
3. Look for custom controls. If there are any, plan to provide testability support.
    a. Ensure that the tool can recognize the control as an object with a name, class and location.
    b. Ensure that the name used by the tool to identify the control is unique. It won't do if multiple windows or controls are given the same identifiers.
    c. Ensure that the tool can verify the contents of the control. Can it access the text of a text field? Can it determine whether a check box is checked? (This step doesn't apply to some controls, like push buttons, that don't have contents.)
    d. Ensure that the tool can operate the control. Can it click a push button? Can it select the desired item from a menu? (This step doesn't apply to some controls, like static text fields, that can't be operated by the user.)

The following sections provide a detailed examination of these processes. Along they way this paper highlights several testability issues faced by GUI test automation:

- A lack of standards makes GUI testability often a matter of trial and error.
- GUI control recognition technology requires instrumentation. The difficulty of doing this varies.
- GUI test automation requires programmers to adhere to naming standards.
- Custom controls represent a major challenge for test automation. Many techniques are available, but it's often a matter of trial and error to determine which will work. Successful techniques often reduce veracity or require calls to testability interfaces.

9. User Interface Standards and Compatibility

The lack of standard testing hooks into user interface technologies has often made compatibility a major issue for GUI test automation. This is one reason why test tools must often be hand customized, and why GUI test automation must often resort to workarounds that reduce veracity.

When selecting a GUI test tool, you must ensure its compatibility with the tools and technology you're using to develop your GUIs. Just reading product literature isn't enough: claims are usually vague, incomplete or out-of-date. You have to actually try the tools out to verify whether a particular test tool will be able to execute and verify the interfaces created by the development tool. Occasionally teams use development technology for which no compatible test technology exists. They end up having to build extensive test tool customizations or else forgo automated testing entirely.

Standards for user interface testability have been slow to develop. Indeed, interface development technology vendors have been slow to acknowledge the importance of testability. For example, a comprehensive guide to selecting user interface development tools cited 66 issues to consider in the areas of usability, functionality, flexibility, portability, support and cost without any mention of testability (Valaer & Babb, 1997).

Even when they have an interest, programmers often have a hard time ensuring their user interfaces will be compatible with specific tools. All GUI test tools make assumptions of how the windows and controls in the interface will be structured. Their GUI drivers expect to be able to recognize and operate controls using a set of standard protocols or messages. Programmers who want to make compatible controls need to know these assumptions but may have a hard time getting this information from test tool vendors. Some vendors claim this information is proprietary and refuse to disclose it. Others only release it contingent to non-disclosure agreements. It may take several calls just to find out what a vendor's policy is.

For several years I was hopeful that some standardization and disclosure would reduce the compatibility problems plaguing GUI test automation. But it seems that the test and development tool vendors lack the motivation to work together. There has been some improvement, however. For one, there has been some consolidation in user interface technologies. With fewer choices, it's easier for tool vendors to cover them. Secondly, the development of standard component technologies (J2EE, .NET, COM, VB) allows tool vendors to provide the glue needed to make their tools easier to customize. The availability of reflection has been particularly helpful. But a lack of a standard for method signatures of GUI controls ensures that automators will have to do some hand stitching when using custom and new technologies.

Another problem is that as standard computer graphical interface technologies mature and are made more testable, more and more applications are moving to thinner client technologies: they are using web browsers, often on handheld computers or cell phones which again lack the testability support for smooth automated testing.

Thus gaps remain. Testers continue to encounter product interfaces and controls that require specific customization before the GUI test tools will operate. Handling these issues often calls on significant skill and time from test automation experts (Hayes 2000, Zambelich 2001).

10. Recognizing User Interface Controls

GUI test tools must instrument the system in order for them to be able to recognize user interface controls. Sometimes this instrumentation is to the software application; other times it is to the operating system. Sometimes it is automation; other times it must be applied manually. Regardless, GUI test automation requires testability instrumentation.

Automation experts use a number of techniques to provide GUI testability. Many techniques are specific to a particular test tool and interface technology, whether it be using WinRunner with Netscape, SilkTest with Java Swing, or Robot with Visual Basic. This paper, however, can't address these issues in detail. Instead it describes some general principles that have been used in multiple cases.

These principles are based on my several years experience as a QA Partner and SilkTest tool expert, tips shared with various colleagues, as well as Lesuer (2001), Zambelich (2001), Segue (1998) and Mercury (2000). Discussing techniques for supporting custom controls is always popular at testing conferences such as STAR and on forums such as QAForums.com.

Simply getting the test tool to recognize the various window controls, including the text fields, list boxes, check boxes and menu lists that constitute modern user interfaces, presents a challenge. There are an endless variety of window controls, sometimes known as widgets or simply "objects". Test tools require some kind of instrumentation in order for them to identify the controls as such—and not as simply patterns on a bitmap.

Without being able to recognize the controls as controls, the tools must resort to dealing with the interface in purely analog terms: sending mouse and key events to locations on the screen and reading and trying to interpret bitmap patterns. This was all the first generation GUI test tools were capable of in the early 1990's. This analog approach was unsuccessful. There were a number of problems, but the most serious was that they were unable to know whether they were actually working with the desired controls.

Imagine a test in which a dialog box appears and the OK button needs to be pressed to dismiss it and continue. Analog methods would wait a "suitable" amount of time for the dialog to appear and then click on the location where the OK button was expected. If the dialog took longer than expected to appear, or if it appeared in a different location, the test script would fail. These kinds of problems were often intermittent and hard to diagnose. As a general label, such problems were known as "windows synchronization" problems.

A new generation of GUI test tools came along that were able to recognize individual controls. They could "see" the OK button as a push button (or "command button") with the label "OK". The tools could now automatically wait for such an object to appear and then click on it. It didn't matter *where* it appeared. If the button didn't appear as expected, an error to that effect would be logged, greatly increasing diagnosability.

In order to see the controls, they needed internal access to the window manager and event queue, and therefore some kind of instrumentation. Initially, this took the form of modified window managers or code that had to be inserted into the main event loop of the application code. Such intrusions weren't popular; with time, the tools became better at inserting the instrumentation automatically. Nowadays this typically takes the form of replacing system DLLs (dynamically linked libraries) or modifying hooks in the operating system.

Automatic instrumentation isn't always successful. When it fails, a test tool expert often has to understand the instrumentation need and then apply it by hand. For example, with a Java Swing GUI running on Windows, the operating system-level test instrumentation will only allow the tool to see the outer Java window. It won't be able to "see" the contents unless it also has instrumentation in the JVM (java virtual machine).

Exceptions to the need for instrumentation are "non-intrusive" test tools such as TestQuest, which connect directly at the hardware level. These can't have direct access to the software objects presenting the window controls and therefore must depend solely on analog methods. They require special tuning and customization in order to be able to view and operate interface controls. (Of course, these tools *are* intrusive at a hardware level.)

Thus the first step in GUI testability is simply getting the tool to see the individual controls.

11. Naming Windows and Interface Controls

Effective GUI test automation requires the ability to name and distinguish windows and controls. Sometimes this requires programmers to add suitable identifiers or conform to naming standards. Some development tools make this difficult.

If your product uses a standard, uncustomized set of interface controls that your GUI test tool supports, you may be all set. But you may still see problems with the names or attributes that the test tool uses to identify the controls. In order for automated tests to be useful, they need to be able to identify different controls with different names. And they need to be able to use the same names through multiple builds of the product. For example, a common problem is that different windows will have the same labels. It's not uncommon for early product versions to use generic labels for all windows. This will lead the test tool will see them all as the same. This makes it hard for the tool to "learn" all the controls in the interface; tests won't be able to tell if the desired window is displayed or if it's a different window with the same name. The programmers need to provide unique names for different pages to avoid these complications.

Naming problems may also affect controls. Test tools may have a hard time associating the text labels with the controls they are adjacent to. They typically look for identifying human-readable text to the left of controls, but I have seen instances where the window controls weren't laid out straight. Since the text label was slightly lower than the text field, the tool did not associate it with the control. These kinds of issues may require adjusting the layout to the tool's assumptions. Sometimes it requires tuning the test tool to understand the fonts being used in the interface.

Often using internal control names instead of the human-language labels avoids the problem. Programmers may need to provide these. Using internal names is particularly helpful when the labels are later re-phrased or when translated interfaces need to be tested. Some GUI development tools automatically assign internal names, which can be problematic if they assign new names every time the system regenerates a window.

When testers can use neither external nor internal names to identify the controls, then they must use other techniques. They may identify controls by their ordinal position (e.g. the third pushbutton), their relative position (e.g. the pushbutton 40 pixels across and 125 down) or by use of other properties of the control. These techniques, however, typically result in automated tests that are brittle and hard to maintain and debug. Thus a key issue with GUI testability is being able to name and distinguish windows and controls using stable identifiers.

12. Mapping Custom Interface Controls

Once you have a name for a control, you also have to be able to operate it. If the control is part of a standard toolkit, support for the control is likely already built into the test tool. No further testability support is needed.

A custom control requires additional support before it can be accessed from test scripts. You may need to customize the tool or modify the control's code. Often the best solution includes some of each. Whether a particular control is "custom" may depend on which test tool you are using. One test tool may intrinsically support controls that another treats as custom.

Sometimes a control is "custom" simply because the tool can't recognize the control type. For example, a push button may look like an ordinary push button to you and me, but not to the test tool. The control object is different enough to throw off recognition. In many cases, once you configure the tool to recognize the control type, its normal support for that control will work.

Tools define control types as "classes". They support each class with various methods for operating the controls: a method for pushing buttons, a method for typing text into text fields, a method for scrolling list boxes. You can instruct the tool to map otherwise unrecognized classes to standard classes (with their methods). This *class mapping* can use class names or other properties of the controls to establish the association. For example, your test tool might not recognize a push button in your interface, instead seeing it as a custom control labeled a "thunder button." Customizing your tool's class map with an indication that a thunder button should be treated as a push button will allow it to support this control. (Segue 1998)

Class mapping is necessary with certain software applications created with MFC (Microsoft Foundation Classes). These applications generate the class names for their controls at run time (rather than compile time); consequently the names change from run to run. These class names appear with an "Afx" prefix ("application frameworks"). Handling them requires specialized class maps. (Nyman 2001)

Testers also use the class mapping technique to instruct a test tool to ignore internal container classes. Programmers and development tools use these structures, invisible to users, to organize window objects. Ignoring them makes test scripts easier to understand. But many custom controls require more work than simply class mapping.

13. Supporting Custom Interface Controls

When it works, class mapping is the simplest way to support custom controls. This section describes other methods that also do not require modifications to the product code.

Are these testability suggestions? Without some sort of support for custom controls, they can't be tested automatically. The ideas in this section don't require changes to the product code (or the control code) but they do make the product more testable.

As discussed above, first you must ensure recognition (can the tool "see" the control?) and identification (can it uniquely identify it?). Then, you must address operation (can it manipulate it?) and verification (can it check its value?)

To develop the necessary testability support, list the actions your tests need to make with the control and the information they'll need to verify from it. The existing tool support for similar controls often serves as a useful guide. Generally you should encapsulate whatever tricks or techniques you make use of within library methods or functions. Thus they won't complicate the test scripts themselves. This also makes maintenance easier should you later find a better

approach. Basing the interfaces on similar classes improves consistency and understandability. (This exposition follows Lesuer 2001.)

Finding tricks and techniques that work often involves trial and error. Suggestions include:

- Key events
- Initializing controls
- Mouse events
- Character recognition
- Clipboard access
- Custom windows
- External access
- Internal interfaces

These techniques are explained in this and the following section.

Key Events. A technique that is elegant, when it works, is simply to use key events. For example, you may find that the HOME key always selects the first item in a custom list box and that the DOWN key then moves the selection down one. Knowing this makes it a simple matter to write a function, say CustomListSelect (int i), that emits the correct key events for selecting the desired list item. Many controls respond to various keys. Experiment with CONTROL, ALT, DEL, HOME, END, SHIFT, SPACE and all the arrow keys.

Initializing Controls. In the list box example, the HOME key explicitly puts the control into a known state (first item selected). This is important. If you assume that a control starts in a particular state, you are likely to introduce bugs into your testability support.

Consider a custom text box. If you assume that it always starts out empty, your test support will work for tests that create new records, but may fail when your tests edit existing records. Your test code for entering text probably needs to clear the box before entering new text.

Mouse Events. Mouse events can also be used. Suppose you have a row of buttons that appear to your tool as a single object. You may be able to compute the location of the individual buttons and send mouse-click events to those locations. (A mouse click is actually composed of a mouse-down event followed immediately by a mouse-down event.)

These techniques provide methods for operating controls. You'll also need to observe the state or contents of controls in order to verify their contents.

Character Recognition. Some tools have character recognition technology built into them. This is the same technology that is used in OCR (optical character recognition) except that it works directly with the digital image. It's an analog technique that is very useful for text-based controls. You can use third-party OCR tools when this ability isn't built into the tool you're using. (Structu Rise)

Clipboard Access. Another clever trick is to copy text from the control to the clipboard, which most tools can then read directly (if not they could always paste it into notepad or another application that they *could* observe directly). I've used this technique, using keyboard commands to copy the text from the control. This technique has an unintended side effect, however. You're

effectively using the clipboard as a temporary storage buffer. Careful automators will save the original clipboard contents and restore them afterwards. Otherwise, you are likely to stumble into false alarms when you test the copy/paste features of your software product in conjunction with your custom control.

Custom Windows. Another problem for testability arises from windows missing standard controls. Most windows have minimize, maximize and close buttons, but developers sometimes choose to remove them or disable them. This effectively makes such windows custom. The standard method your test tool uses for closing the window may not work. It's rarely hard to create a custom close method (e.g. ALT-F4 usually works), but this can complicate testing.

These custom windows cause problems for error recovery systems. Many test suites use error recovery mechanisms to reset applications after failed tests. They are essential if you are to avoid cascading failures. *Cascading failures* occur when the failure of one test in a test suite triggers the failure of successive tests. Error recovery mechanisms depend on heuristics for detecting and closing extraneous windows and dialogs. These heuristics will themselves require customization if developers don't conform to the tool's expectations regarding window standards. Until you devise suitable heuristics, your test suites may not run reliably. It's simpler all around just to stick with standard windows.

External Access. When you can't get the control's contents directly, you may find that you have to resort to other sources for the information. You may have to get the information from a different type of control on another screen, or directly access a file or database that stores the information. Expediency often requires such workarounds.

When these techniques fail, you will need to make use of internal interfaces to the controls. Indeed the use of internal interfaces often provides more reliable support for testing.

14. Accessing Internal Interfaces to Custom Controls

Another set of techniques involves making calls to internal interfaces of custom controls. There are generally two sources of custom controls: third-party vendors or in-house developers. With in-house custom controls, your team may need to specify and provide the internal API's, or interfaces, you need for testing. This usually means adding or exposing methods that allow the control to report its content, state, and location. With third-party controls, you don't have this option, but they tend to be delivered with a much richer set of API's than in-house controls. You may have to be crafty, however, to take full advantage of them.

The API mechanisms vary depending on the type of interface technology used. Many Microsoft Windows-based controls (i.e. Win32 controls) will respond to window messages sent to the control's handle. Other controls can be accessed through DLL (dynamically linked library) interfaces. Most Java, COM and ActiveX controls can be accessed by their public methods. Accessing these interfaces challenges many testers. Each test tool provides different levels of support for the different API mechanisms. You often need a test tool expert to implement this support.

Many control APIs include methods for operating the control and changing its state. Many testers fear that directly manipulating the control using these internal interface hooks will lead tests to take a different path than manual testing would, potentially avoiding product bugs.

A strategy that addresses this concern is to use internal methods exclusively for obtaining state and location information about the control. Use this information to figure out where events should be directed using the normal event queue (Lesuer 2001). Even so, automated support for controls often uses different event sequences than manual users would. Because automated testing can never completely replicate manual usage, wise testers will always supplement automated tests with some manual testing.

Accessing the state of a control allows you to both verify the contents of the control (e.g. that a text box contains the correct text or that a list contains the correct items) as well as learn how to operate it safely. Consider a custom tree view control. Tree views contain hierarchical lists of items. Items with others "under" them are referred to as parents of the children. A tree view could be used to display files in a file system, descendents of George Washington or the organization of genus and species within a phylum. Let's suppose that we're presenting genus and species information in a custom tree view. One of the items in this tree view is an octopus: species Octopoda, genus Cephalopoda, phyla Mullusca. Let's examine how we might build support for selecting this item in the tree view.

Let's also suppose that each item in the list has a text label and that duplicate items with the same name under the same parent aren't allowed. Tests of applications with this tree view would need a function for selecting items in the list by name. This function would need to scroll through the list, expanding items as necessary—just as a user would. A call to this function might look like this:

```
TreeViewSelect (PhylaTree, "Mollusca/Cephalopoda/Octopoda")
```

It would also need to signal an error if the requested item didn't appear in the list.

How would we implement such a function? Our strategy would depend on what we had to work with. Let's suppose we can map this control to a list box class. The list box select method would allow selecting by ordinal position, scrolling as necessary. But it wouldn't know how to expand items.

Further, suppose the control's API provides access to the text label, the indentation level and an expansion flag for each item in the tree view. The indentation level indicates how many parents are above the node. The expansion flag indicates whether the node is expanded or not. The API indexes this information in terms of the internal representation of the tree view, which corresponds to the order in the displayed tree view when all items are fully expanded.

To make use of this API, we'd first need functions to map between the indexes of the currently displayed tree view and the fully expanded tree view. The API provides the information needed to define these mappings; the algorithms are left as an exercise for the reader.

With these mapping functions, we're now ready to write the tree view select function. It must select by name (e.g. "Mollusca/Cephalopoda/Octopoda") expanding nodes as necessary. Our function must first parse its argument into nodes (three, in our example).

Then it must check to see if each node (save the last) is expanded. Obtain this information from the API. Place the expansion logic in its own function. A call might look like this:

```
TreeViewExpand (PhylaTree, "Mollusca")
```

Let's suppose that we use the strategy of sending a mouse click to the expansion hot spot. Our function needs to compute the location of the hot spot. This function needs to be able to obtain the following information from the internal interface: the absolute position of the control (from the built-in control support), the relative vertical position of the item in the list (from the list box support), and the horizontal position of the hot spot. The latter can be computed from the indentation level in the API, using the index mapping function and some observations of the geometry of the hot spots. We'll need the geometric information to be expressed in pixels. We'll also need to ensure that the list item is displayed; long tree views may require us to scroll before the targeted item is visible.

Another workable strategy would be to simply expand the entire tree view first off. This would be simpler to implement, if the tree view has an expand-all command. This strategy would also veer testing further away from normal customer usage and behavior patterns. This illustrates a general principle: automated testing is easier when you're less concerned with complete veracity. This doesn't indict automated testing, but it does require a testing strategy that uses other techniques for addressing the deviations that are often a practical necessity. But I digress.

After using TreeViewExpand to expand the parent nodes, our selection function must select the final node. We can find its index in the internal representation using the API. Our mapping function maps this to the index in the tree view display. And we can use this to select it using the list box select method. This completes our TreeViewSelect function. If we have an API reporting the text of the currently selected item, we might add a check of this as a post condition in our function. This would help catch potential errors in our testability support.

Clearly testability functions like these require unit tests, both positive and negative. This necessitates building a test application incorporating the custom controls and sample data.

15. Conclusion: Knowing Testability Options

This paper's long digression into GUI test automation testability issues underlines the following conclusions:

Automation requires testability. Whether testing via an API or a GUI, some accommodations to testability are necessary for successful automation. If you're going to have to do it, why not plan for it upfront?

Successful automation requires a focus on testability. Again, regardless of approach, the greater the emphasis on testability within the development team, the more likely automated testing will successful.

Programming interfaces provide more productive support for automated testing than user interfaces. This is an arguable point, and is surely not applicable in all situations. But in many situations, the instability of the user interface and the difficulties of automating GUI

technology indicate that test automation will be productive when they use programming interfaces.

It's ultimately hard to draw a clear and useful boundary between testability and automation, where testability is internal test support code and automation is external support code. Any form of automation changes the system to some degree, whether these changes are considered to be inside the product or external to it are often a matter of interpretation. With time, techniques for making these changes at are more external level are developed.

Test automation always affects test veracity to some degree. Automating tests always makes them somewhat different from actual customer usage. Wise testers will understand these deviations and develop compensating testing strategies, rather than act as though it can be eliminated.

This paper has largely been directed to testers and programmers working in traditional software development methodologies. As the deadline for completing this paper approached, I attended the XP Agile Universe conference (Chicago, August 2002). I came knowing that agile development methodologies—and especially Extreme Programming—already incorporate the three keys to test automation: (1) testers and programmers are expected to work together as a single team, (2) the team treats test automation as a team responsibility and (3) testers are expected to be engaged and testing throughout, often writing tests before code's been written. These tests, called acceptance tests, test the system from a customer perspective. The tests themselves are variously written by product programmers, dedicated testers, or analysts acting in the customer role. They use the testing framework created by the product programmers to execute the tests. This is a significant difference from the traditional arrangement that assigns the responsibility for test automation to an independent testing team.

The range of approaches used was impressive. I was surprised to learn that in case after case agile development teams had evaluated and then rejected commercial GUI test tools. Many found their scripting languages weak, awkward or obscure. ("Heinous" was how one put programmer put it.) Working in a methodology that required everyone to be running tests all the time, they also found it cost-prohibitive to purchase expensive licenses ($3,000-5,000) for everyone on the team and unacceptable to only allow tests to be run on a few machines.

These reports have only hardened my impression that regardless of development methodology, testability is the door to successful test automation and that the keys to that door are cooperation, commitment and early engagement.

16. Conclusion: Making Testability Happen

Some teams think that testing is best when it is as independent from development as possible (ISEB 1999). Such teams may get little benefit from this paper. This paper collects examples of testability features that testers and programmers can use to stimulate their own thinking. It presumes they collaborate, develop trust (but verify), and understand the challenges each role faces.

Testers must realize that testability requires them to understand the software design, review existing design documents and read code. This provides the basis for suggestions and facilitates concrete discussions. They need to focus on finding bugs that are important to

programmers, managers and/or customers. Focusing on minor problems and process issues will undermine the trust they need.

Testers are often reluctant to ask for testability out of fear that their requests will be rejected (Hayes, 2000). Results vary, but I've been surprised at the number of times that requests for testability features have elicited replies that something similar is already in place or in planning for other reasons. Testability is definitely worth asking for.

Programmers must realize that a tough tester is a good tester. Building software entails making assumptions. Good testers test these assumptions. Programmers must be frank with concerns about their software. And they need to realize that no amount of explanation can alleviate many testing concerns. Sometimes nothing but a test will do.

Programmers must share their source code, design documents, error catalogs and other technical artifacts. They also need to be ready to provide reasonable explanations of product design and architecture.

Testability is a design issue and needs to be addressed with the design of the rest of the system. Projects that defer testing to the later stages of a project will find their programmers unwilling to consider testability changes by the time testers actually roll onto the project and start making suggestions.

Some teams are interested in testability but feel like their management won't support investing time in it. In fact testability often saves time by speeding testing and making bugs easier to track down. More and more managers are also realizing that effective testing helps communicate the actual state of a project, making late surprises less likely.

When discussing testability features, testers should focus on features: things the software could do to make it easier to test. Be specific. Testability requests are sometimes met with concerns regarding security ("Testability will open up back doors that can be used by hackers."), privacy ("Testability will reveal private information") or performance ("Testability will slow down the software."). Sometimes these issues are raised as an insincere way of forestalling further discussion. More conscientious teams find there are often ways to address these concerns. There is a basis for concerns with security. A security flaw in the Palm OS that allowed anyone to access password-protected information took advantage of a test interface (Lemos 2001). It's fair to take a serious look at how testability hooks and interfaces might be exploited. There are two ways to protect interfaces: one is to simply remove them from production code. This makes many people nervous because it means that many of the software testing suites and techniques developed for the product won't be able to executed against the final production version of the software. The other method is to use encryption keys to lock testability interfaces. As long as the keys are secret, the interfaces are inaccessible. A third method, that doesn't recommend itself, is to keep the test interface a secret and hope attackers don't discover it.

Privacy issues arise with logging. One solution for this is to log information in a form that can be reviewed by users. If they can see how the information is being logged, they're likely to be more comfortable sharing their logs when errors occur. As mentioned in the paper, blanket concerns about performance have no merit. Testability code should be added as appropriate, and performance concerns should only be raised if profiling actually indicates that it is causing a problem. If so, then it will need to be reduced.

There are surely risks entailed by testability. But a lack of testability has serious risks as well.

Some testers use opportunities to discuss testability as an opening to vent a litany of complaints about programmers' lack of discipline, poor documentation or chaotic processes. Airing such complaints helps ensure that such opportunities won't arise again. If testers need information or cooperation, they should be specific and let programmers know how this will help them.

Sometimes testers are convinced that the design is chaotic ("a big ball of mud": Foote and Yoder 2000). Sometimes they are right. But what is the point of making a scene? If it really is bad, then it is going to be easy to find bugs: they'll be popping out everywhere. Testers shouldn't be quiet about their concerns, but there is no point in being emotional. The fact is that many programmers accept less-than-ideal designs because of project constraints. They know they are taking a risk. Testers who are uncomfortable with these risks need to demonstrate how large that risk really is. That's what testing is all about.

Testability features often result in direct value to customers as well. This includes testing hooks that developed into user scripting languages cited earlier or accessibility hooks that make software easier to use both by visually impaired users and by testing tools (Benner 1999). It's often hard to predict how users or other products will take advantage of the increased ability to observe and control a software application.

Testability takes cooperation, appreciation and a team commitment to reliability. In the words of President Reagan, trust, but verify.

Appendix. What Is Testability?

This paper defines testability in terms of visibility and control, focusing on having reliable and convenient interfaces to drive the execution and verification of tests. This definition is meant to broadly include software design issues that affect testing. It is meant to stimulate thinking.

This appendix surveys other definitions for testability and the contexts in which you may find them useful.

Some define testability even more broadly: anything that makes software easier to test improves its testability, whether by making it easier to design tests and test more efficiently (Bach 1999, Gelperin 1999). As such, Bach describes testability as composed of the following.

- *Control*. The better we can control it, the more the testing can be automated and optimized.
- *Visibility*. What we see is what we test.
- *Operability*. The better it works, the more efficiently it can be tested.
- *Simplicity*. The less there is to test, the more quickly we can test it.
- *Understandability*. The more information we have, the smarter we test.
- *Suitability*. The more we know about the intended use of the software, the better we can organize our testing to find important bugs.
- *Stability*. The fewer the changes, the fewer the disruptions to testing.

This broader perspective is useful when you need to estimate the effort required for testing or justify your estimates to others.

The IEEE defines software testability as "the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met." (IEEE 1990) This gives us an unpunctuated clod of ambiguous grammar and vague sentiments. In other words: no help.

Voas defines testability in terms of observability. To wit, he defines it as the inverse of the likelihood that a fault could lay undetected after running a set of tests. This happens when a fault is triggered but nonetheless fails to propagate and cause a failure (Voas & Friedman 1995). This is a good thing for the user, but not for the tester. Voas defines the apparatus needed to turn this definition into a measurement. It may be useful when measuring the visibility of software to testing.

References

Adams, Ed (2001). "Achieving Quality by Design, Part II: Using UML." The Rational Edge, September. http://www.therationaledge.com/content/sep_01/m_qualityByDesign_ea.html

Bach, James (1999). "Heuristics of Software Testability."

Benner, Joe (1999). "Understanding Microsoft Active Accessibility", in *Visual Test 6 Bible* by Thomas R. Arnold II, IDG Books.

Bray, Brandon (2002). "Compiler Security Checks In Depth," MSDN, February, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vctchCompilerSecurityChecksInDepth.asp

Cohen, Noam (2000). "Building a Testpoint Framework," Dr. Dobb's Journal, March.

DiMaggio, Len (2000). "Looking Under the Hood," STQE Magazine, January.

Foote, Brian and Joseph Yoder (2000). "Big Ball of Mud" in *Pattern Languages of Program Design 4,* Harrison et al, eds. Addison-Wesley. http://www.laputan.org/mud/mud.html

Fowler, Martin (2000). *Refactoring*, Addison-Wesley.

Fowler, Martin (2000a). *UML Distilled,* Addision-Wesley.

Fowler, Martin (2001). "Separating User Interface Code," IEEE Software, March.

Friedman, Michael & Jeffrey Voas (1995). *Software Assessment: Reliability, Safety, Testability*, Wiley.

Gelperin, David (1999). "Assessing Support for Test," Software Quality Engineering Report, January.

Guckenheimer, Sam (2000). "What Every Tester Should Know About UML," STAR East, May.

Hayes, Linda (2000). "I Want My Test API," Datamation, December. http://datamation.earthweb.com/dlink.resource-

jhtml.72.1202.|repository||itmanagement|content|article|2000|12|06|EMhayesquest12|EMhayesquest12~xml.0.jhtml?cda=true

Houlihan, Paul (2002). "Targeted Fault Insertion," STQE Magazine, May.

IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std 610.12-1990.

Information Systems Examination Board (ISEB) (1999). Software Testing Foundation Syllabus, version 2.0, February. http://www1.bcs.org.uk/DocsRepository/00900/913/docs/syllabus.pdf

InstallShield (1998). "Creating a Silent Installation." http://support.installshield.com/kb/display.asp?documents_id=101901

Johnson, Karen (2001). "Mining Gold from Server Logs," STQE Magazine, January.

Kaner, Cem (1998). "Avoiding Shelfware: A Manager's View of Automated GUI Testing."

Kaner, Cem (2000). "Architectures of Test Automation." http://kaner.com/testarch.html

Lemos, Robert (2001). "Passwords don't protect Palm data, security firm warns," CNET News.com, March 2. http://news.com.com/2009-1040-253481.html

Lesuer, Brian (2001). "Supporting Custom Controls with SilkTest," Test Automation Conference, August.

Marick, Brian (2000). "Using Ring Buffer Logging to Help Find Bugs." http://visibleworkings.com/trace/Documentation/ring-buffer.pdf

Maguire, Steve (1993). *Writing Solid Code.* Microsoft Press.

Mercury Software (2000). *WinRunner Advanced Training 6.0: Student Workbook*.

Meyer, Bertrand (1997), *Object Oriented Software Construction*.

Mitchell, Jamie L (2001). "Testability Engineering," QAI.

Mullins, Bill (2000). "When Applications Collide," STQE Magazine, September.

Nyman, Noel (2001). Personal communication.

Nyman, Noel (2002). Personal communication.

Payne, Jeffery, Michael Schatz, and Matthew Schmid (1998). "Implementing Assertions for Java," Dr Dobbs, January.

Philips, Todd (2002). "Testing Applications with AppVerifier," MSDN, June. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnappcom/html/AppVerifier.asp?frame=true

Pettichord, Bret (1999). "Seven Steps to Test Automation Success," Star West, November 1999. http://www.io.com/~wazmo/papers/seven_steps.html

Pettichord, Bret (2000). “Three Keys to Test Automation,” Stickyminds.com, December http://www.stickyminds.com/sitewide.asp?ObjectId=2084&ObjectType=COL&Function=edetail

Rubenstein, David (2002). “Who Knows What Evil Lurks in the Hearts of Source Code?,” SD Times, June 1. http://www.sdtimes.com/news/055/story4.htm

Structu Rise (undated). Textract, a screen text capture OCR library. http://www.structurise.com/textract/index.htm

Valaer, Laura and Robert Babb (1997). “Choosing a User Interface Development Tool,” IEEE Software, July.

Segue Software (1998). *Testing Custom Objects: Student Guide*.

Whittaker, James A (2003!). *How to Break Software*, Addison Wesley.

Xconq. http://dev.scriptics.com/community/features/Xconq.html

Zambelich, Keith (2001). “Designing for Testability: An Ounce of Prevention,” unpublished.

Developing Test Cases from Use Cases for Web Applications

Orhan Beckman, Ph.D. and Bhushan Gupta

Abstract

Traditionally test cases are developed from system requirement specifications. This methodology inherently restricts customer-centric software testing. System testing derived from system requirement specifications misses or ignores dimensions of quality, such as usability and end-to-end solution integrity related to actual usage. Without adequate knowledge of how the end customer will actually use the system, test engineers are left with the impossible task of validating quality across isolated system components and functionality. An alternative technique of developing test cases from the use cases effectively centers product testing on actual usage scenarios of the system. Use cases contain the system functionality that delivers the core value to the customer. Deriving test cases from use cases helps to ensure that testing does in fact measure the product's ability to deliver core value to the customer.

This paper first establishes the characteristics of a good use case and then describes a method to derive effective test cases. It explores challenges such as: keeping a test case self-contained and addresses the issues of boundary conditions and negative testing in a web development environment. While the technique is effective, it is not a panacea. It is not possible to develop test cases comprehensive enough to address all possible system variations. Also, system boundaries and error conditions must be built into the use cases in order to effectively address them in testing.

Introduction

Use cases are abstractions of events that take place in a system. They provide succinct representations of how a system will actually be used. Wiegers [1] emphasizes how use cases support the development of valid system specifications. When characterized by actors, events and outcomes, a set of use cases can effectively represent the functions a system will enable. A systematically derived set of use cases, coupled with review and approval from partners, provides assurance that a common understanding of the product under development exists and the product's behaviors are aligned with customer needs.

Use cases provide a solid foundation from which to start developing test cases. Heuman [2] highlights the importance of use cases with regard to test cases. But information beyond what is captured in a typical use case is needed in order to derive test cases. While use cases often capture the functional aspects or requirements of the system to be tested, other dimensions such as compatibility, conformance and usability [3], should be reflected in subsequent test designs. Our paper will enumerate these additional information requirements as well as describe pertinent techniques.

Use Case Definition

A use case, in essence, describes how a customer (actor) uses a product in pursuit of a goal. The term 'use case' is described in the literature as a "contract between the stakeholders of the system about its behavior" [4], a "sequence of actions performed by a system to achieve an observable result" [2], and "interactions between the system and entities external to the system" [5]. A use case, in this paper, is defined as "a description of how an actor uses the system to achieve a specific goal".

Use Case Characteristics

The use cases can be classified as customer-centered and technology-centered. Customer-centered use cases are in stark contrast to technology-centered use cases that describe how the system works and what the actor must do to make it work. Technology-centered use cases may have internal validity but often lack external validity. High internal and external validity, indicative of customer-centered use cases, serve to increase the tester's ability to derive valid test cases and accurately predict the quality of the system through testing.

There are three dimensions that differentiate customer-centered use cases from technology-centered use cases. First, customer-centered use cases serve to maintain a customer perspective in technology-centered environments. Second, customer-centered use cases are more traceable to external customer goals, behaviors and environments than technology-centered use cases. Lastly, customer-centered use cases cover a broader scope of customer experiences.

Customer Perspective

Customer-centered use cases are written from the customer's point of view. Customer-centered use cases describe *what* the actor must be able to do with the system and not the details behind *how* the system works. This means that overt behaviors and cognitive processes are explicitly represented but 'under the hood' steps are not. For example: 'enter zip code' and 'choose method of shipment' should be captured but 'module X sends XML to module Y' should not. Use cases, written at any level, should be void of unwarranted design details. Design details belong in a technical specification or external reference specification.

Traceable

A use case is externally traceable, or has external validity, to the degree it correlates with actors' goals, behaviors and environments. These dimensions of external validity are captured in the Use Cases and Customer Profiles. External validity is established through the validation of assumptions with the actors in the system. This contact with the system actors may take various forms such as customer visits, concept tests, subject matter expert reviews, and usability tests.

Internal traceability (internal validity) is as important as external traceability. A use case has internal validity to the degree it correlates with an understanding of the system shared by internal stakeholders. Internal stakeholders may include executives, managers and individual contributors in the areas marketing and research and development. The Product Concept use case (described below) provides an initial focus and is often the first opportunity to foster a cross-organizational understanding of the system under development. The two forms of traceability are mutually exclusive and therefore must be pursued independently.

Scope of Customer-Centered Use Cases

Customer-centered use cases capture the customer's experience over the breadth of the product or system's lifecycle. Lifecycle stages, highlighted below in the Meta use case discussion, cover not only the 'use' of the system, which is often the most salient aspect of the system, but also the preparation and support of the system which must be accurately specified, built and tested in order for the system to function properly and deliver value.

Use Case Representation

Use cases can be represented at different levels of abstraction as described in this section.

Product Concept Level

The Product Concept use case consists of a brief description of actors, goals and events in the system. It is to be shared widely with both internal and external stakeholders of the system under development. At the highest level of abstraction is the Product Concept use case. This is similar to a user-oriented operational view of a system described by Wasserman [6] and the Operational Concept described by Chipannis [7]. This 10,000-foot view of the system is kept concise so that an executive, manager, engineer, tester, or customer can, in a 'quick read', understand the value the system is designed to deliver. It also summarizes who interacts with the system to receive the value (beneficiary), how the value is delivered, and who enables the system to deliver its value (support role).

Meta Level

At the next level down, the Product Concept is parsed to yield what we call Meta use cases. The Merriam-Webster Dictionary defines 'meta' as, "more comprehensive: transcending". Meta use cases err on the side of breadth rather than depth through 500-foot views of how the system functions to reach lifecycle milestones such as "install, use, service and support the system". Meta use cases provide a more detailed view of the Product Concept and are traceable to its source. Traceable, in this context, means the details in the Meta use cases support, not contradict, the tenets outlined in the product concept. Meta use cases are especially valuable to system stakeholders focused on program-level concerns such as how the system will be installed, used, supported, etc. Figure 1 shows an example of a Meta use case. Expect Meta use cases to evolve early in the product development cycle as design assumptions are tested and refined. Meta use cases are broken down further to provide the additional details needed to generate test cases.

Goal Level

At this third and final level, Meta use cases are decomposed into Goal use cases. Less abstraction allows for a more detailed view of an event or, for example, how an actor interacts with the system to achieve specific goals such as 'register for a class, 'send an e-mail ', and 'check on the status of a class'. Use cases written at this level provide enough detail to leave little to the imagination how the actor interacts with the system to achieve specific goals. As with the transition from Product Concept to Meta use cases, the Goal use cases support, and do not contradict, the processes described at the Meta level.

Goal use cases are written at a level such that it describes how the actor interacts with the system to achieve an outcome significant to the actor. "Enter first name" is probably not a valid use case because it does not constitute a meaningful achievement; it is more likely a step in the process that achieves a recognizable goal such as "Register for Class".

One can share a goal-directed use cases with a target customer (actor), for the purposes of providing constructive feedback. Once use cases are defined they can be used in many phases throughout the product life cycle such as requirements, design, development and testing.

Figure 2 illustrates a use case map in traditional Universal Modeling Language format [8]. In this example, the Employee has six Goal use cases represented and the Instructor has one. It should be noted that the Employee and Instructor, share the use case "Login" (USE 3.1).

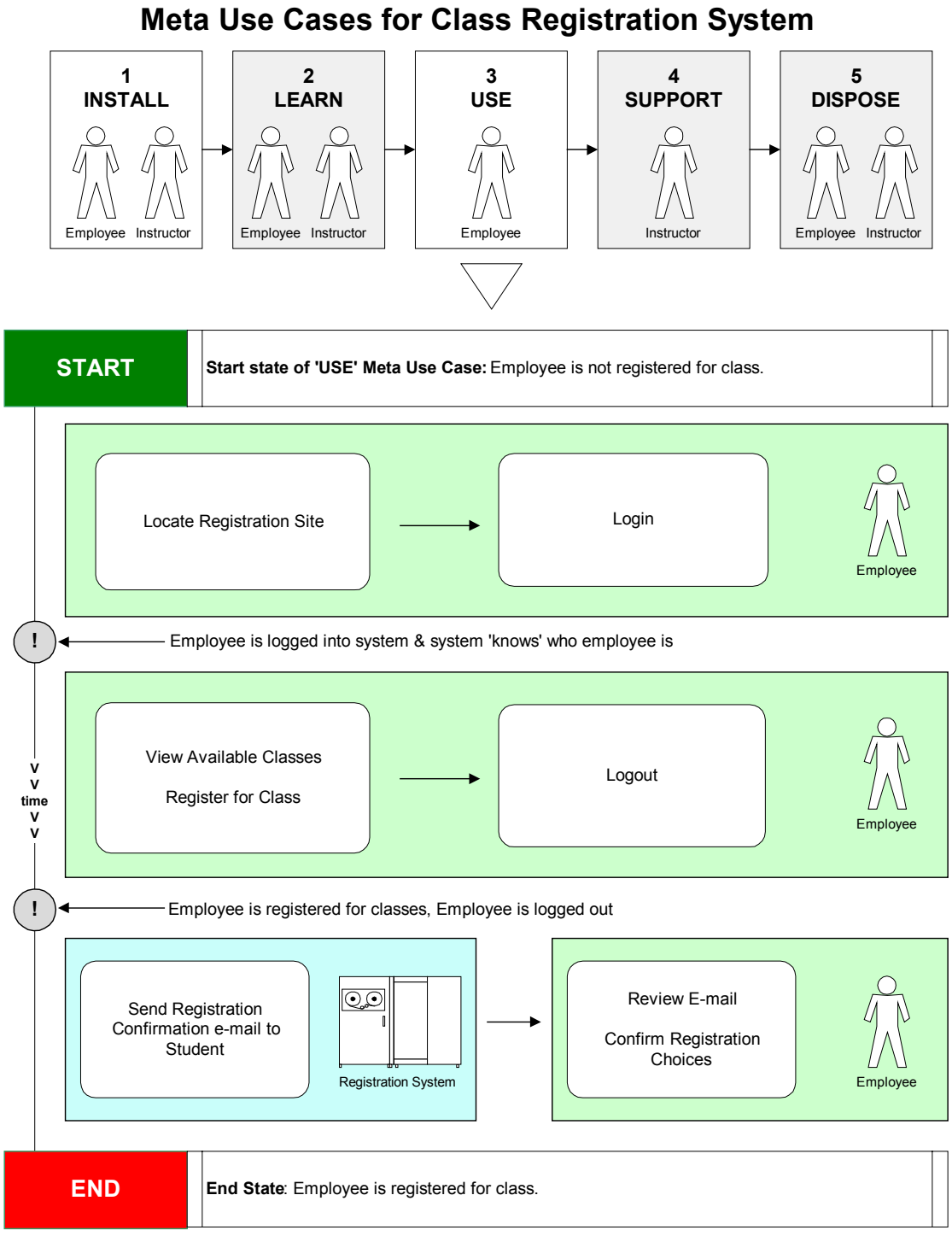


Figure 1. Meta Use Cases for Class Registration System (with USE example)

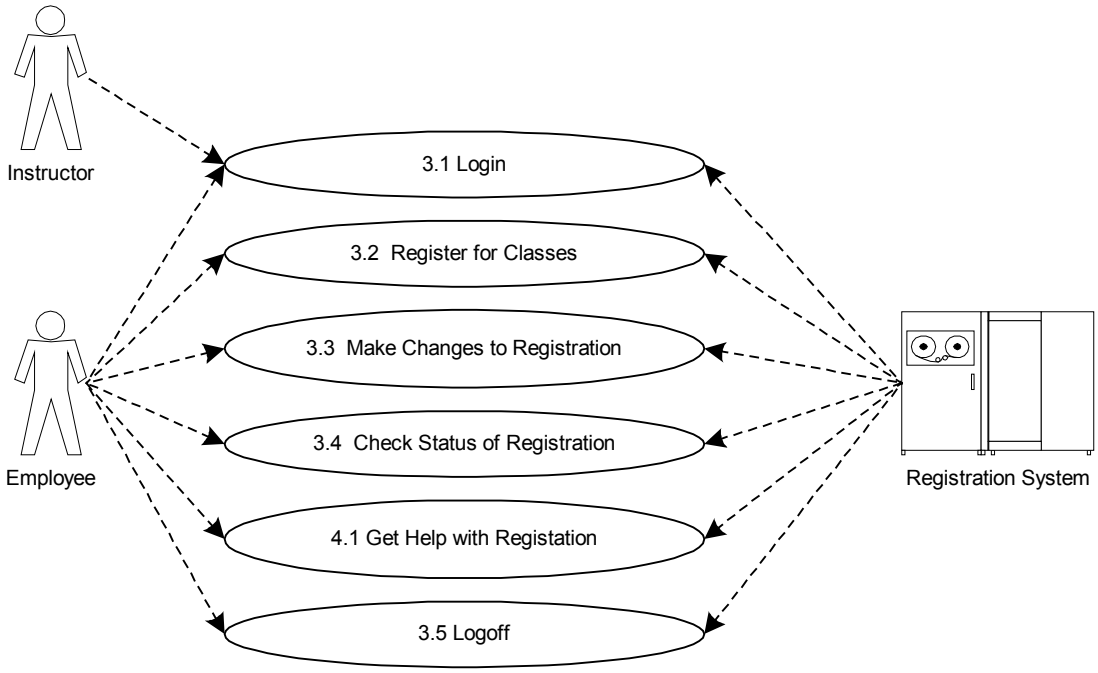


Figure 2. Goal-Centered Use Case Map

The use case 3.1 is further defined in Table 1. This more formal description includes characteristics of the use case such as the Pre-condition, which is the state of the world before the use case begins, the Normal Sequence, which describes the steps the actor takes to reach the goal, and Variations, which outline alternate paths the actor may take.

Title	USE 3.1, Login
Description	Actor logs into the system
Actors	Employee, Instructor
Pre-condition.	Actor has access to a web browser
Trigger event	Employee OR Instructor want to enter the system
Normal Sequence	1. Actor finds the web site 2. Actor enters login ID 3. Actor enters the password
Post-condition	Actor is logged into the system.
Extensions	3.2 Register for Class 3.3 Make Changes to Registration 3.4 Check Status of Registration

	3.5 Logoff
Variations	If Actor enters invalid login ID then “Login erroneous “ result If Actor enters invalid password then “Login erroneous “ result
Issues	Some issues will come up as the use case is defined. Rather than wait for a place to put the information, the Issues field is used to keep track of issues as they come up.

Table 1. Goal-Centered Use Case Example: “Login”

An equally important extension to use cases is the Actor Profile (Table. 2) that describes characteristics about an actor and his/her environment. These data, which may include the amount of experience with the system, education level, environment, tools and special considerations, provide additional context to the use case.

Title	Employee
Description	Someone currently employed at HiTech Company
General Goal	Attend class to improve technical skills
Environment	Location: Office desk, Corporate Library (Internet terminal), Cyber Café Terminal: 256 colors to 24 bit color Terminal: 10 inch to 32 inch screen Terminal: 640 x 480 to 1024 x 768
Tools (Applications, OS’s)	Operating Systems: Microsoft (95,98, NT, 2000), Macintosh OS (>=10), Unix (Linux) Browsers: (Internet Explorer, Netscape Navigator) Applications: (MS Office Suite, AOL Instant Messenger, MS Net Meeting)
Experience	High School education and zero or more years of college education
Special considerations	*Some users may have special needs that need to be considered. For example: age, education level, language, anthropometry, etc.*

Table 2. Actor Profile Example: “Employee”

At this point in our Corporate Class Registration example, the Product Concept is defined and approved by internal and external stakeholders. Meta Use Cases have been generated and approved. The Goal use cases adequately define the details supporting the Meta Use Cases. Also, the actors in the system have been sufficiently described. The use cases boast internal and external validity, adequate scope and

a customer perspective. In short, a strong customer-centered foundation has been established from which the process of test case derivation can begin.

Anatomy of a test case and its relationship with a use case

Test cases share some elements with use cases. Both are triggered by an input, executed by an actor and result in an output. Preconditions for a test case must be met before it can be initiated. The "happy day scenario" in a use case is equivalent to positive testing while the use case variations represent other key aspects of a test case such as negative and boundary testing. Just as UML diagrams can represent use cases, state diagrams can represent test cases.

If established properly, the use cases assure an exact behavior of the system. Actor profiles establish various system configurations to be supported. The system itself is treated as an actor and its profile provides the system environment. This lays the foundation for test case development for system functionality, negative, boundary and configuration testing. The following sections describe the methodology that will lead to a complete test suite for the "Corporate Registration System". It elaborates the Login test case to illustrate how the methodology works.

Establishing Test Cases

The main test cases are summarized in the Meta use cases and detailed in the Goal use cases. For our example, the tests cases in the Use Meta use case are listed in Table 3.

Test Case	Actors	Configurations
Login	Employee (Primary) Instructor (Primary) System (Secondary)	MS Windows 2000, IE 5.0, Linux 6.0 Mac OS 9.0, Netscape 6.1 Weblogic 6.0, Apache, Tomcat
View Classes	Employee	Same as Login
Register for a Class	Employee	Same as Login
Get Help	Employee Instructor	Same as Login
E-mail Notifications	System Employee Instructor	Same as Login
Logout	System Employee Instructor	Same as login

Table 3. List of Test Cases Generated from Meta Use Case – "Use"

Optimization and Prioritization of Test Cases

The list of test cases developed is normally optimized because we are considering all possible events that a system supports. A use case may have one-to-many relationships with the actors. There is a unique input in these relationships but the output will vary based upon the actor. For example, both the employee and the instructor provide the same input (ID and Password) but the output in the case of the employee is the class list while in the case of instructor it is the list of classes he/she is teaching. A single test case with the actor variation will suffice for both aspects of the system. This method is illustrated in Fig. 3. It shows where both the employee and instructor login to the system.

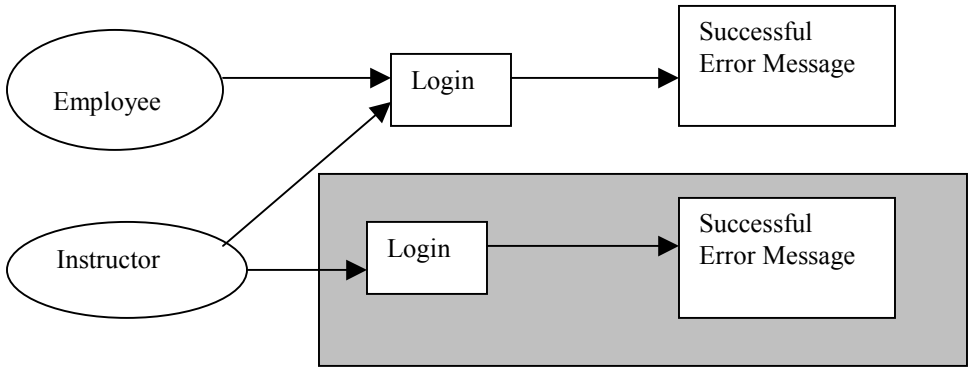


Figure 3. Optimization of Test Cases

A test case can be prioritized based upon the frequency or occurrence of the use case from which it is derived. Once established, a use case can be assigned a frequency (high, medium, low) that can easily be transferred as priority to the corresponding test case. For example, the "Get Help" use case may have a relatively low frequency of occurrence and thus a corresponding low priority. A similar approach can also be taken for the prioritization of negative, boundary, and other types of testing within a test case.

Elaborating Test Cases

Once a test case has been established it can then be elaborated with the help of Goal level abstraction. The following table shows the elaboration of the login use case.

Test Case: Login and its variations

Preconditions: The primary actors (Employee, Instructor) have access to a web browser. The primary actors have been registered and have established a login name and a password.
Post-conditions: An actor is successfully logged in.

Test Procedure: Positive testing and negative testing

Action	Input	Output
Login as an employee	Known valid ID Known valid password	Class offering page displayed
Login as an instructor	Known valid ID Known valid password	Class roster page displayed
Login as employee	Known invalid ID Known valid password	Error message
Login as employee	Known valid ID Known Invalid password	

Table 4 Test cases representation from Goal Level Abstraction

Using the test types proposed by Iberle [3] the other test types applicable to Login are:

- Boundary testing with the login ID and password string lengths 0 and 15 (a large string).
- Usability testing to evaluate if the actor can use it.
- Configuration testing to determine that the system will run on all intended platform/operating system configurations
- Conformance testing to evaluate compliance with internal and external standards.

Advantages

- Since customer-centered use cases represent internally and externally valid usage scenarios, test cases derived from them have similar levels of validity
- Use cases support the test derivation process including negative, boundary and alternative scenario testing.
- Test cases are optimized and prioritized based upon the actual system usage.
- Test case derivation can begin as soon as the system is conceptualized. Normally, the test developers wait until a system has been entirely specified.

Conclusion

Effective test cases can be derived from use cases once an adequate actor characterization has been established. Although this requires a better understanding of the use cases and their description, it also leads to a sound system design, development and testing. Based on the importance and frequency of a use case, a test developer can optimize and prioritize test suites with a minimal amount of effort thereby providing a highly effective testing environment.

Our experience although limited, shows that current representation of use cases does not adequately characterize the actors. The UML representation of a use case often stops at state transition diagrams. For a successful derivation of test cases, it is necessary to characterize the actors with their profiles. One might argue about the amount of work involved in specifying the actor profiles. Customer-centered use cases provide a solid system foundation that support customer-centric system development. Further work is necessary to find the optimal level of characterization and we are continuing our research in this area.

References

1. Wiegers, K.E. Process Impact, http://www.processimpact.com/articles/usecase.html
2. Heumann, J. (2001). Testing from the Beginning, Use Case at Work, Proceedings, Pacific Northwest Software Quality Conference.
3. Iberle K. (2000). Step by Step: Test Design, Software Testing and Quality Engineering, September/October.
4. Cockburn, A. (2001) Writing Effective Use Cases, Boston: Addison-Wesley.
5. Kulak, D. & Guiney, E. (2000). Use Cases: Requirements in Context, Boston: Addison-Wesley.
6. Wasserman. A.S. (1989). Redesigning Xerox: A Design strategy based on operability. In E.T. Klemmer (Ed.) Ergonomics: Harnessing the Power of Human Factors in Your Business. Norwood, NU: Ablex.
7. Chippanis, A. (1996). Human Factors in Systems Engineering, New York: John Wiley and Sons.
8. Quantrani, T. (2000). Visual Modeling with Rational Rose 2000 and UML, Boston: Addison-Wesley.

Appendix – A

The Discovery Center, an education service of a HiTech company, has an on-line course registration system for its employees. Using the company intranet, an employee can log on to the registration system with the URL: discovery.hitech.com and start the registration process. If the employee provides an accurate login ID and password, the courses offered in the current month are displayed. If the login ID and password do not match, the employee is given two more chances to login. If all three logins fail, the web interface to the user is closed.

Upon a successful login, the system provides the courses being offered in the current month with a course number, title, instructor, and date and time of offering. When the employee clicks on a desired course, she is presented with the registration form with course number and title already filled out. When she enters her employee number, the information relevant to her is also filled out. She checks the information for correctness and submits her request. Upon a successful registration, the system generates two emails: one for the employee to confirm the registration and the other for employee's supervisor informing her about the registration. The employee has the capability to view her registration and cancel it any time she wants. This being a wealthy enterprise, there is no penalty on late withdrawal.

If the course she selects is closed, she is placed on a waiting list. Later on if availability occurs, the employee is registered for the course and an e-mail is sent to her supervisor. At any time an instructor can also view the course participants by following the same login procedure as employees that are registered for classes.

Appendix – B

Goal-Centered Use Case Example: "Register for Classes"

Title	**USE 3.2** Register for Classes
Description	Register for classes for next semester. This is the actor's *first* use of the system.
Actors	Student (see Student Profile below)
Pre-condition.	Student is enrolled in the university or college. Student is at a workstation with a browser open and access to the Internet Student is logged onto the system (Use Case 3.1)
Trigger event	Student is ready to register
Normal Sequence	Student finds class registration web site Student identifies self to system Student enters and confirms new password for system Student reviews classes being offered for next semester Student selects classes Student registers for classes System informs user that registration is complete
Post-condition	Student is registered for classes for next semester.
Extensions	Student Logs out System sends e-mail registration confirmation to student Student receives e-mail Student confirms registration choices
Variations	If Student Cancels out of system before done registration is complete then the any selections the student has made up to that point will be lost If Student Does not log out then the system will log the student out automatically after 2 minutes of inactivity If Student registers *before* or *after* the valid registration period then the student

	will be informed of this status and will not be allowed to register If Student is not currently enrolled at the university then the student will not be allowed to register If Student tries to register for class for which he/she does not have the prerequisite then student will be informed of this status and will not be allowed to register for the target class If Student tries to register two or more classes that conflict then student will be informed of this status and will be forced to make a choice between the conflicting classes If student wants to finish registration at a later time (say after talking to the academic advisor) then the student can log on to make changes at a later time (see use case 3.2 – Make Changes to Registration)
Issues	Some issues will come up as the use case is defined. Rather than wait for a place to put the information, the issues field is used to keep track of issues as they come up.

Modeling:
A Picture's Worth 1000 Words

Elisabeth Hendrickson

Abstract

Whether you are in the enviable position of having detailed design artifacts to guide your test effort or are in the more common position of having to discover what the system does as you go, pictures can be a big boon in your testing. Diagramming the architecture and behavior of the system can help you in planning test configurations, designing tests, coordinating efforts, and discussing implications of changes. In this talk, Elisabeth Hendrickson illustrates various styles of diagramming and shows you how creating these diagrams can help make your test effort more effective. Explore ways to derive tests from state charts and architectural maps - Learn how pictures enable you to test a design before you have code - Discover how diagrams can help improve communication throughout the project team.

Elisabeth Hendrickson *is an independent consultant specializing in software quality, management, and testing. She has over a dozen years of experience working with leading software companies. An award winning author, Elisabeth has more than 20 published articles and is a frequent invited speaker at major software quality and software management conferences. You can reach her at* ***esh@qualitytree.com*** *and read more about her ideas on quality and testing at* ***www.qualitytree.com****.*

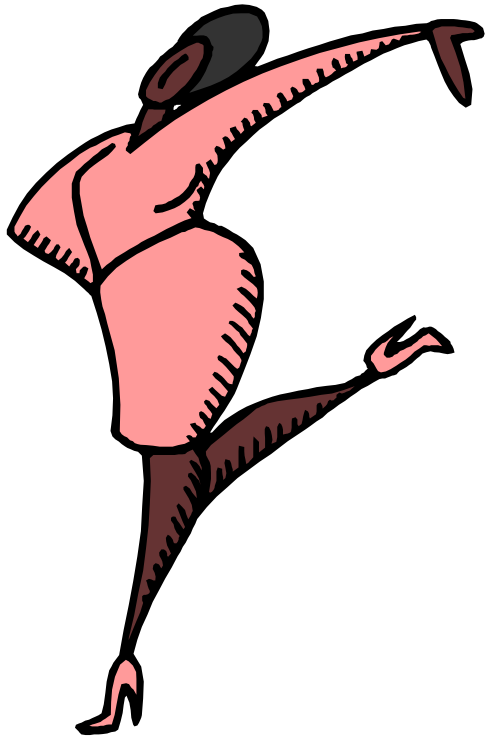
Modeling for Testers: A Picture's Worth 1000 Words

Elisabeth Hendrickson

Quality Tree Software, Inc.

www.qualitytree.com

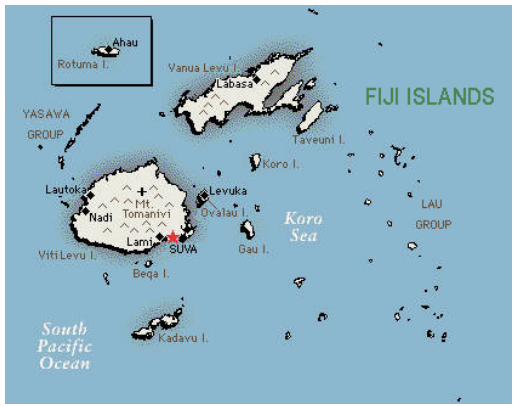
What's a Model?

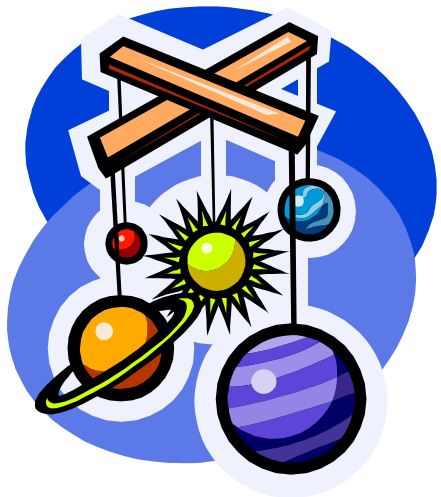


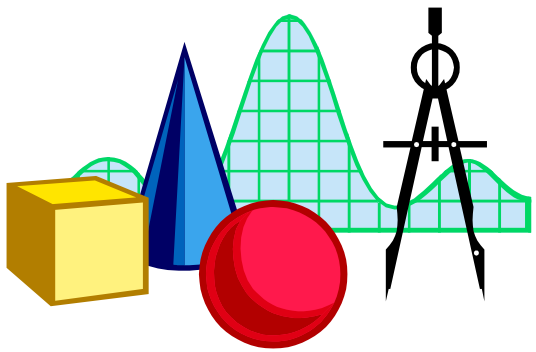
A model:

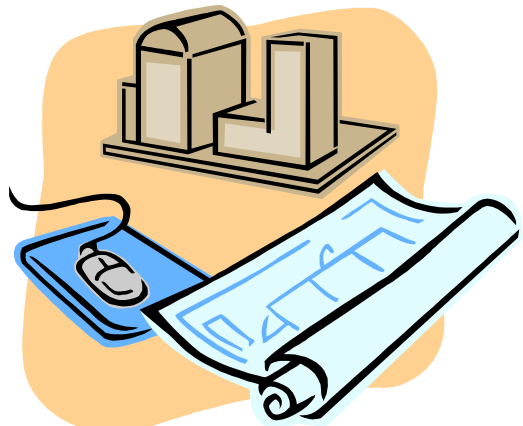
- Is an abstraction or simplified representation of the system from a particular perspective
- Supports investigation, discovery, explanation, prediction, or construction
- My be expressed as a description, table, graphical diagram, or quantitative mathematical model
- Is not necessarily comprehensive

Models in Everyday Life

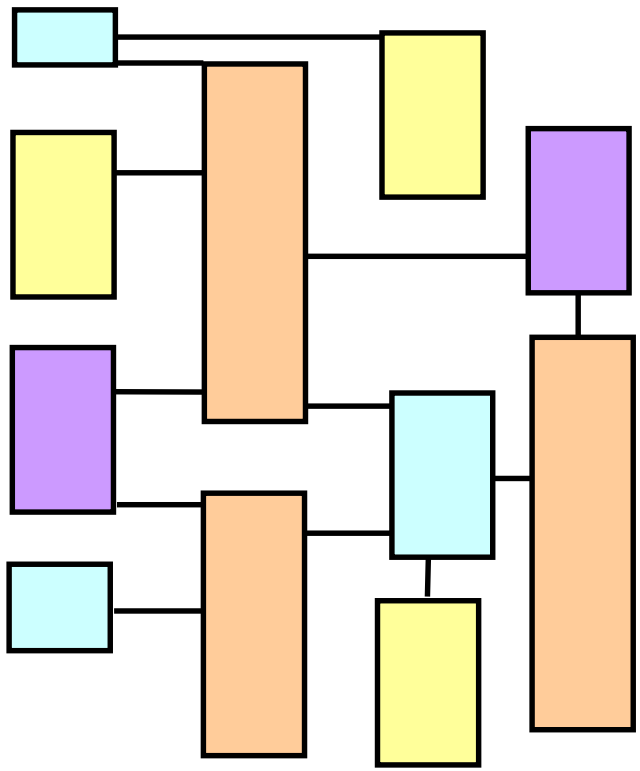








Software Models Abound



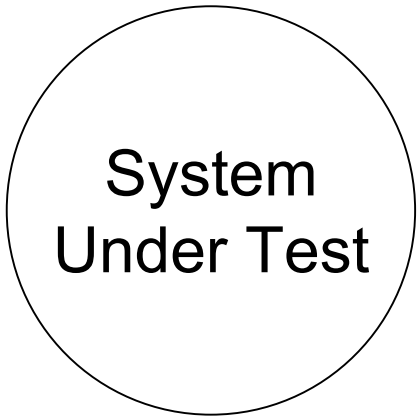
Examples:

- Flow Charts
- Data Flow Diagrams
- Entity-Relationship Diagrams
- State Diagrams
- Deployment Diagrams
- Class Diagrams
- Use Cases
- Activity Diagrams
- State Transition Tables

Two Perspectives on Software

Behavioral/Dynamic

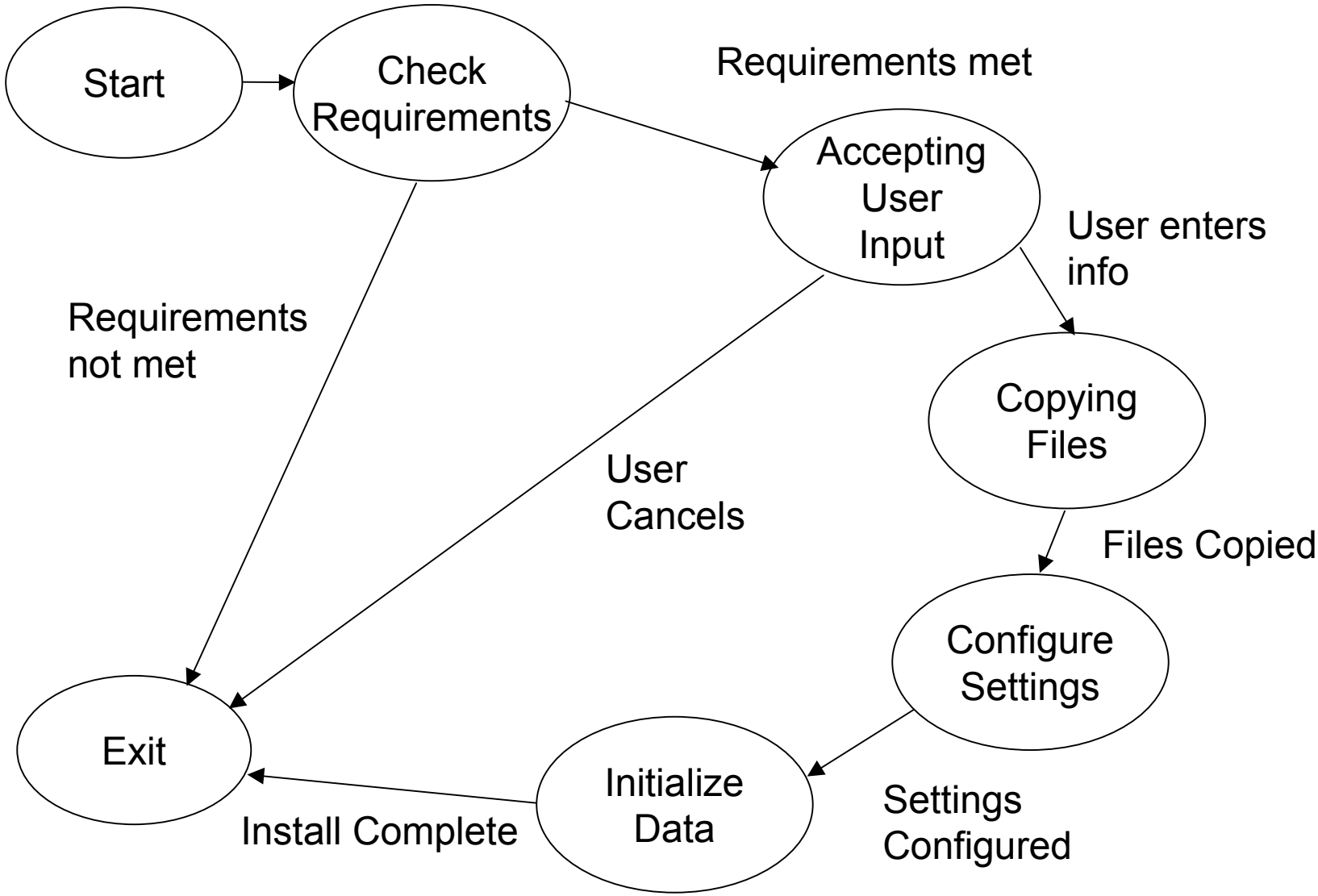
What does the system do and when?



Structural/Static

What pieces and parts make up the system?

Example: A Typical Installer State Diagram



Definitions

- **States** describes behavioral modes.

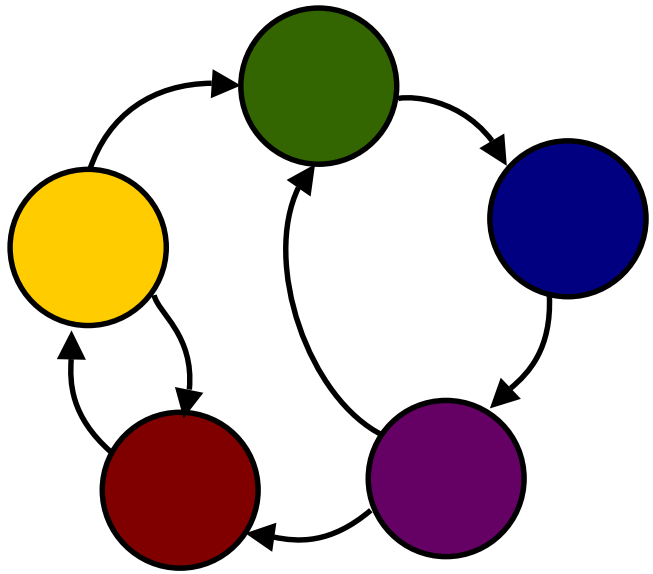
  Example: when projecting slides for a class, I put PowerPoint in Slide Show mode. This is a state.

- **Events** prompt transitions between states.

  Example: to put PowerPoint in Slide Show mode, I click on the Slide Show mode icon.

- The **State Model** describes the states and events.

Identifying States



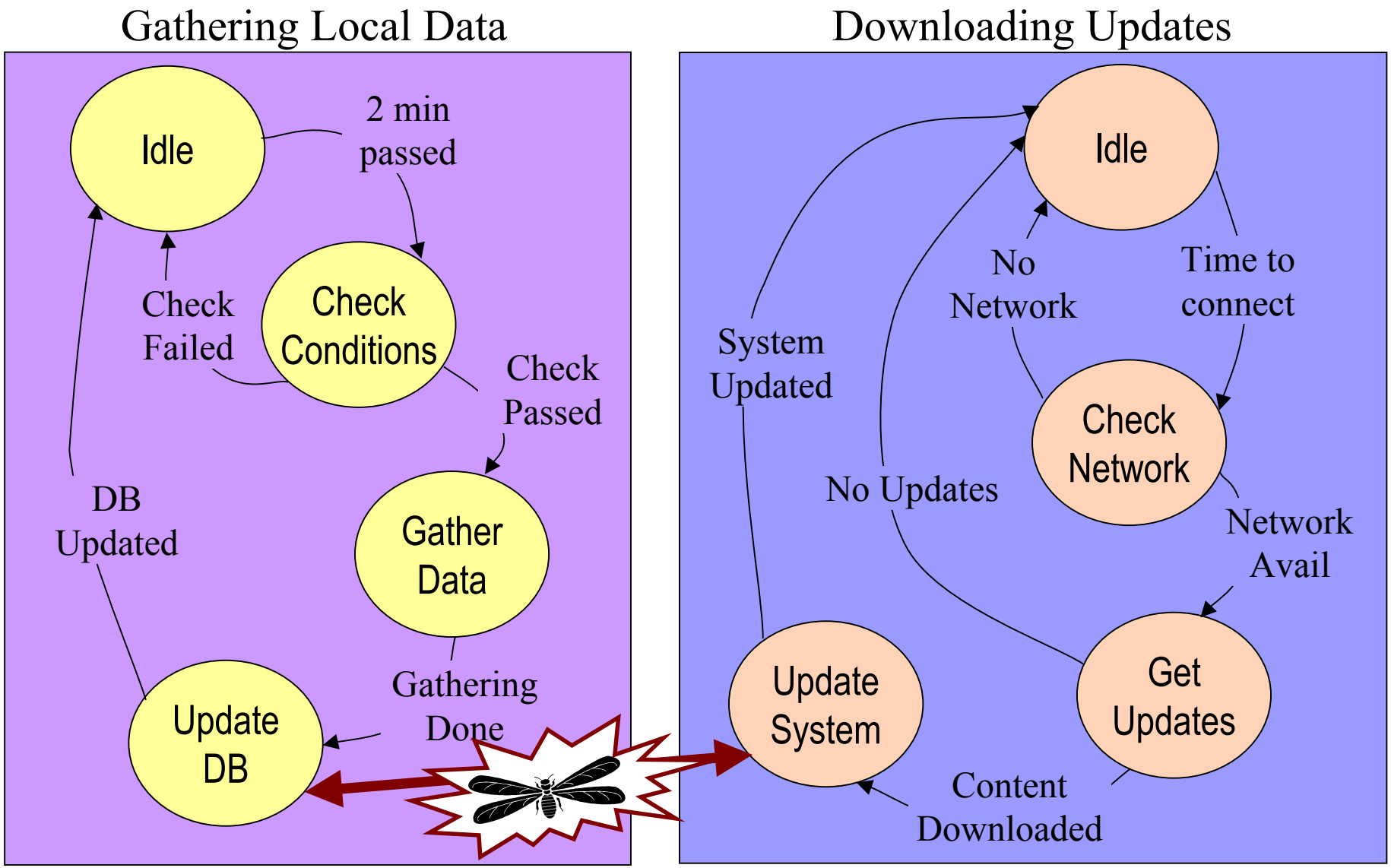
States can be identified by asking:

- What can I do now that I couldn't do before?
- What can't I do now that I could do before?
- How do my actions have different results now than they did before?

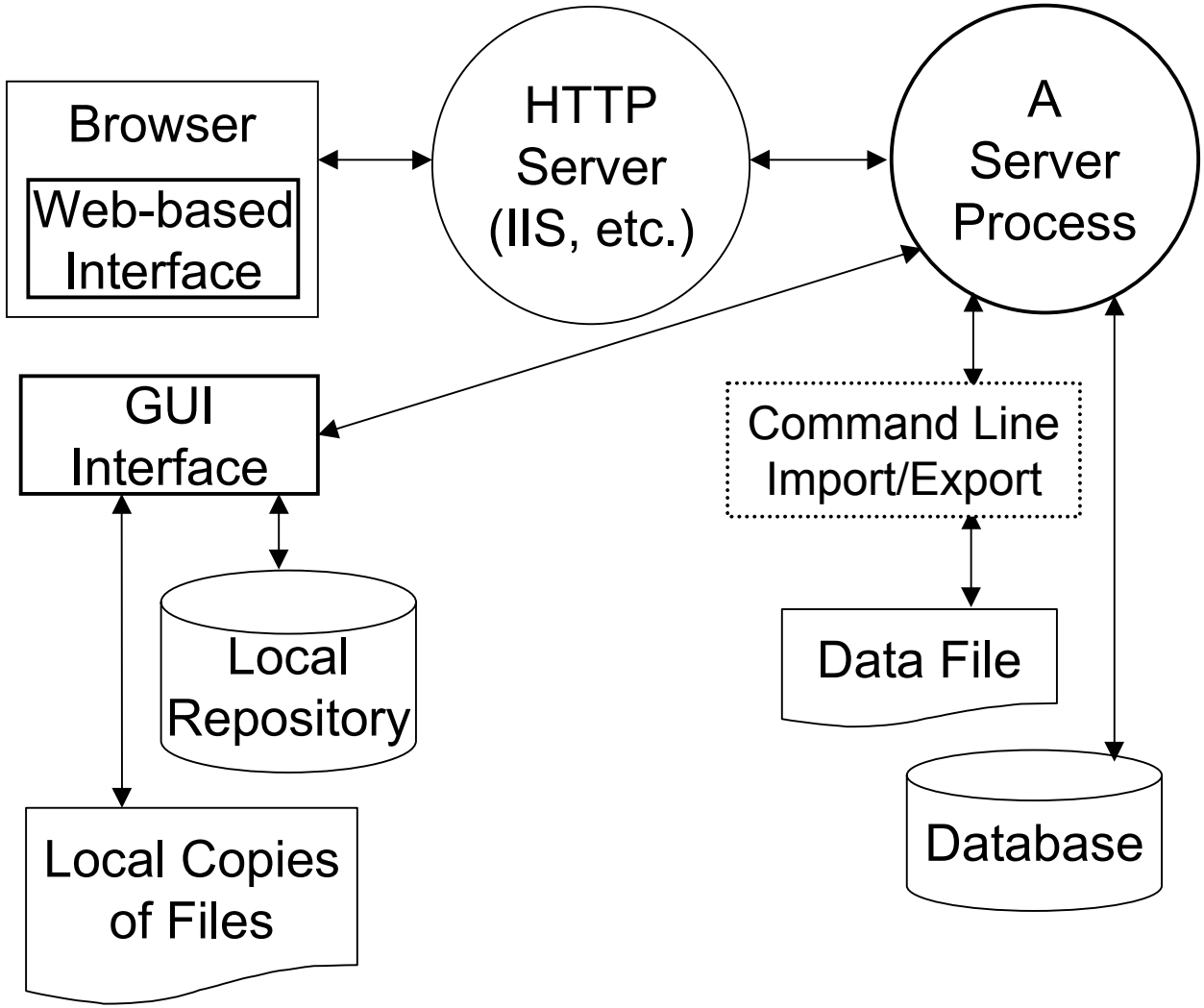
How to Use State Diagrams

- Apply every controllable event to every state. Examples:
  - Push buttons, launch software, or take other user-controllable actions when you aren't supposed to.
  - Cause triggering events from other software to occur.
- Force exits from every state. Examples:
  - Shut down the software as it's starting up.
  - Close windows when they least expect it.
  - Force the computer into hibernation (close the lid on a laptop).
  - Kill the software from the task manager or process list.
- Where two state machines interact, try combinations of states.
  - Are there any combinations that are illegal?
  - Is it possible to get the system into that combination state?

Interacting State Models

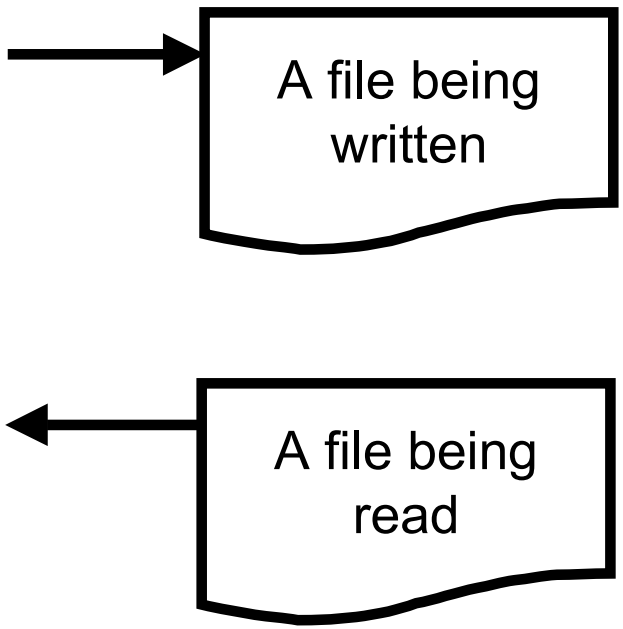


Example: A Typical System Architecture



How to Use Maps: Reading and Writing Files

When you see this...



Try this...

- Disk full
- No write permissions
- No read permissions
- Any of the path name tests you came up with in the Variables section
- File exists if it shouldn't or doesn't exist if it should
- File is locked by another process
- File is in the wrong format
- File is huge
- File is 0 length

How to Use Parts Maps: Connections

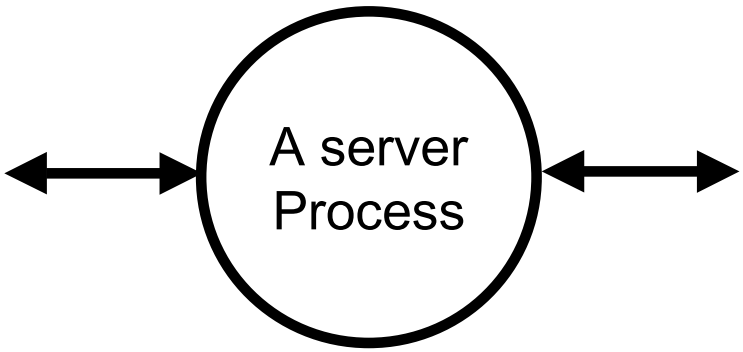
When you see this...

Try this...

- Disconnect the network
- Change the configuration of the connection.
- Slow the connection down (28.8 modem over Internet or VPN)
- Speed the connection up

How to Use Parts Maps: Server Processes

When you see this...

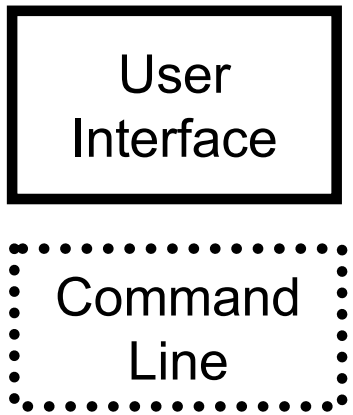


Try this...

- Stop the process in various states such as reading and writing
- Take down the server at various times, like when the client is waiting on data
- Run another program to hog all the CPU and memory on the server

How to Use Parts Maps: User Interfaces

When you see this...



Try this...

- Run multiple instances of each client application on the same machine.
- Log in as the same user on different machines running the same client.
- Log in as the same user on different clients.
- Log in on multiple clients and access the same data at the same time.

Who Makes These Models

- Managers?
- Architects?
- Developers?
- Marketers?
- Technical Writers?
- Technical Support?
- No, no. Surely not. Testing?!?!

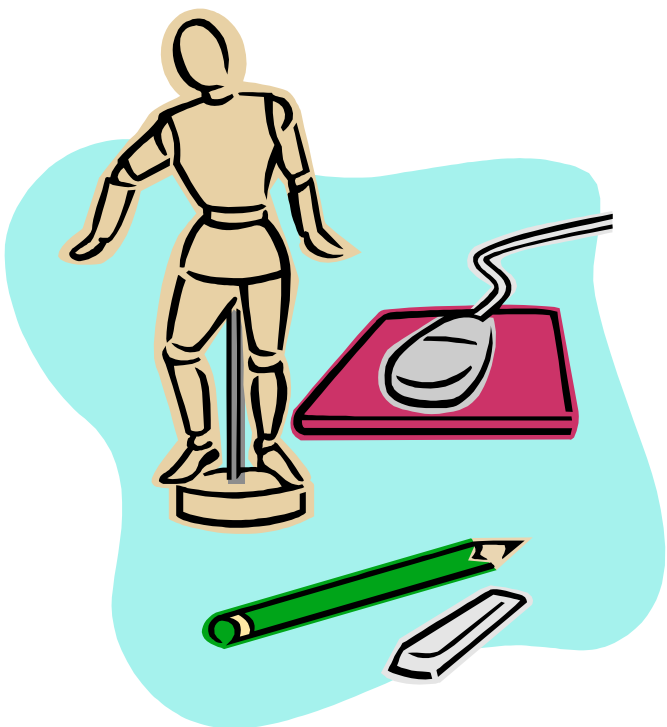
Everyone models. The trick is to articulate your models to share with others.

Tips for Modeling

- Start simple: begin with just one small piece of the software and one style of model and build from there.
- Try to find out what's going on under the covers: use memory monitors, process lists, comparison utilities, etc.
- Begin using your model right away.
- Get feedback early and often: show your model to others.

A Model Doesn't Have to be Perfect…

…It Just Has to be Useful.



A model is useful if it:

- Is better than not having any model
- Gives you new ideas
- Facilitates gathering information, including corrections to the model
- Enables you to communication more effectively.

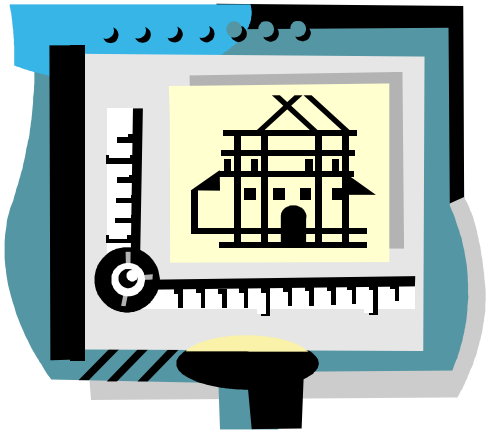
Some Questions to Ask About Models

- What is being described and from whose perspective?
- Is it static or dynamic?
- Is it qualitative or quantitative?
- Is it descriptive or predictive?
- Is it comprehensive or partial?
- What is abstracted out of this model?
- How can I use this model to find bugs?
- What kinds of bugs are invisible to this model?
- How could different people interpret the model differently?

Questions to Ask With Models

- Is this correct, at least as far as it goes?
- What's missing?
- How does *this* pass information to *that*?
- Which parts are you responsible for?
- Who is responsible for these other parts?
- What do each of the parts do?
- What happens if *this* is unavailable?
- What could go wrong *here*? How could I tell?

Go Forth and Model!



Modeling facilitates

- Information sharing
- Early testing
- Discovering hidden dependencies
- Gaining a better understanding of what you're testing

So don't wait for someone to hand you a model. If you don't have one, make your own. (And even if you do have one, make your own.)

Acknowledgements

I am grateful to the participants of the LAWST 13, LAWST 14, and Sim-LAWST 1 meetings in which the participants shared their experiences with modeling.

- The participants of LAWST 13 were: III, Chris Agruss, Sue Bartlett, Hans Buwalda, Anne Dawson, Marge Farrell, David Gelperin, Sam Guckenheimer, Elisabeth Hendrickson, Doug Hoffman, Mark Johnson, Karen Johnson, Cem Kaner, Brian Lawrence, Hung Nguyen, Bret Pettichord, Harry Robinson, Melora Svoboda.
- The participants of LAWST 14 were: Chris Agruss, Sue Bartlett, Hans Buwalda, Fiona Charles, Anne Dawson, Marge Farrell, Sam Guckenheimer, Elisabeth Hendrickson, Doug Hoffman, Bob Johnson, Mark Johnson, Cem Kaner, Brian Lawrence, Dave Liebreich, Mary McMann, Hung Nguyen, Harry Robinson, Melora Svoboda, Jo Webb
- The participants of Sim-LAWST 1 were: III, Sue Bartlett, Paul Downes, Marge Farrell, Rochelle Grober, Sam Guckenheimer, Elisabeth Hendrickson, Doug Hoffman, Brian Lawrence, Serge Lucio, Frank McGrath, Bret Pettichord, Melora Svoboda, Andy Tinkham, Jo Webb, Melissa Wibom

References

- Hendrickson, E. “A Picture’s Worth 1000 Words,” *STQE Magazine*, September/October 2002
- Winant, B. “Modeling Practice and Requirements,” *Stickyminds.com*. Available: http://www.stickyminds.com/se/s3565.asp
- Kruchten, P. “Architectural Blueprints—the “4+1” View Model of Software Architecture,” *IEEE Software*, November 1995. Available: http://www.rational.com/media/whitepapers/Pbk4p1.pdf
- The Model-Based Testing website:
  - http://www.model-based-testing.com
- To learn more about UML, check out:
  - http://www.omg.org/gettingstarted/what_is_uml.htm
  - http://www.rational.com/uml/

Tailoring Agile Development Methods to Fit Your Organization

Created for the 2002 Pacific Northwest Software Quality Conference

I would like to thank Kathy Iberle with Hewlett Packard, whose dedication and interest in this topic made her the best editor I could ever have asked for.

Tylene Áldássy
Engineering Release Manager
Corillian Corporation
3400 John Olsen Place
Hillsboro, OR 97124
(503) 629-4328
taldassy@corillian.com
tylene8@msn.com

Author Biography

Tylene Áldássy has over 20 years of experience in managing software development projects and teams in both the financial and health industries, using both mainframe and web development languages.

Tylene's focus on process improvements began 15 years ago and has been one of her core initiatives ever since. She has managed major projects using traditional development life cycles but about four years ago shifted to agile development methods as time to market demands have increased. Tylene is currently the Engineering Release Manager at Corillian Corporation, a major industry leader in online financial software.

Abstract

'Tailoring Agile Development Methods to Fit Your Organization' will define what is meant by agile methodologies and will focus on how to determine whether an agile method is best for your development project and if so, how to best apply it. It will take you through a step-by-step process for assessing if/what agile concepts your organization can successfully deploy.

This paper will occasionally refer to some of the best practices contained in three of the leading methodologies: eXtreme Programming (XP), Rational Unified Process (RUP), and SCRUM. It will address circumstances under which different methods may be best utilized.

This paper will not go into the details of selected methodologies but rather address the concepts of agile development and how to adopt and adapt them to your organization. Most features of agile development philosophies, like those in other traditional development methodologies, are just common sense and it is how those concepts are applied that will contribute to your success or failure.

What is Agile Development?

Agile development is the term used for any of a number of iterative development methods. It represents lightweight processes, which focus on frequent feature delivery and integrated team development.

The following diagram represents a pictorial view of a conceptual agile development cycle. Some agile development methods suggest that the full development of requirements and all testing be included in each iteration, while others put some of the requirements development up front.

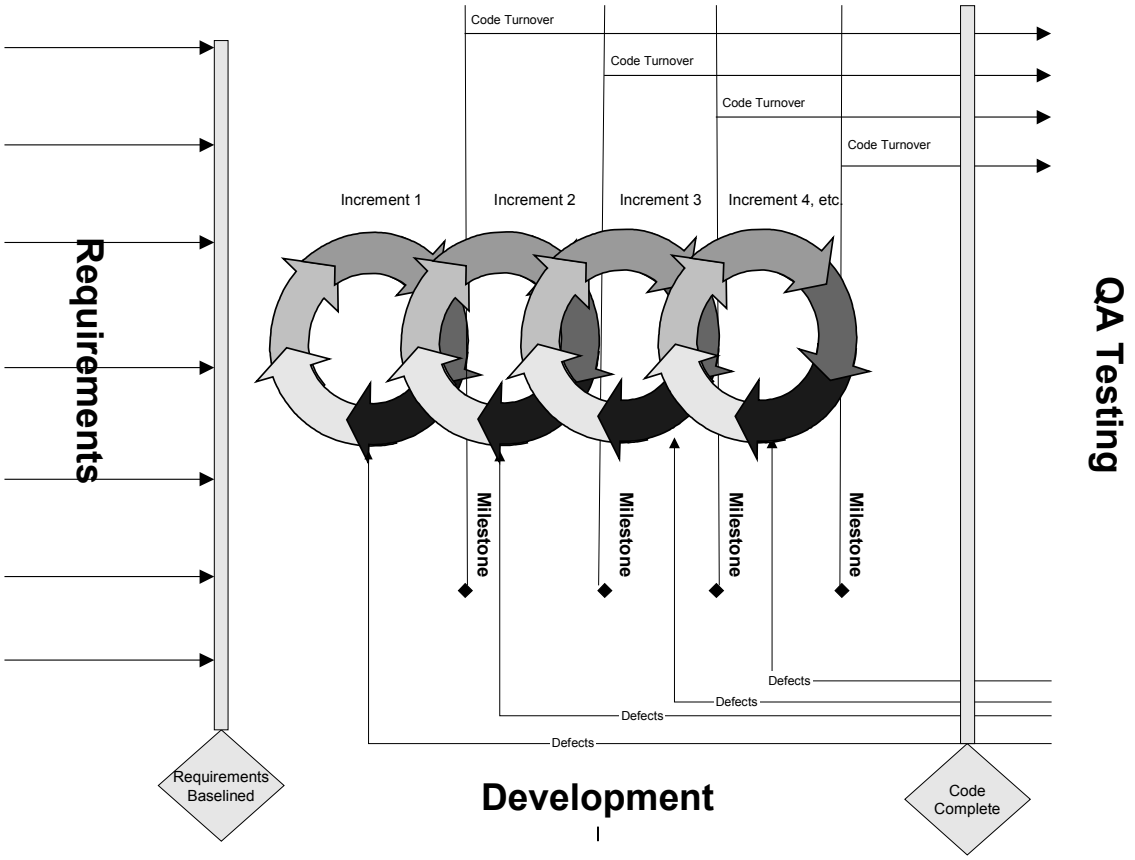


Figure 1 – A Generic Agile Development Method

In this example, the initial high-level requirements are drafted and reviewed before the development begins.

Within development, each circular cycle represents a 2-4 week effort where the five arrows represent:

- Requirements refinement
- Design
- Code
- Test
- Defect resolution and reporting

Repeat as needed until all defects from that iteration are fixed, deferred, or cancelled. All high severity defects must be fixed before exiting the iteration.

One of the key objectives of agile development is to be able to deliver some key functionality earlier in the process than in more traditional 'waterfall' methodologies. For companies who sell their software, this allows flexibility as to when to close out development and prepare to ship a product. It provides the ability to defer selected feature functionality if market demands, enabling the company to react more quickly to market needs.

Most agile methods support Simple Design, Continuous Integration, Collective Ownership and Velocity Driven Re-planning.

Planning for a project using an agile method differs somewhat from the planning you may be used to. The first step to planning is really to do release planning.

1. Identify all the features you expect to deliver
2. Dissect them into small iterations (2-4 weeks long)
3. Lay them out into an initial global delivery schedule (Release Plan)
    a. Task level planning will take place on the first day of each iteration
    b. This is much easier when the end date can have some flexibility so that as you measure the team's velocity in each iteration, you can re-plan your release plan, as appropriate. Velocity re-planning refers to comparing actuals to estimates from the iteration you are just ending and re-planning the next iteration (or two) based on factors that you may have missed or misestimated in the iteration you are completing. Velocity is the rate at which development (and all other iteration work) can be completed by the available team.
4. Communicate the global schedule through Release Management along with all other products' delivery schedules. (Release Management is not addressed much with regard to agile methods but is really key to making the whole process work, when there is any integration between products/systems, at all)
    a. For maximum efficiency, a short Release Management meeting should be held weekly to review all schedules, schedule changes (implementing a change control process that includes notifications to a Release Management group is especially effective), and dependencies.

Agile development methods also allow the opportunity for the product to be reviewed numerous times during development, which may result in refining requirements more clearly or catching design defects earlier and at less cost than other development models.

Agile development methods are not for all projects or even all organizations. That analysis is key to making the right decisions regarding organizational processes and development procedures.

There are a number of published Agile Development Methodologies, most of which have a lot of similarity to each other, with a twist here and a tweak there. They may use different names for similar components. This paper is not attempting to promote any given methodology but rather to identify the concepts for use in assessing if or how agile development might work within a given organization.

Each agile methodology has its own benefits and constraints, which may be magnified in either direction depending on how they are applied and how they naturally adapt to the development environment. This is probably the most significant reason for conducting an assessment of your organization's business and technical environment (as defined in a later section) before adopting any methodology.

Key Components of Agile Development

Logically, not all components of a given methodology are adaptable to all organizations, based on the specific needs of the company. The best fit for a given company may be to use the majority of one methodology, complemented by a few components of another, while the best fit for another may be to adapt selected practices from a variety of methodologies.

There are a few features of agile development which many consider to be too extreme to allow them to move to an agile model. For more on this, and how to deal with those features, see the Risk/Constraints section of this document.

The following list defines some of the key components of agile development in general, and are extracted from XP, RUP and SCRUM selectively and logical agile development practices generally. While RUP (Rational Unified Process) is not generally regarded as an agile method, it has many factors which fall into the agile model (short iterative periods of mini-life cycles, collaborative development, and use cases – which are like stories)

Short Iterative Development Cycles

Short, iterative cycles (called Sprints, in the SCRUM methodology) are a mainstay of the agile world. Cycles should be no longer than 3-4 weeks and should complete with feature functionality developed and tested.

QUESTION? What happened to architecting the solution for the best technological approach before beginning development?

ANSWER! Your first couple of iterative cycles may consist of planning and architecture based on the high level requirements you initially receive. Or you may choose to spend even less time on the architecture (maybe just enough to ensure that you are developing the solution within the best product or using the technical standards your company employs) and focus on designing each solution based on the functionality you will be delivering in a given iteration. Let's take an example.

EXAMPLE: Your Company makes inventory software. The following high-level requirements have been given to you for this project.

1. Add product type codes that will classify the inventory into user defined groups
2. Add regions for where the inventory exists
3. Add product expiration dates
4. Expand zip code fields from 5 digits to 9 digits

Those requirements are not enough to determine what all needs to be added or changed within the system, but may be enough to plan out generally where the changes need to occur and what can be delivered in each iteration.

So, you plan for approximately seven three-week iterations. The first iteration or two (focused on architecting the solution), may define all the systems that need changing and what files or databases may need to be changed, but you will not define what fields, what screens, etc. It may also include some prototyping for client server applications. Iterations 2-5 may each drill down the requirements, design, code, test and deliver one (or more) of the features requested in the initial requirements. At the end of each iteration, at least one feature is complete (except for any final Integration or Scalability testing which may need to be run with multiple product systems). A couple of stabilization and integration iterations may be inserted. The final iteration or two will be primarily focused on integrated testing with other products or systems. This would need to be coordinated with any other projects and takes place at a release management level, if your organization is structured for it.

Under agile development, requirements allow for more change. There is more flexibility in solution development. The impacts of finding out the need for a requirement or design change later in the project are not as extensive, as there had to be no commitment to the functional specifications before the development began.

QUESTION? How do you expect to be able to deliver anything in 3-4 weeks? Our analysis phase alone is at least twice that long!

ANSWER! It all goes back to the old Project Management tenet that states that tasks are too long if they cannot be completed in three day. Same thing for the features defined for a given iteration, -the deliverable set is too much if it cannot be completed within three weeks. You may not be able to deliver a complete feature but rather functions within a feature.

For example – the requirement is 'Add the ability to auto-adjudicate car insurance claims, as much as possible without human intervention, by using the incoming standardized repair code.' This is a huge requirement and obviously requires a lot of breakout. Once the features are

parceled out enough, you can drill down more as part of your iteration. Here is what a few of the iterations might look like:

1. An early iteration might just deliver the architecture
2. One iteration might be the reject handling to address those claims that cannot be fully auto-adjudicated
3. Another iteration might deliver the handling of 5 specific repair codes

QUESTION? How does a typical iteration play out?

ANSWER! Generally, it works something like this following example:

- Half to one day of intense planning to lay out the activities of the iteration day by day
- A couple of days to drill down the requirements for the feature(s) being delivered. The product manager/business analyst is fully engaged at this point, with engineers and QA participating
- A couple of days of designing the change solutions at a detailed level
- Week two may be all coding
- Documentation is created throughout the iteration
- Week three may be coding and testing the new features or all testing

Feature Planning and Prioritizing

Mapping each feature identified in the requirements to a release plan at the beginning of the process complements the concept of agile development. A release plan is a skeleton layout of what features will be delivered in which iteration, much like a project plan except that instead of scheduling phases, you are queuing up feature functionality into planned iterations. Initially, this will change relatively frequently as the process and estimates are tuned up. Release plans are very important for managing dependencies between products being developed. They are also a key success factor to reaching a common point for Integration Testing, if appropriate.
By prioritizing each of the features or functionality to be delivered and scheduling the higher priorities for delivery earlier in the process, you are providing flexibility that will give you options later in the process.

It is always a good idea to also identify where you have dependencies on changes in other products or by third parties and to schedule those up front as well. Then when different systems' schedules change, you are not at their mercy and can continue on your own identified path of development.

Refactoring

Refactoring is revisiting your architecture/design on an on-going basis. This should be done as a part of each iteration, at a minimum.

Agile development expects a certain amount of rework and refactoring. It enables some tuning in the earlier stages of the project. This is especially beneficial when utilizing newer technologies, where you may learn more as you go.

Feature Teams

Development teams are small and tightly integrated, consisting of engineers, business analysts and product managers (user representatives), quality assurance testers, documentation analysts, and the project manager. Not all team members will play equal roles in each iteration but the expectation is that all are committed to delivering the iteration in its entirety before moving on to the next iteration.

Daily Standup Meetings

Daily standup meetings last only fifteen to thirty minutes. They address upcoming tasks, schedule, communication between team members, and defect reporting and resolution. Solutions are not discussed in this meeting. Those are taken offline. The meetings generally occur every day at a stated time and are usually facilitated by the project manager or development manager.

These meetings are called standup meetings because, in practice, nobody sits down. Everyone stands up, thus guaranteeing that the meeting will be short and productive. Some proponents encourage standing in a circle and having each person speak to the status of their tasks from the previous day and their targets for the current day.

Daily standup meetings may be a difficult concept to accept or for your team to accept but they are really quite effective. I have found that there is a lot more hesitation in a more established and formal work environment.

> Example: In a company where the development staff's average age is probably around thirty and they wear jeans and t-shirts, you may find that they don't even question it. After all, some of their best ideas came from hallway conversations and it is not uncommon for impromptu quick meetings to emerge to solve the latest challenge using the newest technology.
>
> It may be a little more unfamiliar to the team who are all in business attire and where a team meeting is a very structured occasion, always facilitated by the manager.
>
> The standup part of this concept really just sets the tone of the meeting - quick, efficient, and full of updates. Many teams really sit down once they have gotten into the habit of whizzing through the meeting. I find it funny - there is a noticeable difference on how timely people are in getting to the standup when you really stand up, because all of their teammates start leaning and propping themselves, if they have to wait on people. Subtle, but effective.

Stories and Use Cases

These two components come from different methodologies and can be used alone or in conjunction with each other. Stories and Use Cases are two techniques which support the lightweight documentation model of most agile methods.

Use Cases (RUP) support requirements development. How will the function be used? Use Cases define the function in business (user) terms. In the case of our example…

> A clerk at CarWorld, an auto parts store needs to get a hold of a distributor cap for a 1963 Bentley. He knows the company has carried them in the past but his particular store does not carry them anymore. CarWorld has a published policy that if a car part exists, they can track one down. The first thing the clerk needs to check is if his company already has one available in another region.
>
> The clerk does a global company search by inputting the manufacturer's part number into their inventory system.
>
> The result returned to the clerk will be one of the following:
>
> 1. Product not found
> 2. Product found in region #x

A *Story* is basically a logical explanation of a feature. Imagine that you are describing some new function that some software has to a friend. You would do it in the form of a story. For more on Stories, refer to *Planning EXtreme Programming*, by Kent Beck and Martin Fowler.

A *Storyboard* is a sequencing of *Stories* (XP). They do exactly what you would expect them to do - they tell a story. By placing the stories in a specific order, it defines what work should be done when. Storyboards help to identify dependencies within the system or with other products' systems.

Task Selection and Estimation

In most agile methods, the concept of having team members select tasks among themselves and then estimate them is embraced. The practice encourages each person to take personal responsibility for the delivery of a task. When tasks and estimates are just assigned, people tend to do one or both of the following:

1. Lower the sense of urgency or importance with which an effort is accomplished, citing poor estimates by someone else.
2. Resent any extra effort required, as there is no sense of ownership in the estimation effort.

There may sometimes be tasks, which are lower on the list of desirables, but in a fully integrated team environment, the sharing of these tasks generally is pretty level. This is not too much of an

issue, in general, because in software development there are usually lots of different types of people who enjoy different aspects of the job.

Testing and Automation

The more testing automation that can be done, as you are developing product/system code, the more effective the agile methods will be. The less time it takes to run regressions and standardized non-feature specific tests, the more time will be spent on verification of the new feature functionality and the quicker the defect turnaround process becomes.

This is an important part of using agile development. If baseline testing is done manually, it inhibits the rapid development cycle because one of the *key rules is that the entire team stays in the current iteration until it is complete and passes its exit criteria.* However, automation has its drawbacks as well. Too much automation leaves the door open to oversight (nothing beats the human eye) and may also be inadvertently coded to actually force a positive result to a test. We don't often test our tests. Common sense would dictate that a combination of manual and automated tests makes sense. I have yet to see an environment where automated testing is the primary functional test, although test automation of the features to be delivered in an iteration is often considered to be a deliverable of the iteration. I believe that the best value of automated testing is on highly repeatable functions (maybe those chosen to be a regression test set) or those where large volumes of tests case executions are required.

Other Components

While the above components provide the essence of agile development, there are a few other features worth noting.

Pair Programming is where development tasks are done in teams of two, who closely and continuously integrate and review each other's work. While this is a specific component of the XP method, it is really a form of code review, which in any form adds value to the quality of delivery and is something you may want to consider including in each iteration.

Measuring velocity and re-planning accordingly is an on-going activity in most agile methods. By scheduling short development cycles, errors in estimation can be detected and corrected early in the project, which is a prime benefit. Velocity means the rate at which the development is occurring. Keep in mind, however, that estimates are not the only thing that might have caused fluctuation in the iteration delivery schedule. A key premise of the agile concept is to embrace change - changes to requirements and changes to design - to enable more accuracy up front and reduced change later in the project.

This assessment may also allow early detection of ideal vs. actual daily productive hours. If you planned on all team members having eight productive hours a day, you may have already overestimated how many resource hours are really available. This is true using any methodology but may not be evident as early as in the more traditional methods. Even planning for six

project-specific productive hours a day may prove to be ambitious, depending on your environment. If team members are also distracted with production support, meetings, time and status reporting, technical problems, and any other number of work-related activities, the project ideal hours may need to be lowered in future iterations.

General Observations

It is important to realize that some factors of agile development reduce or eliminate tools and techniques that development and business teams have become accustomed to relying on.

- XP tends to neglect or diminish the business modeling and architecture activities and puts less emphasis on documentation. It is important to realize this going into it and if your organization has many products that are highly integrated with each other, some adaptation of the methodology may be necessary from the onset. Otherwise, the result may be total abandonment of the methodology, which if applied more effectively could have significantly improved the velocity of output.
- RUP is tools intensive and is developed around the Rational Suite of UML (Unified Modeling Language), requirements, design, testing and source control products. It may seem like a lot of overhead for an agile method.
- SCRUM is more of a wrapper methodology that complements other methodologies, especially XP. It is a process for building software solutions incrementally in more complex environments.

Jim Highsmith, *Agile Software Development Ecosystems,* states, "Scrum's practices are nearly all project management and collaboration related, whereas XP's are predominantly software development and collaboration related." The ideal agile method for your organization may be a conceptual melding of best practices. The assessment of your organizational needs can help you to make better choices initially in selecting how to move toward the implementation of an agile methodology.

Agile development methods may appear to minimize the need for planning, but in application, that need is primary. A well-planned release of multiple projects is critical to successful use of agile development.

Consider how it would play out if one team goes off and plans delivery of their features without taking into consideration when other teams are delivering their supporting changes. Although it would be manageable using a waterfall methodology (because all delivery comes at the end of a development phase), it could be counterproductive to one or more projects' iteration completions. Remember, an iteration is not closed out until it is complete in all aspects, which includes any changes needed by other systems.

Benefits of Agile Development

1. One of the key benefits to using an agile development method is that you deliver feature functionality in its entirety in each iteration. By doing so, you could more quickly react to market demands by shortening your overall time to market by dropping functionality planned for a later iteration.

2. Agile development allows you to see more of the change much earlier in the project, so decisions for change can happen more efficiently and at less cost.

3. Agile development acknowledges that changes to requirements happen and allow the integrated team to react to them more quickly.

4. Speculative development (prototyping an idea) does not cost a lot of development time and can assist in making better business decisions.

Risk/Constraint Areas

(And how to minimize or mitigate them)

Integration

One of the things that stands out the most about agile development is that it really appears to be meant for single projects without dependencies on any other development projects. If an organization jumps into an agile model without thinking about their whole development area from a release management perspective, they may well find themselves with lots of projects out of sync with each other and with lowered productivity.

However, a well planned release using agile methods can be extremely successful and allow a more complex development organization to reap the benefits, just as fully as one with more simple stand-alone products. Some of the key success factors to successful release planning and integration are:

1. Keep a high-level, global schedule of any products that have dependencies between each other
2. Track dependencies closely and make sure that they are reflected in a logical fashion between products. For example, if one team needs changes to a database in another system in order for them to deliver 'x' function, then the iterations of both products must be lined up to deliver the database changes before the function that needs them.
3. Make sure that any change to one team's planned feature delivery schedule is communicated to other affected development teams. Weekly Release Management meetings with all affected managers, project managers, product managers, etc., can catch a lot of failure points before they happen. This small amount of time can save mammoth amounts of rework or delay.

4. Time is allotted in the global schedule for all integrated products or systems to run together and have end-to-end testing done. In reality, there are very few changes that we make to this complex web of technology that does not affect some other system. No new technologies or methodologies are going to eliminate that, so it is critical to manage to it.

Documentation

In exploring specific agile development processes, you may discover that documentation is not noted as a key deliverable. This is one of the most debated aspects of agile development in general. True, you may not have the volumes of documentation that you would in a more traditional model, but there is documentation. This is one area that you should evaluate extensively within your own organization before adopting any agile method.

This is always one of the first points of contention among managers discussing whether to consider moving to an agile model or not. After all, one of the main tenets of the Agile Alliance is "Working software over comprehensive documentation".. *taken directly from the Agile Alliance website at www.agilealliance.com.*

I have been around the block, or more accurately around the world, on this one. In fact I am somewhat of a misnomer in that I am a strong advocate of agile development and also a very strong advocate of documentation. After all, how can we expect anyone to use our wonderful products, if they can't install it and they don't know how to execute its features?

First of all, let's not regard the quote above as a severe rule. It is not likely that the Agile Alliance thinks that end user documentation does not need to be created, but we don't necessarily see that being addressed in most agile methods books. So, let's think of it logically. We know we need user documentation. That can simply be created as an expected deliverable out of each iteration and the iteration should not be regarded as complete until the functions of that iteration are fully documented. The documentation team members are as integral a part of the iterative development team as the developers and testers.

Secondly, let's assess what documentation actually is diminished. Here is one way to address the issue of system documentation during the iterations. Start of the iteration with a JAD session (Joint Application Development). JAD's have been around for quite some time and in my opinion are even more valuable in an agile world than they were in a traditional development environment.

During the JAD, one or many team members are designated to capture information during the JAD (using a laptop during the session prevents any rework) and send it out to the rest of the team following the session. This works very well when drilling down requirements or when crafting the architecture or solution design. It does not need to be a major effort and can be captured in a simple format. This will serve to reiterate the decisions made during the JAD, keep communication open and collaborative, and result in a draft which can be a major data source for the functional or technical documentation you deliver at the end of the iteration and then at the end of the project.

This is a good example of how important it is to assess how you are going to implement and agile method. What kind of a situation would your company be in if you leapt into agile development full force, following some methodology to the letter without weighing the importance of releasing your product to the market with guidelines as to its usage?

Requirements

Requirements are always a primary topic of conversation when discussing agile methods. How detailed should the requirements be before beginning development? This varies greatly depending on the environment. If you are working on new product development and the 'inventors' or product managers are actively part of the project team, the detailing of requirements may more easily fit into the iterative model. A prototype model can easily be reviewed and altered by the team as part of the delivery cycle.

If your user representative (product manager, business group, etc) can only devote a limited amount of time to the project, then more detailed requirements are needed before beginning the development. As we all know from experience, even in this model, more requirements drill-down will occur.

The use of requirements cards is a great way to enhance your agile experience. Each requirements card should contain the following type of information generally:

1. Requirement number – open field – should allow for a naming convention to be established by the team
2. Requirement category – if applicable
3. Statement of the requirement
4. Fit criterion – this is a statement of how you would test to know that the code met the requirement. It is also a good place to state more specifics about the requirement (drill-down)
    a. Example – Requirement states 'Add the ability for customer to modify their customer profile information'. Fit criterion might be:
        i. Allow update capabilities to name, address, phone number, credit card number, x, x, and x
        ii. Do not allow updates to customer account number
        iii. Provide a link to an update screen from the following screens:
            1. Home
            2. My Account
        iv. Require authentication before allowing access to customer profile update screen
        v. Display current information
5. Dependencies
6. Priority – how important is this feature. Note that the requirement may be deferred until a later date if the delivery date is at risk
7. Estimate of effort to complete
8. Actual effort it took to complete

a. Estimates and actuals can then be captured and tracked by the project manager for use in refining how much can be done in each iteration or for improved project planning overall. Project Managers may choose to incorporate these numbers into a Cone of Uncertainty (semi-standard project management practice, not specific to agile development) model later

This sounds like a lot, doesn't it? Well, it isn't. Most of this can fit on a standard 5½" x 8½" card (half a standard letter sheet).

Developers work from these cards, keep them updated with effort hours and requirements changes, and turn them into the project manager a couple of times during the iteration and voila, that is all they have to deal with for entering their time or from documenting requirements. Moments, not hours.

The whole team benefits from using the cards. The manager or project manager can gather them up to produce more detail on their project plan and track the time estimated/spent, the documentation team member can use them to document user functionality, and the product manager can use them to keep the requirements document up to date, if this is something that your model utilizes.

Simple, quick, useful, all the while reducing the risk often expressed by this part of the agile development philosophy.

Architecture

If the product is new and it is a stand-alone product, the situation is more conducive to designing the solution as you go. However, as most products or systems integrate and send data to and from other entities, both internal and external, it is generally necessary to do some level of architectural planning before the small teams go off to work in the iterative world. Assuming that whatever the project team develops will be adapted to by the other systems is simply not realistic.

Assessing Your Business and Technical Environments

Now that you know about the concepts of agile development and some of the techniques of the specific methodologies, you need to beginning looking at whether all or part of the methods will be of benefit to you. There are three options that may result from this assessment.

1. Do not venture into the agile development space
2. Plan to shift to an agile method but adapt it to better fit your organization
3. Adopt a select agile method and follow it according to its properties

I. *What general type of business is your organization in?*

a. Does it provide a product that is sold to other organizations?
b. Does it provide a product that is sold to public consumers?
c. Does it provide services and is doing internal system development to support those services?

(a) If you sell your product to other organizations, agile development works best in the following manner:

- ✓ Make sure that you have product managers who know the market and have a stronghold into the world of those organizations.
- ✓ Make sure that product manager has the ability to develop an initial marketing level set of requirements, which adequately represent the needs of the organizations you are targeting for sale.
- ✓ Make sure that product manager has time allocated to work closely with the development team throughout each iteration.

(b) If you sell your product to public consumers, agile development works best in the following manner:

- ✓ Developers are users of the product, or
- ✓ Business Analyst or Product Managers who use the product are on the project team
- ✓ Product Managers keep a pulse on the industry through trade shows, industry periodicals, and news releases

(c) If you are doing development for internal systems and have access to user representatives:

- ✓ Continue your analysis
  - o Are your systems relatively free standing or tightly integrated?
  - o If integrated, is the communication between product areas clear and consistent?
  - o Are your systems mainframes or client server?
    - ▪ Enhancements to a mainframe system require that more change be introduced before turning the product over, as the process to compile and test is not as rapid as with client server applications.

II. *Is your organization in a position to do integration testing between iterations to ensure that interfaces between systems are designed effectively and the all systems can transport data to and from each other smoothly?* (See Risk/Constraints are above for more information on Integration)

III. *Do you have to deal with mandated dates?*

- ✓ Having to work against an unmoving deadline does not mean that you can't use Agile Development but it does mean that you need to reassess how you will apply it.

- As with the more traditional life cycles, working with mandated dates simply means that you may need to de-scope how much you are delivering.
- So, what if you are using a traditional methodology and you find out that you did not scope how much could be delivered efficiently enough? You would probably figure that out about halfway through your coding phase, if you are lucky. So now what would you do? Try to make it anyway and risk the deadline or de-scope now and deal with the impacts to your design? Either one is going to be very time costly and will endanger your final delivery date

✓ Using agile development would allow you to deal with the situation by simply chopping off iterations (which, if well planned – scheduling higher priority features earlier in the release plan). There would be some recovery to deal with but the risk would be less.

IV. Do you have management support, throughout your organization for process changes?

✓ Recognize the fact that you will probably stumble a little
- Make sure your organization is prepared for the bumps and is willing to ride it out. This applies to any process change.

In Summary

Taking the step into the agile universe is a big one and cannot be entered into lightly. It is not enough to read a book or go to a seminar and come back ready to roll. Agile methods provide a lot of benefits but it is important to take a good look at them first. Question the process you are looking at. Try to imagine where you may experience problems, based on the complexion of your company. Above all, make sure that you have management buy-in and that they realize that the first time will not be perfect and that you will be molding the process for a while to come.

References

Fowler, M. and Highsmith, J., "The Agile Manifesto", a paper summarizing the formation of the Agile Alliance and its purpose.

Fowler, M. and Beck, K., "Planning Extreme Programming", copyright 2001 by Addison-Wesley

Krutchen, P., "The Rational Unified Process, An Introduction, Second Edition, copyright 2000 by Addison-Wesley

Cockburn, Alistair, "Agile Software Development", copyright 2002 by Pearson Education, Inc.

Using XML for Test Case Definition, Storage and Presentation

Michael Ensminger (mensming@ieee.org)
PAR3 Communications

Abstract:

The current interest in web services fuels the rise of XML as a popular technology. The advantages provided by XML in application development and enabling application integration apply equally to supporting the test effort. This paper provides a short overview of XML and related technologies, and an example of the application of these technologies to test management. A simple test case DTD and example XML document is developed. This XML is then transformed into an HTML page via an XSL stylesheet. These simple examples demonstrate the power of this approach and also identify some of the issues that are encountered during implementation. Building on the simple example, a more complex (and realistic) definition of test suite XML is developed using XML schema. Finally, some additional XML documents and their uses in a test management system are discussed.

About the author:

Michael Ensminger is Director of Quality Assurance at PAR3 Communications. He brings extensive experience and education to engineering management, having managed engineering teams and processes to produce high-quality web sites, desktop applications, enterprise, and data products. He holds a BS in Mathematics and a MS in Computer Science.

Prepared and presented at the 2002 Pacific Northwest Software Quality Conference

Extensible Markup Language (XML) has been called the "duct tape of the internet" by some in the media. Like duct tape, which can be regarded as a standard "tool", XML has crept into the software developer's toolbox. This paper discusses how a test team can add XML to their standard set of tools as part of a test management system. In particular, we develop an XML document type that can be used to store a test case and later expand that document type to describe a full test suite. Our goal is to keep the presentation logic outside of the XML document. We then develop an XSL stylesheet that will be used to display the test cases to the tester as an HTML document.

Motivation to use XML for test case management

Personnel experience / tool availability

My employer, PAR3 Communications, uses XML extensively. We use XML, sent over HTTPS connections or within messages sent via MQSeries queues, to communicate between the various components of the PAR3 solution. As such, all of the development and test personnel are intimately familiar with XML and its uses. Extending the use of XML to store test case data was a natural progression. Also, since we are already using XML in the product proper, the team has access to various XML support tools. As the popularity of XML continues to rise, the availability and capabilities of both commercial and open source tools improve.

Easily processed / extended / transformed

XML provides several other advantages for use in test case management. XML documents are easily processed. MSXML (http://msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/msdn-files/027/001/766/msdncompositedoc.xml) from Microsoft and Xerces (http://xml.apache.org/xerces2-j/index.html, http://xml.apache.org/xerces-c/index.html, http://xml.apache.org/xerces-p/index.html) from the Apache Software Foundation provide all of the validation and parsing capabilities we needed. Once parsed, the various components of an XML document may be manipulated easily. XML documents are also easily extended with little risk of breaking existing code. In most cases, existing code will ignore the additional elements resulting from an extension. Finally (and perhaps most importantly), XML provides the ability to be transformed and through this mechanism it is possible to easily separate the data and presentation layers.

Web services

If you pick up any recent trade magazine, you will find a reference to web services. Web services are essentially another way to integrate applications, this time using XML and HTTP. As a way to explore web services, we discuss the development of a homegrown test case management system using this technology. The first step towards development of such a test case management system is to represent test cases as XML documents and present the test cases as HTML pages.

Brief technology overview

This paper is not meant to be a tutorial on XML, DTDs, XML schema, XSL or Web Services and the supporting technologies. However, the following definitions and references are supplied to provide the proper context for the rest of the paper.

XML - Extensible Markup Language. A meta language, that allows users to define their own set of conventions for marking up data. The primary advantage of XML over other data formats and representations is that it allows data to be represented in a platform independent fashion. The set of conventions and other standards developed around XML provide the real advantages to XML. For additional information, see http://www.w3.org/XML/.

DTD - Document Type Definition. Mechanism to describe an XML document type. By including or referring to a DTD, data structures and some data content in an XML document can be validated.
For additional information, see http://www.w3.org/TR/REC-xml#dt-doctype.

XML schema - Another (stricter) mechanism for describing an XML document type. The XML schema standard intends to improve on DTDs. Some of the improvements are support for data types and additional data validation. Unlike a DTD, an XML schema description of a document type is itself an XML document.
For additional information, see http://www.w3.org/XML/Schema.

XSLT - Extensible Stylesheet Language for Transformations. Language that allows XML documents to be transformed into some other format. A typical use is to format an XML document for display as an HTML document. Using an XSL stylesheet, data and presentation can be effectively split.
For additional information, see http://www.w3.org/Style/XSL/.

Web services are the current fad in the technical press. It is the latest silver bullet regarding application integration. Web services revolve around three standards.

SOAP - Simplified Object Access Protocol. An inter-application communications framework based on XML and HTTP.
For additional information, see http://www.w3.org/2000/xp/Group/.

UDDI - Universal Description, Discovery and Integration. Directory service for discovering available web services.
For additional information, see http://www.uddi.org/.

WSDL - Web Services Discovery Language. Standard for describing a web service including input, outputs and calling mechanism.
For additional information, see http://www.w3.org/TR/wsdl.

A simple test case

Our discussion begins with the simplest of test cases. While there is no universal agreement to the contents of a test case, the following items may be considered the minimum:

- Some type of unique identifier
- A short description of the purpose of the test case
- Setup steps
- Expected results

Throughout this discussion, keep in mind the goal of separating the test case data from its presentation.

Sample XML test case documents

Below are some examples of test cases defined in XML that incorporate the four points above. The first example shows a typical user interface test case:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- Simple test case example 1 -->
<simple-test-case id="1001">
   <case-shortdesc>Error handling for temperature setting greater than
         maximum</case-shortdesc>
   <case-setup>
      <setup-step>Display the temperature adjustment screen</setup-
         step>
      <setup-step>Enter a temperature value greater than 250
         degrees</setup-step>
      <setup-step>Press the submit button</setup-step>
   </case-setup>
   <case-result>
      <result>The maximum temperature exceeded error popup is
         displayed</result>
      <result>The input screen remains displayed</result>
   </case-result>
</simple-test-case>
```

The same structure can be used for a developer's unit test:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- Simple test case example 2 -->
<simple-test-case id="1002">
   <case-shortdesc>Module: Temperature, Store Setting Value</case-
         shortdesc>
      <case-setup>
         <setup-step>Execute the method, setTemperature passing in a
         valid value</setup-step>
         <setup-step>Force the temperature module to persist</setup-
         step>
   </case-setup>
   <case-result>
      <result>Temperature value is stored in currentTemperature field
         in the configuration file</result>
   </case-result>
</simple-test-case>
```

Each one of these documents can be validated against a DTD to verify that it is well formed and valid. In terms of a test case management system, the data validation logic is simplified by taking advantage of this capability.

Simple test case DTD

A DTD describing the test case XML may look like this:

```
<!-- simple_test_case.dtd -->
<!ELEMENT simple-test-case (case-shortdesc, case-setup, case-result) >
<!ATTLIST simple-test-case
   id ID #REQUIRED
>

<!ELEMENT case-shortdesc ( #PCDATA ) >

<!ELEMENT case-setup ( setup-step+ ) >

<!ELEMENT case-result ( result+ ) >

<!ELEMENT setup-step ( #PCDATA ) >

<!ELEMENT result ( #PCDATA ) >

<!-- End simple_test_case.dtd -->
```

This DTD describes an XML structure that consists of a simple-test-case element. The element has one attribute, id, meant to satisfy the requirement of some type of identifier for the test case. Within the element, there must exist exactly one case-shortdesc element, one case-setup element and one case-result element (in that order). The case-shortdesc provides for a short description of the purpose of the test case. #PCDATA indicates that character data can exist between the start and end case-shortdesc elements. The case-setup element provides for setup steps. Within the case-setup element there must be one (or more) setup-step elements. Each setup-step element contains character data. Similarly, the case-result element may contain one or more result elements. Finally, the result element contains character data describing the results.

Other items that might be included

We will use this simple document type for a while in our discussion. However, it is useful to discuss how this document type could be extended. In most cases, some information regarding the test case will be desired. Sample information would include the author, date created, date modified, test case version, etc. Adding a test-case-info tag or some other construct could easily extend the document type. Such additions should have no effect on our display logic until we wish to display the additional data. Other elements we may wish to add are environmental needs, special procedures, intercase dependencies, etc. Test automation notes are also useful to include. Properly designing the document type will allow the test case structure to be extended while minimally impacting existing documentation and supporting tools.

While we have been discussing the definition of a test case, in most cases the test personnel will be interacting with a suite of test cases. It is fairly easy to create a test suite document type that allows for multiple test cases to be included. Later in this paper, we will explore such a document definition as a full-blown example.

Storage of XML

Test management requires that the test cases and the associated data be somehow stored in a persistent manner. The way to do this runs the gamut from text files, Microsoft Word or Excel documents, relational database systems, commercial test management systems, etc. Similarly, many of these options are available for storing test case (or any) XML documents.

Flat file

XML documents themselves consist of plain text containing the data and markup tags representing the structure of the data. The simplest manner of storing XML documents is in individual text files. Storing these documents in a location accessible to a web server allows a browser to retrieve the XML documents and apply XSL stylesheets with no additional work. Source code control systems are easily used to track changes to test cases as individual files. However, this simplicity comes at a price. Organization of individual files grows exponentially more difficult as the number of files grows. Maintenance, especially making the same change across multiple test cases, becomes more difficult as the number of files increases.

Relational database

By far, the most common way to store XML data is in a relational database. Much has been written about the "object impedance mismatch" between the relational model and XML structure. This mismatch refers to the fact that XML data are hierarchically organized, and not necessarily exist in normalized table form. And in some cases, significant effort may be required to map between data in an XML structure and in a relational database. However, for our purposes the relational database can easily be used to store the data. Consider the various elements in our simple test case XML. The following SQL statements can create a database schema for storing our test case data:

```
create table simple_test_case (
   CASEID     integer not null,
   SHORTDESC  varchar(255) not null,
   PRIMARY KEY (CASEID),
   UNIQUE (CASEID))

create table case_setup (
   CASEID     integer not null,
   SEQUENCE   integer not null,
   STEPTEXT   varchar(255) not null,
   PRIMARY KEY (CASEID, SEQUENCE),
   FOREIGN KEY (CASEID) REFERENCES simple_test_case)

create table case_result (
   CASEID     integer not null,
```

```
    SEQUENCE   integer not null,
    RESULTTEXT varchar(255) not null,
    PRIMARY KEY (CASEID, SEQUENCE),
    FOREIGN KEY (CASEID) REFERENCES simple_test_case)
```

This database schema allows us to easily store our test case data. For each test case, there is a row in the simple_test_case table. This table contains all of the data regarding the test case except for the individual setup steps and individual results. The case_setup and case_result tables store the individual setup steps and results respectively. In each of these tables there is a foreign key relationship to the proper simple_test_case item. Each step / result contains the case id, a sequence number (to maintain the original order defined in the XML) and the proper text.

Once an XML document is parsed, storing the data from the individual elements into a database is straightforward. Likewise, reconstructing the XML from the database is more an exercise in string manipulation than SQL know-how.

XML database

With the rise in popularity of XML, database vendors have been developing products to support it. Two different approaches are common. The first extends the traditional relational database feature set by adding native support for XML. Microsoft, Oracle, and IBM have all taken this approach by allowing XML to be a native data type and providing a suite of tools for manipulating XML. The second approach is the development of so-called XML databases, designed from the ground up to support XML. Many of these databases are just coming on the market and currently are used in specialized applications.

Either approach would work with regards to a test case management system. Each provides a method for storing XML with a minimal amount of programming. Similarly, the data can be retrieved as XML. Some of these tools can be configured to apply the XSL transformation when the data is retrieved, once again simplifying the development of a test case management system. The author's experience with either approach is limited and the most that can be said is that the jury is still out.

Using XSL to present the test case data

One of the motivations for using XML to store test case data is separating presentation from data. One way to achieve this aim is to encapsulate the presentation logic in an XSL stylesheet. A common use is to apply an XSL stylesheet to transform the data in an XML document into an HTML document for presentation in a web browser. The XSL stylesheet contains all of the logic necessary to select, transform and reformat the data for display. This XSL stylesheet contains instructions in a transform language, XSL, which can handle data that are of variable number or length, missing data, optional data, data elements to be ignored, non-XML data related items such as links to status recording applications, etc. If output to multiple browser types must be supported, the XSL stylesheet can contain the logic and different HTML output required to support browser specific features. Conditional logic allows the presentation to vary based on particular data values. Whenever the presentation needs to be changed, one source file, the XSL stylesheet, need be modified and the presentation change is applied to all test cases.

A more generic advantage of XSL is transforming the original document into a form needed by another application. Above, we described how to use XSL to transform the XML data into a form needed by a web browser, namely HTML. XSL can be used to transform the data into another XML document, a TEX document, a plain ASCII text document, a binary document such as PDF or Microsoft Word (if you know the file format), etc. Multiple XSL transformations can be applied chaining the output of one to the input of another to achieve the desired result. A non-presentation example of XSL usage is transforming the original test cases in the XML document into a series of SQL statements, which would insert the test case data into a relational database.

Simple XSL stylesheet / example transformation

Consider the following simple XSL stylesheet to transform the simple-test-case XML document into HTML for display:

```
<xsl:template match="simple-test-case">
   <h3><xsl:value-of select="@id"/>: <xsl:value-of select="case-
         shortdesc"/></h3>
   <b>Setup:</b>
   <ol>
      <xsl:apply-templates select="case-setup"/>
   </ol>
   <b>Expected Result:</b>
   <ol>
      <xsl:apply-templates select="case-result"/>
   </ol>
   <hr/>
</xsl:template>

<xsl:template match="case-setup">
   <xsl:apply-templates select="setup-step"/>
</xsl:template>

<xsl:template match="case-result">
   <xsl:apply-templates select="result"/>
</xsl:template>

<xsl:template match="setup-step | result">
   <li><xsl:value-of select="."/></li>
</xsl:template>
```

Note that this stylesheet does not generate the necessary HTML header and body tags. The transformation starts by matching a simple-test-case element in the input (XML) stream. Once a match is found, the first template in the file is invoked, which is specifically defined for processing simple-test-case elements. This template emits a header level 3 HTML element, followed by the test case id, a colon, a space, and then the contents of the case-shortdesc element. Some other formatting and output are done. Then the processor is told to apply the case-setup template items, which in turn applies the setup-step template. This setup-step template will process and format output for all of the setup steps in a consistent manner.

Similarly, the case-result element is handled by yet another template, where results output is generated.

Applying this stylesheet to the first XML example yields the following snippet of HTML code:

```
<h3>1001: Error handling for temperature setting greater than
         maximum</h3>
<b>Setup:</b>
<ol>
   <li>Display the temperature adjustment screen</li>
   <li>Enter a temperature value greater than 250 degrees</li>
   <li>Press the submit button</li>
</ol>
<b>Expected Result:</b>
<ol>
   <li>The maximum temperature exceeded error popup is displayed</li>
   <li>The input screen remains displayed</li>
</ol>
<hr>
```

Applying this stylesheet to the second XML example file yields:

```
<h3>1002: Module: Temperature, Store Setting Value</h3>
<b>Setup:</b>
<ol>
   <li>Execute the method, setTemperature passing in a valid value</li>
   <li>Force the temperature module to persist</li>
</ol>
<b>Expected Result:</b>
<ol>
   <li>Temperature value is stored in currentTemperature field in the
         configuration file</li>
</ol>
<hr>
```

The stylesheet becomes more interesting when you include multiple test cases in one XML document to create a test suite and include a mechanism to provide feedback. A more full featured example is presented below.

Beyond the basics: XML schema definition and additional XSL stylesheets

For our simple-test-case example, a DTD provided a fairly adequate definition of the elements and types of data in our XML document. However, when we defined the database schema for storing our test case data, some restrictions were put in place that are not reflected (nor can they be) in the DTD. Thus it is possible to have an XML document that passed validation against the DTD but will be invalid in another (say, database storage) application. For example, as defined in the database schema, the ID must be an integer; description, setup steps and individual results are at most 255 characters, etc. XML schema was developed to allow a finer level of detail to be specified for XML content, and permit more extensive validation on parsing, before the data gets to the application.

Simple test case DTD transformed into XML schema

XML schema definitions themselves are XML documents. Below, the simple-test-case element is defined in a XML schema document:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:mee_stc="http://home.attbi.com/~mensming/papers/PNSQC02/simpl
    e-test-case"
    targetNamespace="http://home.attbi.com/~mensming/papers/PNSQC02/sim
    ple-test-case">

   <xsd:element name="simple-test-case" type="SimpleTestCaseType" />

   <xsd:complexType name="SimpleTestCaseType">
      <xsd:sequence>
         <xsd:element name="case-shortdesc" type="VarChar255Type" />
         <xsd:element name="case-setup" type="CaseSetupType"  />
         <xsd:element name="case-result" type="CaseResultType" />
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:integer" />
   </xsd:complexType>

   <xsd:simpleType name="VarChar255Type" >
      <xsd:restriction base="xsd:string" >
         <xsd:minLength value="1" />
         <xsd:maxLength value="255" />
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:complexType name="CaseSetupType" />
      <xsd:sequence>
         <xsd:element name="setup-step"   type="VarChar255Type"
            minOccurs="1" maxOccurs="unbounded" />
      </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name="CaseResultType" />
      <xsd:sequence>
         <xsd:element name="result" type="VarChar255Type" minOccurs="1"
            maxOccurs="unbounded" />
      </xsd:sequence>
   </xsd:complexType>

</xsd:schema>
```

In this schema, more information is given to help the XML parser validate a simple-test-case data element. The simple-test-case data element is defined as being composed of a sequence of sub-elements, namely a case-shortdesc, a case-setup and a case-result. Further the simple-test-case data element is defined as having an attribute called an "id", which must take an integer value. The subsequent definitions for the sub-elements follow, each definition containing additional definitions and restrictions on the data as appropriate.

More advanced (and realistic) test case / suite XML schema

Now that we have developed a full-fledged XML schema definition for our simple test case document, let us extend it. We define a test suite element type that includes some of the enhancements we discussed earlier. Consider the following example:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- More complete test suite example -->
<PNSQC02-test-suite suite-type="Smoke" suite-name="Set Target
        Temperature" suite-abbr="STT">

   <test-suite-info name="FirstCreated" value="06/30/2002" />
   <test-suite-info name="LastUpdated" value="07/15/2002" />

   <suite-note>This test suite should be run on every build received
        from the development team.
   </suite-note>

   <test-case id="2001">
      <test-case-info name="CreatedBy" value="MEE" />
      <test-case-info name="CreatedOn" value="06/30/2002" />
      <test-case-info name="UpdatedOn" value="07/15/2002" />

      <autotest-note>Use automated testing script: set_temperature.scr
      </autotest-note>

      <case-shortdesc>Able to enter a valid temperature</case-
         shortdesc>

      <case-purpose>Verify the behavior of the temperature option
         value</case-purpose>

      <case-setup>
         <setup-step>Select "Edit", "Options" from the main
         menu</setup-step>
         <setup-step>Select the "Temperature Tab"</setup-step>
         <setup-step>Enter a target temperature between 75 and 250
         degrees</setup-step>
         <setup-step>Press the "Apply" button</setup-step>
      </case-setup>

      <case-result>
         <result>The options window is dismissed</result>
         <result>The new target temperature is the value
         entered</result>
      </case-result>
   </test-case>

   <test-case id="2002">
      <test-case-info name="CreatedBy" value="MEE" />
      <test-case-info name="CreatedOn" value="06/30/2002" />

      <case-shortdesc>Error handling: Temperature above maximum</case-
         shortdesc>
```

```
      <case-purpose>Verify the error handling of the temperature option
         value</case-purpose>

      <case-setup>
         <setup-step>Select “Edit”, “Options” from the main
         menu</setup-step>
         <setup-step>Select the “Temperature Tab”</setup-step>
         <setup-step>Enter a target temperature above 250
         degrees</setup-step>
         <setup-step>Press the “Apply” button</setup-step>
      </case-setup>

      <case-result>
         <result>The options window continues to be displayed</result>
         <result>An error message is displayed giving the valid range
         of temperatures</result>
         <result>The original target temperature is still set</result>
      </case-result>
   </test-case>

   <test-case id=“2003”>
      <test-case-info name=“CreatedBy” value=“MEE” />
      <test-case-info name=“CreatedOn” value=“06/30/2002” />
      <test-case-info name=“UpdatedOn” value=“07/15/2002” />

      <case-shortdesc>Error handling: Temperature below minimum</case-
         shortdesc>

      <case-purpose>Verify the error handling of the temperature option
         value</case-purpose>

      <case-setup>
         <setup-step>Select “Edit”, “Options” from the main
         menu</setup-step>
         <setup-step>Select the “Temperature Tab”</setup-step>
         <setup-step>Enter a target temperature below 75
         degrees</setup-step>
         <setup-step>Press the “Apply” button</setup-step>
      </case-setup>

      <case-result>
         <result>The options window continues to be displayed</result>
         <result>An error message is displayed giving the valid range
         of temperatures</result>
         <result>The original target temperature is still set</result>
      </case-result>
   </test-case>

   <test-case id=“2004” inactive=“true”>
      <test-case-info name=“CreatedBy” value=“MEE” />
      <test-case-info name=“CreatedOn” value=“06/30/2002” />
      <test-case-info name=“UpdatedOn” value=“07/15/2002” />

      <case-shortdesc>Set the default target temperature</case-
         shortdesc>
```

```
      <case-purpose>Verify the default target temperature option
         value</case-purpose>

      <case-setup>
         <setup-step>Select “Edit”, “Options” from the main
         menu</setup-step>
         <setup-step>Select the “Temperature Tab”</setup-step>
         <setup-step> Enter the default target temperature below 75
         degrees</setup-step>
         <setup-step>Press the “Apply” button</setup-step>
      </case-setup>

      <case-result>
         <result>The options window is dismissed</result>
         <result>The new default target temperature is the value
         entered</result>
         <result>The original target temperature is still set</result>
      </case-result>
   </test-case>

</PNSQC02-test-suite>
```

This example is a little bit longer but contains much more information. First, it is a suite of test cases. When the suite was created and last modified is included. The suite is classified as a “Smoke” test. There is a note mentioning that the suite should be executed with each build received from the development team.

Four test cases are defined, one of which is marked as inactive (perhaps because it was defined to test functionality that has not been implemented). Each test case includes information regarding the original creator, when created and, in one case, when updated. One test case includes an automation note referring to the script that is used to execute that test case. There is a description and purpose for each test case as well as setup steps and expected results.

The structure of this example is defined below via an XML schema.

```
<?xml version=“1.0” encoding=“UTF-8” ?>
<xsd:schema xmlns:xsd=“http://www.w3.org/2001/XMLSchema”
    xmlns:mee_tsuite=“http://home.attbi.com/~mensming/papers/PNSQC02/te
    st-suite”
    targetNamespace=“http://home.attbi.com/~mensming/papers/PNSQC02/tes
    t-suite”>

   <!-- XSL annotations -- use for documentation, copyrights, etc. -->
   <xsd:annotation>
      <xsd:documentation>Example Full Fledged Test Suite
         Schema</xsd:documentation>
   </xsd:annotation>
   <xsd:annotation>
      <xsd:documentation>Copyright 2002 Michael E. Ensminger.  All
         Rights Reserved.</xsd:documentation>
   </xsd:annotation>

   <!-- Define out root element of the document -->
```

```
<xsd:element name="PNSQC02-test-suite" type="TestSuiteType" />

<!-- Define the contents and attributes of the data type
      TestSuiteType -->
<xsd:complexType name="TestSuiteType" >
   <xsd:sequence>
      <xsd:element name="test-suite-info" type="InfoType"
         minOccurs="0" maxOccurs="unbounded" />
      <xsd:element name="suite-note" type="NoteType" minOccurs="0"
         maxOccurs="unbounded" />
      <xsd:element name="test-case" type="TestCaseType"
         minOccurs="0" maxOccurs="unbounded" />
   </xsd:sequence>
   <xsd:attribute name="suite-type" type="SuiteType"
      use="required"/>
   <xsd:attribute name="suite-name" type="VarChar255Type"
      use="required"/>
   <xsd:attribute name="suite-abbr" type="SuiteAbbrType"
      use="required"/>
   <xsd:attribute name="language"   type="xsd:lang" />
</xsd:complexType>

<!-- VarChar255Type - non-empty string with a max length of 255 -->
<xsd:simpleType name="VarChar255Type" >
   <xsd:restriction base="xsd:string" >
      <xsd:minLength value="1" />
      <xsd:maxLength value="255" />
   </xsd:restriction>
</xsd:simpleType>

<!-- InfoType - Used to specify test case information -->
<!-- Possible name values: author, version, when_created,
      when_updated, etc. -->
<xsd:complexType name="InfoType" >
   <xsd:attribute name="name" type="xsd:NMTOKEN" />
   <xsd:attribute name="value" type="xsd:string" use="required" />
</xsd:complexType>

<!-- NoteType - Allows for tags (mixed content) and other items to
      be included. -->
<xsd:complexType name="NoteType" mixed="true" >
   <xsd:sequence>
      <xsd:any namespace="##any" processContents="lax" minOccurs="0"
         maxOccurs="unbounded" />
   </xsd:sequence>
</xsd:complexType>

<!-- TestCaseType - Extension of the simple test case type -->
<xsd:complexType name="TestCaseType">
   <xsd:sequence>
      <xsd:element name="test-case-info" type="InfoType"
         minOccurs="0" maxOccurs="unbounded" />
      <xsd:element name="case-shortdesc" type="VarChar255Type" />
      <xsd:element name="case-purpose"   type="VarChar255Type" />
      <xsd:element name="case-note" type="NoteType" minOccurs="0"
         maxOccurs="unbounded" />
```

```
         <xsd:element name="autotest-note" type="NoteType"
            minOccurs="0" maxOccurs="unbounded" />
         <xsd:element name="case-setup" type="CaseSetupType"  />
         <xsd:element name="case-result" type="CaseResultType" />
         <xsd:element name="dependency" type="xsd:integer"
            minOccurs="0" maxOccurs="unbounded" />
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:integer" use="required"/>
      <xsd:attribute name="inactive" type="xsd:boolean"
         default="false" />
   </xsd:complexType>

   <!-- SuiteType - enumerate types of test suites -->
   <xsd:simpleType name="SuiteType" >
      <xsd:restriction base="xsd:token">
         <xsd:enumeration value="Smoke" />
         <xsd:enumeration value="Integration" />
         <xsd:enumeration value="Load" />
         <xsd:enumeration value="Performance" />
      </xsd:restriction>
   </xsd:simpleType>

   <!-- SuiteAbbrType - test suite abbreviations maximum 4 characters
         long -->
   <xsd:simpleType name="SuiteAbbrType" >
      <xsd:restriction base="xsd:token" >
         <xsd:minLength="1" />
         <xsd:maxLength="4" />
      </xsd:restriction>
   </xsd:simpleType>

   <!-- Case setup may contain 1 or more setup steps -->
   <xsd:complexType name="CaseSetupType" />
      <xsd:sequence>
         <xsd:element name="setup-step" type="VarChar255Type"
            minOccurs="1" maxOccurs="unbounded" />
      </xsd:sequence>
   </xsd:complexType>

   <!-- Case results may contain 1 or more results -->
   <xsd:complexType name="CaseResultType" />
      <xsd:sequence>
         <xsd:element name="result" type="VarChar255Type" minOccurs="1"
            maxOccurs="unbounded" />
      </xsd:sequence>
   </xsd:complexType>

</xsd:schema>
```

This XML schema definition is more complicated in addition to being more complete. Eight custom types are defined. One, NoteType, allow mixed content – an element of this type may contain text as well as other unspecified elements. This could allow us to break our rule regarding separating data and presentation by placing HTML markup tags within the note. It also allows us to leave open the possibility of defining additional custom types that could be

included in the notes sections. Also examine the SuiteAbbrType. This type takes advantage of the enumeration capabilities to predefine a set of values.

One additional note regarding this schema, unrelated to the test content. This schema defines the necessary namespaces (used to allow multiple XML elements to exist with the same name without conflicting with one another). The "xsd" namespace contains those elements that are used to define items in the XML schema definition. The "mee_tsuite" namespace is defined and then set as the default namespace for the elements that are defined. We will now develop an XSL stylesheet for this document type.

More advanced XSL transformation

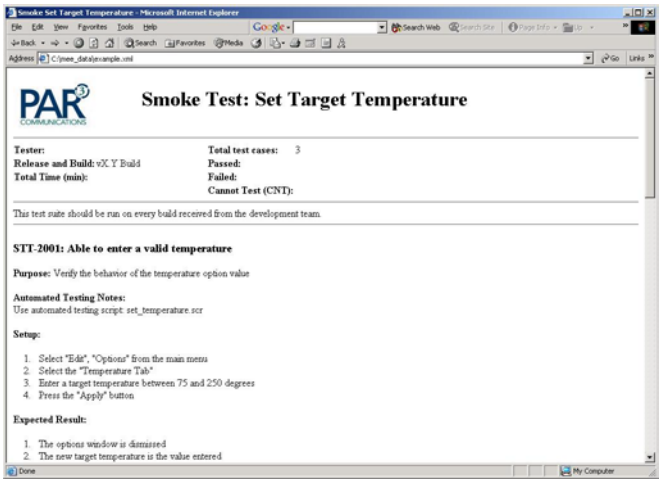
The following XSL stylesheet is more complicated than our previous example. Since the PNSQC02-test-suite element (defined in the schema above) allows for certain sub-elements to occur 0 or more times (such as the autotest-note and dependency elements), this must be taken into account when generating the output. Also, the addition of an inactive flag requires us to exclude those cases from display.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet   version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="html" version="4.0" encoding="iso-8859-1"
    indent="yes" />

<!--   use   -->
<!-- &copy; use &#169; -->
<!-- <hr> use <hr/> -->
<!-- <br> use <br/> -->
<!-- <p> use <p/> -->

<xsl:template match="PNSQC02-test-suite">
   <html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-
         8859-1" />
      <title>
         <xsl:value-of select="/PNSQC02-test-suite/@suite-type"/>
              Test: <xsl:value-of select="/PNSQC02-test-
              suite/@suite-name"/>
      </title>
   </head>
   <body>
      <table WIDTH="100%" BORDER="0" CELLPADDING="0" CELLSPACING="0" >
         <tr><td>
         <img SRC="/~mensming/papers/PNSQC02/images/YourLogo.jpg"
              ALT="Your Company or Project Logo" BORDER="0" />
         </td>
         <td>
         <h1><xsl:value-of select="/PNSQC02-test-suite/@suite-
              type"/> <xsl:value-of select="/PNSQC02-test-
              suite/@suite-name"/></h1>
         </td></tr>
```

```
      </table>

      <hr/>
      <table BORDER="0" CELLSPACING="0">
         <tr><td><b>Tester: </b></td>
         <td></td>
         <td><b>Total test cases: </b></td>
         <td align="left"><xsl:value-of select="count(//test-case) -
              count(test-case[@inactive='true'])"/>
         </td></tr>
         <tr><td><b>Release and Build: </b></td>
         <td>vX.Y Build
              &amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;&amp;#160;</td>
         <td><b>Passed: </b></td>
         <td></td></tr>
         <tr><td><b>Total Time (min): </b></td>
         <td></td>
         <td><b>Failed: </b></td>
         <td></td></tr>
         <tr><td></td>
         <td></td>
         <td><b>Cannot Test (CNT): </b></td>
         <td></td></tr>
      </table>
      <hr/>

      <xsl:apply-templates select="suite-note"/>
      <xsl:apply-templates select="test-case"/>

      <table WIDTH="100%" BORDER="0" CELLPADDING="0" CELLSPACING="0">
         <tr><td align="center">Copyright &amp;#169; 2002, Your Company.
         All rights reserved. Confidential.</td></tr>
      </table>
   </body>
   </html>

</xsl:template>

<xsl:template match="suite-note">
   <xsl:value-of select="." disable-output-escaping="yes"/> <hr/>
</xsl:template>

<xsl:template match="test-case">
   <xsl:if test="@inactive!='true' or not(@inactive)" >
      <h3><xsl:value-of select="/PNSQC02-test-suite/@suite-abbr"/>-
         <xsl:value-of select="@id"/>:&amp;#160;<xsl:value-of select="case-
         shortdesc"/></h3>
      <b>Purpose:</b>&amp;#160;<xsl:value-of select="case-purpose"/><p/>
      <xsl:apply-templates select="case-note"/>
      <xsl:apply-templates select="autotest-note"/>
      <b>Setup:</b>
      <ol>
         <xsl:apply-templates select="case-setup"/>
      </ol>
      <b>Expected Result:</b>
```

```
         <ol>
            <xsl:apply-templates select="case-result"/>
         </ol>
      <b>Test Case Dependencies (must be run before this test
         case):</b>
      <ol>
         <xsl:apply-templates select="dependency" />
      </ol><hr/>
   </xsl:if>
</xsl:template>

<xsl:template match="case-setup">
   <xsl:apply-templates select="setup-step"/>
</xsl:template>

<xsl:template match="case-result">
   <xsl:apply-templates select="result"/>
</xsl:template>

<xsl:template match="setup-step | result | dependency">
   <li><xsl:value-of select="."/></li>
</xsl:template>

<xsl:template match="case-note">
   <b>Test Case Notes:</b><p/>
   <ul><xsl:value-of select="." disable-output-escaping="yes"/>
   <hr/></ul><p/>
</xsl:template>

<xsl:template match="autotest-note">
   <b>Automated Testing Notes:</b><br/>
   <xsl:value-of select="." disable-output-escaping="yes"/>
   <p/>
</xsl:template>

</xsl:stylesheet>
```

Applying this XSL stylesheet to our XML produces the following HTML screen:



Other possible XML / XSL transformations

Now that we have seen the power of XML in conjunction with XSL, consider some other areas within test management where this combination could also be used. We could represent test case results using an XML document that looks something like this:

```
<?xml version="1.0" encoding="iso-8859-1"?>

<!-- Sample test case results -->
<sample-test-case-results>

   <case-result id="1001" status="Pass">
      <scheduled-date>20020610</scheduled-date>
      <execution-date>20020611</execution-date>
      <tester>MEE</tester>
   </case-result>

   <case-result id="1002" status="Pending"></case-result>
      <scheduled-date>20020610</scheduled-date>
      <execution-date>20020610</execution-date>
      <tester>MEE</tester>
   </case-result>

   <case-result id="1003" status="Fail">
      <scheduled-date>20020610</scheduled-date>
      <execution-date>20020610</execution-date>
      <tester>MEE</tester>
      <case-result-note>Error displayed: Unable to execute</case-
         result-note>
      <case-result-note>Program hang</case-result-note>
      <bug-id id="ABC123" />
   </case-result>

</sample-test-case-results>
```

This XML structure could be extended to include other elements, such as the full results of all test case executions. An XSL stylesheet could display a summary report with just the totals (passed, pending, failed). Yet another stylesheet could display the full detail. Yet another could display a list of bugs entered as a result of test case execution. Another could display only those cases that failed or were executed by a particular tester. The pending test cases could be grouped by tester to produce a test assignment report. Such versatility demonstrates the advantages of separating data from the presentation.

Additional XML elements could be created from scratch or by extension. Consider the test documents described in IEEE Std 829-1999 - Software Test Documentation. An XML / XSL approach could be used to describe the contents of the documentation and present it as HTML.

- Test Plan
- Test Design Specification
- Test Case Specification
- Test Procedure Specification

- Test Item Transmittal Report
- Test Log
- Test Incident Report
- Test Summary Report

Plans for additional work

Over the years, I have evaluated multiple test management systems -- commercial, shareware and open source. In all cases, the functionality / cost ratio were not advantageous to the business. With the rise in popularity of open source software, I would like to see the development of a full-featured test management system. I believe the combination of XML as an easily customizable and extendable data format in conjunction with XSL provide the basis for developing such a system. At my company, we are using this functionality today using the flat file storage approach. Developing a database driven system is the next logical step.

Conclusion and acknowledgements

We have completed our development and use of an XML document containing the definition of a test case, and expanded it to describe a suite of tests. Along the way, we have examined different ways to formally describe such an XML document. We have successfully excluded presentation logic from the test case data in the XML document. To display the test case data, XSL stylesheets were developed and modified without impacting the underlying data. I hope this spurs the imagination of the reader in how they could apply XML and its related technologies in his/her own day-to-day test management chores.

I would like to thank my co-workers at PAR3 Communications for their contributions to my understanding of the various XML technologies. Special recognition goes to Michael Jacobs who has put many of these ideas into practice.

References

MSXML
http://msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/msdn-files/027/001/766/msdncompositedoc.xml

XERCES http://xml.apache.org/xerces2-j/index.html
http://xml.apache.org/xerces-c/index.html
http://xml.apache.org/xerces-p/index.html

BOX00 Box, D., Skonnard, A., and Lam, J. Essential XML: Beyond Markup Boston, MA: Addison-Wesley, 2000

BLEZ01 Blezek, D., Kelliher, T., Lorensen, B., and Miller, J. "The Frost Extreme Testing Framework", Quality Week 2001 Conference Proceedings, May 29 - June 1, 2001 San Francisco: Software Research Institute, 2001

CRIS01 Crispin, L. "The Need for Speed: Acceptance Test Automation in an Extreme Programming Environment", Quality Week 2001 Conference Proceedings, May 29 - June 1, 2001 San Francisco: Software Research Institute, 2001

HUNT00 Hunter, David, et. al. Beginning XML Birmingham, UK: Wrox Press Ltd., 2000

IEEE98 IEEE IEEE Std 829-1998, IEEE Standard for Software Test Documentation New York, NY: IEEE, 1998

KAY00 Kay, Michael. XSLT: Programmer's Reference Birmingham, UK: Wrox Press Ltd., 2000

WALM02 Walmsley, P. Definitive XML Schema Upper Saddle River, NJ: Prentice Hall PT, 2002

The Soft Side of Peer Reviews

Karl E. Wiegers
Process Impact

Abstract:

Peer reviews are as much a social interaction as a technical practice. Asking your colleagues to point out errors in your work is a learned—not instinctive—behavior. This paper describes some of the cultural and interpersonal aspects of peer reviews and inspections that must be considered when trying to install a review program in an organization. The idea of "egoless programming" is described as it relates to reviews. Suggestions are provided for how a reviewer should present issues to the author in a nonjudgmental way. Some aspects of management attitudes and behaviors are discussed, including ten signs of management commitment to peer reviews. A case study illustrates the risks of managers using defect counts from inspections to evaluate individual authors. Several reasons why people do not perform reviews and ways to overcome them are explored, including factors related to lack of knowledge, cultural barriers, and simple resistance to change. Some of the benefits that people performing different project roles can enjoy from a successful peer review program are itemized. The paper also addresses some aspects of holding reviews that involve participants from different cultures.

Biography:

Karl E. Wiegers is Principal Consultant with Process Impact, a software process consulting and education company in Portland, Oregon. His interests include requirements engineering, project management, risk management, software process improvement, and peer reviews. Previously, he spent 18 years at Eastman Kodak Company, where he held positions as a photographic research scientist, software developer, software manager, and software process and quality improvement leader. Karl received a B.S. degree in chemistry from Boise State College, and M.S. and Ph.D. degrees in organic chemistry from the University of Illinois. He is a member of the IEEE, IEEE Computer Society, and ACM.

Karl's most recent book is *Peer Reviews in Software: A Practical Guide* (Addison-Wesley, 2002). He also wrote *Software Requirements* (Microsoft Press, 1999) and *Creating a Software Engineering Culture* (Dorset House, 1996), both of which won Productivity Awards from *Software Development* magazine, and more than 150 articles on software development, chemistry, and military history. Karl has served on the Editorial Board for *IEEE Software* magazine and as a contributing editor for *Software Development* magazine. He is a frequent speaker at software conferences and professional society meetings.

You can reach Karl at:

Process Impact
11491 SE 119th Drive
Clackamas, OR 97015-8778
503-698-9620
kwiegers@acm.org
www.processimpact.com

The Soft Side of Peer Reviews[1]

Karl E. Wiegers
Process Impact

Peer review—an activity in which people other than the author of a software deliverable examine it for defects and improvement opportunities—is one of the most powerful software quality tools available. Peer review methods include inspections, walkthroughs, peer deskchecks, and other similar activities. After experiencing the benefits of peer reviews for nearly fifteen years, I would never work in a team that did not perform them.

However, many organizations struggle to implement an effective review program. Many of the barriers to successful peer reviews are social and cultural in nature, not technical. This article explores some of the social and psychological aspects of having people review each other's work, ways to overcome resistance to reviews, and issues regarding management involvement. The suggestions provided here might help your peer review program succeed where others have failed.

Scratch Each Other's Back

Asking your colleagues to point out errors in your work is a learned—not instinctive—behavior. We all take pride in the work we do and the products we create. We don't like to admit that we make mistakes, we don't realize how many we make, and we don't like to ask other people to find them. Holding successful peer reviews requires us to overcome this natural resistance to outside critique of our work.

Busy practitioners are sometimes reluctant to spend time examining a colleague's work. They might be leery of a co-worker who asks for a review of his code. Questions arise: Does he lack confidence? Does he want you to do his thinking for him? "Anyone who needs their code reviewed shouldn't be getting paid as a software developer," scoff some potential review resisters. These resisters don't appreciate the value that multiple pairs of eyes can add.

In a healthy software engineering culture, team members engage their peers to improve the quality of their work and increase their productivity. They understand that time spent looking at a colleague's deliverable isn't time wasted, especially when other team members willingly reciprocate. The best software engineers I have known actively sought out reviewers, having learned early on how much they can help. Indeed, the input from many reviewers over their careers was part of what made these developers the best at what they do.

Gerald Weinberg introduced the concept of "egoless programming" in *The Psychology of Computer Programming*, originally published in 1971 and re-released in 1998. Weinberg recognized that people tie much of their perceived self-worth to their work. You can interpret a fault that someone finds in an item you created as a shortcoming in yourself as a software developer, and perhaps even as a human being. To guard your ego, you don't want to know

---

[1] This paper was originally published as "Karl Wiegers on Humanizing Peer Reviews" in *STQE*, March/April 2002. It is reprinted (with modifications) with permission from Software Quality Engineering. It is adapted from *Peer Reviews in Software: A Practical Guide* by Karl E. Wiegers (Addison-Wesley, 2002).

about all the errors you've made, and you might rationalize possible bugs as product features. Such staunch ego-protection presents a barrier to effective peer review, leads to an attitude of private ownership toward individual contributions within a team project, and can result in a poor quality product. Egoless programming enables an author to step back and let others point out places where improvement is needed.

Note that the term is "egoless programming," not "egoless programmer." Developers need a robust enough ego to trust and defend their work, but not so much ego that they reject suggestions for better solutions. Similarly, the egoless reviewer should have compassion and sensitivity for his colleagues, if only because their roles will be reversed one day.

A broad peer review program can only succeed in a work culture that values quality in its many dimensions, including freedom from defects, satisfaction of customer needs and business objectives, timeliness of delivery, and the possession of desirable product functionality and attributes. Recognizing that team success depends on helping each other do the best job possible, members of a healthy culture prefer to have peers—not customers—find defects. They understand that reviews are not meant to identify scapegoats for quality problems. Having a co-worker locate a defect is regarded as a "good catch," not a personal failing. In fact, reviews can motivate authors to practice superior craftsmanship because they know their colleagues will closely examine their work.

Tips for the Reviewer

The dynamics between the reviewers and the work product's author are a critical aspect of peer reviews. The author must trust and respect the reviewers to be receptive to their comments. Conversely, the reviewers must respect the author's talent and hard work. The ways in which reviewers speak to authors indicate whether their culture favors respectful collaboration or competitive antagonism.

Reviewers should focus on what they observed about the product, thoughtfully selecting the words they use to raise an issue. Saying, "I didn't see where these variables were initialized" is likely to elicit a constructive response; the more accusatory "You didn't initialize these variables" might get the author's hackles up. You might phrase your comments in the form of a question: "Are we sure that another component doesn't already provide that service?" Or, identify a point of confusion: "I didn't see where this memory block is deallocated." Direct your comments to the work product, not to the author. For example, say "This specification is missing Section 3.5 from the template" instead of "You left out section 3.5." Reviewers and authors must work together outside the reviews, so each needs to maintain a level of professionalism and mutual respect to avoid strained relationships.

While a draft of my book *Peer Reviews in Software* was undergoing peer review (naturally!), one reviewer expressed a concern by writing, "How in the world have you managed to" miss some point he thought was important. Then he added, "Good grief, Karl." I respect this reviewer's experience and value his insights, but perhaps he could have phrased that bit of feedback more tactfully. Expressing incredulity at the author's lack of understanding is not likely to make the author receptive to a reviewer's suggestion. While a reviewer might let an inappropriate comment slip out accidentally during a discussion, it's inexcusable in written feedback.

You do not want your reviews to create authors who look forward to retaliating against their tormentors. Moreover, an author who walks out of a review meeting feeling personally attacked or professionally insulted will not voluntarily submit his work for review again. Bugs

are the bad guys in a review, not the author or the reviewers. The leaders of the review initiative should strive to create a culture of constructive criticism, in which team members seek to learn from their peers and do a better job the next time. Managers should encourage and reward those who initially participate in reviews and make useful contributions, regardless of the review outcomes.

Problem: Why Don't People Do Reviews?

Lack of knowledge about reviews, cultural issues, and simple resistance to change (which often masquerades as excuses) contribute to the underuse of reviews. Many people don't understand what peer reviews are, why they are valuable, the differences between formal and informal reviews, or when and how to perform them. Some developers and project managers don't think their project is large enough or critical enough to need reviews, yet any piece of work can benefit from an outside perspective. There's also a widely held perception that reviews take too much time and slow the project down. In reality, any technique that facilitates early quality improvements should shorten schedules, reduce maintenance costs, and enhance customer satisfaction. The misperception that testing is always superior to manual inspection also leads some practitioners to shun reviews.

Several cultural issues can create unpleasant review experiences. The fear of management retribution if defects are discovered can make an author reluctant to let others examine his work. Another cultural barrier is the reviewer's attitude that the author is the most qualified person to work on his part of the system—who am I to look for errors in his work? This is a common reaction from new developers who are invited to review an experienced colleague's deliverables. There is also the paradox that many developers are reluctant to try a new method unless the technique has been proven to work, yet the developers don't believe the new approach works until they've successfully done it themselves. They don't want to take anyone else's word for it.

Cultural biases that run deeper than workplace attitudes can play against review participation. For instance, our educational system grades people primarily on individual performance, so collaborating is sometimes viewed as cheating. There's an implication that if you need help, then you must not be very smart. Therefore, it's not surprising that developers often resist asking for help through reviews. We have to overcome the ingrained culture of individual achievement and embrace the value of collaboration. In addition, authors who submit their work for scrutiny might feel their privacy is being invaded, being forced to air the internals of their work for all to see. This is threatening to some people, which is why the culture must emphasize the value of reviews as a collaborative, nonjudgmental tool for improved quality and productivity.

Then there are the excuses. People who don't want to do reviews will expend considerable energy explaining why reviews don't fit their culture, needs, or time constraints. Resistance often appears as NAH (Not Applicable Here) Syndrome: we don't need no stinkin' reviews. There is the attitude that some people's work does not need reviewing. Some team members can't be bothered to look at a colleague's work. "I'm too busy fixing my own bugs to waste time finding someone else's. Aren't we all supposed to be doing our own work correctly?" Other developers imagine that their software prowess has moved them past the point of peer reviews. "Inspections have been around for 25 years; they're obsolete. Our high-tech group only uses leading-edge technologies." These excuses could reflect the team's cultural attitudes toward quality, indicate resistance to change, or reveal a fear of peer reviews. You must address these existing barriers to establish a successful review program.

Strategy: Overcoming Resistance to Reviews

Lack of knowledge is easy to correct if people are willing to learn. A one-day class that includes a practice review session gives team members a common understanding about the process. Managers who also attend the class send a powerful signal about their commitment to reviews. Management attendance says to the team, "This is important enough for me to spend time on it, so it should be important to you, too" and "I want to understand reviews so I can help make this effort succeed."

Dealing with cultural issues requires that you understand your team's culture and how best to steer the team members toward improved software engineering practices. What values do they hold in common? Is there a shared understanding of, and commitment to, quality? What previous change initiatives have succeeded and why? Which have struggled and why? Who are the opinion leaders in the group and what are their attitudes toward reviews?

Larry Constantine described four cultural paradigms found in software organizations: *closed*, *open*, *synchronous*, and *random* (see "Work Organization: Paradigms for Project Management and Organization," *Comm. ACM*, October 1993). Understanding which paradigm your team's culture most closely resembles can give you some clues about how to introduce a peer review program.

A **closed** culture has a traditional hierarchy of authority. You can introduce peer reviews in a closed culture through a management-driven process improvement program, perhaps based on one of the Software Engineering Institute's capability maturity models.

Innovation, collaboration, and consensus decision-making characterize an **open** culture. Members of an open culture want to debate the merits of peer reviews and participate in deciding when and how to implement them. Respected leaders who have had positive results with reviews in the past can influence the group's willingness to adopt them. Such cultures might favor review meetings in which discussion of proposed solutions is common, although the inspection process emphasizes finding—not fixing—defects during meetings.

Members of a **synchronous** group are well aligned and comfortable with the status quo. Because they recognize the value of coordinating their efforts, they are probably already performing at least informal reviews. A comfort level with informal reviews makes it easier to implement a more formal inspection program.

Entrepreneurial, fast-growing, and leading-edge technology companies often develop a **random** culture populated by autonomous individuals. In random organizations, individuals who have performed peer reviews in the past might continue to hold them. The other team members probably don't have the patience for reviews, but they might change their minds if quality problems from the resulting chaos burn them badly enough.

Whichever category best describes your work culture, people will want to know what benefits any new process will provide to them personally. Table 1 lists some of the benefits that various project team members might reap from reviewing major life cycle deliverables. Not only will the team benefit, but the customers come out ahead as well. They'll receive a timely product that is more robust and reliable, better meets their needs, and increases their own productivity.

Table 1. Benefits from peer reviews for different project roles.

Project Role	***Possible Benefits from Peer Reviews***
Project Manager	• Shortened product development cycle time • Increased chance of shipping the product on schedule • Reduced field service and customer support costs • Reduced lifetime maintenance costs, freeing resources for new development projects • Improved teamwork, collaboration, and development effectiveness • Reduced impact from staff turnover through cross-training of team members • Better and earlier insight into project risks and quality issues
Developer	• Less time spent performing rework • Increased programming productivity • Better techniques learned from other developers • Reduced unit testing and debugging time • Less debugging during integration and system testing • Exchanging of information about components and overall system with other team members
Maintainer	• Fewer production support demands, leading to a reduced maintenance backlog • More robust designs that tolerate change • Conformance of work products to team standards • Better structured and documented work products that are easy to understand and modify • Better understanding of the product from having participated in design and code reviews during development
Quality Assurance Manager	• Ability to judge testability of product features under development • Shortened system-testing cycles and less retesting • Ability to use review data when making release decisions • Education of quality engineers about the product • Ability to anticipate quality assurance effort needed
Test Engineer	• Ability to focus on finding subtle defects because product is of higher initial quality • Fewer defects that block continued testing • Improved test design and test cases that smooth out the testing process
Requirements Analyst	• Earlier correction of missing or erroneous requirements • Fewer infeasible and untestable requirements because of developer and test engineer input during reviews

Although there are many individual rewards from conducting peer reviews, also address the larger question of "What's in it for *us*?" Sometimes when you're asked to change the way you work, your immediate personal reward is small, although the team as a whole might benefit in a big way. I might not get three hours of benefit from spending three hours reviewing someone else's code. However, the other developer might avoid ten hours of debugging effort later in the project and we might ship the product sooner than we would have otherwise.

Influential resisters who come to appreciate the value of reviews might persuade other team members to try them, too. A quality manager once encountered a developer named Judy who was opposed to "time-sapping" inspections. After participating under protest, Judy quickly saw the power of the technique and became the group's leading convert. Since Judy had some influence with her peers, she helped turn developer resistance toward inspections into acceptance. Judy's project team ultimately asked the quality manager to help them hold even *more* inspections. Engaging developers in an effective inspection program helped motivate them to try some other software quality practices, too.

Management Involvement

The attitudes and behaviors that managers exhibit affect how well reviews will work in an organization. While managers want to deliver quality products, they also feel pressure to release products quickly. Managers need to learn about peer reviews and their benefits so they can build the reviews into project plans, allocate resources, and communicate their commitment to reviews to the team.

Watch out for culture killers, such as singling out certain developers for the humiliating "punishment" of having their work reviewed. For example, my colleague Phil once told me that his manager demanded code reviews whenever a project was in trouble, with the unstated objective of finding a scapegoat. Unfortunately, Phil was the first to fall victim to such a review. The review team found only minor issues, but the manager then went around complaining that the project was late because Phil's code was full of bugs! Understandably, Phil soon quit this job.

On a similar note, I recently heard from a quality manager at a company that had operated a successful inspection program for two years. Then the development manager announced that finding more than five bugs during an inspection would count against the author at performance evaluation time. Naturally, this made the development team members very nervous. It conveyed the erroneous impression that the point of the inspection was to punish people for making mistakes. Such evaluation criminalizes the mistakes that we all make and pits team members against each other. This misapplication of inspection data could lead to numerous dysfunctional outcomes:

1. Developers might not submit their work for inspection to avoid being punished for their results. They might refuse to inspect a peer's work to avoid contributing to someone else's punishment.

2. Inspectors might not point out defects during the inspection, instead reporting them to the author offline so they aren't tallied against the author. This undermines the open focus on quality that should characterize peer review.

3. Inspection teams might endlessly debate whether something really is a defect, because defects count against the author while issues or simple questions do not.

4. The team's inspection culture will develop an unstated goal of finding few defects, rather than revealing as many as possible.

5. Authors might inspect very small pieces of work so they don't find more than five bugs in any one inspection. This leads to inefficient and time-wasting inspections.

When presented with these risks, reasonable managers will rethink their intention to misuse the review data. If an unreasonable manager insists on using the data in this way, the team won't be successful with reviews. Managers may legitimately expect developers to submit their work for review and to review deliverables that others create. However, managers must not evaluate individuals based on the number of defects found during those reviews.

Without visible and sustained commitment to peer reviews from management, only those practitioners who believe reviews are important will perform them. Management commitment goes far beyond providing verbal support or giving team members permission to hold reviews. Figure 1 lists several clear signs of management commitment to peer reviews. If too many of these indicators are missing in your organization, your review program will likely struggle.

Figure 1. Ten Signs of Management Commitment to Peer Reviews.

1. Providing the resources and time to develop, implement, and sustain an effective review process.
2. Setting policies, expectations, and goals about review practice.
3. Ensuring that project schedules include the time needed to perform reviews.
4. Making training available to the participants and attending the training themselves.
5. Never using review results to evaluate the performance of individuals.
6. Holding people accountable for participating in reviews and for contributing constructively to them.
7. Publicly rewarding the early adopters of reviews to reinforce desired behaviors.
8. Running interference with other managers and customers who challenge the need for reviews.
9. Respecting the judgment of an inspection team's appraisal of a document's quality.
10. Asking for status reports on how the program is working, what it costs, and the team's benefits from reviews.

Review Your Way to Success

If you're serious about the quality of your work, you'll accept that you make mistakes, seek the counsel of your compatriots in finding them, and willingly review your colleagues' work products. You will set aside your ego so you can benefit from the experience and perspective of your technical associates. When you have internalized the benefits of peer reviews, you won't feel comfortable unless someone else carefully examines any significant deliverable you create.

On the other hand, even if you care about quality, you'll be reluctant to participate in reviews if your environment is not supportive of such quality practices. Perhaps you can lead by example, inviting other team members to look at your deliverables. Maybe you can contribute to leading a nascent review program and help steer managers and practitioners to the effective and routine application of reviews. But if holding reviews would be too unpleasant in a hostile environment, you might want to find a culture that better supports your personal quality philosophy.

[SIDEBAR] Reviews without Borders

Increasingly, software projects involve teams that collaborate across multiple corporations, time zones, continents, organizational and national cultures, and native languages. The review issues include both communication logistics and cultural factors; the latter pose the greater challenge. Different cultures have different attitudes toward critiquing work performed by another team member or by a supervisor. People from certain nations or geographical regions are comfortable with a more aggressive interaction style than are others, who avoid confrontation. A review participant from the more reserved community might feel that someone from the assertive domain is dominating the review, while an assertive participant wonders why his quiet counterpart isn't contributing to the discussion. If you face such a multicultural challenge, learn about ways to get members of different cultures to collaborate and adjust your expectations about peer reviews.

One company encountered cultural barriers on a project that involved collaborating development teams in Singapore and the Netherlands (see Erik P.W.M. Van Veenendaal's "Practical Quality Assurance for Embedded Software," *Software Quality Professional*, June 1999). Developers in Singapore were not accustomed to having others comment on their work. They could take well-meaning comments personally, especially if they were presented semi-publicly during an inspection meeting. To deal with this, the company matched a co-author from the Netherlands with each work product from Singapore and held all inspections in the Netherlands. This approach succeeded, but it sidestepped the underlying cultural issue and essentially permitted the Singapore developers to avoid engagement in the inspections.

When you plan reviews for cross-cultural development projects, be aware of these interaction differences and consider which review approaches will work best. Discuss these sensitive issues with review participants to make everyone aware of how their differences will affect the review process. If the participants are geographically separated, hold an initial face-to-face training session to surface the cultural and communication factors so the team can determine how best to function when the team members are separated.

The Soft Side of Peer Reviews

Karl E. Wiegers
Process Impact
www.processimpact.com

Agenda

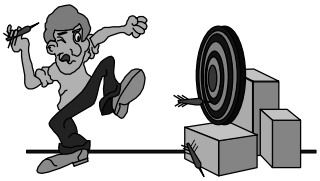
- Define “peer review”
- Barriers to peer reviews
- Who benefits from peer reviews
- Egoless programming
- Peer reviews and managers
- Peer review do’s and don’ts
- Multicultural reviews

What is a Software Peer Review?

An examination of a software work product by people other than its author in order to identify defects (departures from specifications or from standards) and improvement opportunities.

Objectives of Peer Reviews

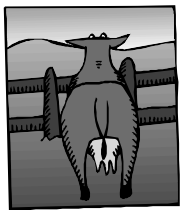
- To reveal errors in function, logic, or implementation
- To ensure that a work product satisfies its specification
- To check for conformance to standards
- To give managers insight into product quality
- To communicate technical information in a project
- To enable someone besides the author to support or modify a work product
- To identify process improvement opportunities



Why Don't People Do Reviews?

Cultural Barriers to Peer Reviews

- Fear of public ridicule or criticism
- Fear that management will hold defects against the author
- Previous negative review experiences
- Concern that reviews will slow the project down
- Preference for fixing, not preventing, bugs
- Managers who overrule review outcomes
- National or team cultures that avoid criticism

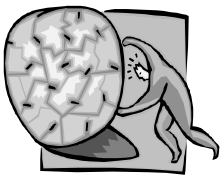


Signs of Resistance

- "My work doesn't need reviewing."
- "Anyone who needs reviews shouldn't be getting paid as a programmer."
- "I can find bugs faster by testing."
- "No one around here is qualified to review my work."
- "Reviews are an old technique. Our high-tech team uses only the latest methods."

Overcoming Resistance to Reviews

- **Education**
  - ✓ train reviewers and review moderators
  - ✓ educate team about return on investment, cost of quality
  - ✓ show what's in it for various project participants
- **Cultural Factors**
  - ✓ understand the underlying causes of resistance
  - ✓ understand how to instill change in your culture
  - ✓ have respected opinion leaders show the way
- **Management Leadership**
  - ✓ set expectations
  - ✓ attend the review training
  - ✓ publicly reward early adopters



Who Benefits from Peer Reviews - 1

Developer

- less time spent performing rework
- increased programming productivity
- confidence that the requirements are right
- better techniques learned from other developers
- reduced unit testing and debugging time
- less debugging during integration & system testing
- exchanging information with other team members

Maintainer

- fewer production support demands
- more robust designs that tolerate change
- conformance of work products to team standards
- more maintainable work products
- better understanding of the product from having participated in design and code reviews

Who Benefits from Peer Reviews - 2

Test Engineer

- can focus testing on finding subtle defects
- fewer defects that block continued testing
- improved test design and test cases

Requirements Analyst

- earlier correction of missing or erroneous requirements
- fewer infeasible and untestable requirements

Project Manager

- increased chance of shipping on schedule
- earlier awareness of risks and quality issues
- reduced risk from staff turnover through cross training

Who Benefits from Peer Reviews - 3

Development Manager

- shortened product development cycle time
- reduced field service and customer support costs
- reduced lifetime maintenance costs
- improved teamwork and collaboration
- earlier insight into risks and quality issues

QA Manager

- can judge the testability of product features
- shortened system-testing cycles, less retesting
- can use review data when making release decisions
- quality engineers who understand the product
- ability to anticipate quality assurance effort needed

Egoless Programming

- Our self-image is tied to the work we do.
  - ✓ defects in our work imply “defects” in ourselves, so...
  - ✓ we resist hearing about defects in our work, and…
  - ✓ we rationalize bugs into features
- Ego-protection creates some problems:
  - ✓ poses barrier to peer review
  - ✓ leads to attitude of private ownership of team products
  - ✓ can lead to quality problems
- Practitioners of egoless programming:
  - ✓ trust and take pride in their work, but value outside input
  - ✓ respect and trust their colleagues

Ways to Present Issues

- In the form of a question: “Does another component already provide that service?”
- As a point of confusion: “I didn’t see where this memory block was deallocated.”
- As an observation on the work product: “This specification is missing Section 3.5 from the template.”
- But not to the author: “You left out section 3.5.”
- Avoid sounding accusatory: “Did you ask the account managers if they really needed this feature?”
- Don’t harp: “Once again, you need more code comments.”

Risks of Criminalizing Defects

- Developers might not submit their work for review.
- Developers might refuse to review someone else’s work.
- Reviewers might not point out defects they see.
- Authors might hold unofficial “pre-reviews”.
- Review teams might debate whether or not something is a defect.
- The culture will have an implicit goal of finding few defects.
- Authors might review small bits of work to avoid finding too many bugs.

Peer Reviews and Managers

- The conventional wisdom is “no managers.”
  - ✓ can inhibit discussion
  - ✓ author’s fear of retribution
- I think it depends.
  - ✓ first-line manager can participate at author’s invitation
  - ✓ managers can often contribute technically
  - ✓ key issue is mutual respect and trust
- Managers need their work reviewed, also.
  - ✓ project charters, plans, schedules
  - ✓ soliciting review is a culture builder
- Managers must demonstrate commitment to reviews.

10 Signs of Management Commitment

1. Providing resources to establish and sustain the review program.
2. Setting policies, expectations, and goals.
3. Ensuring that project schedules include time for reviews.
4. Making training available.
5. Never using review results to evaluate individuals.
6. Holding people accountable for participating in reviews.
7. Staying the course even in the face of schedule pressure.
8. Running interference with other managers and customers.
9. Respecting the review team’s document appraisal.
10. Asking for status reports on the program.

Installing Reviews into 4 Types of Cultures

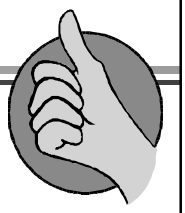
- ***Closed*** **(**traditional hierarchy of authority)
  - ✓ go through a management-driven process improvement program
- ***Open*** **(**innovative and collaborative; consensus decision-making)
  - ✓ involve team in discussion about how and when to do reviews
  - ✓ opinion leaders can show the way
- ***Synchronous*** (well-aligned; comfortable with status quo)
  - ✓ may already be doing informal reviews; build from that
- ***Random*** (entrepeneurial, independent workers)
  - ✓ team members who have done reviews before might continue
  - ✓ the others probably don't have the patience for reviews

[Constantine, Larry L. "Work Organization: Paradigms for Project Management and Organization," *Comm. ACM*, Oct. 1993]

Multicultural Peer Reviews

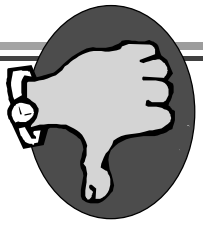
- Frank criticism isn't acceptable in some cultures.
  - ✓ publicly airing defects in a meeting is uncomfortable
  - ✓ offering suggestions to managers isn't done
- Distance separation requires distributed reviews.
  - ✓ need a capable moderator
  - ✓ rotate meeting times to spread the inconvenience
  - ✓ hold face-to-face initial meeting to establish rapport
- If necessary, collect input anonymously
  - ✓ use peer deskchecks or passarounds

Peer Review Do's



- Cultivate the attitude: "We prefer to have a peer, rather than a customer, find a defect."
- Regard finding a defect as a "good catch".
- Check your egos at the door.
- Critique the product, not the producer.
- Emphasize finding, not fixing, defects.
- Avoid being distracted by style issues.
- Make everyone's work subject to review.

Peer Review Don'ts



- Use data collected from reviews to punish or reward individuals.
- Attribute defect data to individual authors.
- Track who finds the most defects.
- Use reviews to find scapegoats for project problems.
- Use untrained or reluctant review moderators.
- Let reviewers make personal, offensive, accusatory, or provocative comments.
- Include participants who are not there to find defects.

The Soft Side of Peer Reviews

You cannot *review* quality in.

You must *build* quality in.

For Further Information

Ebenau, Robert G. and Susan H. Strauss. *Software Inspection Process*. New York: McGraw-Hill, 1994.

Freedman, Daniel P. and Gerald M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews, 3d Ed*. New York: Dorset House, 1990.

Gilb, Tom and Dorothy Graham. *Software Inspection*. Reading, Mass.: Addison-Wesley, 1993.

Radice, Ronald A. *High Quality Low Cost Software Inspections*. Andover, Mass.: Paradoxicon Publishing, 2002.

Weinberg, Gerald M. *The Psychology of Computer Programming, Silver Anniversary Edition*. New York: Dorset House Publishing, 1998.

Wheeler, David A., Bill Brykczynski, and Reginald N. Meeson, Jr. *Software Inspection: An Industry Best Practice*. Los Alamitos, California: IEEE Computer Society Press, 1996.

Wiegers, Karl. "The Seven Deadly Sins of Software Reviews," *Software Development*, vol. 6, no. 3 (March 1998), pp. 44-47.

Wiegers, Karl E. *Peer Reviews in Software: A Practical Guide*. Reading, Mass.: Addison-Wesley, 2002.

Software Development Process in a Non-Software Centric Company

By
Sandy Raddue

Presenter Biography

Sandy Raddue is presently a Staff Software QA Engineering Team Leader at Cypress Semiconductor. She started in the wonderful world of Software Engineering in 1978, as her company's Punched Paper Tape expert. Since then, she has worked on a variety of projects and products, from video games to satellite telecommunications systems. Sandy has done development work as well, but has made Software Quality her mission. She holds a Certified Computer Programmer certificate with Scientific Specialization from the Institute for Certification of Computer Professionals (ICCP), and also holds a Certified Software Quality Engineer certificate from American Society for Quality (ASQ). She is a member of the ASQ and IEEE, and a former member of the IEEE Software Engineering Standards Committees.

Kudos

Many, many thanks to my paper reviewer, Manny Gatlin. This paper is of far better quality due to Manny's involvement.

The author may be reached at the following:
C/o Cypress Semiconductor
9125 SW Gemini Dr. Suite 200
Beaverton, OR 97008
Email: sur@cypress.com

ABSTRACT:

For those who work in the “pure” software industry, where deliverables are strictly software in its many forms, such as applications, games, and the like, the understanding of software engineering and software technology, and the role of software quality and process, are fairly well understood, though often debated. Support and adherence to known methods may be varied.

How does one approach software development, quality, and process issues in a company where the actual shippable product contains no software, or where software just has a “support” role e.g. software is used as a support tool (programming software) or as a component in a larger system (drivers), or the like.

I refer here to software created as device drivers, development tools to target specific hardware, reference designs, or other product lines that are unusable without support software. I am not, however, referring directly to embedded systems where software is an integral part of a manufactured system.

For my definition of the “Non-Software centric” company, I refer to companies where software itself contributes little or no revenue directly to the bottom line. For upper management, the questions often arise regarding the importance, expense, and complexity of the software.

I will attempt to share my experience gained in years of experience as the software development process expert, using this business model at a semiconductor company, and dispel the following myths. I will start by explaining the business model used in the Programmable Products area of Cypress Semiconductor.

PAPER:

There are several myths that are pervasive in a non-software company.
The first myth – Software is easy – it’s “just” a CD, but silicon is expensive to make, hard to manufacture. This myth comes from years of hearing “it’s a simple fix in software”. This is the most dangerous myth, because this is a cultural issue. In a true silicon company, software can gate a silicon release, but there is NO high level understanding of software by the upper level management.

I often use the following analogy – every decision point in software is like a silicon gate – count the complexity in a piece of software compared to a chip – the complexity can be astounding. This is effective for those managers who want to understand the differences, but not effective for the “pure silicon” folks. Out of necessity, their world is narrowly focused on chips, masks, and manufacturing technology. I’m sure that at some level they ask themselves “What kind of complexity is there when all you have to do is duplicate a CD?” For these people, to quote a colleague [1], “The properties of semiconductors are hard, tangible constraints within which to work, while in software there are no analogous constraints – in other words, physics does not bound the

solution space." As software becomes more complex, the defect rate increases, thus contributing to higher costs both in development and maintenance. Complexity is a development issue; the hard part of software is in the requirements, design, and implementation of code, not in the "manufacturing" of multiple copies. Media defects are the only real manufacturing issues that are of concern when shipping software. The focus of most manufacturing companies is on quality control; i.e., making sure that the products going out the door meet some minimum threshold of defects per million (DPM) based on sampling random products from manufacturing lots. There are no sampling sizes based on lots in software; the focus is on quality assurance, i.e., making sure that the development process is followed correctly and meets minimum criteria at each specified development gate (e.g., requirements specification, architecture, detailed, design, coding, testing, etc.).

Most people are familiar with defects per thousand lines of source code (KSLOC) as a reasonable measure of software quality, but the distinction between software measures of defects and hardware measures of defects is rather distinct. In hardware, the defect measurement is an average defect density across an entire production run. In other words, for a product that has a measured DPM of 2, on average only 2 parts in every million will be defective. For software, if the defects per KSLOC are measured to be 4, for example, each and every copy of the software will be expected to have at least 4 defects per KSLOC. It's a distinction between hardware and software quality that often confuses people.

This is where the highest levels of upper management (VP level or equivalent) come in to play. If they understand that software is an integral part of the product, the directive for software quality as a requirement can come from the top down. It then will be treated as something that at least shows up on the radar. Understanding may still be lacking by the worker bees, but management support is a place to start.

Second myth – The customer doesn't really care about the quality of the software – it's something we have to have checked off so we can ship silicon! In reality, the software is a tool the customer uses to complete their design. The software quality is an indicator to the customer about how much we care about the silicon – if the software is easy to use, and a comfortable, well performing tool, they expect the silicon to be as good. If the software is cumbersome, and not well thought out, they expect the silicon to follow suit, and this costs us design wins in the marketplace. It is also far cheaper for an engineer at a customer company to perform a quick review of the software than it is for that engineer to build a prototype board, or some other method of evaluating the silicon.

This attitude is pervasive even with drivers and other supporting software that ships as part of the hardware product.

The personal computer world as a whole has become accustomed to software with "bugs". The term "bug" should be replaced by defect in general usage. If this were to occur, the public might take the problems more seriously. Cars are not supposed to ship

with defects. Cars do ship with warranties. Often these warranties are used to make the car meet the customer's expectations (requirements). If the car does not meet the customer's expectations, there are laws that protect the consumer. Although the topic of software warranty is outside the scope of this paper, I'd like to mention the following: when one installation of software contains a defect, ALL installations of the same software contain the same defect. This is different than cars, or any manufactured product. Possibly the use of the term "bug" has trivialized the issues to the consumer, who expects blue screens and computer crashes and data loss. If the software is too "buggy", it gets put on the shelf.

Cypress's customers are engineers. Engineers don't have time in product development schedules to spend chasing down defects in tools and determining workarounds to these defects. They will just move on to another tool without as much overhead and down time. Engineers make design decisions based on the quality of the whole product – not just the advertised features of the silicon. If the silicon performs the tasks it is supposed to perform, that's great. BUT – if the customer can't either access the features via the tools, or can't access the silicon due to a defect in programming software, one of two things is going to happen. 1) If the customer isn't too far down the design path, and can change the chips they've selected, they might decide to do that to keep their schedules. 2) The customer can decide to work with the silicon vendor. The silicon company will have to be extremely responsive to customer needs, providing tested service packs and patches almost upon request.

The costliest error (to the "non-software centric" company) regarding software errors and customer support is probably illustrated by the "Pentium flaw" issue that was discovered in 1994. The reader is aware that Intel had a flaw in the floating-point arithmetic of the Pentium/Pentium II chip. What the reader may not be aware of is that this flaw can be traced back to a software error – an error in a script that generated a lookup table within the processor. [2] The estimated cost to Intel alone was $475 MILLION dollars in replacement cost for defective processors. The cost to the company's reputation was far greater. In addition, it was estimated that for each of the millions of processors that were replaced, it cost the customer approximately $289 in the cost of reconfiguration, testing, and customer downtime. Smith-Barney estimated that it cost them $200,000 to replace processors in 600 computers in New York. In addition to the crushing financial burden caused by this software defect, Intel lost credibility with the marketplace, because they did not react quickly, they downplayed the severity and impact of the defect, and insisted in shipping flawed inventory for many months following the discovery of the defect. They only agreed to act "reasonably" after lawsuits and class action suits had been filed. Surely this software was as important as the silicon. Intel did learn from this blunder. The next "Pentium flaw" to be discovered was quickly disclosed to the marketplace, and motherboards were recalled and replaced rapidly, with minimum fuss.

Third myth – we don't sell the software – so why does everybody keep saying it has value? We don't make anything from it! The software does not usually add resource

dollars to the bottom line. This is a true statement. But, since the software is NECESSARY to make the silicon {functional, configurable, programmable, etc}, there can be no chips sold without the software accompanying them to make them work. But the software does have value – at a minimum, the cost to develop it.
When software is necessary to support silicon, the value of the software surpasses the value of the CD, and often surpasses even the development cost. Today's complex chips are far more expensive to develop, and without the software, that cost is also wasted. Without the software, or with software that contains a lot of defects, the company would have no revenue from silicon, resulting in jobs lost, money wasted, customers lost. There is also the high cost of software maintenance and multiple releases. In addition, often the software is used internally to characterize the silicon prior to producing the silicon, and used after the first run of silicon to verify the chip. Therefore, the value is also internal to the company as a development tool!

A Cypress Vice President has stated the following regarding the impact of the quality of the software released by our company:
"If we want a billion dollars of sales a quarter, and this product line accounts for 25% of that revenue, the quality of the software directly affects $250 million dollars, for better or worse. If the software isn't good, and if it doesn't do what it's supposed to, customers will take their $250 million elsewhere. "

Fourth myth – the competitor's software is no better than ours. What's the big deal? This myth stems from the idea that the innovation has to be in the silicon – the software doesn't really matter to the customer. Of course, this myth comes from the silicon developer, who feels that they are the sole owners of the "neat stuff".

The software is the first impression the potential customer has about our company and out products. This is the place we can really excel – if we can be better than our competition, we can grab market share by the buckets. If we don't differentiate ourselves here, our silicon will appear to be the same as well.

The cost to repair software defects is often underestimated as well. The cost of a new mask for a silicon layer can be upwards of $100K. The "engineering time' to fix a software defect may just be a couple of hours. The real costs, such as regression testing, cost of updates, loss of existing customers, loss of future sales, and loss of credibility in the marketplace – are ignored. In addition, the assumption is often made that a change to software will always correctly fix the problem. As we all know, that is not always the case.

In addition, to maintain an intellectual property edge, software must be developed in-house. If it is not, you must accept the risk of giving away the company's hard-earned IP to have this information integrated into other tools. The competitive advantage is one of the main reasons to develop design tools in-house.

What we've done to contradict these myths:

History-
After rushing software development during a “trivial” silicon implementation that required extremely complex software support, of course the software didn’t meet the silicon schedule. When the original schedule was prepared, the silicon design team was completely oblivious to the complexity required from the software. The company had an overall “interlocking milestone” specification, but all it said at the time was that the software release would gate the silicon release. There was no information provided about how the software milestones interacted with the silicon ones. There was also no mechanism for consultation or collaboration regarding schedules, features, or other requirements of the project. In addition, the software team suffered the punitive effect of a late delivery that didn’t support silicon!

Solution-
Now, when a new product is proposed, a whole product plan is prepared. This plan contains silicon, software, hardware (boards, cables, etc) and all other collateral required for a product to ship. The schedule is negotiated, and an overall project schedule is prepared considering all these items, with resources and responsibilities assigned accordingly. Interlocking milestones are determined, with milestone gates clearly identified.

This has resulted in a lot of discontent among the silicon designers, but it has helped the systems and software groups deliver better products than ever before, and meet the needs of both the silicon designers and the marketplace.

Conclusion – The quality of the software does matter – as much as if we were selling only a software product – maybe more. Appropriate attention must be paid to ALL elements of a product for a successful market launch.

[1] Gatlin, Manny, Intel Corp.
[2] http://www.intel.com/support/processors/pentium/fdiv/wp/4.htm
Pentium® Processors Statistical Analysis of Floating Point Flaw
Intel White Paper

Climbing CMM Levels in a Small Organization
By Kimberly Ritchie, Wacom Technology Corporation

Abstract

Incorporating process into our daily engineering activities has been an ongoing task for the past two years. We chose to use the Capability Maturity Model as our guide. Our first step was to combine the Key Process Areas (KPAs) into a project document template that we call the Engineering Project Description (EPD). Before beginning, the last formal document released was done so in February of 1998. We started using the EPD in the Fall of 2000. Since that time we have seen success using the newly adopted EPD process. Once we became familiar with using the EPD, we ventured into other areas to determine if they too could be bettered by process. We began creating non-project based documents such as engineering practices. We are slowing climbing the CMM levels and are seeing good results. It has been a slow process but once we started, it's been very easy to continue climbing.

Biography

Kimberly Ritchie is a PC Software Engineer for Wacom Technology Corporation in Vancouver, Washington. Kim is also the documentation control specialist for Wacom. Prior to Wacom, she worked as a programmer for the Texas Transportation Institute. She graduated from Texas A&M University in College Station, Texas. Kim is also a member of IEEE.

Kimberly Ritchie
Wacom Technology
1311 SE Cardinal Court
Vancouver, WA 98683

kritchie@wacom.com
360-896-9833

GLOSSARY

CMM – Capability Maturity Model – process model used to improve software quality by improving process. This improvement is done in levels. Each level contains several key process areas.

KPA – Key Process Area – focus area defined in a CMM level. Identifies a goal to be reached at the specific CMM level.

INTRODUCTION

This paper describes the past, ongoing, and planned efforts to incorporate process into the daily work of a small organization. For this paper, the "small organization" mentioned is the Software Engineering Group at Wacom Technology Corporation. Wacom manufactures several product lines of graphics tablets and develops the associated software as well. The Software Engineering group is broken into platform teams. There are three PC engineers, three Mac engineers and one UNIX engineer. It is our responsibility to: 1) add support to the existing code base for new products; 2) implement new features; 3) correct any issues logged by the QA team, such as defects. Requests may come from our internal Product Manager or any of the international Product Managers.

Most of these requests could be combined into projects. Not having the manpower or time needed to produce the number of documents typically supporting a software project, it would benefit us greatly to combine several documents into one document. This new document would include requirements, design specifications, management items, and implementation details. It would also contain metrics for manpower usage and schedule tracking. The creation of this project document would be the beginning of our process improvement. Before going into detail about the current process, I am going to give a brief history of where we were at the time we decided to improve our software process.

HISTORY

Several years ago, Wacom released a new line of tablets with enhanced features and transducers. It was determined that the existing code was not a good foundation on which to base the new product line. The code would be entirely rewritten. Following IEEE standards, several documents were produced to support this large project. This included a requirements document, a design document, a management document, and an implementation document. Once the project was completed, most changes to the code included new hardware support, added features, and defect corrections. These hardly were considered to be large projects. There was no time or staffing available to create the number of documents typically used for software projects. There was a period of no process after that.

THE BEGINNING

Several of the Software Engineers attended the PNSQC 2000 Workshop titled: "Making the CMM Work: Streamlining the CMM for Small Projects and Organizations." Our manager had been reading papers on CMM and had looked favorably on process

improvement. Being a small group, less than 10 engineers, we decided to look into what we had learned about CMM and determine what, if anything, we could implement into our group. We chose CMM as a model because, as a small company, we did not have a process expert in-house and it made sense to model after established "best practices." The main hurdle to overcome was deciding where to begin. As mentioned earlier, each request from management was handled individually with no formal process involved. We decided to combine these requests into defined projects. This way we would have a collective set of requests instead of handling one at a time. We could then document each project's set of requirements. The problem encountered was that, according to IEEE standards, several documents are needed to support software projects. We did not have the time or manpower to produce the documents for our small projects. It was then determined that we could combine all of the documents into one document for each project.

THE ENGINEERING PROJECT DESCRIPTION

We began focusing on a document that could be used to define any one of the projects in which we may be involved. It would include the requirements, design specifications, management items, and implementation details. The Engineering Project Description or EPD was generated. We decided to combine and consolidate Key Process Areas (KPAs) of the CMM into the EPD. Being a small organization, we prioritized the KPAs according to our specific needs. The basic format of the EPD was generated using IEEE documents as guides. Refer to Appendix A for the general outline and format of the EPD.

The EPD is a template that, when followed, will be used to create a specific project description. The titles follow the following format: "Engineering Project Description – XYZ Project". XYZ Project is, of course, replaced with the project name.

We have modified the EPD over the past two years. The current EPD sections are described towards the end of the paper.

THE FIRST YEAR

We began using the EPD for all project documents in December of 2000. In addition to the creation of the EPD template, document control was also established. All documents require approval before they are released. Once the documents are released, they require a Document Change Order or DCO before any changes are made to the document. Each EPD is reviewed by peers and managers to verify the content of the document.

An EPD template was created and put under document control. This template would be a starting point for an engineer writing a project description. An EPD procedure document was also created to help engineers use the EPD template. We did not do much refining of the EPD template the first year. We basically got used to having documents for each project. Some difficulty was encountered with keeping up with the documentation. Once changes in requirements were made, changes to the existing project description were also required. We found that this was beneficial because we had proof that requirements were changing and this was a detriment to schedule management.

THE SECOND YEAR

Once we felt comfortable with our existing EPD document, a process auditor was contracted to assess our existing CMM level and possible areas of improvement to focus. Of course, implementing the EPD did not catapult us over levels. We did, however, get valuable information as to the direction we should go in order be at a solid level two. One key suggestion was the addition of metrics into the EPD. We did not want to add metrics just for the sake of adding them. We wanted data that, if collected, would be beneficial in the long run. We added actual dates next to the estimated dates for the defined milestones. We also added actual times next to the estimated times under the Staffing section. This has helped engineers make better estimates in future projects. We have incorporated the project documents into our time sheet input. A project number is associated with a time slot entered. This makes calculating the actual time values more precise.

Since the beginning of December of 2000, we have created close to 30 requirement documents using the EPD template.

The following sections detail our current EPD template.

EPD SECTION DESCRIPTIONS

Title Page

The title page contains the document's title, document number, document's current revision, and a list of the required department's approval needed to release the document. A document numbering system is used so each document will have a unique identifier. The version would be a number for all non-released documents, such as those being dispersed for comments, and would become a letter on final release.

Preface

The Purpose, Scope, and Audience sections contain boilerplate comments about the Engineering Project Description. Any specifics to the project are spelled out in the following sections. The Reference Documents contain any documents that are specifically needed or referenced. The Glossary defines any new terms to be used in the document.

Project Summary

The Objectives section provides an overview of the project. It also describes the objectives that will be accomplished by the conclusion of the project. The Deliverables section lists the deliverables associated with the project such as software, reports, or completed release procedures. The Major Activities and Milestones section provides a broad description of events within the project and the timing of the conclusion of those events. Estimated dates are included with the activity defined. This makes it simple to see how the project is progressing. Once the project has been completed, the actual date is entered next to the estimated date. This section is mapped to the Software Project Planning key process area of Level 2. An example of an entry in the Major Activities and Milestones section is listed below:

Initial Release to QA by 10/14/02 (actual 10/13/02)

Requirements

The Base Version subsection lists the software version that the required changes or additions will be based upon. This section is mapped to the Software Configuration Management.

The remaining subsections were developed based upon our business needs and also IEEE procedures. For example, the Tablet / Transducer section details any new tablet or transducers added to the existing software package. The Requirements subsections list any differences from the base version for each respective section. Each subsection can be broken into as many subsections as needed. A list of the remaining subsections can be found in Appendix A. This maps to the CMM Requirements Management KPA of Level 2.

The lists of requirements for each section are individually numbered so that if a test plan were written, it could map back to a specific requirement. This maps to the CMM Software Quality Assurance KPA of Level 2.

Plan

The Responsible Engineer subsection lists the engineer responsible for the projects documentation and implementation. The person listed is responsible for verifying the progress of the project and also addressing any issues that come up during implementation. This subsection maps to the CMM Software Project Planning KPA of Level 2.

The Assumptions, Dependencies, and Constraints subsection lists what the section heading implies.

The Risks subsection describes any risks associated with the project. For each identified risk, there is an associated entry under the Risk Management and Mitigation section.

The schedule section describes, at the task level, the proposed schedule for the project. Milestones are clearly marked with the dates defined. This usually consists of a chart generated by scheduling software. This section maps to CMM Software Project Planning and Software Project Tracking and Oversight KPAs of Level 2.

The final subsection, Resources, is detailed below.

Resources

The Staffing subsection lists the staffing necessary to complete the project. Each person required is listed with the estimated number of engineer-days required for project completion. Once the project is completed, the actual number of engineer-days is entered next to the estimated days. An example of an entry under the Staffing subsection when the project has been completed is listed below:

Kim Ritchie, PC Engineer 35 days (actual 30 days)

The remaining subsections were developed using our needs for a project description. Each subsection can be broken into subsections in needed. A list of the remaining subsections can be found in Appendix A.

Process

The Reporting, Schedule Management, and Project Retirement sections are boilerplate for all project descriptions. They do not require additional text. The Risk Management and Mitigation section does require additional text.

The Reporting section states that status reports will be made to engineering management at the weekly status meetings. Reporting will also include schedule tracking and issue discussion. The Reporting section maps to CMM Software Project Tracking and Oversight KPA of Level 2.

The Risk Management and Mitigation section lists the actions required to mitigate the risks listed under the Risks section. Each risk listed in the Risks section requires an action to be listed. This section also states that risk re-assessment will be made at each milestone prior to release.

The Schedule Management section lists the actions taken if there is a deviation from the schedule. For our purposes, if minor slips in the schedule will be resolved by dropping defect corrections in reverse, priority order. If there are no defect corrections to be dropped, the project will be re-scoped with Product Management coordination. The project description will then be updated with the changes.

The Project Retirement section lists the criteria for project retirement. Once the project has been completed, all actual values will be added next to the estimated ones and the Revision of the document will be changed to “OBS”. This signifies the document is obsolete.

BEYOND THE EPD

The EPD just handles process for defined projects. We have also put focus on non-process documents, such as Engineering Practices, Documentation Practices and Release Procedures. We have created an additional 15 of these documents so far. These additional documents map to the CMM Organization Process Definition KPA of Level 3. The Engineering Practices document details the training procedure for newly hired and existing engineers. This maps to the CMM Training Program KPA of Level 3. A Configuration Management document is also planned to define our software control process.

RESULTS

Following a major software release last summer, a post-mortem was held to review “the good, the bad and the ugly” from the life cycle of the project. This review included Product Management, Quality Assurance, and Software Engineering. Most of the good comments were related to our newly adopted practices. The project documentation provided clear details that were easily testable. There were no questions as to the requirements of the project. The engineering schedule was met without any issues. Following the post-mortem, the Engineering group realized that the reason we did well on this project was due to our existing practices. It easily gave us the confidence to continue on and continue optimizing for the best results.

CONCLUSION

Looking back over the past two years, one thing has been essential in establishing our existing process… ability to change. Our group had to move in an entirely different direction than we were previously going. We are still continually changing the way we do things. We are continually optimizing our practices to better suit our needs while still striving to climb CMM levels.

REFERENCES

Paulk, M., Curtis, B., Chrissis, M., Weber, C., *Capability Maturity Model for Software, Version 1.1*, CMU/SEI-93-TR-024, http://www.sei.cmu.edu/pub/documents/93.reports/pdf/tr24.93.pdf.

IEEE Recommended Practice for Software Requirements Specifications, Std. 830-1993, IEEE Standards Collection, Software Engineering, 1994 Edition.

Appendix A
Engineering Project Description Template Sections

Title Page
Preface
- Purpose
- Scope
- Audience
- Reference Documents
- Glossary

Project Summary
- Objectives
- Deliverables
- Major Activities and Milestones

Requirements
- Base Version
- Tablet / Transducer
- Platform
- Interfaces
- Behavior
- Look and Feel
- Installation
- Defect Corrections
- *Other Requirement Sections not defined by the template*

Plan
- Responsible Engineer
- Assumptions
- Dependencies
- Constraints
- Risks
- Schedule
- Resources
  - Staffing
  - Equipment
  - Software
  - *Other Resources not defined by the template*

Process
- Reporting
- Risk Management and Mitigation
- Schedule Management
- Project Retirement

The Services Oriented Applications Lifecycle: A Proven Methodology for Developing Quality Web Services

Abstract: Web services proponents claim that their architectures will enable businesses to develop software applications with greater agility and responsiveness to change. Without the processes to support such flexibility, teams can easily find that the amount of change that they encounter overwhelms their ability to manage it. The E-Services Lifecycle was developed to enable development teams to successfully manage Web services development programs with high degrees of risk and change. It has been tested with over a dozen teams inside Hewlett-Packard, and has been demonstrated to reduce time to market and increase flexibility for the business without sacrificing quality.

Author's Biography:

Katherine Radeka is a software development process consultant for Hewlett-Packard. She coaches Web services development teams to adopt service oriented analysis, evolutionary delivery and program management best practices to improve quality and shorten time to market.

She has over ten years of experience in Information Technology, and has spent the past two years focused on using evolutionary delivery to improve Hewlett-Packard's ability to deliver solutions internally and externally. The teams benefit from faster time to market, a deeper understanding of customer needs, and improved means of managing the changes and risks associated with Web services development.

Katherine Radeka
Hewlett-Packard Company
18110 SE 34th Street
Vancouver, WA 98683

katherine_radeka@hp.com
(360) 212-0375

The Services Oriented Applications Lifecycle: A Proven Methodology for Developing Quality Web Services

Introduction

For the past two years, major industry leaders such as Sun, Microsoft, Hewlett-Packard and IBM have heralded a new paradigm – services oriented computing. This new way of thinking about building software promises to deliver systems that are more responsive, enabling the businesses to be more agile in the face of change, and creating a new kind of IT organization, which is seen as an essential strategic partner in business decisions.

If an organization embraces Web services but tries to develop them in the same way they developed legacy systems or implemented large off-the-shelf software packages, they lock themselves into the same behaviors which lead to long development cycle times, schedule overruns and disappointed customers. These problems are amplified in Web services development, since the technology and customer interactions required to build them do not respond well to traditional software development techniques. These projects have high levels of risk and change. Any development methodology for services developers must address this, without overburdening the team.

Web services provide both the opportunity and the imperative for organizations to think differently about software development. Agile development methodologies such as eXtreme Programming (XP), Crystal, SCRUM and RAD enable developers to profit from the strengths of Web services development, and mitigate the risks. The Services Oriented Applications Lifecycle pulls together best practices from a variety of agile software development methods into a framework that is optimized for Web services development. Pilot teams have demonstrated that the SOAL methodology delivers these benefits to services oriented development teams:

- Faster time to market with no loss of quality
- Increased responsiveness to business change
- Increased change and risk management capabilities
- Improved visibility into project progress

These benefits enable teams to harness the power of services oriented development to improve business results and build an IT organization with the agility to respond to a changing environment.

What is a Web Service?

A Web service is a service delivered over the Web – a complete service, instead of just a replacement for something that can be done off-line. It is more than just shopping or booking travel on-line. A Web service takes something that a user does frequently, and makes it happen automatically. It is proactive – it looks for opportunities to be helpful. If the last flight back home is cancelled, it makes a hotel reservation. If a customer's favorite author publishes a new book, it notifies the customer that the book is available and reserves a copy.

The customer of a Web service can be a person, a device or even another Web service. A printing service may route a document to a specific device and set the printing options. A payment service might handle all of the credit card processing procedures for other Web services. So long as the services is a way to communicate, Web services can be combined and recombined to eliminate more manual work from the system for the developers, and ultimately the end user.

At a technical level, Web services use standard protocols to enable inter-component communication. Each component service performs one step of a business process very well. The components can reside on different systems, even at different companies. An aggregator service coordinates communication between the different components to fulfill an instance of a business process. The overall Web service fulfills a significant task for the user, once that cannot be done with a standard Web application.

A Services Oriented Development Methodology

With any technology that has not been around for very long, the customers themselves have a difficult time articulating what they would want such a service to do, until they have had the opportunity to try one out. This leads to a great deal of requirements instability during development. At the same time, the technologies and standards for building Web services are immature and evolving rapidly. This creates a great deal of project risk.

Teams who are new to Web services development tend to have similar problems. The team may need to integrate with component services that are located literally anywhere. New technologies may not work as expected. The levels of change and risk to manage can overwhelm the project team.

The Services Oriented Applications Lifecycle is a program management framework that addresses the specific challenges faced by Web services development teams. Over a dozen teams are actively using the methodology inside Hewlett-Packard, and the results demonstrate that these teams deliver real solutions faster, with a greater fit between the expected and achieved results. The specific techniques recommended by the methodology enable teams to successfully complete Web services projects, while managing the high levels of change and risk that are inherent in Web services development.

Focus on Service

For a Web services customer, the key word is "service." That implies a more meaningful relationship with the consumer than a Web application would typically have. In exchange for the convenience of using a Web service, the customer gives information and control to the service. The bookstore knows a customer's favorite authors, and has permission to commit to a purchase. The travel service knows what credit card to use to secure a hotel reservation, and has the authority to make a reservation.

To maintain such an intimate customer relationship, a Web service must uphold high standards of quality. The customers who are using a Web service must trust the Web service to do what it says it will do. Since the customers relinquish more control to the Web service in exchange for having to do less work, it is important that the Web service function properly, to enable the customers to develop trust. This is not an area where pushing out a bug-filled version 1.0 will be acceptable.

Web services also must have the flexibility to evolve, to gather information from the environment and learn from it. Human-delivered services naturally evolve in response to changes in the environment and feedback from the customer. Each time a trainer gives a class, she learns more about what works and does not work to help her students learn the materials. Each training session is more effective than the last one, because the trainer has the benefit of the feedback from previous sessions. Web services need that same ability to learn.

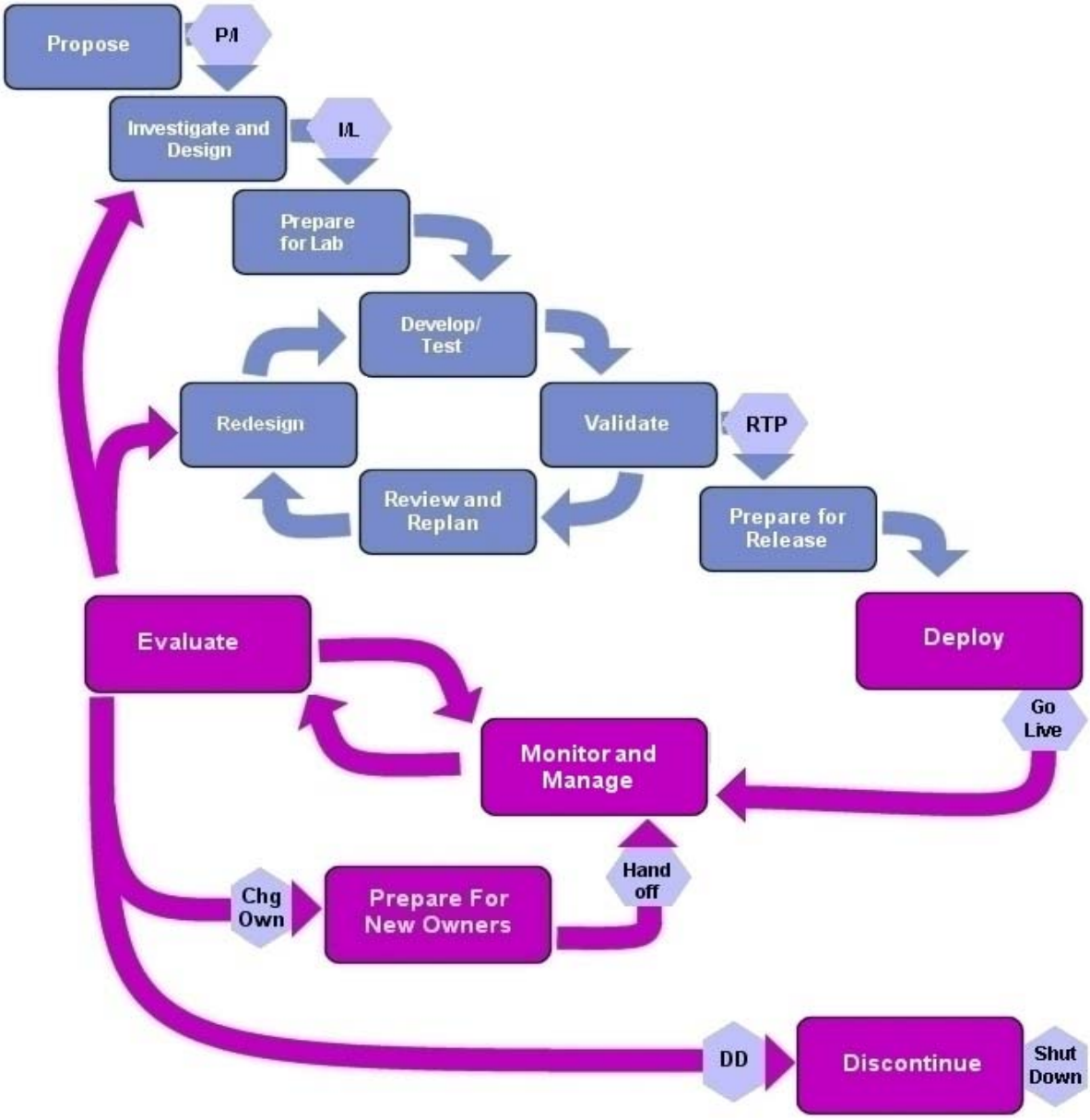


Figure 1: Services Oriented Development Lifecycle

The Services Oriented Applications Lifecycle (SOAL)

Last year, a team of services oriented developers from across Hewlett-Packard decided to look at the specific challenges and issues that Web services teams faced. They documented their findings in the Services Oriented Applications Lifecycle (SOAL), shown in Figure 1. It describes the best practices and processes that are more likely to help Web services teams accommodate the

high levels of change and risk associated with Web services development. It enables teams to deliver a high quality service that can learn from customer responses and develop solid customer relationships that improve the Web service's chances to be successful.

SOAL defines phases, development cycles and checkpoints. Phases, represented by rounded rectangles, are periods of time when activities take place. Blue reflects those phases that are most visible to the development team, while purple phases have direct impact upon the customer base for the service. The phases are described in detail below. Development cycles are groupings of phases which go around in a circle:

- The Development Cycle includes the Develop/Test, Validate, Review/Replan and Redesign phases. It represents the team's iterative development work.
- The Production Cycle includes the Development Cycle, plus the deployment and production phases. It describes the evolutionary delivery of the service, with incremental deployments as the team develops new features.

Checkpoints are represented by purple hexagons, and represent major decision points in the lifecycle. The acronyms for these decision points are the ones that Hewlett-Packard uses, but the decisions are common to all lifecycles:

- P/I (Proposal-to-Investigation): Management agrees to provide resources for an investigation into the merits of an idea for a new service.
- I/L (Investigation-to-Lab): Management agrees to provide resources for developing a new service.
- RTP (Release-to-Production): The entire team agrees that the service is ready to be released to customers.
- Go Live: The team confirms that the service has been released.
- Change Owners: Management decides to transfer ownership to a new team.
- Handoff: The new team confirms that it is ready to accept ownership.
- DD (Decision-to-Discontinue): The team decides to discontinue a service.
- Shut Down: The team confirms that the service has been discontinued.

A Web Services Implementation of Evolutionary Delivery

The SOAL team discovered that those Web services teams who started with the expectation that a high rate of change would be the norm and who put an iterative development lifecycle in place to manage it, tended to be more successful than those teams who attempted to control or minimize change and followed more sequential lifecycles. They were able to release their products earlier, and the Web services had fewer defects and greater customer satisfaction.

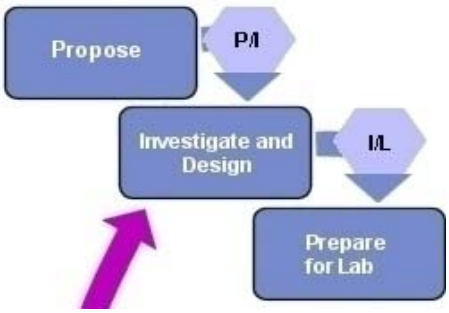
SOAL is an adaptation of evolutionary delivery to the world of Web services. The standard evolutionary delivery lifecycle contains one loop of iterative development. Each development loop contains a little requirements gathering, a little design, some development and testing. At the end of each loop, the development team releases a little more of the application until the application is done (or at least development is discontinued). Then the application passes into a typical "maintenance" or "support" phase before going into obsolescence.

With Web services, projects have a difficult time reaching the "maintenance" phase in the standard evolutionary lifecycle. Because the technology is new to the users, and because the

technologies are immature, there is a lot of new information which is not available to the team until the Web service is in production. A Web services team that does not capitalize on the opportunities presented by this new information risks losing the valuable relationship that the Web service has with its customers.

The diagram shows that the SOAL methodology emphasizes the steps that the team needs to take once the Web service is in production. While most lifecycles based on evolutionary delivery have one kind of development cycle, Web services teams need two interlocking cycles: one to manage the iterative development process, and one to capture information from current customers, and use it to evolve the Web service. The Prepare for New Owners branch explicitly addresses the need to hand off development responsibility since that is both common and a significant source of risk for services oriented developers. The diagram also shows that even after the rate of change has slowed, Web services must receive periodic evaluations to ensure that the Web service's value justifies the expense of managing it in production.

Preparatory Phases



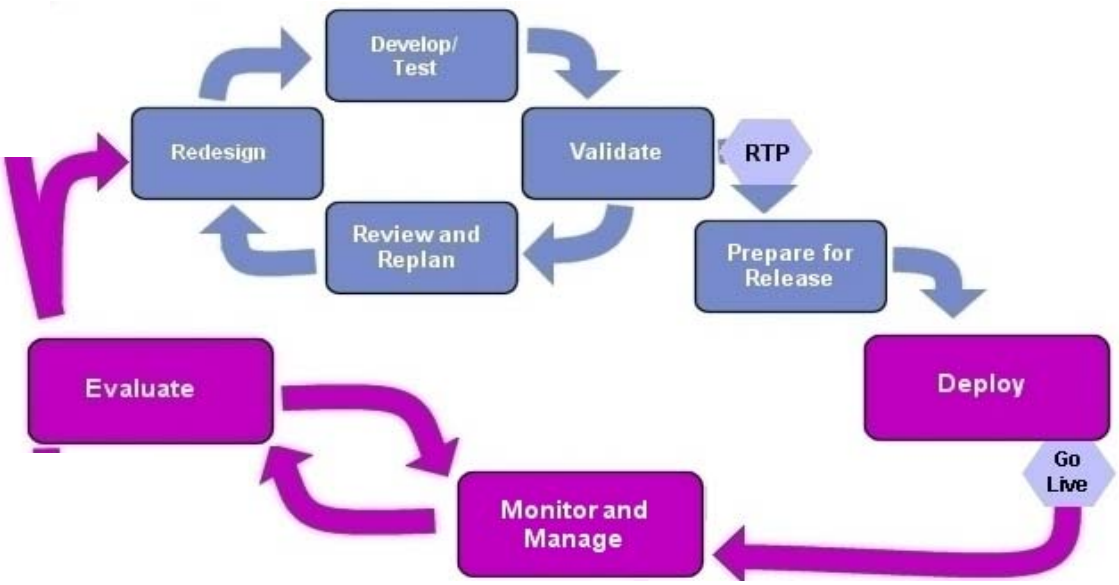
Before plunging into development, the team passes through three preparatory phases. **Propose** is the idea phase, the phase for building management buy-in to take a closer look at a potential Web service offering. The primary outcome is a high level description of the problem or opportunity, and how the Web service is going to address it. The **P/I checkpoint** validates this preliminary investigation, and management's agreement to fund a more detailed investigation.

Investigate is for determining feasibility: is the problem or opportunity really what it appears to be, and is the Web service the solution? The activities include initial requirements analysis, some initial design, perhaps some early prototyping as well as market or ROI analysis. At the conclusion of this phase, the team passes the **I/L checkpoint**, where management agrees to fund the service's implementation.

The **Prepare for Lab** phase is the team's "deep breath" before plunging into development. It provides the team with the opportunity to ensure that they have the infrastructure that they need to move rapidly through development cycles. This is important in all projects, but it is especially important in Web services projects, where the architecture must be reproduced on multiple systems to allow for smooth transition from the development environment through testing to the production environment.

It is a time for doing all those things that the team will not have time for, once development is underway. This includes a laundry list of things like setting coding standards, getting the team up and running on configuration management and the build process, making agreements with Test. The team also does their first detailed design work during this phase.

Two cycles of evolutionary delivery



There are two interlocking cycles of evolution: the **Development Cycle** (the inner loop), and the **Production Cycle** (the outer loop). The diagram shows how they are integrated into one process flow, essentially allowing the team to benefit from the best of evolutionary prototyping and evolutionary delivery. It illustrates the two main feedback loops: the initial feedback that comes from demonstrating early work to potential customers, and the feedback the team begins to receive once the service is in production.

Viewing the development process in this way enables teams to benefit from everything that they are going to learn, during development and after the Web service goes into production. It prepares them to think about how they are going to capture the information – how they are going to track the consumer's experience with the service, and how they are going to measure performance. The feedback loop is already in place, and everyone from management on down is prepared to act on new information. The team shares the understanding that for Web services, the real development work begins after the first version is in production.

The process flow diagram also shows that a team does not have to do a Release to Production for every development cycle. New functionality, especially, benefits from some user feedback prior to release, and perhaps even a limited release at first. In practice, a team may do several Development Cycles between Production releases.

Development Cycle

The core development cycle consists of four steps:

- **Develop/Test**: the team writes code and does unit-level and component-level testing.
- **Validate**: they test their work against user expectations, through demos and system-level testing.
- **Review and Replan**: the team, sponsors and other key stakeholders review progress to date and plan where to go next.
- **Redesign**: an explicit step for updating all project documentation to reflect the team's current understanding of what they have done, and what they will do.

This development cycle facilitates change management by providing a natural point in the cycle to scan the environment for change. If a team keeps its development cycles short enough, customers and management are never far away from a pre-scheduled opportunity to let the team know about changes that impact them, during the Validate and Review phases. This helps prevent side conversations that interrupt the team during their development phases.

They also have high visibility into the project's progress. Every two to four weeks, they have an opportunity to see what the team has done so far. After the Review and Replan step, the program manager has an accurate picture of the progress that the team has made, and how it compares with management expectations.

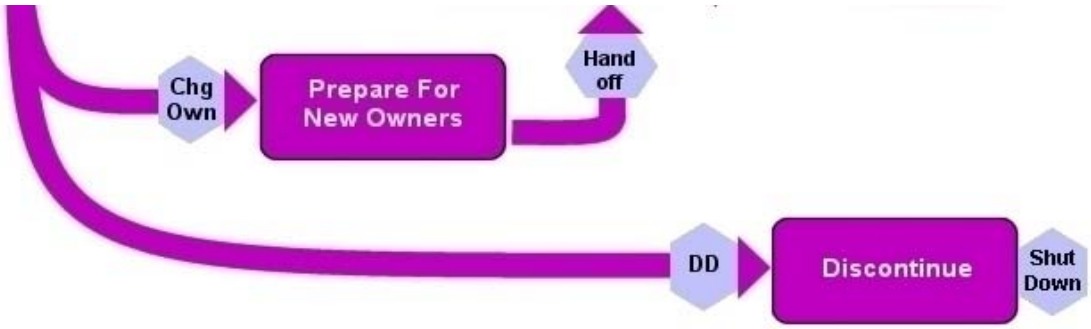
Production Cycle

Once the service is ready for its first release to real customers, it goes into the Production Cycle, which includes the Development Cycle and adds four phases to manage the running version of the service:

- **Prepare for Production**: time for the business (Marketing, Finance, Customer Support) to prepare themselves for the Web service implementation. Depending upon the type of service, this may include training, organizational change implementation, and pre-launch communications.
- **Deploy**: the process of moving the Web service into its production environment.
- **Monitor and Manage:** the closest thing in SOAL to a typical "maintenance" phase. Unlike waterfall lifecycles, however the team does not stay here long before they move into the next development cycle.
- **Evaluate:** Review the results of this cycle, and determine what to do next.

The Production Cycle ensures that feedback from real customers makes it back to the development team, to be addressed in future releases. The Evaluate phase has multiple branches depending upon the feedback the team has received and what they decide to do about it: do a minor enhancement that feeds back into the development cycle, do a major enhancement or upgrade that requires a return to Investigate and Design, leave it as-is by returning to Monitor and Manage, change the development team responsible for the service or Discontinue the service.

Post-Production Phases



After a service has been in production for some time, it may pass through a Prepare for New Owners branch:

- **Change Owners checkpoint**: the management decision to move the service's ownership from one team to another.
- **Prepare for New Owners phase**: the work of preparing the new team for the transition.
- **Hand Off checkpoint**: confirmation from the new team that they have what they need to take over the service.

These phases are called out explicitly, because it is easy to discount the amount of work involved in creating a smooth transition that minimizes the risk to existing customer relationships.

End of life is another time when it is easy to underestimate the amount of work required to decommission a service. Customers may need a transition plan, and the company may have other legal or business requirements to fulfill. SOAL handles this in three steps:

- **DD checkpoint:** the management decision to discontinue a service.
- **Discontinue phase:** the work of turning off the service, and communicating the change to those affected by it.
- **Shut down checkpoint:** the confirmation that the service has been turned off.

Best Practices for Using SOAL

In addition to the process flow itself, the SOAL methodology has some specific recommendations for managing the overall development process for Web services, drawn from the team's experiences with agile development methodologies:

- Set a short, regular development cycle and timebox all development within the cycle. Timeboxing is a technique which sets an absolute time limit for each phase. Work that is not done may slip to a later cycle, but the cycle itself is declared to be done when the time is up. The team may not review as much as they intend if something happens, but the reviews never slip. This is a major difference for a team that is used to postponing reviews when they run into unexpected difficulties, but experience shows that it provides immediate visibility into problems that might affect the schedule.
- Assign a theme to each development cycle. This gives the team something to focus on. Typical themes include logical subsets of the functionality, compliance with a specific standard, support for a specific platform. It enables the team to point to something concrete that they have accomplished at the end of every cycle.
- Only do detailed planning for the next two or three iterations. Beyond that, there is likely to be so much change that it is not worth the time to plan earlier. It also takes a project manager some time to understand how much a team can realistically accomplish in a development cycle.
- Release early, without sacrificing quality: Teams that use the SOAL methodology typically release smaller, more focused Web services than they may have originally intended. They may decide to target one specific kind of customer or provide one specific service out of a suite of planned services. This enables them to get something out there in the real world, and start gathering information from the environment earlier than if they waited until everything was finished before releasing. This does not get them off the hook on quality – whatever they release must work well.

Results from the Field

To test out the SOAL methodology, the team recruited several projects representing three different kinds of Web services: an end-to-end enterprise IT Web service for getting and posting information from multiple data sources, a workflow management Web service, a service that is aimed at Hewlett Packard's external development partners, and a service that is aimed at consumers.

These pilot teams have demonstrated that the SOAL methodology delivers these benefits to services oriented development teams:

- **Faster time to market with no loss of quality:** These teams reported that the SOAL methodology enabled them to release workable Web services early without sacrificing quality. By focusing on those things that were absolutely needed for the first release and postponing anything else until after the first release, one team was able to meet a critical deadline for demonstrating their Web service at a trade show, despite a pre-SOAL schedule that was three months too long

- **Increased responsiveness to business change:** The teams reported that they were also able to get much closer to what the customer actually needed, since the built in feedback mechanisms were in place. The built-in processes to manage change and risk enabled them to make better decisions at each point in the process. The enterprise IT project had a hard deadline of ninety days to deliver as much functionality as they could. By using a two week development cycle, they were able to continuously steer the project towards the areas that would deliver the most business value.

- **Increased change and risk management capabilities:** One of the teams was strongly impacted by Hewlett-Packard's merger with Compaq. The lifecycle's built in feedback loops enabled the team to mitigate the risks by shifting their development priorities in response to the latest information. This enabled them to quickly demonstrate that their service would deliver maximum value in the new organization, and that the development team was responsive to business needs.

- **Improved visibility into project progress:** Since teams never go more than a few weeks without a review, it is easy for management to steer the teams' efforts. Midway through the enterprise IT project, organizational roles and responsibilities changed for key components of the program. Since management had accurate information about the project's scope and status, they made good decisions about where to place the project in the new organizational structure.

.

One promise of services oriented computing is that teams will be able to reuse services developed by other teams. The process enabled the teams to share the Web services they developed with other teams, so that they could be incorporated into new solutions. One project developed into an entire group that is now entirely dedicated to building Web services for other Web services to consume, providing common functionality around revenue capture, notifications and printing.

After we had fine-tuned the process, we began broader trainings and adaptations. To date, more than a dozen teams have used the SOAL methodology, and as a result, Hewlett-Packard's services developers are gaining a reputation for agility, high quality and responsiveness that results in improved profitability for their business partners.

The Road to Insanity
– New Goals, Less Staff, Shorter Schedule, Better Quality

Authors:
Louis Testa,
Nancy Munoz
GE Medical Systems, formerly MedicaLogic/Medscape
(503) 531-7000

Abstract
Development organizations sometimes face expectations from management that appear unreasonable in that they may severely constrain all of the elements of the development matrix: functionality, schedule, resources, and quality. This paper describes the authors' experience with an over-constrained expectation, and the variety of techniques used to meet the goals set.

Author Biographies
Louis Testa received his BS in Engineering at California Institute of Technology (CALTECH) in 1978, and his MS in EE at UC Berkeley. His master's thesis project was a software program for simulating electronic circuits using RC modeling. His first position was at Silicon Systems where he designed a dozen different integrated circuits, and later managed software development to aid in the creation of integrated circuits. His next position was at TSSI/Summit Design, where he worked in several capacities: developer, manager, and Director of Engineering. TSSI produced software that generated test programs for commercial ATEs. In 1996, he moved to IMS as Director of Advanced Development in the Virtual Test Division.

Louis Testa was with MedicaLogic (now GE Medical Systems) for 3 years, severing as Director of Product Development; Director of Engineering; and Director of Process, Planning and QA. MedicaLogic produces Electronic Medical Record software. He can be reached at lct203@hotmail.com.

Nancy Munoz received her BS in Computer Science from Winona State University. She also holds a BA in Urban Planning from the University of Washington. She has 14 years experience in software development and testing of communications and business application software. Her first job in the computer industry was with IBM implementing a TCPIP test suite for the Department of Defense. Later she moved to ADP Dealer Services in Portland where she worked on their accounting application and systems software.

Nancy Munoz is currently the Software QA Manager at GE Medical Systems Information Technologies (formerly MedicaLogic) where has seen it grow from a small startup to a public company.

The Road to Insanity
– New Goals, Less Staff, Shorter Schedule, Better Quality

Introduction
MedicaLogic started 2001 with a development plan in place that was already half executed. Two separate shocks altered the landscape: First, new management came in and changed the product direction and features. Second, a number of engineering and QA staff were laid off. The new goals were to get the product out quicker, with fewer people, and with better quality.

This submission is about our experience in responding to these challenging directives. It is broken into two sections. The first part is a chronology of Logician 5.5 development and the techniques that were applied to make it successful. The second part describes quality measurement techniques used on several projects to improve understanding of project status.

Logician 5.5 Release Story

Logician 5.5 Release Challenge
Introduced seven years ago, Logician is MedicaLogic's flagship product. It is a large application for managing workflow and documenting patient information in an outpatient medical clinic. Moderate to large size releases were introduced almost annually ever since.

The Logician 5.5 project work began after Logician 5.4 was shipped in August of 2000. The original release was feature driven, with a target date of September 2001. By the end of January 2001, many Logician 5.5 development projects were well into the implementation phase. However, a change in senior management significantly changed the product direction. The QA department was directed to reduce project staff by one third, cancel some projects, add other projects, and ship the results two months earlier. The ship date was non-negotiable and had to be July 31st, 2001, 6 months later.

The first step was to quickly pull together an estimate of the new work and time it would take. The resulting estimate was a bare minimum of 8 months. The goal was to find a way to deliver it two months quicker. QA and engineering agreed to split the problem in two by cutting one month from development and one month from testing time. From a QA perspective, this required doing all the final testing in 13 weeks instead of 17 weeks required in the last release. It also had to be done with 12 QA engineers instead of the 17 used in the last release cycle.

Modeling the Release
The core methodology was to carefully model the current situation and then use this model to explore different options. The model was iterated on until the desired result was reached. The model was then used to monitor our progress toward the end goal and continually estimate likely outcomes.

Creating a model requires understanding past behavior. To do this, it is useful to have past data showing staffing, project tasks, and schedules. When some of this data isn't available, making best estimates and documenting them is still worthwhile.

The initial model was set up in a spreadsheet program, following these general guidelines:

- Create a block diagram of the development process for the effort.
- Create the initial spreadsheet worksheet to represent this high level rollup as a timeline.
- Expand each major development segment into the identifiable sub-components. It is usually easier to put each sub-block on separate worksheets and link their output to the summary worksheet page.
- Expand each section down to simple models that can be tied to past data or reasonable assumptions. Write up the assumptions in the spreadsheet as these will get tested and reviewed through-out the process.

Other construction recommendations:

- Keep units consistent across the entire model. In practice, working days work well as it is a good compromise between too much and too little detail. However, keep the conversion back to calendar days at the summary page of the spreadsheet.
- Add comments and notes to explain formulas used.
- Account for the fact that everyone is not available 100% of the workweek to work on the project. Mandatory company meetings, training, vacations, secondary projects, and sick time take up a predictable amount of time over the entire team.
- Allow that people are not 1-1 fungible between tasks when building the model.
- Keep the model simple but complete enough to be useful.
- Continue building and reviewing the model until you believe it.

The Logician 5.5 high-level steps are illustrated in Figure 1. Each block is terminated by a review milestone that is part of the high-level schedule and plan. The top-level spreadsheet shows these blocks and the end dates. A brief discussion of these blocks follows.

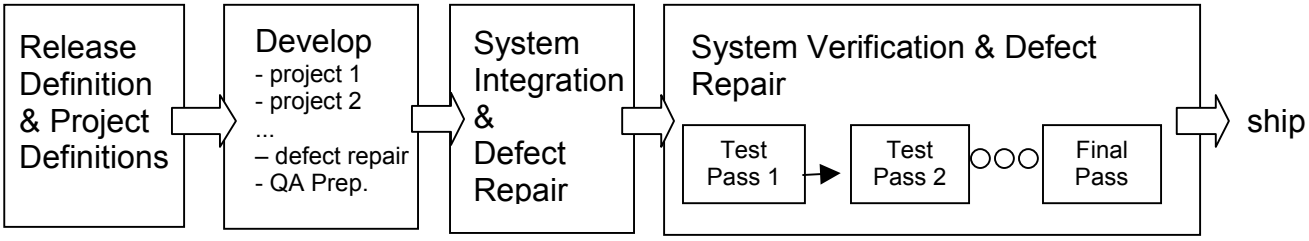


Fig 1. Top level view of Logician model elements

Definitions Block Model

This step ends with a completed high-level definition of the release as a whole and descriptions of the project efforts. A fixed time allocation for definition was set. This was broken down further into specific steps needed to complete the definition.

Develop Block Model

The development phase consists of a number of parallel efforts: individual projects, defect repair, and QA preparation for the final Verification. Each individual project is broken down into detailed definition, design, coding, and unit/module test steps. Defects found during any part of the Develop phase are assumed to be repaired prior to system integration. The model for the Develop phase should show a number of separate parallel efforts where the long pole item sets the completion date. This part of the model can be complex enough that it is preferable to use a Project Management tool to do the schedule layout and shifting, then take the resulting plan and model that.

Some efforts that take place during development aren't always accounted for. One in particular is time for defect repair. As Logician has been through a number of releases, there are often defects that didn't get repaired in past releases that are to be repaired in the current release. An example would be a defect that received only a quick patch treatment in the last release that is now getting a better repair. Another example would be a medium priority defect that had a workaround from the prior release and time didn't permit its repair earlier. Defect time can be estimated by adding newly created defect estimates to counts of known defects to be repaired. It is also important to account for a percentage of improperly done repairs.

Newly created defects can be estimated based on past history. An alternate approach is to estimate the new development effort in days and multiply by a constant (average number of defects per day of effort).

Existing defects are known, but it is important to review all of them early in the process to decide which will be fixed and stick to this list.

Failed defect repairs will yield additional time. A simple approach is assume a failure percentage based on your past history or industry numbers. Failed defect repair rates from 5% to 20% are not uncommon.

System Integration Block Model

Historic integration times are the best resource for the System Integration Block model. It is important to assume that a number of defects will be discovered and require repair during this phase. Performance issues in the code are often resolved at this point. Extra time from the historic numbers was added in the 5.5 release to allow for full defect detection and early resolution.

System Verification and Defect Repair Block Model

The System Verification block covers the test passes through the entire product. In each pass, the code is tested, and defects are discovered, repaired, and verified. Each pass through the product has both a fixed and variable time element: the time to test the product is fixed; the time to identify and repair the defects is variable. The model should reflect this.

The number of passes through the code has a huge impact on the total time to complete the system checkout. It is driven by the number of defects left in the product at the end of System Integration: a perfect product would require one pass through to verify it.

To calculate the number of test passes, one has to make assumptions about incoming total number of defects, the defect discovery rate, the failed verification rate, and what an acceptable level is before the testing changes from full evaluation to spot checks of fixes.

To model the time for each pass, consider the number of tests to perform, time per test, estimated defects to discover, repair and verify, and fixed times associated with preparing the test pass and building/packaging the code. Separate estimates need to be made for those parts of the test that are automated vs. manually tested. In Logician's case, there were about 35,000 tests total with 40% of them automated and 60% manual.

Overall, our model accounted for all of these elements and was our starting point for evaluating how to improve the outcome.

Process Changes for Logician 5.5

Once the model was in place, improving the projected result was the next step. Improving the outcome required a number of changes in the way things were done. For each change, it was important to understand its impact on the big picture, not just on the local effort. There are

many cases where improving the time or effort for one team can make things worse overall for the release. Consistently looking at the big picture effects and modeling them is always important.

A number of specific improvements are discussed in this section.

Task Prioritization, Estimation & Scheduling Improvements

Setting up the plan is the most important part of the task. One main rule should apply when doing this: Don't lie to yourself because it is expedient. Write down all assumptions, as these will get tested during development.

In calculating costs of tasks, it was necessary to figure out the true cost of tasks and make best estimates. This included accounting for true effective time of staff. Company meetings, mandatory training, vacations, sick leave, and other time consumers were estimated and accounted for. Scheduling realistically was essential, even when the answer wasn't what was desired. This analysis was a great starting point for real optimization.

When approaching an impossible release date, it is often good to start with the assumption that it is possible, and ask how would it have been achieved. That is, start with the desired schedule outcome and ask how time would have to be allocated across the different process milestones to achieve this. From there, the same question can be asked of the individual process step – what would have to happen for the desired outcome to occur.

Starting with the required schedule and working back to the implementation was applied to several elements of the Logician 5.5 schedule. To illustrate, consider the System Verification step. The previous Logician release had required 3 full system passes and 3-4 rounds of spot checks to complete. The only way to meet the 5.5 schedule goal was to have 2 full passes and 1-2 rounds of spot checks. From the model discussed earlier, this sets an upper limit on the maximum number of defects that could be delivered to QA at the start of this step.

Optimizing the Schedule

A number of approaches were applied to optimize the Logician 5.5 schedule. First, all work that could be was moved out of the critical path. This was done in several areas: as the packaging and database upgrade work was pulled out of the critical path by doing trial run prior to System Verification. Also, performance and scalability testing was done earlier and repeated during stabilization. Only significant speed problems were addressed and then double-checked.

Second, we looked for improvements that could be made by shifting effort from one block in the schedule to the other. Some of the work done during System Verification could be prepared during the Development step. Examples of this include: doing legacy defect repair and checking them during the development phase instead of during System Integration; and pre-testing and optimizing the build environments during the Develop phase instead of at the start of System Integration.

Third, we involved the whole team in the discussion of optimizations. A large number of changes were evaluated and some implemented. Examples include: simplifying the defect review process by standardizing the review method; not starting test passes when outstanding known-and-to-be-repaired defects exist; and clear communication of status, especially as the release approaches a milestone.

Defect Management Improvements

Defects often get ignored as a major schedule concern until the development effort is complete. This is a mistake. Defect management controls a large percentage of the overall schedule.

Consider that if the product were perfect out of engineering, it would only require one test pass evaluation to verify this. Then, it would be ready for shipment.

To manage defects well, it was necessary to get the incoming rate of defects at the start of the System Verification step as low as possible. Our goal was then to aggressively tackle defects in the first test pass to ensure that they were found and corrected.

For Logician 5.5, a defect model was set up at the beginning. All known defects were reviewed and categorized as to their priority. A set of known defects were assigned to developers to fix for the release, preferably early in the timeline. Added to known defects was a count of expected defects. This was calculated based on the number of weeks of development effort and a constant. The constant, based on historical data, reflected expected defect introduction per week of effort. The model assumed that these defects would be present prior to system integration and then estimated how many would get found and fixed, during each stage. Estimates for secondary failures due to each bug repair were included – typically this number would average 10%.

The team also emphasized the importance of fixing each defect correctly. Lowering the failure rates in defect repair would shorten the System Verification time considerably and could easily remove a test pass. A huge contribution to this succeeding is simply to have all development management telling the staff the same message: do it right the first time instead of do it quickly. A second factor was the assignment of QA staff paired up with engineers for each defect. They would discuss what was to be done, fix it, and then review the results.

Project and Release Definition Improvements

Because of the tight timeline and constrained resources, getting the project definitions right for the release was critical. Success required the marketing team to clearly define the goals early and not change them, which they exceeded in doing. Having a cooperative instead of adversarial relationship between development and marketing is necessary to make tight schedules. This sometimes depends on the personalities of the people involved. However, it can be improved by development management taking an open book approach with marketing – show them all the data and estimates and ask for suggestions. Most marketing managers will respond positively and help in making the trade-offs necessary. This was the approach used for the Logician 5.5 release.

To avoid late discoveries of problems in the definitions of projects, cross-functional teams were set up to review high level definitions before a huge amount of effort was put into writing them up. A cross-functional team usually included members from engineering, marketing, QA, documentation, support, and the field.

Development Step Improvements

It is a general axiom of development that QA doesn't determine product quality, the software engineers writing the code do. A key driver is having the engineering management emphasizing quality instead of schedule dates of individual milestones. In past efforts, making intermediate milestone dates to please top management was a priority. However, this often led to the problems being improperly fixed and causing bigger delays later. The Logician 5.5 team demonstrated that it was better to be a few days late on early milestones to make sure the job was done right than to move ahead and fix those problems sometime in the future at a greater cost.

System Integration Step Improvements

Having a distinct System Integration period with specific start and stop criteria was an important difference in the Logician 5.5 release cycle from previous releases. After the start of the

integration period, no more enhancements to the product would be made. Initial system performance was measured and major regressions were fixed. The requirement to end the integration period was that all known defects that were required to be fixed for the release had to be fixed and verified. This improvement led to a clean handoff of the product to QA for System Verification.

System Verification Step Improvements

The System Verification of the 5.5 release was a major time component of the release cycle. Because Logician is used in clinical settings where it can affect healthcare, quality has to be a key focus. There are over 35,000 tests that are run to verify the product, which take considerable time as only 40% of them are automated. Overall, it took 17 weeks of effort to complete the verification process in the Logician 5.4 release. Our goal was to reduce this time to 13 weeks for Logician 5.5.

Several factors went into the reduction of the verification time. First, cutting down the incoming defect rate to System Verification was a key element – one less test pass was required. Second, a few days of "smoke tests" had been run in the past to ensure the build was acceptable for evaluation. These were eliminated and the focus instead was on getting top quality deliverables right out of engineering. Third, decisions about which defects found required fixing used to be made by a committee a few times a week. This introduced delays and extra overhead time for the reviewing committee. For Logician 5.5, the engineering and QA managers would meet on the spot and make a decision on internally found defects, saving considerable time. Fourth, the customer support team and field engineers were involved early in looking at the product so that they could identify problems early in the verification cycle. Finally, the QA work during a test pass was reorganized to cut total time. This is illustrated in Figure 2.

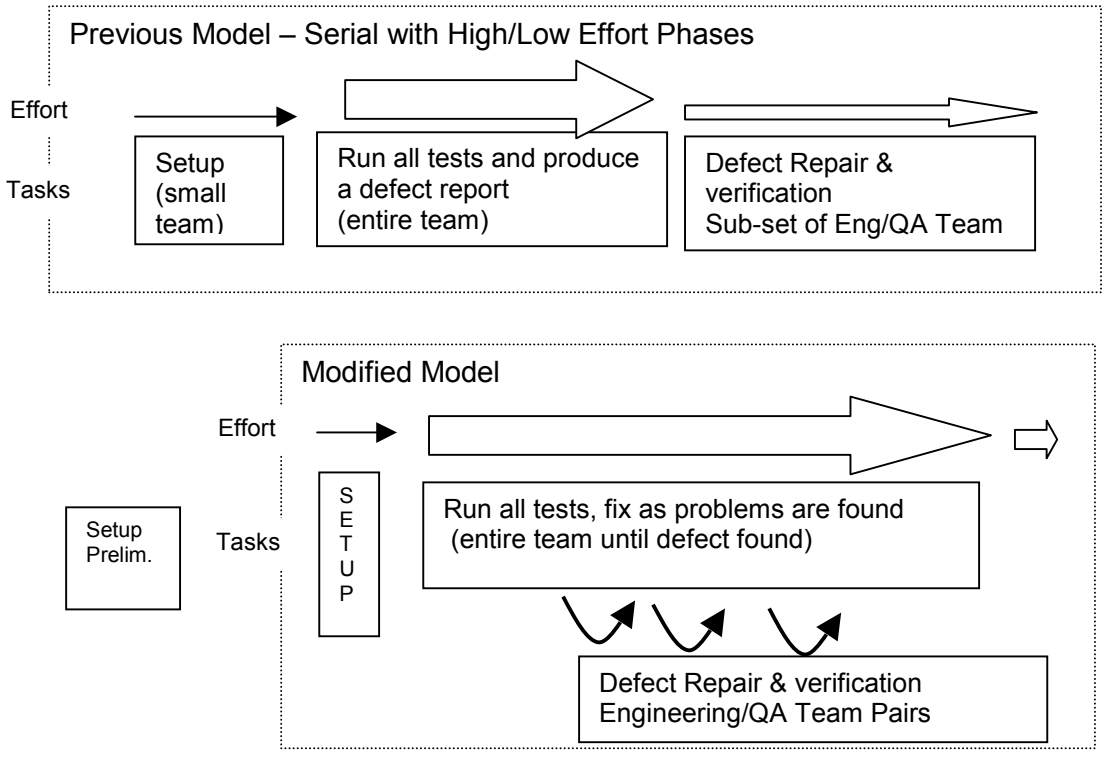


Fig. 2 Test Pass Workflow – Before & After

Comparing the new model against the old model makes two things evident. First, part of the preparation time for the test pass was moved out of the critical path. This included some of the database setup, database upgrade testing and debugging, and install testing and debugging. These tasks were all done well prior to the needed time, debugged, and automated. They became turn-key efforts when needed. Second, instead of running through a full test pass and then producing a report of defects to fix at the end, direct repair of any found defect was done. For each defect, an engineer and QA pairing was made to define the fix, implement it, and verify it was correct. As the overall test pass was made on a specific build that was left unchanged during the pass. Verification of defects was done on spot builds or nightly builds before the code was checked in.

Monitoring Logician 5.5

A successful release required monitoring. This was done through weekly checks on schedule. Each element of the model was examined weekly and corrections were made as assumptions proved true or false. At any given week, the model would always give an estimated delivery date for the release which was the best given the information at that time. We made management aware of the projections even when the model showed later than desired delivery, and continue to work on improving the projected outcome. This allowed us to tackle the problems early when the maximum schedule leverage is available.

It is true that some work environments won't let people admit the truth about schedule status. In these environments, everyone has to stick to the fiction that the release will be on time and there are no problems. In general, it is better to get problems out early and address them rather than sweep them under the rug.

Results from Logician 5.5

Overall, the release was a huge success. The product shipped on the day desired, July 31, 2001. Even more remarkable was that the last month prior to release was relatively calm and regular instead of high panic often associated with pending releases given that so much was at stake.

Quality improved with the release as well. Four months after the release, our customer support group issued a release report based on customer calls and problems identified. The number of post release defects reported in the first 4 months after release were half the number that had been reported in the 5.4 Logician release. Customer sentiment was that the quality had improved as well. This substantiated the common feeling of development team members that this was the highest quality release they had participated in.

Quality Standard

The Challenge

The QA team was assigned a goal in March of 2001: to establish a quality standard that could be applied to all of the software product lines. The quality standard had to:

- Provide a consistent scoring method for all products
- Be relatively easy to calculate
- Provide an ongoing evaluation of the product quality while it was still in development
- Make sense to the existing development team
- Affect both developer and manager behavior in a positive way

In addition, the standard was to be targeted at several audiences: programmers, QA staff, managers, and senior management.

Creating a standard required reviewing existing standards in place. As things currently stood, the decisions to repair a defect was made by a variety of different people from different teams: Product Marketing, QA, Engineers, Engineering Management, Project Managers, and sometimes Senior Management.. A management team determined readiness to release. This method presented several problems: it couldn't be compared across projects; it depended too much on the teams reviewing readiness and their view of quality vs. schedule; and it didn't predict quality.

The existing defect ranking method had several problems. Defect ranking was based on multiple scales making it difficult to compare defects except through discussing each one. Information from the evaluation discussion wasn't stored in the defect-tracking database. Instead, it was stored separately in meeting notes. Finally, rankings couldn't be compared well between releases.

Building a Standard

The development of the standard started by collecting ideas from the team on what the standard should look like. Several different discussion meetings were held to generate ideas. The steps were: first focus on getting a common understanding of the goal, list as many different approaches as possible, compare and rank the different proposals, then go into detailed discussion on the highest ranked proposals.

A variety of different methods were considered in several discussion meetings. It was clear that a defect-based method was part of the desired solution, because defects are familiar, measurable, and quantifiable. The major problem with a defect-based solution is that it only can be measure after the product is built. Typically, this is the last 10-20% of the project life cycle. In addition, there needed to be a good method of calculating an overall *number* that that would be meaningful. This *number* is referred to in this paper as *defect score:* a single decimal number that sums up the defectiveness of the product at a point in time.

To supplement the defect measure, it was necessary to choose another measure that could be quantified earlier in the development cycle. A subjective measure had several advantages over a pure defect based measure. The biggest was that opinions could be determined well before defect based scores would show any problem. A number based on defect counts would only show a problem during the middle of testing, not early in the development. Solving major quality problems that late in the development cycle would result in shipping delays. On the other hand, a subjective measure can be evaluated right from the start of the project.

Figure 3 illustrates the relative times that each measure is active.

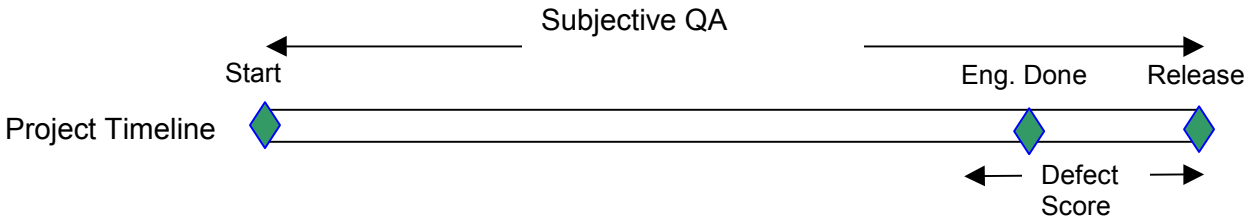


Fig. 3, Applicable Times for Quality Measures

The next section explains how each measure was implemented.

Subjective Measure

The subjective measure needed to be quantified. A simple 1-5 numerical ranking scheme was chosen. A 5 would represent and outstanding effort, 3 the minimum allowable level for shipment, and 1 total unacceptable. Table 1 shows definitions for each level.

Score	Quality Summary	Description
5	Outstanding	Product: Customer will be very pleased with end result. Development Practice: Best Practices
4	Good	Product: Customer will be satisfied with end result. Development Practice: Good
3	Min Acceptable	Product: Just meets customer needs/expectations and may have a number of annoying defects and features. Development Practice: Fair/Poor
2	Poor – below min. acceptable	Product: Key functionality can be made to work, but defects and poor definition mar the product, making it undesirable. Development Practice: Poor/Ad-hoc
1	Unacceptable	Product: Product does not perform a useful function or is too defect ridden for the customer to care. Development Practice: Unprofessional, Ad-hoc

Table 1, Subjective Ranking Scale

Scores could be reported in 0.5 increments. In addition to ranking the product, individuals would be expected to write a comment explaining any overall ranking of 3 or less. The subject score would be the average value of the staff. The minimum and maximum values would also be reported. Lowest scores on a project often were indicators of a specific problem. Any score 3 or less required the submitter to supply comments indicating what the specific deficiencies in the project were.

Initially, more detailed information was collected about sub-categories of performance on the project. These categories were: Overall Process, Schedule Vs Resources Assigned, Requirements Quality, Projected Customer View of Resulting Quality, Amount of Late Changes, and Perceived Code Quality. These proved to be time consuming to evaluate, collect, and analyze, so they were later dropped. It was sufficient that QA engineers reviewed the project in each of the areas, gave an overall score, and then supplied specific comments as in what areas they felt were deficient.

To simplify the entry, a web page was created with a simple form with pull down menu items. A user had to select their name, their project, assign the ranking, and then fill out a comment box. The user clicked the “Submit” button, which mailed it direct to the QA manager for accumulation.

Requests for information were made every two weeks. Sample plots illustrating 2 months of scores are shown in Figure 4.

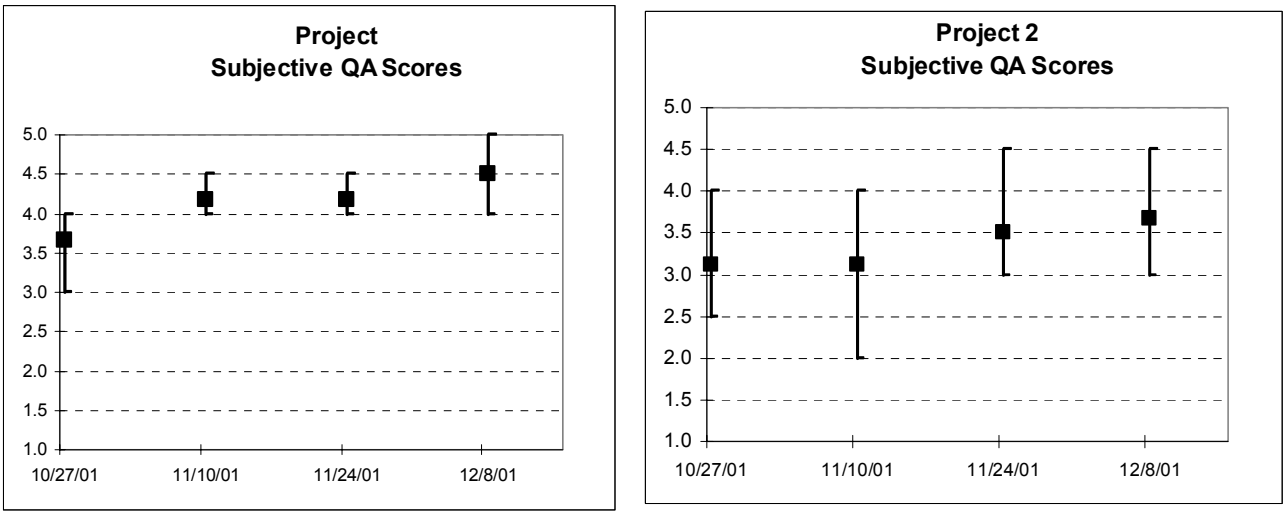


Fig. 4, Plots of Subjective Scores during Development

Some overall lessons came out of this effort. First, some of the QA staff found the continuous feedback to management to be empowering. Having the chance to describe problems other than defects to all development management improved morale. On the other hand, some programmers and managers felt that it was an unfair treatment to solicit feedback from one group. It was agreed to expand the group reporting on quality problems. Second, in the process of pointing out problems, some of the feedback could be interpreted as attacks on groups. The best solution was for QA management to provide summaries of issues to the other managers, rather than the specific comment. Third, the trends in the data and the comments proved to be the most valuable aspects of the system.

Overall, the system worked well as it provided several different levels of management regular feedback on progress while there was still time to solve problems.

Defect Ranking & Defect Score

A method of calculating a defect overall score and acceptable values for release presented several problems. First, previously multiple ranking methods were being used on a single defect in some cases: how bad is it broken, what is the harm, and what is the priority. This made it difficult to understand how to compare different defects. Second, there was no consistent method of putting each individual defect into each category. Different reviewers would give different ratings, often driven by their personal desire to see that defect fixed. Third, there was no overall release criteria or cut line for defects. It was often the subjective judgement of QA, Engineers, and Engineering managers on a case-by-case basis whether to fix a defect or not. This lead to a huge time commitment and many earlier decisions being questioned later.

Ultimately, every defect-rating scheme boils down to two major pieces: A classification method for individual defects and a formula to sum the defects into a value. Building a classification method for individual defects where each defect resolves into a single number works best. It is easier to classify defects, easier to sum the total, and easier to make comparisons between different projects and times.

The first problem was to set a method by which each defect was ranked. A single ranking criterion was used, ranking each defect with a single number from priority 1-5 (P1-P5) showing its overall priority. One complication was the many different types of factors that would lead one to prioritize a defect high -- examples being security issues or data problems. The solution was to build up a table with columns for each priority ranking and rows for each category of concern to be considered. Individual boxes would describe the conditions that set the ranking in that category row.

It is important when setting up categories to decided on relative weightings between each ranking. One could set conditions for P1 defects as 2x a P2 or some other multiple of the core condition. It is probably easier to assign absolute meanings to each priority category and then work back to the descriptions from there. For example, we chose P1 & P2 categories as "do not ship with these" categories. P3 items were acceptable for shipment, but must be kept to a minimum. P4 items were undesirable but not critical problems and P5 were small unimportant errors.

A simplified table that illustrates the method is shown below. Normally, each box in the table has a text description which indicates the condition that triggers the column's priority ranking. In this table, X's are shown instead of the full description. The text description may include several conditions that may have to be met to get a particular priority assigned.

Priority ->	**P1 if any below**	**P2 if any below**	**P3 if any below**	**P4 if any below**	**P5 if any below**
Safety	X	X			
Security	X	X			
Legal Liability	X	X			
Data Integrity	X	X	X		
Revenue	X	X	X		
Updates	X	X	X	X	
Functionality Broken	X	X	X	X	X
Functionality as specified & built, affects usefulness:	X	X	X	X	X

Table 2, Defect Priority Ranking by Category

It was noted that different size efforts (in development time) generated different amounts of opportunities to create defects. Consequently, the relative ranking of mid-level defects would depend on some measure of the opportunities to create those defects. Using lines of code wouldn't work because some of the development effort took weeks and reduced the amount of code in a section. Ultimately, we decided to normalize based on the number of programmer weeks from start to hand-off to QA. A number of allowable defects per weeks of effort was determined. This in turn led to a normalizing factor "n" to be used on lower level defects.

One more item was need to make a formula. It needed to incorporate the "must-do" defects in it to show current status during debugging. To do this the P1 defects were assigned a value of 2 and the P2 defects a value of 1. The resulting formula was
Defect Score = P1*2 + P2*1 + P3/n
where P1, P2, P3 are the total counts of open defects in each priority category. The resulting defect score is a number that represents "the number of reasons not to ship the product". P1's count as 2, P2's as 1, and "n" P3's as 1.

The defect count data was relatively easy to extract from our defect-tracking program on a daily basis. This information could be quickly turned into a graph showing the overall "defect score"

and trend line for each product being tested and defects were repaired. It became a useful management tool to track problems and our progress on defect discovery. A second graph using a similar scoring was added to track an overall score for defect verification.

One item was contentious: It had been the practice in the past to put defects into a deferred category and then ignore them for the release. However, deferring repair of a defect does nothing to change the product's quality. Consequently, this score included "deferred" defects in the overall ranking. Therefore, P1 & P2 defects couldn't be deferred. P3 defects could be deferred up to the "n" limit. A sample plot showing defect scores on two different projects is shown in Figure 5.

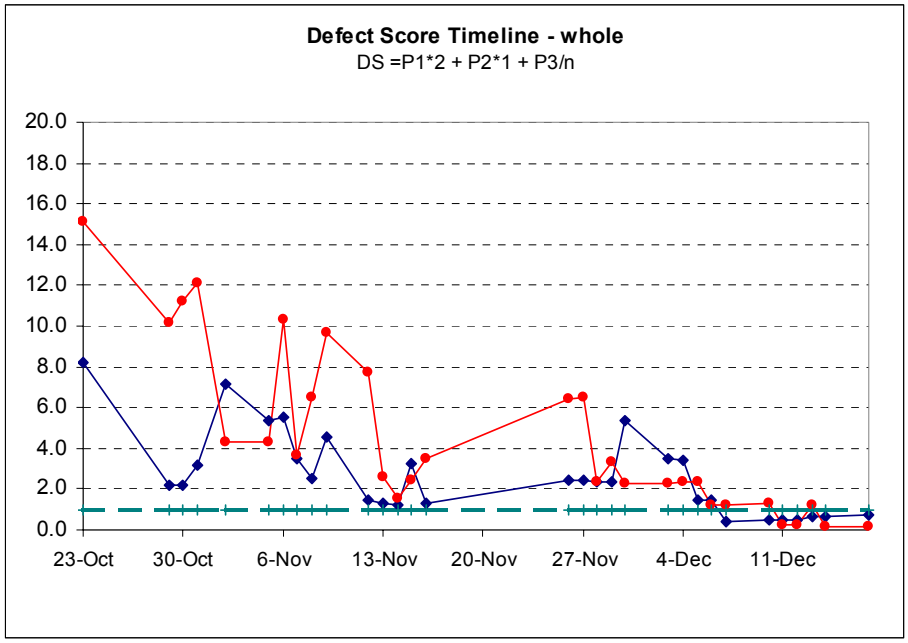


Fig. 5, Defect Score Plots for Multiple Projects

Overall, the defect score worked well. It provided a tracking method for identifying the current state of the project as well as set a goal that needed to be achieved prior to shipment.

Conclusions

Overall, the techniques used during 2001 to improve quality on the Logician 5.5 release and to set quality standards were successful. Many of these techniques can be generally applied to other work situations, as they are relatively simple to implement and direct the organization toward a quality goal. Of course, many of the best methods for improving quality require full support of senior management and especially the engineering management as they require basic changes in the way work gets done. However, the methods described are good first steps for organizations.

The goal of this paper was to show practical methods one could use to measure quality, improve schedules, and improve quality overall. Our discovery is that application of quality improvement techniques matched to "insane" situations can result in great outcomes: improved quality, schedule, and morale. In fact, that is what makes it all worthwhile.

References

1. Preseton G. Smith, Donald G. Reinertsen, *Developing Products in Half the Time, New Rules, New tools*, Second Edition, Wiley, 1998

2. Robert G. Cooper, *Winning at New Products*, Perseus Publishing, 2001

3. Steve McConnell, *Code Complete*, Microsoft Press, 1993

4. Tom DeMarco, *Controlling Software Projects*, Yourdon Press, 1982

5. Tom DeMarco, *Slack*, Broadway, 2001

6. Tom DeMarco & Timothy Listner, *Peopleware*, Dorset Publishing House, 1987

7. Tom DeMarco, *The Deadline*, Dorset Publishing House, 1987

8. Jim McCarthy, *Dynamics of Software Development*, Microsoft Press, 1995

9. Brederick P. Brooks, Jr., *The Mythical Man-Month, Anniversary Edition*, Addison-Wesley, 1995

10. Caper Jones, *Estimating Software Costs*, McGraw-Hill Professional Publishing, 1998

But Will It Work For Me?

Kathy Iberle
Hewlett-Packard
18110 SE 34th Street
Vancouver, WA 98683
Kathy_Iberle@hp.com
www.kiberle.com

Abstract

This article defines software engineering "practice cultures" as groups of people who hold shared assumptions about their business model and situation. These shared assumptions drive their choices of software engineering practices, often in ways that are not obvious. The author visits a number of practice cultures and explores the relationship between the software engineering practices favored by that culture and situational factors such as project size, cost of fixing post-release errors, regulation (or lack of), and economic drivers. An understanding of that relationship is important when evaluating whether a practice will work in a new environment.

Biography

Kathy Iberle is a senior software test engineer at Hewlett-Packard. Over the past 17 years, she has been involved in software development and testing for products ranging from data management systems for medical test results to inkjet printer drivers and, most recently, e-services. Kathy has worked extensively on researching appropriate development and test methodologies for different situations, training test staff, and developing processes and templates for effective and efficient software testing.

This paper was written for and presented at the 2002 Pacific Northwest Software Quality Conference.

Acknowledgements

My thanks to the participants of the Software Test Managers Roundtable (STMR), whose stories from a wide range of software development environments contributed substantially to this article. Facilities and other support for STMR are provided by Software Quality Engineering, which hosts these meetings in conjunction with the STAR conferences.

Special thanks to the many reviewers who provided informative and thought-provoking comments: James Bach, Bernie Berger, Bob Hollister, Barney Levie, Jonathan Morris, Carol Peterman, Bill Tuccio, and the reviewers from the Software Architecture Group at University of Illinois in Champagne-Urbana: Ralph Johnson, Brian Marick, John Brant, Bill Davis, and Weerasak Witthawaskul.

Why Don't We All Do the Same Thing?

The libraries and bookstores are overflowing with papers and books on software engineering practices. There's thirty years or more of experience behind us – so why haven't we discovered the one right way of doing software? Why do we still use different practices?

Because we're different! Far too often, I hear this when someone simply doesn't want to use the practice in question, or is convinced that she won't be allowed to use it. But in truth, we really are different.

In any flock of software engineers, different groupings can be identified. The members of a group share some common assumptions – assumptions about the business, assumptions about goals, assumptions about users, assumptions about risk. A group of practitioners who share a set of assumptions can be said to belong to the same *practice culture*, even if they don't know each other.[1] People working in the same culture tend to read the same journals, use the same language, attend the same conferences, and use the same practices. Some individuals are completely unaware of the existence of any other practice cultures, and others are completely convinced of the innate superiority of their own practice culture. I often see people coalesce into cultural groupings during lunch at large conferences – the aerospace and medical folks cluster together, happily comparing test plan formats and discussing reliability measurements, while the shrinkwrap software crowd is in another corner talking about agile methods and rapid testing.

Amazingly enough in this imperfect world, the different software engineering practice cultures actually work pretty well. Over time, practice cultures develop methodologies and practices that fit their situation (Bach 1999). They also tend to develop mythologies to explain the "rightness" of their practices. As with other human cultures, there is a strong tendency to believe that all people should recognize the superiority of one's culture and immediately rush to join it. There is also a tendency to believe that everyone already belongs to one's culture, and to be puzzled and hurt when members of another culture do not behave in the time-honored traditions of yours.

Books, journals, magazines, and methodologies are based on implicit, often unspoken assumptions about the situation experienced by the authors. The stated practices usually do work in those situations – but the question remains as to whether the practices will transfer successfully to your situation. If you identify the originating practice culture and understand its assumptions, you can then evaluate whether the assumptions on which the practice is based are likely to be true in your situation also. Recognizing your own practice culture and its differences from other practice cultures will help you:

- learn from other practice cultures
- work with members of other practice cultures
- evaluate new practices for effectiveness in your own situation
- move into a new job in another practice culture

[1] This concept was identified as a "community of practice" by James Bach , (Bach 1999). The term has also been adopted by the knowledge management community with a somewhat different meaning, so I have chosen to use the term "practice culture" in this article, as suggested by Mr. Bach.

Assumptions Behind the Practices

A practice culture is the collective creation of a group of people. It represents the group's common beliefs and practices. Most individuals live in one practice culture and have learned the beliefs and practices of that particular culture. Individuals who move between cultures generally carry their original culture with them and acculturate gradually, often with some degree of "culture shock" during which the practices of the new culture seem very foreign and unreasonable. I would expect that individuals such as contractors who switch back and forth frequently between cultures don't change their practices with every new job, but instead either stick with their original practices or create a cross-cultural amalgam of practices.

When looking at a practice culture, there are several things to consider:

- What assumptions does this practice culture make?
- How do those assumptions lead to "best practices" – that is, practices that are favored within that practice culture?
- Are there "best practices" that don't appear to follow logically from the assumptions? This suggests that there are assumptions unknown to you.
- Do any of the assumptions involve factors that have changed in the last few years?

The assumptions behind a practice are difficult to find because they are usually not listed neatly and explicitly with the practice description. Sometimes assumptions are stated as beliefs or even as incontrovertible truths, rather than assumptions. Other assumptions are left unspoken, because they are so obvious to the people in that culture. This is a trap for the unwary - even smart people from other cultures can miss the point completely.

For instance, if an inch or two of snowfall is predicted, many inhabitants of Seattle plan to stay home for the day. Coming from Michigan, I thought this behavior unnecessarily timid if not downright silly – until it actually did snow. The next morning, I skated downhill for less than a block before deciding to preserve my car and myself by prudently remaining at home. Two of my assumptions about driving in snow had turned out to be incorrect – Seattle has a lot of hills and the roads are not salted. Even though I had walked up and down those hills for months and noticed the lack of rust on the numerous older cars, it hadn't occurred to me that together this would add up to dangerously slick roads.

Assumptions typically are about things other than technology – for instance, James Bach (Bach 1999) lists capabilities, goals, and situation. A discussion of when a practice will work (or not work) usually names at least one and often several of the author's assumptions. For instance, Highsmith discusses a couple of classification methods for criticality when discussing agile methods, saying that regulatory requirements may preclude the use of some agile methods for life-critical software. (Highsmith 2002).

The Major Practice Cultures

So what are the major practice cultures? There are a number of different ways to divide the software engineering community into cultures. A variety of factors for being different are cited by the authors or critics of various practices – project size, customer type, risks incurred, number of regulations that affect the product, and so forth. For this paper, I chose to sort first by *business model* – that is, how do people profit from writing this software? This didn't result in clear groups with no overlaps (as you'll see later) but I am using this division for several reasons:

- The way in which money flows and what profit depends upon appears to either drive or constrain quite a few of the other factors.
- The effects of the business model on practices are frequently overlooked completely by practitioners
- I didn't see a better sorting rule that accounted for all the differences I could see.

Sorting by business model is not completely unprecedented - James Bach's list of practice cultures from his article "Good Practice Hunting" (Bach 1999) is very similar.

Business Model

The primary business models currently appear to be:

- **Custom systems written on contract:** We make money by selling our services to write other people's software.
- **Custom systems written in-house:** We make software to make our own business run better.
- **Commercial products:** We make money by writing and selling software to other businesses.
- **Mass-market software:** We make money by writing and selling software to consumer.s
- **Commercial and mass-market firmware:** We make money by selling objects which happen to require embedded software.
- **Open-Source:** We make software for our personal satisfaction. (This doesn't include all open-source software - some is written in a for-profit environment and later "freed", and some comes from academic projects.)
- **Academic:** We make software to use in our research – in biology, chemistry, physics, psychology, economics, and many other fields as well as computer science.
- **Internet**: There are multiple business models using Internet technology now, and a single practice culture has not yet formed. More on this later….

The underlying economic assumptions in different practice cultures are rarely stated in articles or books about practices. When I switched practice cultures myself, it took a while to realize that the reasons behind the practice differences very often traced back to economics. My new colleagues would mention the economic factors in passing, but often didn't explain the connections between the economics and the practices.

There are, of course, ways to make money and reasons to write software that don't fit neatly into one of the above categories. If a particular combination grows significantly, eventually there will be a critical mass and we see one (or more) new practice cultures arise. For instance, the mass-market practice culture didn't yet exist in 1980, whereas the custom systems practice culture certainly did. Existing practice cultures also grow and change over time, such that a description of the custom systems practice culture today will not be the same as it was in 1980.

Situational Factors

Each business model has a set of attributes or situational factors associated with it. Assumptions about situational factors are more likely than the business model to be stated in a process or practice description. Situational factors are often derived from or limited by the business model – both the way the money flows, and the end purpose of the software itself. For instance, software meant for consumer purchase is unlikely to be life-critical.

Here's a list of situational factors that seem to influence the choice of software engineering practices:

- **Criticality:** The potential for harming the user's or purchaser's interests varies depending on the type of product. Some software can kill you when it fails, other software can lose large sums of many people's money, and yet other software can do nothing worse than waste the user's time.
- **Uncertainty of Users' Wants and Needs**: The requirements for software that implements a known business process (such as the U.S. tax code) are necessarily better known than the requirements for a consumer product that is so new the end users don't even know that they want it.
- **Range of Environments:** Software written to use in a specific company needs to be compatible only with that company's officially supported computing environments, whereas software sold to the mass market has to work with a wide range of environments. The skill sets of users in the mass market also vary more widely than those of users in a specific company.
- **Cost of Fixing Errors:** Distributing firmware fixes is a lot more expensive than patching a single website.
- **Regulation:** Regulatory agencies and contract terms can require practices that might not otherwise be adopted, or may set up a situation in which certain practices become useful. Some situations require process audits, which verify that a certain process was followed to make the product. (The term "audit" is also used in some fields for an activity which verifies that the end product works as intended, which is not the same as a process audit.)

- **Project Size:** Multi-year projects with hundreds of developers are common in some businesses, whereas shorter single-team projects are more typical in other businesses.
- **Communication:** There are a number of factors in addition to project size that can increase the amount of person-to-person communication needed, or make accurate communication more difficult. Some of the factors seem to show up more frequently in certain practice cultures, whereas others appear to be randomly distributed.
    - **Concurrent Developer-Developer Communication:** Communication with other people on the same project is affected by the way the work is distributed. In some organizations, senior staff design the software and junior staff do the actual coding and unit testing (as opposed to having the same individual design, code, and unit test a given component). This practice increases the amount of developer-developer communication needed. Specialization can also cause communication difficulties when taken to the point where specialists start assuming knowledge on the part of others or are themselves missing a piece of the picture.
    - **Forward Developer-Developer Communication:** Maintenance and enhancements require communication with developers forward in time. This is easiest when the developers tend to stick around, and thus the communication is with yourself.
    - **Developer-Management Communication**: Project status needs to be reported upwards, but the amount and form of communication that managers believe they need varies rather considerably.
- **Organizational Culture:** The organization itself will have a culture that defines how people operate. Highsmith (Highsmith 2002) summarizes work by Moore[2] that defines four organizational cultures:
    - **Control –** "Control cultures, like IBM and GE, are motivated by the need for power and security."
    - **Competence –** "A competence culture is driven by the need for achievement: Microsoft is an obvious example."
    - **Collaboration –** "Collaboration cultures, epitomized by Hewlett-Packard, are driven by a need for affiliation."
    - **Cultivation** – "A cultivation culture motivates by self-actualization … and can be illustrated by Silicon Valley start-up companies."

  Organizational culture affects communication practices quite strongly, possibly more strongly than the practice cultures themselves. For instance, a friend of mine who was an in-house software designer for a control-culture auto manufacturer complained one day of the time she spent explaining the difference between 0 and O to one of the "coders" – a person with a few months of training in COBOL and no other background. My reaction "this is weird" probably stems from many years in a collaboration culture, rather than from differences between the in-house software business and the commercial software business. Organizational culture can also drive a practice culture to adopt or cling to practices that are not necessarily a good fit for the rest of the situation - Highsmith discusses the problems of adopting agile practices within a control culture. (Highsmith 2002).

It's important to keep in mind that people's beliefs about the situational factors of their practice cultures are assumptions, which may or may not match the current facts. It's very hard for any one person to have enough data to establish facts. Most assumptions are generalizations of currently true situations, but some represent situations that used to be true but have changed. Assumptions can have a lot of staying power, especially if they get into widely read literature.

It surprises some people to find more than one practice culture within a single company, yet this appears to be not uncommon in companies with a broad variety of businesses. Hewlett-Packard, with its history of autonomous business units, has developed distinctly different practice cultures in commercial software and mass-market software to suit the situational differences between these areas. Microsoft's work, while not covering such a broad range as Hewlett-Packard's, still shows some differences in practice: "Applications like Word or Excel, or consumer products like Works or Complete Baseball, tend to have shorter schedules, smaller teams, and more precise estimates of delivery dates than do system products such as operating systems…. The vision statement and specification document for systems products are more complete and detailed earlier in the lifecycle … Systems products also tend to have longer testing periods." (Cusumano 1995)

[2] The cultural models are based on work in Schneider (1994).

A Visit to Some Practice Cultures

Let's take a look at some of the major practice cultures in the United States. For each one, we'll first look at the business model in more detail, and then explore the situational factors. But wait – are these situational factors fact or fiction? How far can you trust me?

The statements about the situational factors are based on one or more of these sources:

- My own personal experience within Hewlett-Packard: this covers commercial software (mechanical engineering instruments – now part of Agilent; medical products – now part of Siemens; business inkjet printers), mass-market software (consumer inkjet printers), and both commercial and mass-market e-services.
- The experiences of people within Hewlett-Packard with whom I worked closely for an extended period of time: this covers all of the above plus both commercial and mass-market firmware.
- Stories gathered from conference presentations and books by people from a variety of practice cultures.
- Conversations with people from a variety of cultures, many of which occurred at the Software Test Managers Roundtable meetings.
- Stories from the various people who reviewed this paper.

The practice cultures in which I've had personal experience are probably more accurately described, particularly since I've been comparing notes with other practitioners in the same fields for some years. The ones in which I haven't had experience are probably less accurate. With that caveat in mind, let's go exploring…

Custom Systems Written on Contract

Business Model

An entity decides to hire another company to write a custom software system, or to customize and install a system provided by that second company or purchased from a third company. A contract is drawn up to capture the agreement for delivering the system. The profit for the company providing the services comes from staying within the allotted budget and avoiding penalties for late delivery. Normally, the contract requires a detailed set of specifications for the work to be done. Once the customer has signed off on those specifications, the contractor can charge extra for every change request thereafter.

Typical software written or substantially customized under contract for a specific customer includes banking networks, insurance software, billing and payroll systems, and a wide variety of military applications such as missile guidance systems.

Contract software is a very big business in the U.S. The two largest and longest-running purchasers of custom software appear to be finance companies and the U.S. military. Finance and military people have been using computers since the 1940s, and actively discussing what we would today call "software engineering" practices since at least the 1970s, if not earlier. In fact, the very earliest software was all custom-written for a specific use, so this business model has a great deal of history behind it. The history, coupled with the large number of people involved, has resulted in a huge amount of software engineering literature. A substantive portion of this literature is written by companies such as IBM that have historically provided both one-to-a-customer systems and customizable systems meant for more than one customer.

Software that was written under contract for the U.S. Department of Defense (DoD) ended up in this category along with software written under other types of contracts because I didn't find major differences in either practices or assumptions between software written for the DoD and for the rest of the contract world – in fact, it was clear that the other contractors had adopted many practices that originated in DoD work. It does appear that the DoD contracts are more detailed and prescriptive than the non-DoD contracts, but that didn't drive radically different practices as far as I could tell. However, I didn't get much information about DoD work from actual practitioners, so I may be missing something important in lumping DoD work in with other contract work.

The Situation

- **Criticality:** Software failures in finance systems can seriously damage the purchaser's business interests. Software failures in military systems can be life-critical, although quite a bit of software purchased by the military is actually business software for which the consequences of failure are similar to the consequences of failures in finance systems.
- **Uncertainty of Users' Wants and Needs:** Since the purchasers and users are an identifiable group of people, they can be located to find out what they want. In general, they do have a pretty detailed idea of what they want – nobody is interested in your creative re-interpretation of their payroll system. On the other hand, the process to be implemented isn't always well-documented, the users may disagree on steps of the process, their requests may assume technology that doesn't exist, business needs change during the project, and sometimes people just plain change their minds. Given all this, it's hard to understand why the literature in this practice culture is so emphatic about the need to establish all requirements up front – it may be that the contracting process itself forces people to believe that requirements should not change, although conversely the contracting process might be based on experiences from a time when requirements didn't change as much.
- **Range of Environments:** The purchasing organization has usually decreed a small set of target environments to hold down their costs, will offer or require user training if needed, and the users themselves are not such a varied lot as the general populace. This results in a range of environments that is both clearly defined and relatively narrow compared to the other practice cultures.
- **Cost of Fixing Errors:** Generally there are reasonably priced ways to distribute fixes – much of the software will be on servers in an identifiable building, and the location of client software is usually tracked.
- **Regulation:** Software for the Department of Defense must be written in accordance with a huge list of regulations, most concerning the process of producing the software. Finance software is not subject to regulation in the same way, although contracts vary. Not infrequently the contract requires process audits to prove that the organization followed its stated processes. The customer also often expects regular status checks.
- **Project Size:** Big seems to be the operative word here. Really big. Many dozens of people over two years seems merely average, whereas the biggest have hundreds of people over multiple years. There is some data suggesting that smaller projects are really much more prevalent than larger projects (Highsmith 2002), but the literature definitely reflects larger projects. This may be an example where the situation has changed but the assumptions about it have not, or it may be that funding for software engineering research is concentrated on large projects, and the publications reflect that.
- **Communication:** The practice of splitting design and coding between senior and junior staff shows up sometimes in this practice culture. Since the systems and projects are large, there are often separate people or even separate departments for analysis, database design, and so forth. In addition, maintenance contracts can go to people other than the original authors, so both concurrent and forward communications can be complicated. The companies themselves are often large, whether or not the projects themselves are large, which adds extra layers of status reporting.
- **Organizational Culture:** The companies that write software on contract often have a control culture. This seems logical since many of them have ties to the military.

These situational factors and capabilities together lead logically to a set of assumptions.

- **Delivery on time and within budget is very, very important**.
- **Software that reliably delivers correct output is very, very important.**
- **The requirements can and should be known in detail up front**.
- **Projects will be large and communication paths complex.**
- **We must be able to prove that we did what we promised.**
- **We need plans and regular status reports flowing upwards.**

Favorite Practices

Now let's explore how some practices popular in this practice culture make sense, given the situational factors and the assumptions.

Lots of documentation. Documentation is valuable as a communication medium when *project sizes are large*. Written documentation is often more effective than whiteboard or hallway chat when *the communications paths are complex*, which happens when people are widely separated by geography or job. In addition, some documentation is often required to *prove that we did what we promised*, and *requirements known in detail up front* also implies documentation of those requirements.

Capability Maturity Model (CMM). The CMM is a product of the Software Engineering Institute (SEI), which is funded by the U.S. Department of Defense (DoD). The SEI's stated mission is "to provide the technical leadership to advance the practice of software engineering so the DoD can acquire and sustain its software-intensive systems with predictable and improved cost, schedule, and quality". (SEI 2002) The CMM emphasis on careful estimation and project and detailed project management is completely in line with the assumption that one must be *on time and within budget* and with the assumption that *plans and regular status reports* are necessary. Large companies and *large projects* are both assumed by much of the CMM literature, such as Watts Humphrey's classic Managing the Software Process. (Humphrey 1989)

The CMM is also very popular with contractors in other industries because, just like defense contractors, they also need to be *on time and within budget*. In addition, CMM certification has become a selling point for many software contractors, so CMM practices may be adopted in order to get certification.

Waterfall Lifecycle. The waterfall lifecycle was originally invented to give *large projects* enough structure to be able to plan and steer towards *on-time delivery*. Newer "iterative" methods, such as Staged Delivery, plan in smaller chunks, which generally fulfills the need for planning while allowing more flexibility. However, it is easier to coordinate multiple *large projects* using the simpler structure of a waterfall lifecycle. Sometimes you find that the official high-level schedule appears to be a waterfall lifecycle for coordination purposes, and the actual activity on the team level is more iterative.

Formal Requirements and Analysis, Unified Modeling Language, Rational Unified Process: These practices stem from a combination of the complexity of the software, the need for *correct output*, and the practice of having multiple departments implementing different areas resulting in *complex communication paths*. Sophisticated diagramming processes to aid communication make sense here.

Process Audits. Extensive auditing procedures make sense when you must be able to *prove that you did what you promised*.

Rapid Application Development (RAD) and Joint Application Development (JAD). These "rapid" practices were developed as an alternative to the time-consuming practice of getting users and developers to agree on *requirements in detail up front* via an exchange of written documents, with all the concomitant misunderstandings and rework. These practices are consistent with the assumption that the requirements can and should be known up front – they are simply a different approach to getting there.

Cleanroom: Cleanroom is a process pioneered at IBM and supported and used by the Department of Defense and many of its contractors. The aim of cleanroom is "the development of high quality software with certified reliability" (SET 2000), leading to *reliable delivery of correct output.*

Custom Systems Written In-House

Business Model

In-house software is written when a need or opportunity to improve the bottom line of the company can be demonstrated. There are a variety of ways to fund projects – the larger projects require bigger decisions so they are

funded in a more formal way with a need for up-front estimates in detail, whereas smaller projects may not require as much justification. There are rewards for staying within the allotted departmental budget, but overruns are not fatal in the same way that they are in the contractual world. There are some projects that have to meet a deadline (like Y2K), and some projects that can be put on hold if the department runs out of money. Projects may be split up into sub-projects and approved separately over a period of time.

The typical software written in-house includes such things as order-tracking and online timecards.

The Situation

- **Criticality:** Most of the software is essential to the company's business purposes, although there will be occasional experimental projects of less criticality. In an organization whose business is something other than software, the criticality of the software may be underestimated by the people setting the budget.
- **Uncertainty of Users' Wants and Needs**: Many of the projects done in-house are very similar to those contracted out, so logically the level of uncertainty should be much the same as that of the previous practice culture. However, the discussion above pointed out that the traditional assumption that requirements should be stable may be based on economic factors (i.e. the way a contract works) rather than actual stability. Whether or not that is true, the "agile" practitioners in this practice culture are vocally asserting that the requirements in fact are uncertain and do change rapidly. The in-house developers have the opportunity to dramatically reduce uncertainty by talking directly to the end users, an opportunity often not available to developers in other cultures.
- **Range of Environments:** Again, the range of environments is limited by the company's own rules about its people's environments.
- **Cost of Fixing Errors:** The responsible department generally has easy access to the software, although distribution of fixes to a large number of clients can be problematic. If the functionality is centralized on a small number of servers, it can be quite inexpensive to release fixes.
- **Regulation:** Unlike the software provided on contract, generally neither regulatory issues nor process audits play a big role.
- **Project Size:** The size ranges from small to quite substantial.
- **Communication Between Developers:** As far as I have seen, the same people generally do design and coding. However, specialized staff for analysis, database design, and so forth seems typical in this business. Status reporting is of particular importance in this culture because the project budgets are often "overhead" in the larger company picture, and so projects often need to be repeatedly justified.
- **Organizational Culture:** Varies.

Again, the situational factors and capabilities together lead to a set of assumptions that are very similar to those of the contract world, with a few key differences (underlined):

- **Delivery within budget is very, very important, <u>but delivery on time may be less so.</u>**
- **Software that reliably delivers correct output is very, very important.**
- **The requirements can and should be known in detail up front**.
- **<u>The requirements change rapidly and can't be known in detail up front</u>.**
- **Projects can be large and communication paths complex.**
- **<u>We don't have to prove that we did what we promised.</u>**
- **We do need to track costs and results to help demonstrate that what we are doing is cost-effective.**

Favorite Practices

Typically, in-house software development groups use practices not dissimilar to the methods used in contract groups. This may be more of a historical effect than driven by their particular needs.

> **Lighter-weight Documentation:** In-house groups often produce less documentation and have fewer required documents. There's *no need to prove that a contract was fulfilled*, no auditors to satisfy, and the

staff doesn't turn over as fast as that of contracted software so it is possible to carry information in people's heads. However, the *larger projects* still require a certain amount of documentation.

CMM: The in-house groups are less likely to go after formal CMM certification since they aren't trying to sell themselves, but appear to be using CMM methods frequently from what I've seen, often to attain *delivery within budget* and to *track costs and results*.

Formal Requirements and Analysis, Unified Modeling Language, Rational Unified Process: Not surprisingly, in-house and contracted companies that are working on similar projects will often use similar methods to ascertain *requirements in detail* and attain *correct output*.

XP and other agile methods: Agile methods have been pioneered on custom in-house projects, and the authors write persuasively about dealing with *rapidly changing or uncertain requirements*. Most agile methods assume a small project size, colocated development staff, and very easy access to the user, all of which are common but not universal attributes of in-house projects. Agile methods are also easier to implement in this practice culture than in the contract culture because the projects don't all have to fit into a fixed-price bidding process, which assumes that a project can be accurately estimated at its beginning.

Drafting Inexperienced People: I've been regularly alarmed by stories at the Software Testing and Reliability conference that tell of assigning people from other, unrelated jobs to head up testing for projects such as insurance software, or of hiring developers who are obviously seriously underprepared to take on the job. While this undoubtedly goes on in all practice cultures to some degree, it seems to be more common when the hiring organization's main business is not software-related.

Commercial Software

Business Model

Commercial software is software written by one company and sold to other companies for use in their own businesses. Retailers or other middlemen may sell the software off-the-shelf, or it may be sold directly to the customer. It may be strictly a piece of software, or may accompany a particular piece of hardware as processing and control software. If extensive customization services are included in the sale, rather than just the software itself, then the company belongs to the "custom software under contract" practice culture, since the contract ends up driving a lot of the economics.

In the simplest terms, the profit on this software is the difference between how much it cost to make, and how much it can be sold for – which is the product of sales volume and price per item. Sales volume depends on the attractiveness of the product in a competitive market. Attractiveness of commercial software is based on features, reliability, and service – glitzy new features for the sake of being new are not generally rewarded, particularly when selling to large companies. In fact, IT departments are notorious for not wanting frequent new releases, which they would have to test and integrate into their existing support structure. Price per item is subject to competition, so it can't be jacked up infinitely. The math works out to suggest that moderate budget overruns can be made up eventually by sales volume, which is certainly what I've observed. (This analysis assumes that there is competition – presumably a monopoly can behave differently.)

There are market windows for commercial products, often based on trade shows, but they are not as frequent or as powerful a force as the consumer market windows. "The need to develop and refine a central set of functions that operate correctly and efficiently drives the releases of system products like Windows NT and Windows 95; the target ship date is of secondary importance." (Cusumano 1995)

Typical commercial software includes database management systems, operating systems for commercial use, network management software, and drivers for peripherals used mainly in a business setting such as high-end laser printers. Medical and avionics software includes medical records systems, diagnostic programs, and air traffic control systems. Despite Microsoft Office's undoubtedly large sales figures, I don't believe Office is an archetype

for this practice culture. I say this because I don't see commercial software developers flocking to practices espoused by Microsoft – in fact, there seems to be a healthy dose of skepticism. One might also note that Office is purchased routinely by home users, whereas enterprise network management software is not.

The *regulated industries* are a subset of the commercial software vendors that work in areas regulated by a federal agency, generally to ensure the safety of the citizenry. Medical products (regulated by the Food and Drug Administration) and aviation software (regulated by the Federal Aviation Administration) are two of the biggest players in this culture. While these two business areas don't generally think of each other as sharing a common culture, I've observed a surprising level of similarity.

The Situation

- **Criticality:** There is a range from life-critical to mission-critical to mission-important. Medical products and airplanes are usually life-critical, operating systems and network software are mission critical, and printer drivers are merely important. Regulated products generally are life-critical.
- **Uncertainty of Users' Wants and Needs:** The requirements for products aimed at an existing business activity are generally reasonably well determined, but not as specific as the requirements for custom software. The users in the regulated industries tend to be very conservative, which results in a low level of uncertainty. For medical products, everything from data accuracy and precision to report formats must match earlier precedents. Aerospace software requirements are dictated by the specific equipment being controlled, so again there are considerable specific requirements. Getting approval for a completely new device in a regulated area is incredibly painful, so the number of experimental new products is, I think, less than in the non-regulated businesses.
- **Range of Environments:** The environments are the sum of all environments used by the companies to whom you wish to sell – this is a larger set than what's used in a single company, but tends not to include everything possible. For instance, compatibility with the America OnLine browser is unlikely to be a requirement in an online payroll deduction package, whereas compatibility with multiple recent versions of Internet Explorer and Netscape is. In the regulated industries, it is common to limit the environments in which a product is warranteed to run – this reduces the reliability problem space to a tractable size.
- **Cost of Fixing Errors:** The cost of fixing errors in released software is significantly higher than in the custom-software practice cultures. The software has moved into many people's hands, so patches must be distributed widely – this has gotten easier with the advent of the Internet, but it's still not trivial. Some products sell a maintenance contract to cover the cost of distributing upgrades, while others give them out for free. In the regulated industries, patches have to be distributed to all affected parties, so records of who owns what have to be kept for years. Worse, if it's a serious bug, the patches have to go out right away. Just because a target system is now sitting in a country with unreliable phone service and light-fingered customs agents does not excuse the vendor from getting the patch to the customer somehow, some way.
- **Project Size:** Projects can be small, or fairly substantial. Aerospace projects seem to run larger than medical products, probably because airplanes are bigger than your average medical device.
- **Regulation:**
  - **Non-regulated industries:** Generally there are no required process audits by any external agencies, including the purchasers. There are industry standards in some businesses such as telecommunications, which are often set by a consortium of the companies themselves. The customers have high expectations for correctness and reliability of the software, and the larger customers have some serious clout both financially and legally if the software doesn't live up to expectations. Extensive and detailed tracking of post-release problem reports seems to be common.
  - **Regulated Industries:** Many people think that practices in the regulated industries are dictated in detail by the federal government. This is in fact not true (unless working under government contract). There are some required practices, such as hazard analyses in the design stage, but the primary expectation is documented compliance to one's own stated practices, whatever they may be. There are extensive audits to demonstrate compliance. Each and every serious post-release problem must be reported to the vendor and the vendor must demonstrate both tracking and resolution of the reports in audits.
- **Communication between Developers:** Most companies appear to operate in what I think of as the "default" mode – the same person designs, writes, and unit tests a piece of code, and most people are permanent

employees rather than contract employees. These practices make both concurrent and forward communication simpler than in the preceding practice cultures.

- **Organizational Culture:** Varies, but control cultures seem to be less prevalent than in the custom software market.

Assumptions

The situation and capabilities again lead to some common assumptions:

- **Delivery on time is important, but budget overruns can probably be made up.**
- **The customers expect correctness and reliability.**
- **The software must work in a variety of different environments.**
- **We are expected to know about and solve problems after release.**
- **(In regulated industries), we need documentation, and we need to do what we say we do to pass audits.**

Favorite Practices

The commercial software vendors seem to be in a middle ground between the purveyors of custom systems and the mass-market software vendors. There are some large vendors whose staff has published extensively (IBM, Kodak, Rockwell), but many of these also operate in the contract software practice culture. It can be difficult to tell if a particular piece of work from these sources is independent of the contract culture. There is little published that identifies itself as clearly from the commercial software practice culture, and practitioners complain that there is little written about them. This may suggest that the total number of practitioners isn't very high, or maybe they just don't write much. It may also be that the commercial vendors don't benefit from revealing their practices to their competitors, whereas the Department of Defense benefits from spreading its favorite practices to more potential contractors.

Key Customers. The definition of *correct behavior* for Hewlett-Packard's cardiology products was heavily influenced by the opinions of a few key customers, who were themselves influential cardiologists. These key customers were frequently consulted during design and participated in field testing of new devices. I have heard of similar practices in other companies as well. These key customers are in a long-term relationship with the company, rather than being randomly selected as is done for focus groups.

"Real" Beta Test. Commercial software typically undergoes field trials when there is still time to fix the problems that are found. HP cardiology software underwent a series of field tests in doctors' offices under very close scrutiny by both developers and medical personnel – often the software was instrumented to keep a complete record of everything that went on, and any errors had to be fixed immediately. This practice was considered part of ensuring *correctness and reliability*.

Release Criteria and Quality Planning. I've observed quality planning used extensively in Hewlett-Packard for commercial software in both the regulated and non-regulated fields. In the regulated fields (now part of Agilent or Siemens), release criteria had to be set well before the release date, and the product had to pass its own release criteria. As a result, the release criteria were carefully chosen to be both achievable and sufficient to ensure *correctness, reliability, and safety*. In both fields, formal quality planning is used to decide what techniques and practices will be employed to achieve the intended levels of quality – everything from requirements to design methods to testing practices are carefully considered for efficacy and efficiency. Practitioners at other medical companies have told me that they use similar practices, and I believe this to be a fairly widespread practice in commercial software.

Inspections and reviews. Inspections and reviews appear to be heavily used in both regulated and non-regulated fields, because of their demonstrated effectiveness in achieving *correctness and reliability*.

Total Quality Management. TQM is still a favorite in the conservative regulated industries, because of the emphasis on managing practices by measuring their results, and the traceable documentation structure, which *supports audits*.

Tailored versions of CMM and ISO 9001 practices. ISO 9001 certification is expected in some areas of this industry – while this may not actually improve software practices, it does help with setting up the necessary record keeping. In the regulated industries in particular, *defects in the field are religiously reported*, which provides a substantial amount of information on which to base the choice of practices from the CMM or other models. A pragmatic approach is quite typical, so CMM-style management practices may be used to achieve *delivery on time*, but certification isn't sought unless there is some reason to think that the software buyers are interested in it. It is not uncommon to voluntarily seek out certifications such as ISO 9000 if this appears likely to confer an advantage over competitors or if the certification is required in some country in which the software will be sold.

Compatibility, Configuration, and Usability Testing. Due to the *variety of both users and user environments* supported, the commercial software vendors often spend more time and effort on these types of testing than the authors of custom-written software.

Variety of lifecycle models. The contractual issues which demand requirements up front are not present in commercial software so there's room for some variety. Methods that are very light on documentation such as XP are not popular in the regulated industries because the methods don't produce the *documentation to support audits*. I've seen staged delivery used with considerable success, and forms of evolutionary delivery as well. Teams working with hardware groups often have a high-level plan in a simple waterfall format in order to synchronize their work with the hardware development.

Mass-Market Software

Business Model

Mass-market software is software for home or small business use. (Firmware is in a category of its own because some of the situational factors have a huge impact on the business model, as we will see later.) The mass-market world is very different compared to any of the preceding cultures. Moving from a regulated or contract-driven organization into a mass-market organization can produce prolonged culture shock. At first glance, the practices used in this industry can appear crazy – but on further investigation, there are assumptions behind the behavior that do make sense.

The software is usually sold via a retailer or other middleman to the consumer, rather than purchased directly from the company that wrote it. The mass-market software market appears to be driven by market behavior which predated the invention of consumer software by quite a few years – frequent releases of "new and improved" items, heavy advertising to convince consumers to buy software which is discretionary at best, and a reliance on seasonal market patterns. If you miss Christmas or start of school, you can lose a huge percentage of sales for the entire year. Releasing new, cool features before the competition is a very strong factor in this market.

Typical mass-market software includes tax-return software, greeting card programs, games, and drivers and associated software for mass-market peripherals such as low-end inkjet printers and digital cameras. I hesitate to put mass-market operating systems in this category, since Microsoft is by far the largest player in the field and they seem to have at least one foot in the commercial software practice culture (as described by Cusumano 1995).

The Situation

- **Criticality:** If consumer software fails, people lose time and are often annoyed, but seldom lose large sums of money and are rarely at risk of injury or death. (Note that this refers to *software*. Failures in the firmware in your car may well be able to kill you.)

- **Uncertainty of Users' Wants and Needs:** The customers typically are a mixed bunch and difficult to contact. It can be quite challenging to get a truly representative cross-section. The most extreme case is a brand new type of product, where the customers can't tell you what they want because they've never even thought of doing that with software before. For products that have been around a while, the end users may be pretty sure what they want, but the marketing department usually makes up for it with lots of dandy new ideas.
- **Range of Environments:** Mass-market software has by far the widest range of environments in which it must operate, and those change constantly, making high levels of reliability very difficult and expensive to achieve.
- **Cost of Fixing Errors:** The software support life is quite short compared to commercial software. Patches are often distributed via the Internet now, which has reduced the cost of patch distribution quite substantially. (However, there is a practical limit on how large an upgrade can be downloaded via the Internet, with which Microsoft is continually flirting.) The vendor is under no obligation to find and notify the affected customers.
- **Project Size:** Projects generally run between one and a few teams, and are usually less than a year long. Larger projects require extremely high sales volumes to recoup costs at consumer-level prices.
- **Regulation:** No audits, no comprehensive regulation – in fact, unlike the other businesses, it is common to include a warranty disclaimer in the End User License Agreement, which must be accepted in order to install the software. While the legality of such a disclaimer is doubtful (Kaner 1998), the situation basically boils down to little or no regulation.
- **Communication between Developers:** As in commercial software, developers typically do all the work on "their" code. Geographic separation of developers seems to be more common in this environment than in the others, although my evidence for this is purely anecdotal.
- **Organizational Culture:** Varies, but the control model doesn't seem to be prevalent. Competence cultures (such as Microsoft) are more common, and cultivation cultures appear in the smaller companies.

The situational factors lead to a set of assumptions that are very different from the other practice cultures:

- **If we don't hit the market window, we're dead.**
- **This is a competitive market.**
- **The requirements are ill-defined and vary widely.**
- **If we satisfy 80% of our customers, that's good enough.**
- **If we ship with bugs, we can fix them later (as long as they're not too bad).**

Favorite Practices

The traditional, documentation-heavy methods designed for enormous multi-year projects simply don't work in this practice culture. Sometimes this has resulted in organizations ditching *everything* from the older cultures, and starting anew with a code-and-fix approach. This, coupled with the prevalence of just-out-of-school programmers in this culture, sometimes results in entire organizations that cannot choose to adopt potentially useful practices because literally no one in the building knows about them. Sometimes the practitioners rediscover or reinvent a practice on their own, and you'll see a breathless announcement of an invention that bears a very strong resemblance to practices in use for years in other practice cultures. To be fair, adapting a practice to a radically different situation is harder than simply adopting one by the book, so it's not surprising that the transfer of practices into the mass-market practice culture from other cultures is often not high.

Some popular practices are:

Focus groups and formal usability testing. These techniques to ascertain the *ill-defined and varying requirements* are heavily used by the larger consumer software vendors.

Lightweight planning and project management. Projects often must complete within a few months in order to *hit a market window* or *beat the competition*, so they tend to be shorter and smaller, which requires less formal planning.

Lightweight documentation. The requirements are usually not written out in detail, and design documentation may be almost non-existent.

"Good Enough" decision-making. The "Good Enough" movement is based on the idea that defect-free software in the mass-market situation is both unattainable and likely unnecessary, and focuses attention on planning to "understand the problems and benefits of a situation well enough to eliminate (or prevent) the *right* problems and also deliver the *right* benefits." (Bach 1997). The definition of "right" is based on both customer needs and business realities – *satisfying 80% of the customers* is often good enough. This is a difficult practice for many practitioners in this and in other practice cultures to accept, since admitting that software does ship with defects is often equated with "not valuing quality" or otherwise not being on board with official company goals, so the adoption of "Good Enough" methods has been slow. Oddly, the commercial software vendors use release criteria and quality planning to make decisions in ways that seem very similar to the "Good Enough" decision-making practices, although the intended reliability and correctness levels are usually considerably different.

Rapid Testing, Exploratory Testing, et. al. Many consumer software vendors use these testing techniques, which look for the important bugs first, rather than systematically testing against an extensive list of requirements (which doesn't usually exist anyhow). Since *the requirements are often not well known*, exploratory testing techniques are used to look for user dissatisfiers.

Agile Practices. These are designed specifically to work with *ill-defined user needs* that tend to change during the development period. Agile methods are actually easier to use on in-house projects, such as the C3 project described by Beck (Beck 2000), than on mass-market projects, because it's much easier to talk to the users frequently. However, agile methods are also used in the consumer market, as described in Iansiti and MacCormack's work (Iansiti 1997). There's also a strong tendency in the mass-market to claim that eXtreme Programming is being used when the only XP practice in use appears to be the lightweight approach to documentation.

Code, then Test. This is the practice of finishing all the coding first and then fixing all the defects – this doesn't preclude up-front design so it isn't quite the same as McConnell's code-and-fix. (McConnell 1998). The practice of coding everything before fixing anything tends to be a really bad match for the mass-market culture since the work piles up at the end and makes the release date uncertain, which this culture can ill afford due to the *fixed market windows*. The only reason I can think of for the persistence of code-then-test is that it gives the illusion of rapid progress towards the all-important release date, at least during the coding phase.

Firmware

Business model

The software is made as an adjunct to a physical product, such as a toy or automobile or printer. The products themselves are subject to market forces that depend on whether they are mass-market products, commercial products, and so forth. However, the situational factors are dramatically different from those of the other practice cultures, and I think they overwhelm the effects of the market forces. For instance, I found that the practices for software used in HP's cardiology division and its inkjet printer division were substantially different, but the firmware practices were quite similar.

The Situation

- **Criticality:** The products run the gamut from toys to automobiles to missiles, so some are life-critical while others are not.
- **Uncertainty of Users' Wants and Needs**. The requirements of firmware are set by the hardware for which it is developed. Generally the requirements are easy to discover by asking the hardware developers, who are readily available. However, the requirements do change as the hardware design develops over

time, and there is a common practice of expecting firmware workarounds for hardware problems discovered late in the game.

- **Range of Environments**: Firmware is generally expected to work in only one environment – the hardware that is using it. However, if that hardware connects to hardware or software that varies widely (like a specific printer connects to a wide variety of PCs), the firmware will probably have to handle some of the variation itself**.**
- **Cost of Fixing Errors:** Fixing a firmware error requires re-flashing the firmware or replacing the entire chip, which in turn requires manufacturer or service personnel getting their hands on the product. For anything sold in the mass-market, recalling millions of items makes this prohibitively expensive. For items with lower sales volume, simply finding the items may be practical, but all the shipping and work on them is economically practical only if the item costs thousands of dollars. This is a huge factor in driving software engineering practices for firmware developers – you simply cannot afford mistakes, even on low-criticality products.
- **Regulation:** Firmware in medical and aviation products is regulated just like the commercial software in those fields. The regulatory picture for mass-market firmware is considerably different than for mass-market software – you are not expected to accept an "End User License Agreement" excusing an auto manufacturer from responsibility for the results of a firmware defect in your car.
- **Project Size:** Firmware has historically been smaller than software because it has to fit in permanent memory, but in the last decade the chips have gotten a lot bigger and the size of firmware projects has increased. The amount of code in an automobile, which is spread over multiple processors, is both amazing and a little scary.
- **Communication between Developers:** Most firmware developers seem to be permanent employees and I have not heard of any instances of the junior-senior division of labor.
- **Organizational Culture:** I have no real information on this.

The situational factors lead to a reasonable set of assumptions:

- **The firmware must be of high reliability and correctness.**
- **We have to work closely with the hardware developers.**
- **The hardware will change during development.**

Favorite Practices

Waterfall Lifecycle. Firmware developers often use a waterfall lifecycle because they *work closely with hardware developers*, whose work is usually sequential rather than iterative, and a waterfall lifecycle is easily synchronized with that.

Realtime Design Methods. Firmware developers use a number of design methods that are not needed in other fields to deal with realtime input from the devices that the firmware controls.

Evolutionary Delivery. Surprisingly, some firmware teams do use some varieties of agile methods, wherein the hardware team plays the role of the customer. This seems to help with the *changing hardware requirements.*

Inspections and reviews. Firmware developers, even for mass-market products, tend to make heavy use of inspections and reviews to achieve *high reliability and correctness*.

Testing by developers. Firmware teams seem more likely to be heavily involved in testing than their software counterparts. In some cases, this is driven by safety issues – a partially assembled or prototype piece of hardware may not be safe for an untrained person to handle. There is also a big emphasis on "white-box" testing to help achieve a high level of *reliability and correctness*, in which a development background of some sort is a necessity.

Open Source Software

A lot of the situational variables are clearly quite different for open source. I'm not personally familiar enough with either the field or the writings about it to do an analysis on open source.

Academic Software

Academic software is written by people in universities to do research – this would include both computer science research and research in many other fields such as biology, chemistry, psychology, etc. Programs used to do research in fields other than computer science are often in use for many years and there can be considerable turnover of the graduate student authors, who may be entirely self-taught. I was first introduced to programming as a chemistry graduate student in the 1980s, when I worked on an extensive set of FORTRAN programs for various types of computational chemistry. At the time, the graduate student authors had very little training and no experience in aspects of software engineering such as structured design or testing. "Favorite practices" were very primitive and sometimes counter-productive (such as the prohibition against indentation favored by one professor). Things have probably changed since then, although it's doubtful that graduate students in other fields have picked up a complete computer science education.

Internet Software

Business Model

The Internet is a technology, not a business model, and as such the technology appears in multiple business models – some closely resembling business models we saw above, and some that are new.

- **Custom Contract Software.** Plenty of people are writing Internet software for other companies – the main difference between this and traditional custom contract software is that the customer companies and the contractors are often much less experienced at drawing up contracts than the traditional customers and contractors in this business model, so the contracts are much less detailed. The software written under contract can be any of the types described in the following business models.
- **In-House Custom Software**: This is generally business software for use within the company - payroll, health insurance, etc. The economic factors for Internet software intended for in-house use seem to be pretty much the same as for more traditional client-server software – the benefits have to justify the cost, and projects can be cancelled or delayed due to budgetary constraints. The IT groups do face some issues with learning so much new technology, which drives some of them to hire or contract with outside programmers.
- **Commercial Software**: Internet software written by one business to be used by another business mostly is sold as "services" to which the customer company subscribes. Some are invisible components of custom services, and others are quite visible to the users, such as services that manage health benefits information or charitable payroll deductions for multiple companies.
- **Mass-Market Software:** There's been a lot of software written for consumers but very little success at extracting money directly from consumers. This is not the same business model as we find in traditional mass-market, where the consumer has to pay for the software – I am not sure what this will become.
- **E-Commerce**: E-commerce sites used by mass-market consumers represent a new business model, where the users are mass-market consumers but the people who pay for the service are not. This strongly resembles the traditional mass-market business model, where time-to-market and competitive new features are more important than cost. Most e-commerce sites allow the consumer to buy a product or service (books, airline tickets, clothes, computers) and a percentage of that transaction pays for the service. Others operate as inducements to buy a product, rather like a type of advertising.

The Situation

- **Criticality:** In-house Internet applications that replace existing applications are expected to meet the same standards for correctness, although the tolerance for availability and usability may be lower than usual.

Consumers expect e-commerce applications to keep track of all the money and orders without fail, so expectations of correctness are quite high, although the customers will tolerate performance that would be unacceptable in shrinkwrap software. For the first few years, the consumer-facing software was being used mostly by "early adopters" who were willing to tolerate a substantial number of problems, but as the technology becomes more widespread, expectations of reliability from consumers are increasing dramatically. The availability requirements can be more stringent than in traditional client-server applications, because it's very easy to pick up customers in multiple time zones, and suddenly you have no "middle of the night" maintenance windows.

- **Uncertainty of Users' Wants and Needs:** For both commercial and consumer-facing software, the service being provided is often either completely new or was previously done in a very different manner, so the uncertainty of requirements is often very high.
- **Range of Environments**: The range of environments depends on whether the end-users are people working at a company with its pre-determined environment and tasks to be done, or are consumers with a wide range of environments and tasks. The number of possible combinations is actually larger than that of shrinkwrap software, because the software must be compatible with not only the operating system but also the browser.
- **Cost of Fixing Errors:** The cost of releasing new functionality and bug fixes to a website is in fact quite low compared to other forms of releasing software fixes. However, changes to improve performance or support loads that increase by orders of magnitude have not proven to be as simple – they often require tearing apart part of the infrastructure and redoing it.
- **Regulation:** The in-house projects are subject to the same regulatory and audit requirements as those of other in-house applications. The consumer-facing software is not actually purchased by the consumer, so there is no End User License Agreement in force. It is unclear at this point what level of responsibility will play out in the consumer market.
- **Project Size:** Project sizes range from quite small to considerable for the larger commercial sites.
- **Communication between Developers:** I am frequently seeing small development groups scattered across wide geographies, but I don't have enough data to say if this is a trend or not.

The assumptions based on the situation and business model appear to be heavily affected by a third factor: the home practice culture of the people in question. For all practice cultures, some of their accustomed assumptions work and some don't. I have seen some individuals and teams consciously inventory their own assumptions and figure out which ones should be kept and which replaced, and others throw the baby out with the bathwater. The latter is more common with teams consisting entirely of bright young people with a command of the technologies but little project experience, although more experienced people can also fall prey to the illusion that the "Internet is nothing like anything that has ever been done before".

Immigrants from the in-house custom software practice culture often need to:

- Discard the emphasis on low cost over time-to-market. This emphasis makes sense when supporting internal efficiency measures, but it holds back the rapid progress needed to succeed in this market. Often teams are not consciously aware of which of their company policies or standards are designed to keep costs low.
- Keep their skills with handling large disparate systems: configuration management for systems, change management, database design, security, proactive maintenance, etc. Immigrants from practice cultures that didn't make large systems often don't have these specialized skills.
- If not already using agile methods, investigate switching. Collection of requirements in detail up-front does not seem to work in this business model.

Immigrants from the mass-market practice culture often need to:

- Learn the skills associated with large disparate systems.
- If not already using an appropriate project planning methodology, learn one of the agile methods.
- Plan to learn methods for producing high-reliability code – you'll need it soon.

All the Internet immigrants, regardless of their original culture, seem to have a tendency to drop practices to which they are accustomed because of the compressed schedules typical in this field. The results are not as bad as one

might think, because to date the focus on time-to-market and low customer expectations for reliability have made taking some shortcuts reasonable. However, this is probably not going to last as expectations for reliability increase.

Internet Immigrants Learn From Each Other

It is not uncommon for immigrants from more than one practice culture to work together on the same project - many companies contract out applications or hire complete teams of new programmers to work with existing in-house staff. The meeting of minds can be antagonistic or mutually beneficial, depending on the attitudes of the people involved.

A good example of a mutually beneficial relationship is Bill Tuccio's story of a joint project. (Tuccio 2002) Bill worked for a dot.com company that contracted with a theater chain to create a system to sell movie tickets online. The dot.com folks worked directly with the theater chain's IT department. Initially there were some difficulties. The IT people at first didn't take extra steps to push the infrastructure orders through – they didn't see the artifacts that they were accustomed to seeing, and assumed the project couldn't possibly meet its "aggressive" schedule. Eventually the two groups started having twice-weekly status meetings and frequent phone calls. It became clear that the dot.com staff was progressing very quickly through the software despite their lack of written documentation, and the IT staff responded by speeding up their part of the work. The dot.com group also found some advantages in the IT folks' methods. At one point, the IT people sent an expert to do a risk analysis – something that no one in the dot.com really knew how to do. The risk analysis identified the need for some extra staffing that turned out to be a very good idea. Overall, the relationship wasn't always smooth, but in general the staff of the two companies took the time to ask "why are you doing that" and adjust their assumptions and their practices as needed. By the end of the project (which did release on time), both sides had learned some useful things from each other.

Evaluating Practices for Effectiveness

Now that you've visited a number of practice cultures, let's return to the original topic of this paper – how can I evaluate a practice that I'd like to adopt? Once you've understood the basics of the practice, the next step should be to determine the assumptions behind the practice, in order to compare those assumptions with your environment. To find the assumptions, you need to know where a technique originated, and in what other contexts it has been used.

Sometimes the description of a technique will also describe the business in which the technique originated – this is more common now than it was in the 1970s, when there were fewer practice cultures. Sometimes the examples and context surrounding descriptions of the technique will identify its origins. Other times, you may have to look for the professional history of the author, which nowadays is often available on their website. Articles in magazines and journals are usually aimed at the practice cultures served by that publication. Sometimes the publication is quite upfront about its audience. Crosstalk identifies its audience right in its title*: Crosstalk: The Journal of Defense Software Engineering*. If the magazine isn't quite that forthcoming, look for their message to potential advertisers. Software Development Magazine: "Discover the many ways we can deliver your message to over 100,000 corporate software development leaders every month". (SDM 2002)

Once you've identified the practice culture in which a practice originated, you can examine the benefits attributed to it against the business model and situational factors for that practice culture. Ask yourself: are any of the benefits dependent upon assumptions or factors that won't be true in your situation?

For example, Sue's company is working on e-commerce products. They are interested in increasing the reliability of their software. Sue knows that medical products are highly reliable, and she also knows that detailed documentation of test plans and procedures is typical in the medical products field. Sue wonders if introducing IEEE 829 standards for test documentation would be helpful.

What are the assumptions behind the use of detailed documentation in the medical field? The medical products practitioners will readily tell Sue that written test plans and test procedures lead to thorough testing, which results in high reliability. Many practitioners will tell Sue that everyone, in every field, should document their test plans and procedures in detail. However, the practitioners may neglect to tell Sue that some of the documents are primarily intended to satisfy FDA audits. People who are not directly involved in the audits may not even realize this. Sue

also may not discover that not all medical products companies follow the IEEE 829 standard exactly as written. Most importantly, Sue may not discover that using the IEEE 829 standard did not teach the medical products testers which tests are useful and which tests are not. They learned that someplace else, which may not be as visible as their use of IEEE 829 standards.

Introducing IEEE 829 standards to Sue's group may improve reliability if they haven't had a systematic method to communicate what to test, what results are expected, and what has already been tested. However, following the entire IEEE 829 requires a great deal of documentation and will probably be overkill in Sue's situation. In fact, if Sue's project has a fixed ship date, precisely following the IEEE 829 standard may actually reduce reliability by spending time preparing documentation that could have been spent on finding bugs.

Once you start looking for the assumptions behind the use of a practice, you'll ask much better questions about the context or situation in which the practice is used, which will clarify the ways in which the practice delivers value in that situation. This insight can help you decide whether to adopt a practice as-is, modify the practice, or stay away completely.

Summary

The practice cultures described in this article are by no means definitive. Despite the length of this paper, I feel I've only scratched the surface of this topic. Undoubtedly each of you will have your own observations about business models, situational factors, and practices, based on your own experience. I've tried to demonstrate the connections between assumptions and practices in order to give you some idea of why some practices seem to "fit" your own situation while others seem ineffective, if not downright stupid.

If you're interested in talking to people from other practice cultures, you'll find them at conferences, online, in classes, in the pages of magazines and journals, perhaps at lunch if you work in a large company with a varied business. You'll learn many more interesting and useful things from these people when you understand that their assumptions don't necessarily match your assumptions, and there may be good reasons for that. Listen, ask questions, and don't jump to conclusions too fast. Remember, it all depends.

References

Bach[1997]: Bach, James; "Good Enough Quality: Beyond the Buzzword"; IEEE Computer; August 1997. (also available at www.satisfice.com)

Bach[1999]: Bach, James; "Good Practice Hunting"; Cutter IT Journal; 1999. (also available at www.satisfice.com)

Beck[1999]: Beck, Kent; *eXtreme Programming Explained: Embrace Change;* Addison-Wesley; 1999.

Cusumano[1995]: Cusumano, Michael; Shelby, Richard W.; *Microsoft Secrets*; Simon and Schuster; 1995.

Highsmith[2002]: Highsmith, Jim; *Agile Software Development Ecosystems*; Pearson Education; 2002.

Humphrey[1989]: Humphrey, Watts; *Managing the Software Process*; Addison-Wesley; 1989.

Iansiti[1997]: Iansiti, Marco; MacCormack, Alan*; Developing Products on Internet Time*; Harvard Business Review; 1997

Kaner[1998]: Kaner, Cem; Pels, David; *Bad Software: What to do When Software Fails*; John Wiley & Sons; 1998.

McConnell[1996]: McConnell, Steve; *Rapid Software Development*; Microsoft Press; 1996.

Schneider[1994]: Schneider, William. *The Re-engineering Alternative: A Plan for Making Your Current Culture Work*. Irwin Professional Publishing, 1994.

SDM[2002: http://www.sdmagazine.com/sdonline/mediakit/, Aug 16, 2002.

SEI[2002]: http://www.sei.cmu.edu/about/about.html, July 1, 2002.

SET[2000]: http://source.asset.com/stars/loral/cleanroom/tutorial/, dated 2000, Software Engineering Technology, Inc.;

Tuccio[2002]: Tuccio, Bill; personal correspondence.

ABAT: A Tool for Automating Basic Acceptance Testing

Julie Fleischer/Rusty Lynch
Telecom Software Program
Intel Corporation
MS CO5-162
15400 NW Greenbrier Pkwy
Beaverton, OR 97005
Julie.N.Fleischer@intel.com
Rusty.Lynch@intel.com

Abstract

A common problem among software testers is starting a test pass and then discovering that the software is missing some expected functionality. This often leads to abandoning the test pass and waiting for another software release with the needed functionality. Unfortunately, in many cases, testers have already spent considerable time preparing the hardware, installing and configuring the software, and starting to test before discovering the lack of functionality, thus resulting in a significant loss of time per tester. This loss of time is difficult to prevent, though, as developers are not always aware of how their changes may affect the entire system, or they may not have the time to test that all pieces of the system are stable before submitting their changes. The Automated Basic Acceptance Test (ABAT) system is designed to assist in solving this problem by automating the tasks of installing and performing basic tests on each build of software and publishing the results to a web interface. Readers who find they are losing time due to testing software missing expected functionality are encouraged to read more and see how they can customize the ABAT system to benefit their organization.

Julie Fleischer is the validation engineer at Intel Corporation who integrated the ABAT system into her group's continuous build system environment and then maintained and expanded the system. She currently maintains the open source website on this system.

Rusty Lynch is responsible for the creation, design, and development of the ABAT system. He also has experience designing and implementing other automated test systems for engineers at Intel Corporation.

In our group, we found that we were losing a significant amount of time starting to test software builds that were missing basic functionality. Generally, the core problems were easy to detect and fix (the wrong files were tagged, a configuration item was wrong, or an important feature was not marked to be built). However, by the time the issue was caught, considerable time had already been lost installing and configuring the software and beginning a test pass. It was especially difficult when these types of problems happened at the end of a project, where the effects were most costly. Not only did release times risk slipping, but also more testers than usual had installed and started testing the build on the chance that the build would be good. When we multiplied the amount of time it took to find the issue by the number of testers that had started testing and the number of times these types of problems occurred, we found that too many man-hours were being lost.

While multiple process and tool solutions to this type of problem exist, in this paper, we will focus on a software tool that assists us in determining which builds have expected functionality. This tool, the Automated Basic Acceptance Test (ABAT) system, determines the basic functionality of a build by automating the process of performing basic acceptance tests. We will start our discussion with a description of what the ABAT system is and what capabilities it provides. Next, we will describe the ABAT architecture and design, including information on how the reader could customize the system to meet his or her organization's needs. Lastly, we will discuss the benefits we experienced after integrating the ABAT system into our validation process.

ABAT Features

The process that ABAT automates is that of using acceptance criteria to define whether a software build is an acceptable software build that warrants further testing by the validation team. Our group's acceptance criteria have been in the form of tests, called "Basic Acceptance Tests" or BATs. These tests evaluate the core functionality of the build, and their results are then analyzed to determine whether that build is an acceptable build. If certain functionality is missing, a formal test pass is vetoed, and only some portions (or none) of the build will be tested. The ABAT (Automated-BAT) offers one more layer of timesavings by automating the following:

- Installing the latest software build on a target machine and rebooting the machine.
- Running a series of basic acceptance tests on the target machine.
- Sending the results of the installation and basic test pass to a web server for publication.

The following sections describe these major functions in further detail.

Software Installation on the Target Machine

The ABAT system downloads a build of software from a build machine onto a target machine. This build of software is then installed on the target machine. In order to have a record of which pieces of the installation were successful and which were not, the ABAT system generates an installation report for publication after the ABAT system has finished. The installation report contains a Boolean status (success or failure) as well as log files generated during the installation of each feature in the build. In our

case, a reboot was necessary after the software installation, so the ABAT system is responsible for rebooting the target machine.

Software Test And Report

After the machine has rebooted, the ABAT system runs through a series of basic acceptance tests on the newly installed software. These basic acceptance tests are broken into separate scripts for each feature. This gives multiple benefits:

- Multiple testers can contribute to the ABAT tests, each writing test scripts for the features with which he or she is most familiar.
- Tests can be added or removed as features are added and removed from the product.
- Since each script is an independent basic acceptance test suite of a feature, these scripts can also be run separately from the ABAT system if desired.

The ABAT system compiles a test report that is also broken out by feature. If all tests for a given feature have passed, the status of testing for that feature is successful; otherwise, it is a failure. The test report also shows all log information generated by a test, including details on specific tests that failed as well as any additional information that may indicate why the failure happened.

Sending Installation and Test Reports to the Web Server

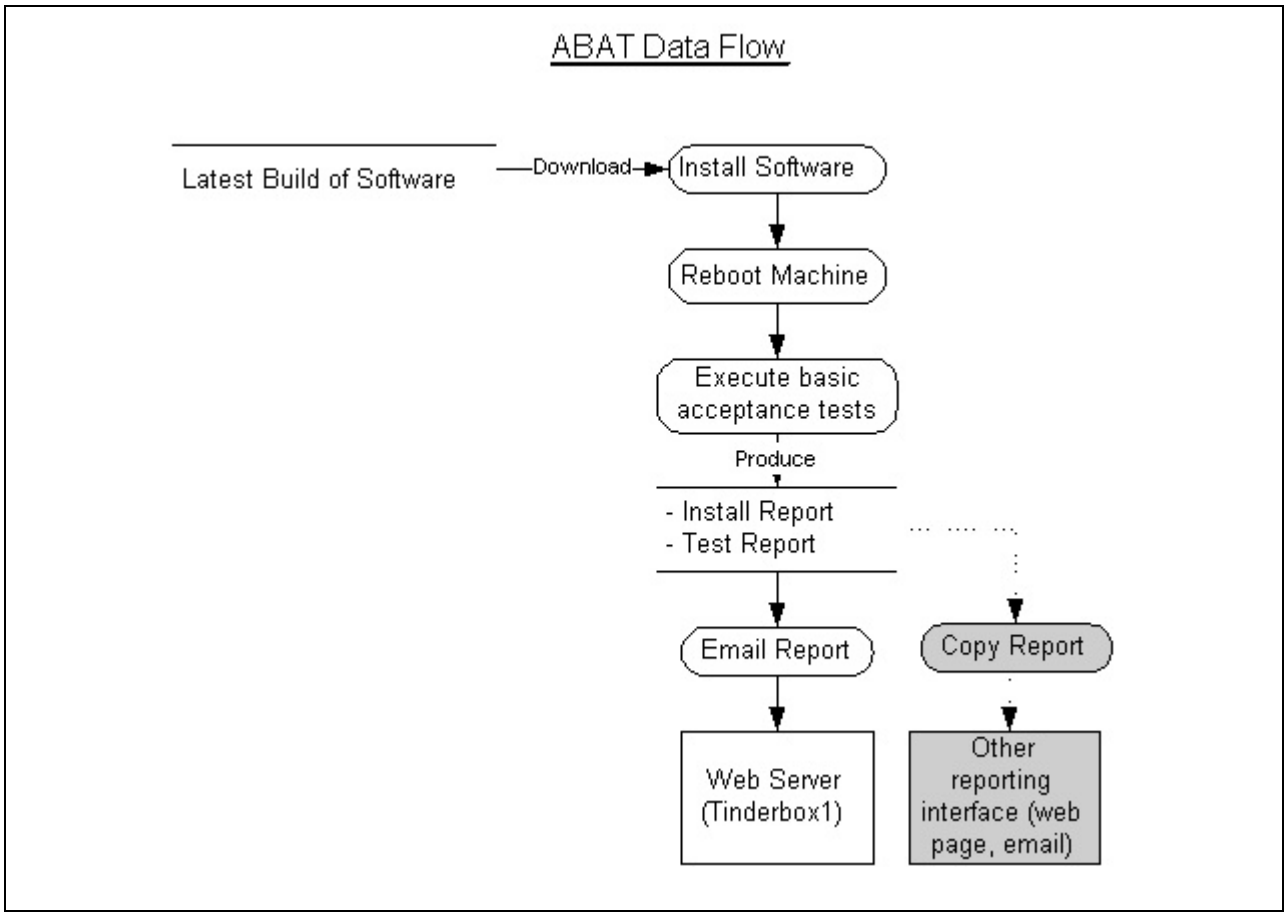
The installation and test reports that ABAT generates are sent to a web server for viewing. The ABAT is responsible for sending this information in a format that is understood by the web server. In addition, ABAT is responsible for instructing the web server on how to parse and display this information so that it is presented in a format that is easy for a reader to digest.

ABAT Architecture

Now that the basic features of the ABAT system have been presented, we have the context from which the architecture of the ABAT system can be discussed.

The basic data flow of ABAT is shown in **Figure 1**. This diagram illustrates how the ABAT system downloads its input (a build of software), installs it on a target machine, runs through tests on it, and then sends, via email, all results (an installation report and a test report) to the output web server.

Figure 1 - ABAT Data Flow Diagram



In order for ABAT to download a build of software, it needs to know the location and format of the build. Upon completion, ABAT needs to send its output in an agreed upon format so that the web server can parse and display the output.

Although the details are out of the scope of this paper, an overview of how our group generates and displays ABAT input and output is worth mentioning.

In our group, a continuous build machine produces the build of software that is the input to the ABAT machine. This continuous build machine rebuilds the software each time a change is made to the source code. It then sends the build to the ABAT machine for basic acceptance test processing. In this way, the current state of our software is always known.

Our group displays ABAT's output on a special web server, which uses the Tinderbox[1] software created by The Mozilla Organization. Tinderbox provides a graphical interface

[1] Tinderbox is open source code licensed by The Mozilla Organization under the MPL.

for viewing both the output of the continuous build process, as well as the output from the ABAT system[2].

The gray boxes in **Figure 1** illustrate where other organizations could choose a different methodology to display their output. We were required by Tinderbox to email ABAT's results to a web server. However, other organizations could choose any transportation mechanism (for example, email, copy, ftp) and send the data to a web server, another machine, or to the inbox of the test team.

ABAT Design

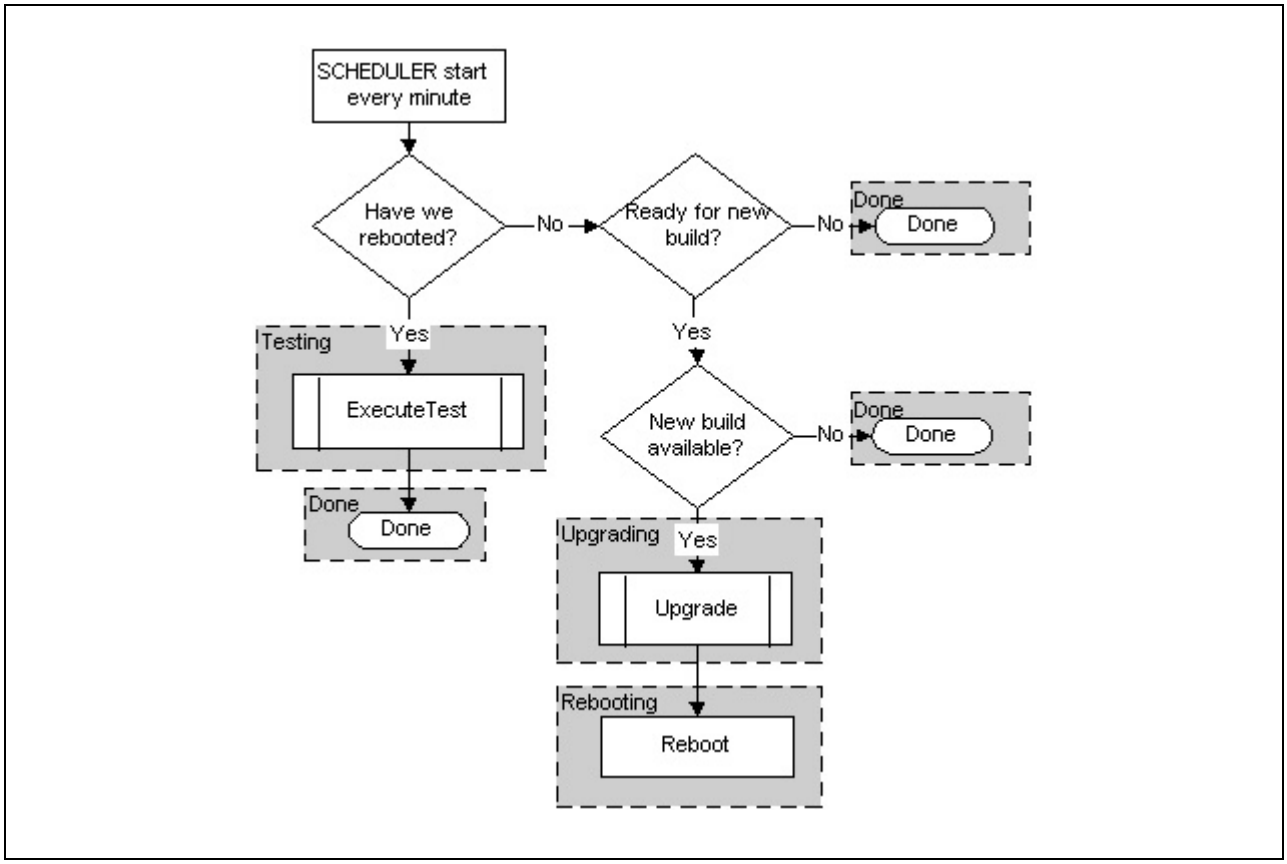
Now we will cover a more detailed description of the ABAT design, including the overall logic of the system and its implementation.

ABAT Logic and State Changes

The ABAT system cannot execute as a single pass through a program for a couple of reasons. 1) The ABAT must stop executing while the reboot is happening, and 2) the ABAT does not know when a new build is available for processing. Because of this, it was decided to run our ABAT system on a scheduled basis, thus allowing ABAT to determine what state the target machine was in and execute the appropriate actions based on that state. This logic is illustrated in **Figure 2**.

[2] For additional details on Tinderbox, or to see how The Mozilla Organization displays Tinderbox output, see http://www.mozilla.org/tinderbox.html.

Figure 2 - Logic of ABAT System with States



The ABAT system is constantly started by a scheduled job on the ABAT machine. Depending on the state of the machine and whether or not a new build is available, the ABAT system installs a new build, runs through the basic tests, or quits.

The state of the machine is encoded in a state file that persists across reboots. It can be set to either:

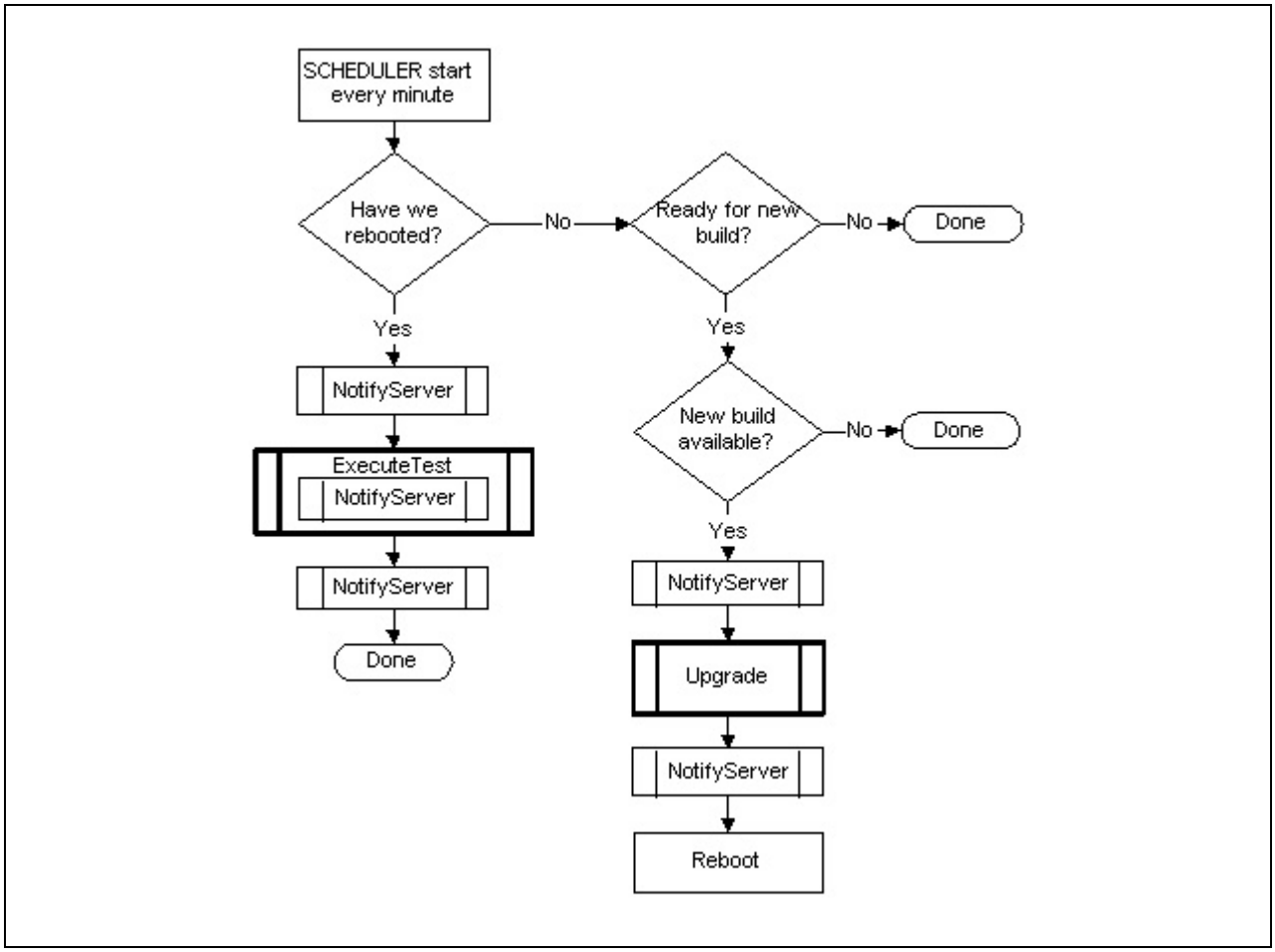
- No file – Never been started.
- Upgrading – Performing an upgrade.
- Rebooting – Rebooting the ABAT machine or just finished rebooting the ABAT machine.
- Testing – Testing ABAT software.
- Done – Done with an ABAT cycle.

The ABAT determines whether or not a new build is available by comparing the timestamp of the build on the build machine with the last timestamp of an ABAT run. If the build on the build machine is newer than the ABAT run, then ABAT downloads the new build.

ABAT Functional Blocks

The ABAT system is broken down into four separate functional pieces. One piece upgrades the machine, another piece executes the tests, a third piece informs the web server of its progress and sends the output reports when finished, and the fourth piece coordinates all the pieces together. Each piece is coded as a command-line Perl script. **Figure 3** illustrates how these scripts fit into the overall ABAT logic; each double-edged box represents a script.

Figure 3 - ABAT Flow with Perl Scripts



A discussion of each of these scripts follows.

`Upgrade` Script

The `Upgrade` script is responsible for upgrading the system and generating the upgrade report. The upgrade report contains a Boolean attribute, which indicates whether the upgrade was successful or not, as well as the output received during the upgrade. This report is stored in a file on the target machine so that it will persist across the reboot and be sent to the web server along with the test report after all tests have been run.

`ExecuteTest` Script

The `ExecuteTest` script is responsible for executing the basic acceptance tests. There is one test script per feature in the system, and the `ExecuteTest` script is responsible for running each of these scripts and storing its output in the test report.

The `ExecuteTest` script requires that the output of the test scripts follow a simple pre-defined format:

```
Line 1 – Title
Line 2 – Description
Middle lines – Log information
Last Line - %%SUCCESS%% if successful  (Anything else is
interpreted as a failure.)
```

Because of the simplicity of this requirement, test scripts can be written in any language that can be executed on the ABAT machine.

Test script creators must also ensure that the main script for their test suite is installed in a specified directory on the ABAT machine. The `ExecuteTest` script will run through all scripts in this directory and parse the output into the test report.

`NotifyServer` Script and Display of Output

The `NotifyServer` script is responsible for informing the web server of the ABAT machine's status during execution and also for sending the install and test reports at the end of the ABAT cycle. `NotifyServer` communicates with the web server via email messages that are composed in the required format of the web server.

In addition to sending email in a form that the web server can parse, the `NotifyServer` function also places the install and test report information into an XML formatted report according to the grammar in **Figure 4**.

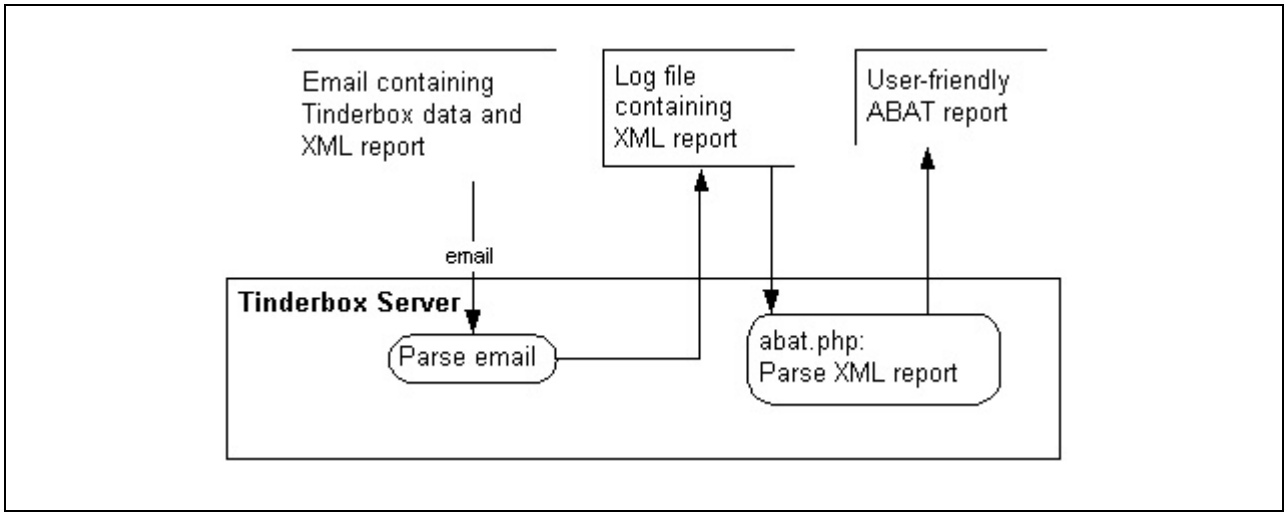
Figure 4 - XML Grammar for ABAT Output

```
<?xml version="1.0"?>
<!DOCTYPE TestReport [
<!ELEMENT TestReport (InstallReport, TestSuite+)>
<!ELEMENT InstallReport #PCDATA>
<!ATTLIST    InstallReport
     result (success, failure) #REQUIRED
>
<!ELEMENT TestSuite (Title, Description, Log)>
<!ATTLIST    TestSuite
     result (success, failure) #REQUIRED
>
<!ELEMENT Title #PCDATA>
<!ELEMENT Description #PCDATA>
<!ELEMENT Log #PCDATA>
]>
```

This grammar shows that the full test report consists of the install report plus output for one or more test suites. The install report contains a Boolean attribute stating the success or failure of the installation as well as additional log data. Each test suite also contains a Boolean attribute stating the success or failure of the test suite as well as a title, description, and any log data collected from the suite.

On the web server, a PHP file is used to parse the XML and display the output in an easy-to-read fashion. **Figure 5** shows this process. The web server (or Tinderbox, in our case) first parses the email that is sent by the `NotifyServer` script into a log file containing just the XML report. Next, a PHP file on the web server reads the XML report and parses it according to the XML grammar described in **Figure 4**. The final parsed HTML file is displayed to the user in an easy-to-read format.

Figure 5 - ABAT Output Flow



`abat` Script

The `Upgrade`, `ExecuteTest`, and `NotifyServer` scripts comprise the basic functional elements of the ABAT system. A main script, entitled `abat`, ties all the functional elements together (see **Figure 3**). The `abat` script is started via a scheduled job on the target machine.

Customizing ABAT

The ABAT system can be customized to meet the individual needs of an organization.

Architecture Customizations: Inputs and Outputs

The first step in customizing ABAT is to determine how an ABAT system will fit into the architecture of an organization. Each organization may have a different way of sending input to the ABAT system and of receiving output from ABAT. Understanding these interfaces will assist the reader in making the correct changes to the input and output systems, as well as to the ABAT system.

Input

As shown in **Figure 1**, input to the ABAT system is a build of software. In order for ABAT to download this build, the build needs to be placed in a predetermined location and have a predetermined name.

In our group, our continuous build machine updated a symbolic link to point to the latest build. Other methods that the reader could use are:

- Place a script on the build machine that a build engineer runs after a build has been generated. This script would place an archive file of the build into a known location that both the build machine and the ABAT machine can access.
- Have the build engineer manually place an archive file of the build into a known location after a build has been completed.

Output

In addition to making ABAT input decisions, the reader needs to determine what format the ABAT output should be in, where that output will be displayed, and how ABAT should send output to its display location.

Recall that the original ABAT system generates output in a format that a Tinderbox server can parse. The `NotifyServer` script then sends this report via email to the web server. The install and test report portion of the output is sent in XML format, and a PHP file on the Tinderbox server parses through the XML output and displays it in a user-friendly format (see **Figure 5**).

If the reader has a Tinderbox server at his or her organization, then customizing ABAT is just a matter of placing the PHP file in the correct location and configuring ABAT to mail results to the Tinderbox server.

However, for readers who do not have a Tinderbox server available, the following customizations are possible:

- ABAT could be configured to send the output email to specific testers or managers. That is, the data flow of the email to Tinderbox (see **Figure 5**) could be replaced with an email sent to certain individuals in the organization. To accomplish this, only an ABAT configuration item needs to change.

  In addition, if the reader would like the output to be mailed in a user-friendly format, the `ExecuteTest` and `NotifyServer` scripts could be modified to change the format of the ABAT output.
- ABAT could also be configured to ftp or copy the output to a web server. In this case, the `NotifyServer` script would be modified to ftp or copy output instead of sending mail. In this case, the web server could perform the operations performed by Tinderbox, as depicted in **Figure 5**.

Design Customizations: Modifying ABAT Scripts

After the reader has determined how the ABAT system can fit into the overall architecture of his or her organization, the next step is to modify certain pieces of the ABAT scripts to manipulate the data.

Install Mechanism

Because the install, or upgrade, mechanism in ABAT is specific to the particular software, the `Upgrade` script would need to be rewritten by the reader. This script would need to de-archive and install the build sent to the ABAT machine. It is not necessary to add code for the reboot, however, as this is an intrinsic part of the ABAT system.

ABAT Suite

The next challenge is gathering a suite of tests to run. Since the ABAT system is designed so that multiple testers can develop test suites for the system, this can be straightforward. The test suites created need only follow the few rules described in the "`ExecuteTest` Script" section in order to run on the ABAT system.

Benefits of ABAT

We experienced a number of benefits from implementing the ABAT system into our validation process:

- The ABAT system detected several types of errors that previously were not caught until a formal test pass had started, such as:
  - Wrong files tagged.
  - Configuration items incorrectly set.
  - Severe bugs in critical functionality.
- The ABAT system provided validation engineers with an overview of the health of the system before formal testing began. If certain features were unavailable in the build, the validation engineers knew this before they tried to test those features.
- The ABAT system could effectively be used to manage releases of software to other validation engineers. We had a set of criteria ("ABAT tests must pass or issues must be documented") to use in releasing software.
- The ABAT system took an order of magnitude less time to run than previous manual BATs took, meaning more ABAT cycles could happen per day. Sometimes as many as five cycles occurred in one day, allowing us to fix small bugs before a release.

Figure 6 and **Figure 7** present some data that illustrate how the ABAT system benefited our group; however, not all of the benefits we saw could be quantified using defect data because:

- Some defects exposed by ABAT were quickly communicated to the build engineer and fixed before defect reports had time to be written. Previously, these types of defects would not have been caught until multiple engineers had started testing.

- Testers had more time to spend looking for more insidious and severe bugs; since less time was spent testing bad builds. A side effect of this is that the overall number of high severity defects could increase.
- There was less risk of having multiple defect reports for the same issue, as ABAT would expose issues that could be fixed before testers would find them.

Figure 6 - ABAT Defect Data in First Release

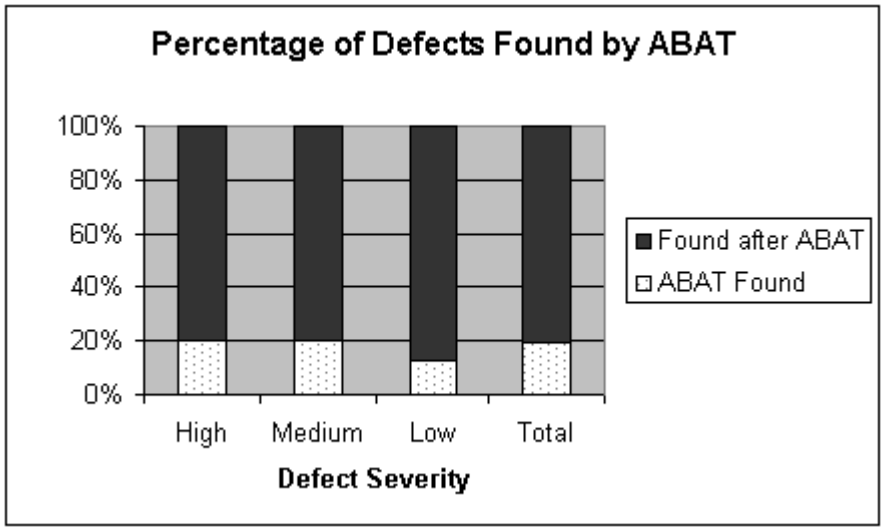
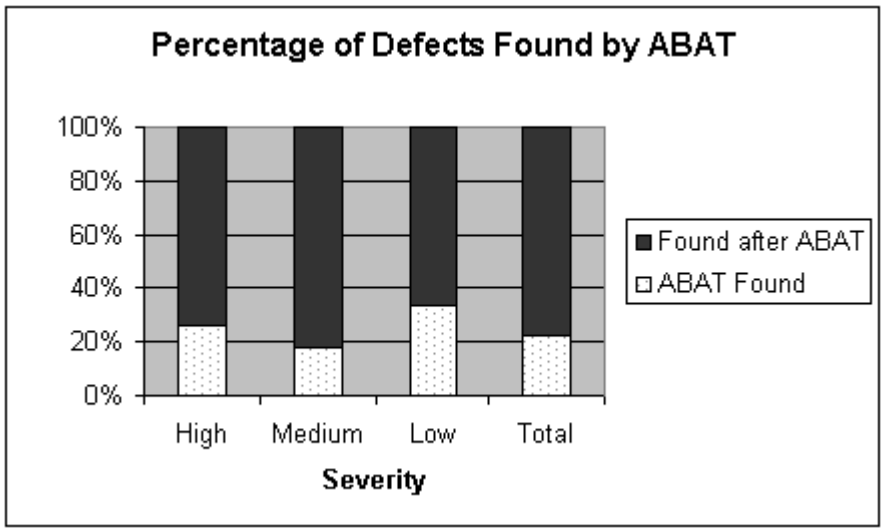


Figure 7 - ABAT Defect Data in Second Release



We didn't start using ABAT until the middle of the first release of our software. Nonetheless, we still saw significant benefits to using ABAT. ABAT detected nearly one-fifth of all of the defects present in the software, including one-fifth of the high-priority defects. The second release of our software (with ABAT in place) is currently in progress, and, again, we are seeing similar benefits as those we experienced during the first release. Slightly over one-fifth of all the defects present are being caught by ABAT,

including one-fourth of the high-priority defects. This data is shown in **Figure 6** and **Figure 7**.

Downloading ABAT Source Code

The ABAT system is an open-source project, and its source code is available under the BSD license. The current home for the ABAT project is on SourceForge at http://abat.sourceforge.net. Source code for ABAT, along with a README file containing more specific setup information, is currently available from SourceForge. If the reader would like more information or would like to contribute to the ABAT project, he or she should contact the ABAT contact person listed on the ABAT SourceForge website.

Conclusion

The ABAT system helped our group find defects, especially severe defects, before those defects made it into the hands of multiple testers, thereby saving our group time, money, and frustration. We found the ABAT system to be a good solution in determining whether a build was acceptable or unacceptable for further testing. As motivation for implementing an ABAT system, we illustrated some of the benefits our group experienced in implementing the ABAT system in our test environment.

This paper discussed the basic features and design of the ABAT system, as well as provided information on how to customize the ABAT system to meet the individual needs of an organization. Additional information and source code for the ABAT system is available from the ABAT Open Source website.

Monkey Testing Revisited
Using Automated Stress Testing

Praveen Hegde, Perry Hunter, Feng Liang, Doug Reynolds
Tektronix, Inc.
P.O. Box 500, M/S 39-732
Beaverton, Oregon 97077
praveen.hegde@tek.com; perry.w.hunter@tek.com;
feng.liang@tek.com; douglas.f.reynolds@tek.com

Abstract

Have you ever thought about paying people off the street to come in and press buttons and turn knobs as a way of stress testing your user interfaces? We have joked about bringing our kids in and letting them do this. As impractical as this may be, we know that we need to test our code in a variety of ways to ensure that we have done our best to find defects before our customers do. Though this practice of manual stress testing is still being used, it has significant weaknesses. The primary weakness is once a problem is found; it is difficult or impossible to reproduce the defect because the tester was not following a pre-defined sequence of events.

This paper describes what stress testing means, what some of the benefits and weaknesses are of automating stress testing, how we implemented automated stress testing to augment other testing strategies, and what some of the lessons learned are with our experience of using automated stress testing. Our Graphical User Interface stress testing has been instrumental not only in finding problems within our software applications but also in the third party tool used to perform the stress testing.

Biographies

Praveen Hegde is a graduate of India's Govt. BDT College; he holds a degree in Electronics & Communication Engineering. He has worked for the past 5 years within the areas of Safety & Mission Critical Software Testing, GUI Testing, and Embedded Systems Testing.

Perry Hunter is a Software Quality Engineer with Tektronix Inc., living in Portland Oregon. He is a graduate (1987) of California's Humboldt State University and has worked in software development and test in a variety of roles for approximately 15 years.

Feng Liang graduated from ASU (Arizona State University, Tempe) in 1998 with his Master of Computer Science degree. He worked at Tektronix Inc. as Software Quality Engineer for the past 3 years. His responsibilities include automated testing and software process improvement.

Doug Reynolds is a Software Quality Engineer at Tektronix with the Instrument Business Unit. He has worked for Tektronix for the past 10 years. He is currently working on a Master of Computer Science and Engineering degree at Oregon Health & Science University (OHSU).

Introduction to Stress Testing

In our organization, software testing parallels the entire software development cycle. This testing is accomplished through reviews (product requirements, software functional requirements, software designs, code, test plans, etc.), unit testing, system testing (also known as functional testing), expert user testing (like beta testing but in-house), smoke tests, etc. All these 'testing' activities are important and each plays an essential role in the overall effort but, none of these specifically look for problems like memory and resource management. Further, these testing activities do little to quantify the robustness of the application or determine what may happen under abnormal circumstances. We try to fill this gap in testing by using stress testing.

Stress testing can imply many different types of testing depending upon the audience. Even in literature on software testing, stress testing is often confused with load testing and/or volume testing. For our purposes, we define stress testing as **performing random operational sequences at larger than normal volumes, at faster than normal speeds and for longer than normal periods of time as a method to accelerate the rate of finding defects and verify the robustness of our product**.

Stress testing in its simplest form is any test that repeats a set of actions over and over with the purpose of "breaking the product". The system is put through its paces to find where it may fail. As a first step, you can take a common set of actions for your system and keep repeating them in an attempt to break the system. Adding some randomization to these steps will help find more defects. How long can your application stay functioning doing this operation repeatedly? To help you reproduce your failures one of the most important things to remember to do is to log everything as you proceed. You need to know what exactly was happening when the system failed. Did the system lock up with 100 attempts or 100,000 attempts?[1]

Note that there are many other types of testing which have not mentioned above, for example, risk based testing, random testing, security testing, etc. A good review of different testing types is provided by Cem Kaner and James Bach[2]. We have found, and it seems they agree, that it is best to review what needs to be tested, pick multiple testing types that will provide the best coverage for the product to be tested, and then master these testing types, rather than trying to implement every testing type.

Some of the defects that we have been able to catch with stress testing that have not been found in any other way are memory leaks, deadlocks, software asserts, and configuration conflicts. For more details about these types of defects or how we were able to detect them, refer to the section 'Typical Defects Found by Stress Testing'.

Table 1 provides a summary of some of the strengths and weaknesses that we have found with stress testing.

Table 1 Stress Testing Strengths and Weaknesses	
Strengths	**Weakness**
Find defects that no other type of test would find	Not real world situation
Using randomization increase coverage	Defects are not always reproducible
Test the robustness of the application	One sequence of operations may catch a problem right away, but use another sequence may never find the problem
Helpful at finding memory leaks, deadlocks, software asserts, and configuration conflicts	Does not test correctness of system response to user input

Background to Automated Stress Testing

Stress testing can be done manually - which is often referred to as "monkey" testing. In this kind of stress testing, the tester would use the application "aimlessly" like a monkey - poking buttons, turning knobs, "banging" on the keyboard etc., in order to find defects. One of the problems with "monkey" testing is reproducibility. In this kind of testing, where the tester uses no guide or script and no log is recorded, it's often impossible to repeat the steps executed before a problem occurred. Attempts have been made to use keyboard spyware, video recorders and the like to capture user interactions with varying (often poor) levels of success.

Our applications are required to operate for long periods of time with no significant loss of performance or reliability. We have found that stress testing of a software application helps in accessing and increasing the robustness of our applications and it has become a required activity before every software release. Performing stress manually is not feasible and repeating the test for every software release is almost impossible, so this is a clear example of an area that benefits from automation, you get a return on your investment quickly, and it will provide you with more than just a mirror of your manual test suite.

Previously, we had attempted to stress test our applications using manual techniques and have found that they were lacking in several respects. Some of the weaknesses of manual stress testing we found were:

1. Manual techniques cannot provide the kind of intense simulation of maximum user interaction over time. Humans can not keep the rate of interaction up high enough and long enough.
2. Manual testing does not provide the breadth of test coverage of the product features/commands that is needed. People tend to do the same things in the same way over and over so some configuration transitions do not get tested.
3. Manual testing generally does not allow for repeatability of command sequences, so reproducing failures is nearly impossible.

4. Manual testing does not perform automatic recording of discrete values with each command sequence for tracking memory utilization over time – critical for detecting memory leaks.

With automated stress testing, the stress test is performed under computer control. The stress test tool is implemented to determine the applications' configuration, to execute all valid command sequences in a random order, and to perform data logging. Since the stress test is automated, it becomes easy to execute multiple stress tests simultaneously across more than one product at the same time.

Depending on how the stress inputs are configured stress can do both 'positive' and 'negative' testing. Positive testing is when only valid parameters are provided to the device under test, whereas negative testing provides both valid and invalid parameters to the device as a way of trying to break the system under abnormal circumstances. For example, if a valid input is in seconds, positive testing would test 0 to 59 and negative testing would try –1 to 60, etc.

Even though there are clearly advantages to automated stress testing, it still has its disadvantages. For example, we have found that each time the product application changes we most likely need to change the stress tool (or more commonly commands need to be added to/or deleted from the input command set). Also, if the input command set changes, then the output command sequence also changes given pseudo-randomization.

Table 2 provides a summary of some of these advantages and disadvantages that we have found with automated stress testing.

Table 2
Automated Stress Testing Advantages and Disadvantages

Advantages	Disadvantages
Automated stress testing is performed under computer control	Requires capital equipment and development of a stress test tool
Capability to test all product application command sequences	Requires maintaince of the tool as the product application changes
Multiple product applications can be supported by one stress tool	Reproducible stress runs must use the same input command set
Uses randomization to increase coverage; tests vary with new seed values	Defects are not always reproducible even with the same seed value
Repeatability of commands and parameters help reproduce problems or verify that existing problems have been resolved	Requires test application information to be kept and maintained
Informative log files facilitate investigation of problem	May take a long time to execute

In summary, automated stress testing overcomes the major disadvantages of manual stress testing and finds defects that no other testing types can find. Automated stress testing exercises various features of the system, at a rate exceeding that at which actual end-users can be expected to do, and for durations of time that exceed typical use. The automated stress test randomizes the order in which the product features are accessed. In this way, non-typical sequences of user interaction are tested with the system in an attempt to find latent defects not detectable with other techniques.

To take advantage of automated stress testing, our challenge then was to create an automated stress test tool that would:

1. Simulate user interaction for long periods of time (since it is computer controlled we can exercise the product more than a user can).
2. Provide as much randomization of command sequences to the product as possible to improve test coverage over the entire set of possible features/commands.
3. Continuously log the sequence of events so that issues can be reliably reproduced after a system failure.
4. Record the memory in use over time to allow memory management analysis.
5. Stress the resource and memory management features of the system.

Typical Defects Found by Stress Testing

At Tektronix we have found that stress testing has been successful at finding asserts, memory leaks, deadlocks, and resource problems, etc. We have found different techniques for identifying and reproducing these defects. The following section defines these types of defects.

A program application assertion occurs when something abnormal in the application occurs that the software designer believes should not happen. Typically when a program assertion occurs the program halts and no further operations may be executed. An assertion is a statement that is considered to always be true. It is this statement that is used as a check against the code to demonstrate program consistency. An assertion statement will not be executed under normal circumstances.

A program application deadlock occurs when two processes are holding resources that each other require. A deadlock situation is not likely to occur under normal circumstances. However, putting the software application under stress is likely to cause a deadlock to occur. A deadlock usually manifest itself by no longer executing commands and/or overflowing input/output buffers as new command requests continue to be exercised.

A program application memory leak is the gradual loss of available memory when the program application repeatedly fails to return memory that it has obtained for temporary use. As a result, the available memory for that application becomes exhausted and the program application begins to slow down or no longer functions. For a program that is frequently opened or called or that runs continuously, even a very small memory leak can eventually cause the program to begin to slow down or terminate. A memory leak is the result of a program defect.

Other resource problems can also occur, for example what happens when the hard disk becomes full? How does the system react under this condition? We have found that there are many other resource problems that can occur within a system. Many of these resource problems can cause adverse effects like page faults, core dumps, etc. which cause the program to terminate unexpectedly.

Requirements for an Automated Stress Test Tool

There are many requirements that could be added to an automated stress test tool but the following is a list of the essential and desired requirements for an automated stress tool. Though some of these requirements have been mentioned before, we will re-hash them here for completeness and further describe some of the necessary attributes for each of the requirements.

Essential requirements of an Automated Stress Test Tool include:

- **Reproducible Command Sequences:** The stress test tool should be able to produce a random yet repeatable series of command instructions. This could be done using a pseudo-random generator, which uses a unique seed value to create a unique sequence of command instructions to be executed.
- **Command Sequence Logging:** Each action executed by the system should be logged in a sequential file, ideally, the file can be played back in whole or in part to reproduce any problems that occurred during the test. The file should also be able to track the seed number and the number of commands executed.
- **Memory and Resource Utilization Monitoring:** At discrete intervals during the automated stress testing, values representing the free and allocated memory and other resources available in the system should be recorded. This allows for memory leaks and resource depletion issues to be identified.
- **Fault Tolerance:** The automated stress test should be able to handle minor failures when exercising the product application. The automated stress test tool should be able to gracefully handle the fault situation, log messages as needed and continue with the test without significant reduction in the rate at which commands are sent to the system, and without itself impacting the available memory, resources or ability of the DUT to respond to simulated user inputs. On the other hand, if a deadlock condition occurs, then the stress test should stop.

Desirable requirements of an Automated Stress Test Tool include:

- The capability to pause the stress test after a set number of command sequences. This allows the checking of status or provides the ability to run the stress test to just before the point where the DUT fails where the remaining operation(s) may be performed in a single step mode or entered by hand while the system is being debugged.
- The capability to set a delay time between command sequences to support different application speeds, i.e. accessing pop-up menus on one application may take longer than another application.

- The capability to increment the pseudo random seed number every set number of command sequences and store the state of the DUT, for example the stress test could run for 500 command sequences, save the state of the DUT, increment the seed number and then run for another 500 commands sequences.
- The capability to execute the command set sequentially (this is mainly used for debugging purposes which can be used to check that the command set is entered correctly and are valid).
- The capability to generate feedback while the test is running so the state of the test can be checked at anytime (test application identifier, number of commands executed, runtime, current seed value, etc).
- The capability to randomize input values within a command. For example, if a command accepts a parameter from 1 to 4 then any of the numbers 1, 2, 3, or 4 should be selectable in a pseudo-random fashion.
- The capability to recognize the test application configuration and make adjustments accordingly.

Given these requirements for an automated stress test tool, our goal was to develop tools that could be used to augment our existing test methods, fulfill the stated requirements and fill the gaps to allow us to provide products that meet or exceed our customers' expectations.

Automated Stress Testing Implementation

Automated stress testing implementations will be different depending on the interface to the product application. At Tektronix, the types of interfaces available to the product drive the design of the automated stress test tool. The interfaces fall into two main categories:

1) **Programmable Interfaces:** Interfaces like command prompts, RS-232, Ethernet, General Purpose Interface Bus (GPIB), Universal Serial Bus (USB), etc. that accept strings representing command functions without regard to context or the current state of the device.
2) **Graphical User Interfaces (GUI's):** Interfaces that use the Windows model to allow the user direct control over the device, individual windows and controls may or may not be visible and/or active depending on the state of the device.

At Tektronix, many of our products also have front panels with actual buttons and knobs. Many of these products also have 'backdoor' programmable access and/or alternative interfaces (like USB), which simulate the actual button pushes and knob turnings. Granted this is not quite the same as actually pushing the mechanical button or twisting knobs but we are able so simulate these and test the software's reactions.

Programmable Interfaces

Tektronix has been using programmable interfaces for many years (starting in the 1970's). These interfaces have allowed users to setup, control, and retrieve data in a variety of application areas like manufacturing, research and development, and service. To meet the needs of these customers, the products provide programmable interfaces, which generally support a large

number of commands (1000+), and are required to operate for long periods of time, for example, on a manufacturing line where the product is used 24 hours a day, 7 days a week. Testing all possible combinations of commands on these products is practically impossible using manual testing methods.

Programmable interface stress testing is performed by randomly selecting from a list of individual commands and then sending these commands to the device under test (DUT) through the interface. If a command has parameters, then the parameters are also enumerated by randomly generating a unique command parameter. By using a pseudo-random number generator, each unique seed value will create the same sequence of commands with the same parameters each time the stress test is executed. Each command is also written to a log file which can be then used later to reproduce any defects that were uncovered.

For additional complexity, other variations of the automated stress test can be performed. For example, the stress test can vary the rate at which commands are sent to the interface, the stress test can send the commands across multiple interfaces simultaneously, (if the product supports it), or the stress test can send multiple commands at the same time.

Graphical User Interfaces

In recent years, Graphical User Interfaces have become dominant and it became clear that we needed a means to test these user interfaces analogous to that which is used for programmable interfaces. However, since accessing the GUI is not as simple as sending streams of command line input to the product application, a new approach was needed. It is necessary to store not only the object recognition method for the control, but also information about its parent window and other information like its expected state, certain property values, etc. An example would be a 'HELP' menu item. There may be multiple windows open with a 'HELP' menu item, so it is not sufficient to simply store "click the 'HELP' menu item", but you have to store "click the 'HELP' menu item for the particular window". With this information it is possible to uniquely define all the possible product application operations (i.e. each control can be uniquely identified).

Additionally, the flow of each operation can be important. Many controls are not visible until several levels of modal windows have been opened and/or closed, for example, a typical confirm file overwrite dialog box for a 'File->Save As…' filename operation is not available until the following sequence has been executed:

1. Set Context to the Main Window
2. Select 'File->Save As…'
3. Select Target Directory from tree control
4. Type a valid filename into the edit-box
5. Click the 'SAVE' button
6. If the filename already exists, either confirm the file overwrite by clicking the 'OK' button in the confirmation dialog or click the cancel button.

In this case, you need to group these six operations together as one "big" operation in order to correctly exercise this particular 'OK' button.

Data Flow Diagram

A stress test tool can have many different interactions and be implemented in many different ways. Figure 1 shows a block diagram, which can be used to illustrate some of the stress test tool interactions. The main interactions for the stress test tool include an input file and Device Under Test (DUT). The input file is used here to provide the stress test tool with a list of all the commands and interactions needed to test the DUT.

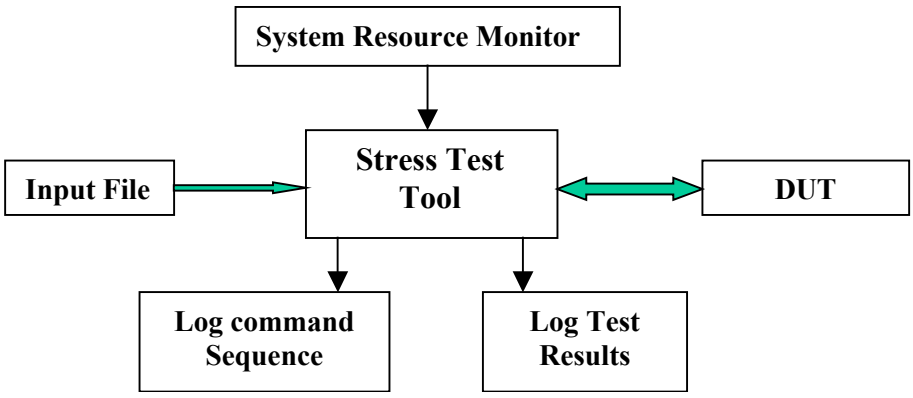


Figure 1: Stress Test Tool Interactions

Additionally, data logging (commands and test results) and system resource monitoring are very beneficial in helping determine what the DUT was trying to do before it crashed and how well it was able to manage its system resources.

The basic flow control of an automated stress test tool is to setup the DUT into a known state and then to loop continuously selecting a new random interaction, trying to execute the interaction, and logging the results. This loop continues until a set number of interactions have occurred or the DUT crashes. The following c-code is an example of the basic structure of an automated stress test tool.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

        long    seed    = 1;                    // Initialize random number seed

short random_number (void) {
        int     increment       = 1;            // Random number increment value
        int     multiplier      = 0x15a4e35L;   // Magic number for random number generator


        seed = multiplier * seed + increment; // Generate new seed value (i.e. random #)
        return ((short) (seed >> 16) & 0x7fff);
} // END random number generator



int main (int argc, char **argv) {
        char    buffer [5][32];

        int     command;                        // Interaction to select from
        int     commands;                       // Total interactions to choose from
        int     count           =  0;           // Number of commands executed
        int     total           = 10;           // Number of commands to execute
```

```
        // create dummy array of interactions
        commands = 5;
        sprintf (buffer[0],"%s", "do file close");
        sprintf (buffer[1],"%s", "do file delete");
        sprintf (buffer[2],"%s", "do file open");
        sprintf (buffer[3],"%s", "do file print");
        sprintf (buffer[4],"%s", "do file save");

        // initialize random number generator and DUT to a known state
        do {    // Loop until total interactions have occurred or instrument crashes
                // Generate which command to select
                command = (int)((((double)(random_number() * commands)) / 32767.0) + 1.0);

                count++;
                // Send command to interface and log to data file; check application crash
                // Here we will just use a printf statement
                printf ("Command %2d is:  %s\n", count, buffer [command -1]);

                _sleep (0);                     // Delay time between sending commands

                // If system resources are to be logged that could be done here
        } while (count < total);                // Only send total commands
}                                               // END of Example Program
```

This c-code example will output "do file …" in a pseudo-random sequence. If the seed value and command buffer are not changed then the program output sequence will be the same every time the program is run. If the seed value were to be changed, then the program output sequence will also change. Likewise, if commands are added to or deleted from the buffer, even the same seed will generate different sequences of commands.

Programmable Interface Stress Test Tool

Tektronix has been using a programmable interface stress test tool for many years. This tool has proven to be highly valuable in catching obscure faults in the DUT that would be difficult or impossible to find using other test methods. Things like assertion failures, memory leaks, pointer problems and the like have been detected and fixed where previous to the implementation of the tool, and these defects would likely have passed through the testing process. Indeed, the tool has uncovered defects that have existed in the field for some time, but are either obscure or so difficult to reproduce that users have never reported them.

The programmable interface stress test tool uses log files, which record the sequence of commands, applied to the product and have been necessary to analyze system failures since the failure may have actually occurred earlier (for example, a memory leak may be detected after a particular sequence of operations, or over a long period of time). One item that was not in our early versions of stress was the ability to track system resources like memory. This has since become a requirement.

The stress test tool command sequences should also be able to handle randomization of parameters within a command, for example if a command takes a parameter of type integer then the stress test should be able to select an integer value within a pre-defined range and reproduce the value on a subsequent re-test. There are many types of input parameters some numeric (like integers and real numbers), some are alphanumeric (like names, labels, etc), and some are enumerations (like selections). We also have also found that supporting macros and other complex algorithms has helped us instantiate random complex math expressions and the like.

The stress test tool should also have some idea on how to terminate on a failure. For the most part a timeout value can be used. The timeout value is the value in which the program application must respond before it is considered to have had a failure. Since each program application is different the timeout value is generally selectable at runtime.

Graphical User Interface Stress Test Tool

Though many of the requirements for GUI stress testing were identical to those for programmable interface stress testing, we found ourselves making design decisions based upon the capabilities of the third party test automation tool. Where our programmable interface stress test tool uses a flat-file listing of programmable interface commands, the GUI stress test tool uses various object recognition methods to identify and act upon each control in the GUI. Each control had its own context defined by the window that contained it, and therefore it was possible to have objects share non-unique object recognition methods. We decided that the best way to store and use this information was to use a database. With the information stored in the database, not only could a unique command sequence be generated but also unique parameters for the enumeration of buttons, checkboxes, or numeric/alphanumeric input fields.

We augment the third party test tool by creating our own library functions. For example, while the test is running, we use custom library functions to handle the output of the command sequences to one log file and the test results to another log file. At the same time, values for critical memory and resource values are captured using API functions and written to the test result log file. One advantage we have found to using our own logging functions is that we have control of these log files when the system crashes and do not lose data due to log database corruption in the third party test tool.

Lessons Learned with GUI Automated Stress Testing

This section points out some of the lessons we have learned from GUI based automated stress testing including tips for design and development, early stages of stress testing, problems found through test execution, and techniques used to isolate defects.

Design and Development

During the development of our GUI-based stress test tool, several issues arose that had not been fully considered at the outset. These were:

Object Recognition

The third party test automation tool could not recognize the custom tightly coupled (embedded) OCX controls in our product applications. This necessitated the development of a custom object recognition library that required significant effort to implement. We also learned that early and well thought-out effort to create an object-recognition map would pay large dividends in reduced maintenance throughout the project. A table (Object Recognition Map) could be used to assign unique recognition information for each object in the product application. Changes made to a single definition of an object can then propagate throughout the code with little or no effort.

A crud example of Object Recognition Map using Notepad® would be to add a layer of abstraction for the 'File->Save' operation. To do this the 'File->Save' operation would be like:

```
#define Notepad_File_Save "File->Save"
```

By de-referencing the 'File->Save' operation to Notepad_File_Save, if the 'File->Save' operation were to change then only the object recognition map would have to be changed. All the underlying test case code would remain the same since it is using Notepad_File_Save. This is very handy while the underlying GUI interface is still under development but the underlying product features are concrete.

Stress Input Data Source Control

The ability to go back and reproduce the same set of operations, using the same input file and seed value, must be maintained. Because of this, we have found that we must keep the input file/database in a configuration management system.

Inadequate Test Logs

The test log capability provided by the third party tool generated as output during script execution was inadequate for the examination of memory and resource values, executed command sequences and other data that would have been beneficial in analyzing the applications behavior during and after the test. Additionally, in the event of a test-tool related failure or system crash, the log file was left in an incomplete state, and at times was rendered inaccessible due to file corruption. Therefore, a custom set of logging functions was developed that collected the necessary data and wrote it off to external files for later analysis.

Early Stages of Stress Testing

Typically when we first start stress testing on a new DUT in the early stages of integration, the stress test may only run for a short period of time. To help isolate faults as each new subsystem is integrated into the application we, will run stress testing for that subsystem only. Once the subsystem becomes stable, we add the commands for that subsystem to the main stress data input file and run the integrated set of commands. This process continues until all subsystems have been individually added to the main stress data input file, allowing the integrated DUT to be fully tested.

The following graph in Figure 2 shows how the number of commands in a given stress test of the DUT increased over time. The flat region of the chart (< 1000 commands) roughly represents the period when integration build was adding new subsystems to the DUT, while the region in which the number of commands increases geometrically represents the period when the system was fully integrated and more difficult-to-detect issues were found.

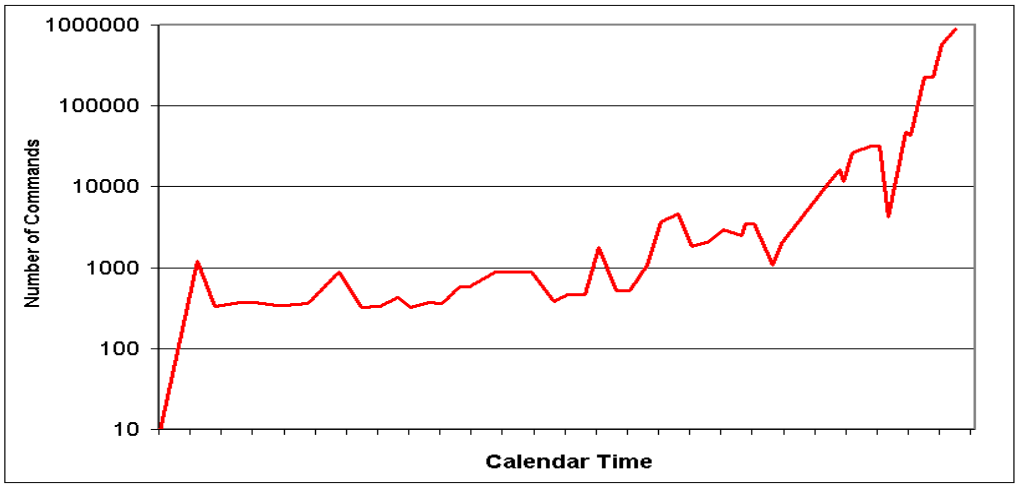


Figure 2: Application Stress Testing Over Time

Execution

Third Party Tool Memory Management Issues

During execution of GUI stress we found that the third party tool had serious memory management issues that would occur when the tests were run. We found that each time we would run our tests, the third party test tool would leave blocks of memory open causing increased swap file usage, which in time increased the time it took to execute a command sequence. Left unchecked, the system would hang or crash well before useful data from the DUT could be obtained. Also, the memory loss from within the third party tool tended to mask memory and resource management issues from within the DUT.

We isolated the memory problems within the third party tool by running them against known 'good' applications like Notepad®. We did this by creating an data input file for the application and then use the stress test tool to stress the application while monitoring the system resources. We found that each time certain stress test tool error conditions were encountered; it left blocks of memory open. We resolved the memory losses within the stress test tool by working with the vendor to fix the defects over time, however, great care needs to be taken to ensure that the stress test tool can work gracefully under the test conditions.

Test Execution Time was Variable

As the stress test runs for longer durations of time, it was noticed that the overall performance decayed (related to memory management issues mentioned above), which lead to increasing time for display of GUI objects. Given sufficient run length, the rate at which commands could be delivered to the system dropped well below what was possible for a normal user to generate. The situation was improved, but not entirely eliminated by implementing a "wait for object" structure in place of the original "wait for time delay" to prevent the script from overrunning the DUT application.

Techniques Used to Isolate Defects

Depending on the type of defect to be isolated, two different techniques are used:

1. System crashes – (asserts and the like) do not try to run the full stress test from the beginning, unless it only takes a few minutes to produce the defect. Instead, back-up and run the stress test from the last seed (for us this is normally just the last 500 commands). If the defect still occurs, then continue to reduce the number of commands in the playback until the defect is isolated.
2. Diminishing resource issues – (memory leaks and the like) are usually limited to a single subsystem. To isolate the subsystem, start removing subsystems from the database and re-run the stress test while monitoring the system resources. Continue this process until the subsystem causing the reduction in resources is identified. This technique is most effective after full integration of multiple subsystems (or, modules) has been achieved.

Some defects are just hard to reproduce – even with the same sequence of commands. These defects should still be logged into the defect tracking system. As the defect re-occurs, continue to add additional data to the defect description. Eventually, over time, you will be able to detect a pattern, isolate the root cause and resolve the defect.

Some defects just seem to be un-reproducible, especially those that reside around page faults, but overall, we know that the robustness of our applications increases proportionally with the amount of time that the stress test will run uninterrupted.

Future Improvements

We are continuously looking at ways to improve our stress test tools. Our GUI stress tool still relies on a third party application as the front end to perform object recognition. We are working on moving away from this dependency by developing an in-house stand-alone stress test tool. This will increase the number of platforms on which the stress test can be run since we will no longer be limited to those supported by the third party application.

Another area for improvement includes standardizing the data input file. Currently our programmable interface stress test tool uses a standard database file that works with all of our applications. We are now applying the same constraints to the GUI stress test tool. This will make it easier to support more applications with fewer stress test tool changes.

Last, we are considering a control-sequencing application that would recursively inspect the DUT to create our database. Currently, it takes significant effort to create a new database since all the control attributes must be entered by hand.

Summary

Our company has a heritage of delivering products of highest quality and reliability. One of the methods we use to ensure this high quality and reliability is through stress testing. Manual stress testing has proven to be very ineffective or impractical. We created an automated GUI stress test tool using a model similar to our existing programmable interface stress test tool that has been effective for many years. Automated GUI stress testing has proven to be an effective way to find elusive and difficult to reproduce defects that were not identified by any other means.

References

[1] Mark Fewster & Dorothy Graham, "Software Test Automation", Great Britain, Addison-Wesley 1999, pp548-550.

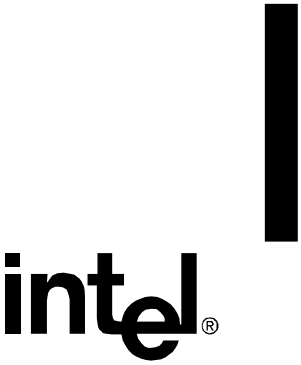
[2] Cem Kaner & James Bach, Software Testing, Analysis & Review Conference (STAR West), October, 1999, Reviews different approaches to the design of software tests.

Keywords

Monkey, Testing, Automated, Stress, GUI

Glossary

Command Set	A set of all commands to be exercised by the stress test tool.
Command Sequence	A sequence of commands generated from a command set using a pseudo-random generator.
DUT	Device Under Test. The application software/hardware to be tested.
GPIB	General Purpose Interface Bus, also referred to as IEEE Standard 488.
GUI	Graphical User Interface.
Load Testing	A type of testing which evaluates how will a system operates under extreme load conditions. Used to test the system capacity.
PI	Programmer Interface is a set of commands that are used to develop custom applications that interact with the DUT.
RS-232	Interface between Data Terminal Equipment and Data Communications Equipment Employing Serial Binary Data Interchange, also referred to as EIA Standard RS-232-C.
Stress Testing	A type of testing which is used to perform random operational sequences at larger than normal volumes, at faster than normal speeds and for longer than normal periods of time as a method to accelerate the rate of finding defects and verify the robustness of our product.
Volume Testing	A type of testing that is used to expose a system to large volumes of data.



Test Case Documentation and Results Tool

August 6, 2002, Jeffrey Robison, Pat Mead

Intel
Research &
Development

Test Case Documentation and Results Reporting Tool.

Abstract

With the current environment of cost cutting, we all have more to do with less time. It's critical in this environment to use tools wherever possible to make our jobs easier and show our progress to the team and management in a clear and timely manner. With this in mind we looked at the test case documentation, test results gathering, and reporting as key areas that could be made easier and better.

This paper will elaborate on how a "Microsoft * Excel" Add-In was created for formatting test cases and charts, how those cases are updated, maintained and managed through the life of a project, how the resulting charts are exported to a web site, what went into writing the Add-In, and some key learning's that were experienced along the way. It will detail the requirements imposed on the Add-In, how it is being used, and the future of it.

Biographies

Jeff Robison

Jeff is a Systems Quality Engineering Lead for Intel's research and development. Jeff graduated with a degree in Electrical and Computing Engineering from Clarkson College (now Clarkson University) in 1977. Jeff's experience has included real time embedded hardware and software design in the defense and avionics industries. Jeff has also been an independent software consultant for such companies as Compaq and Brooktree with expertise in testing and quality assurance. For the past six and a half years Jeff has been working for Intel primarily in software quality. Key areas of expertise include test design for device drivers, streaming audio and video codecs, and most recently in the wireless mobile environment. Jeff can be reached by email at jeff.d.robison@intel.com.

Patrick Mead

Pat is a Systems Quality Engineering Architect/Manager for Intel's research and development. Pat's background includes graduating with a Degree in Computer Science from Brigham Young University in 1985. While there he attended course work that included testing from Vern Crandall. This was one of the only University courses dealing with testing at the time. He has spent the past 18 years working in the field of Software Quality Engineering. The past 16 of that have been with Intel. His first assignment at Intel was working under Glen Myers, author of "The Art of Software Testing". Pat has worked on QA/QE from low level performance measuring tools, to compilers, to multimedia software, to Web based software (including e-commerce), to transaction processing systems. Key areas of expertise include test harness design/building and finding new ways to test new technologies.

Introduction

With the current environment of cost cutting, we all have more to do with less time. It's critical in this environment to use tools wherever possible to make our jobs easier and show our progress to the team and management in a clear and timely manner.

In our case we have a centralized Quality group supporting a multifaceted community that consists of multiple departments, multiple projects, multiple programming languages and multiple Operating Systems. So a key challenging requirement was that any tools we developed for assuring quality needed to be generic enough to be useful on any of these facets of the community.

With this in mind we looked at the test case documentation, test results gathering, and reporting as key areas that could be made easier and better.

It is important to be able to communicate efficiently and effectively the current status of any given project to the team and management at any time. Management and peers may measure the Quality team's competence by how well they report the status of a project as much as measuring them by how they obtain the results. Creating standardized reports and charts that are clear and concise is one way to accomplish this. This creates a common language and understanding among the various departments and projects. The problem becomes how does one repeatedly generate concise reports with the same look and feel so that management can easily and readily understand the progress and health of a project or projects? Further, as projects come and go how does one manage to maintain those reports so that they are consistent across projects and over time?

In most corporate environments, Microsoft Office is a standard tool used and understood by nearly everyone within the company. All of the Microsoft Office applications share a common scripting language, which can be used to perform tasks within applications. This scripting language is the Microsoft Visual Basic for Applications (VBA). Using VBA scripts or macros, a standard, automated, repeatable way of formatting test cases and generating reports can be written and used across all of the projects a Quality group manages.

Extending VBA macros into an Add-In provides a mechanism for extending a Microsoft Application's capabilities. Add-Ins are also easily distributed and installed on users machines enabling them to use this extension to the Office application.

One application of Microsoft Office is Excel. Excel is a spreadsheet application. Excel has built in support for formulas, graphing /charting, reporting, and among many other features, VBA. Using VBA to create an Excel Add-In provides the Quality group with an efficient and effective way of communicating to the various project teams. This extends excel functionality to automate the formatting of test cases, real-time updating of progress charts, and outputting this information to a web site.

This paper will elaborate on how an Excel Add-In was created for formatting test cases and charts, how those cases are updated, maintained and managed through the life of a project, how the resulting charts are exported to a web site, what went into writing the Add-In, and some key

learning's that we experienced along the way. It will detail the requirements imposed on the Add-In, how it is being used, and future improvements.

This paper assumes that the reader has some experience in testing activities, and some programming knowledge of Visual Basic and/or Visual Basic for Applications and will not go into depth on either of these topics. Helpful references will be provided.

Requirements

As is common with all projects, we had to decide what the requirements were needed for the tool. The following requirements were established.

- Requirements for Standardized reporting within the lab:
    - Management (staff) needs status as brief as possible – to grasp a big picture
    - Staff will see the same style of reporting and will come to understand the data/report
    - Dev. team may need more details - Detailed reporting of each test case
    - Report total number of tests
    - Report how many of the tests are completed (or percent complete)
    - Report status of the test cases
    - Report the results graphically making it easy to comprehend
        - Export graphs to web page
        - Export graphs to reports
- Requirements for Tool for test case writing:
    - Can organize test cases into groups or suites
    - Hierarchal tree to fit most projects
    - Must be able to capture
        - Title
        - Description
        - Steps
        - Comments
        - Time (Actual vs. Estimated)
            - Results (Pass, Fail, Blocked, Skipped, N/A)
    - Easily deployable
    - Easy to use and portable

Solution

We decided to use Excel as our solution because every tester in our environment has access to it, it is easy to learn, use and it provides the required reporting and charting capabilities. As with all Microsoft Office applications, Excel has VBA support, which enabled us to write an Add-In providing an easy way to deploy the tool to our testers.

Having written the tool, we next needed an easy way to distribute it to our testers. Using the Packaging and Deployment Wizard include in the Microsoft Office Developer's Kit, we created an installation file that included all the necessary components of our Add-In. Now we were able to give our testers a tool with a user interface that could be used to generate Excel spreadsheets with logically arranged test cases by functionality and devices without requiring any underlying knowledge of VBA.

A little explanation on what we mean by devices may be needed. To make the tool flexible, we refer to devices as anything that one may be testing – an object of sorts. Devices can be physical devices such as PDA's (Palm Pilot's and iPAQs), or hardware configurations (different processors, various amounts of RAM, etc.), or even Operating Systems (Window '98, Windows 2000, XP, etc.). The context of devices is left up to the user of the tool to logically group test cases per worksheet based upon devices. Each worksheet may have the same devices as other worksheets, or different ones, or a combination.

The tool first provides the user a dialog box (Figure 1) that allows the user to create their first sheet, in this case GUI, and add devices to that sheet. Clicking the "Add" button in this example would add the device Server to the Device List.

Figure 1

This dialog provides a quick method for creating the first sheet and subsequent sheets used in the test cases. Sheets are used to logically organize groups of related test cases by device or devices.

Once a sheet and the associated devices are defined, the user can start adding test cases with another dialog box by pressing the "OK" button that provides a tree view list (Figure 2). Figure 2 shows a tree view that has already been populated with sheets (GUI, Memory, Misc., Installation, etc.).

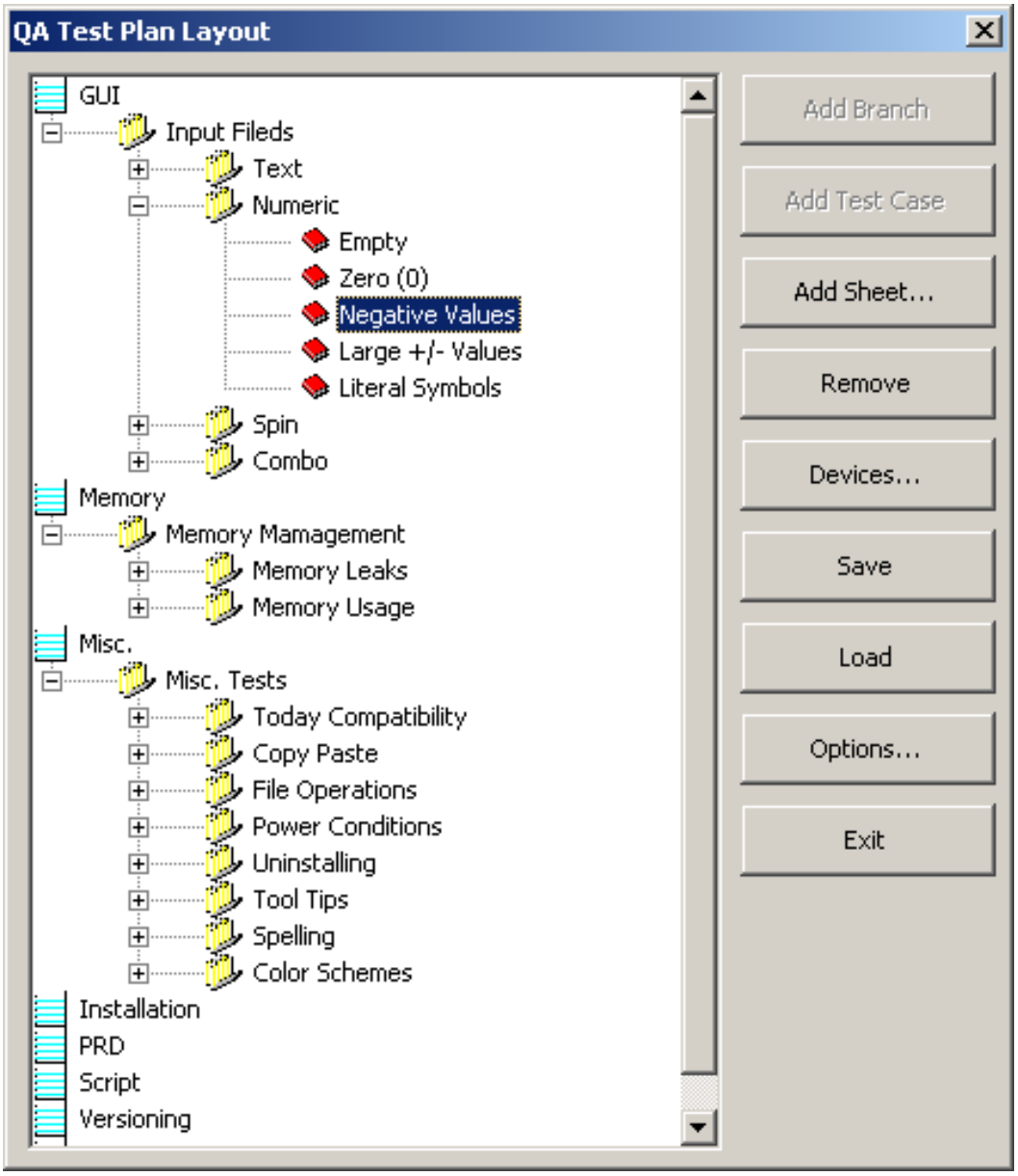


Figure 2

Using the tree view list the user can visually generate a hierarchy of test cases. The hierarchy starts at the sheet level and works its way down to major functions, such as 'Numeric' in sheet 'GUI', that can contain one or more sub-functions, such as 'Negative Values', which in turn can contain one or more test cases. This hierarchy provides a way for the user to logically layout and develop their test cases that can be used for nearly any project.

Devices associated with sheets are not shown in the tree list. To view, add or delete devices on a sheet, select the desired sheet, and click the "Devices…" button. This will bring up the dialog box in Figure 1.

Test cases themselves contain a title for the test, a description of the test, steps for executing the test, and the expected results (Figure 3).

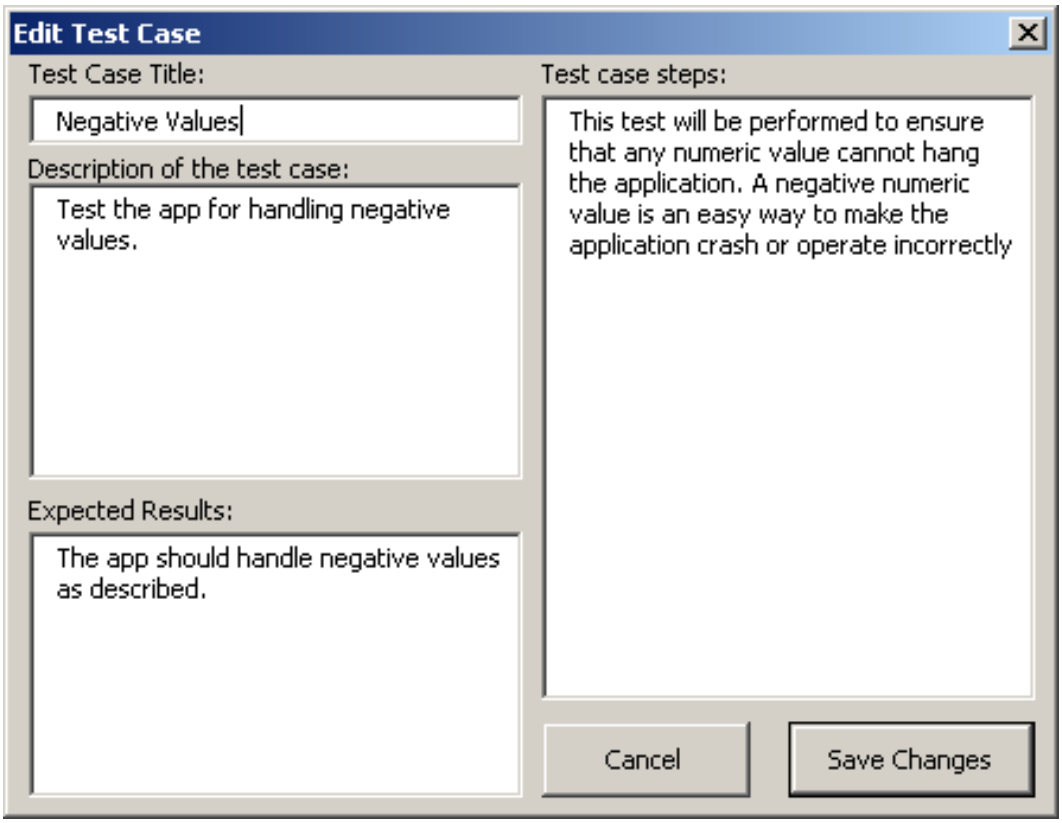


Figure 3

At any time the user can save their data to a hidden spreadsheet within the Excel workbook, save the workbook, close Excel and return to it at a later time and pick up where they left off. The user can also modify their data at any time using the tool.

When the user saves the data in the tree view, they are prompted if they wish to generate the test case documents and reports (Figure 4).

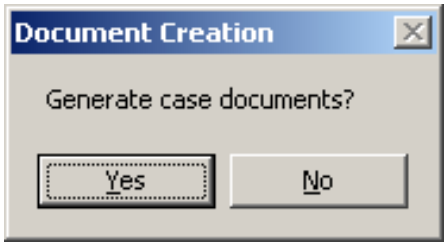


Figure 4

Selecting Yes to this option generates the test case documentation within the Excel workbook. Selecting No saves the information to the hidden spreadsheet without generating the test case documentation. In the case of selecting Yes, a separate sheet is created for each of the sheet names defined in the tree view. For each sheet the test cases are automatically formatted and generated. The test cases are formatted by major functions and by sub functions. Each test case is

automatically assigned a unique number to identify the test case, its title, description, execution steps and expected results filled in. A notes field and a field for each device associated with the sheet are generated. The device field is formatted with a drop down box with the possible state the test could be in. The states are Pass, Fail, Blocked, Skipped, and N/A. Also fields for estimated time and actual time for executing each test case are provided for optionally tracking test time. The last field generated is the final status of each test case (Figure 5). Figure 5 shows a sample of what the tool will generate. The sheet high lighted, PRD, are tests cases that were written against a Project Requirement Document, or PRD. This sheet has only one device associated with it, the Client device as seen in column H. Therefore this sheet only contains tests for the device, Client.

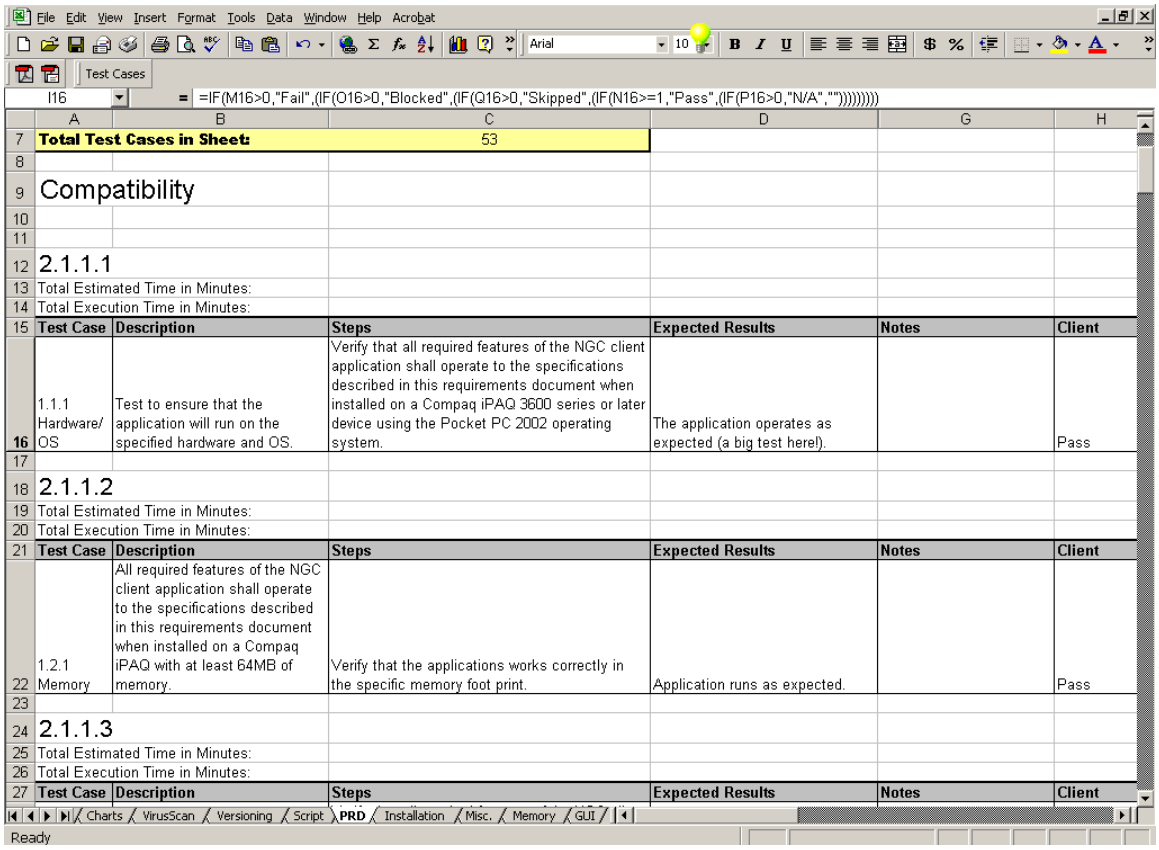


Figure 5

While performing the test cases, the tool can be used for real time reporting. As the test cases are run, the user can enter the results into the spreadsheets using the device's drop down field (Figure 6).

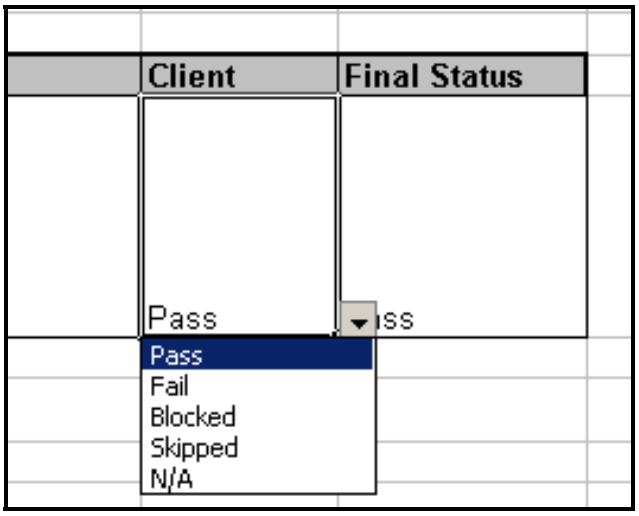


Figure 6

As this is done, the final status for each test case is automatically filled in using applied rules. In addition as the user enters the results, reports and graphs are automatically generated reporting execution time, percent of test cases completed, and pass/fail status for each device.

Figure 7 illustrates the graphs generated, pass/fail by device, and overall testing progress.

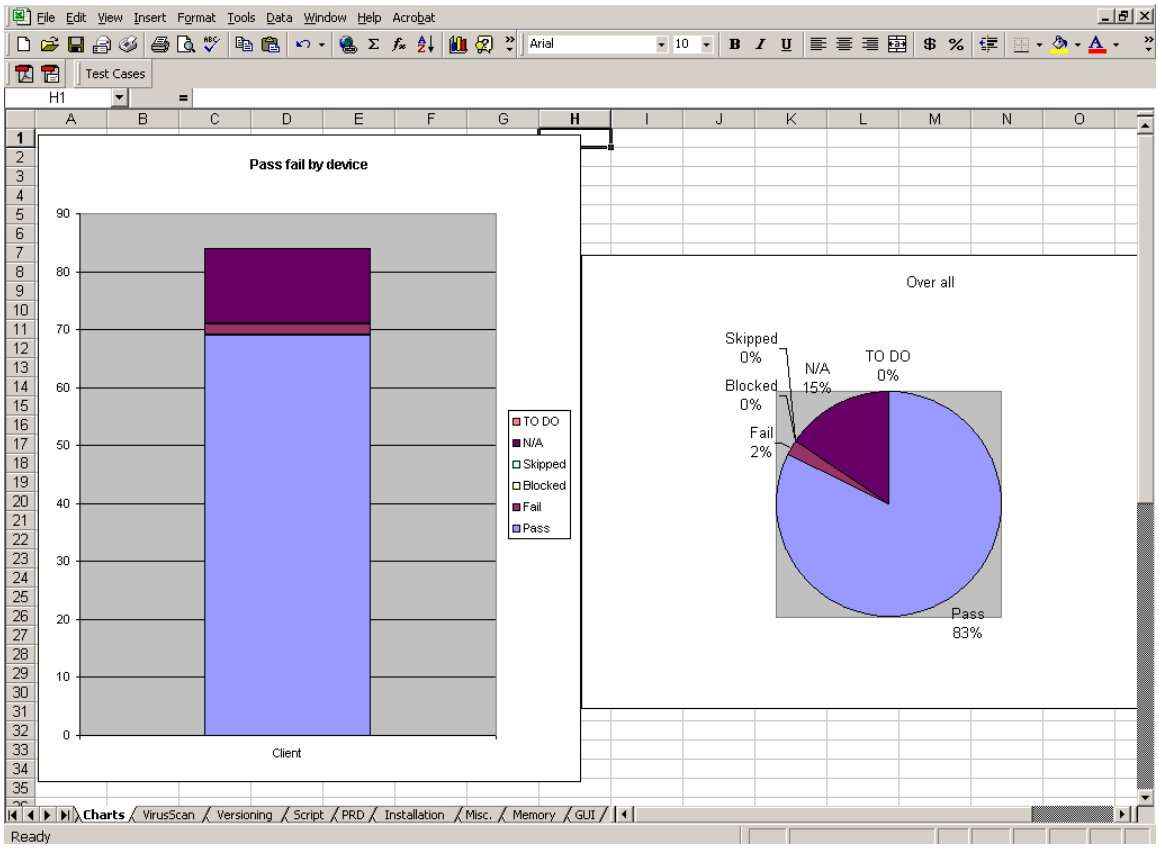


Figure 7

Using only two VBA statements, one can export a chart to a folder. Follows are the two statements to export a chart:

```
Set mychart = Charts(ChartName)
mychart.Export Filename := Chart_Folder & ChartName & ".gif", FilterName:="GIF"
```

The first statement sets the variable mychart to the Charts object (ChartName) to be exported. The second statement uses the Export method of the Charts object to save the chart named ChartName (string variable) to the folder Chart_Folder (also a string variable) in the GIF file format. Other file formats are supported including JPG.

The above lines of code could be used in the Auto_Close subroutine of the Excel workbook to automatically export the charts to a shared folder for a web site. The charts could then be referenced statically in a HTML file, enabling anyone with a web browser to view up to date status of the testing progress and results.

A summary report/sheet (Figure 8) is updated providing a breakdown of the execution time; tests ran for each device (in this case only one – Client), and a final status of all the test cases. This provides management a clear overview on the testing progress and results. More detailed reports are generated on other sheets that are more useful to the development teams. These detailed reports can be used to help identify such things as problem areas in testing. Note that in this report no time estimates have been entered for estimated or actual execution times, therefore time remaining is zero for all cases.

Time Status

Test Matrix	Tester Name	Build #	Test Start Date	Test End Date	Estimated Test Time	Executed Test Time	Remaining Time
VirusScan	Jeff Robison	0.0.98	4/8/2002		0.0	0.0	0.0
Versioning	Jeff Robison	0.0.98	4/8/2002		0.0	0.0	0.0
Script	Jason Landers	0.0.98	4/8/2002		0.0	0.0	0.0
PRD	Jason Landers/J	0.0.98	4/8/2002		0.0	0.0	0.0
Installation	Jason Landers	0.0.98	4/8/2002		0.0	0.0	0.0
Misc.	Jason Landers/J	0.0.98	4/8/2002		0.0	0.0	0.0
Memory	Jeff Robison	0.0.98	4/8/2002		0.0	0.0	0.0
GUI	Jason Landers/J	0.0.98	4/8/2002		0.0	0.0	0.0
TOTAL					0.0	0.0	0.0

Client

Test Matrix	Tester Name	Build #	Pass	Fail	Blocked	Skipped	N/A	To Do	Tota
GUI		0.0.98	2	1	0	0	11	0	
Memory		0.0.98	2	0	0	0	0	0	
Misc.		0.0.98	6	0	0	0	2	0	
Installation		0.0.98	4	0	0	0	0	0	
PRD		0.0.98	52	1	0	0	0	0	
Script		0.0.98	1	0	0	0	0	0	
Versioning		0.0.98	1	0	0	0	0	0	
VirusScan		0.0.98	1	0	0	0	0	0	
Total			69	2	0	0	13	0	

FINAL STATUS

Device	Pass	Fail	Blocked	Skipped	N/A	TO DO	Total
Client	69	2	0	0	13	0	84

Figure 8

Learning's

When we first started developing the tool we started out writing the VBA code directly into an Excel workbook. VBA code written in a workbook cannot be compiled into an executable binary, and therefore can become very large. In our case the code was in excess of 600KB. Also code that is not compiled runs slower than that of compiled code. Therefore we switched over to writing the VBA code in an Excel XLA Add-In. Now the code could be compiled making it smaller (approximately 284KB) and faster. Also we could use the Package and Deployment Wizard (a VBA add on included in the Office Resource kit) to create an installation package making it easy to distribute.

The learning curve was rather easy, as we already knew Visual Basic. The book "Microsoft Office 97 Visual Basic Programmer's Guide" (reference 6) turned out to be a great asset in minimizing the learning curve for VBA. The trickiest part was programming the Tree View Control. That is where Evangelos Petroutsos' article (reference 4) helped to clarify the mysteries of the tree view control.

Several characteristics distinguish an Excel Add-In from a typical workbook file:

- An Add-In has the file extension .xla to indicate that it is an add-in.
- When you save a workbook as an Excel Add-In, the Workbook window is made invisible and cannot be viewed. You can use the invisible Workbook and Worksheets for storing calculations or data that your Add-In requires while it is running.
- Users cannot use the SHIFT key to bypass events that are built into the Add-In. This feature makes sure any event procedures written in the add-in will run at the proper time.
- Excel messages (alerts) are not displayed by code running in an Add-In. In a standard Workbook file, messages appear to verify that the user wants to perform an operation that might result in data loss, such as deleting a Worksheet or closing an unsaved Workbook file. In an Add-In, such operations can be performed without the messages being displayed.
- Pay close attention to the context in which the code is running. When referencing the Add-In Workbook, use the **ThisWorkbook** property, or refer to the Workbook by name. To refer to the Workbook that is open in Excel currently, use the **ActiveWorkbook** property, or refer to the Workbook by name.

Future Enhancements

Future additions to the tool that would enhance it includes:

- Drag and Drop functionality for the tree control providing copy and paste methods.
- Add a description field for each sub function.
- A means to import and export the test cases.
- Support for multiple workbooks.
- A method of "rolling up" the results.
- Linking to Excel Workbook and Add-In to automated testing.

Summary

Faced with the task of providing management and development teams with clear, understandable, concise, up to date test case results and reports, we solved the problem by extending the capabilities of Excel by writing a VBA Add-In. The Add-In provided our testers an easier and more rapid way to create a hierarchy of test cases in a standardized format using an intuitive user interface of dialog boxes and tree views in a timelier manner than previously. The tool provides real time updating, also saving time from manually reformatting the reports and charts.

Both management and the development teams are now able to view the testing progress at both the high level and detailed levels each require. Comunication between the various teams are much more clear and concise that before. Less time is spent in meetings explaining the status of the testing, and more time is spent elsewhere as needed.

Overall the tool has helped our Quality team to more rapidly document test cases, report the status of testing, thus saving time that can be used on other aspects of Quality Engineering.

References:

1) Microsoft Office 2000/Visual Basic Programmer's Guide
http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000550

2) Microsoft Office XP Developer, Developing Office Applications Using VBA
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/modcore/html/deovrAddinsTemplatesWizardsLibraries.asp

3) Microsoft® Excel 2000/Visual Basic® for Applications Fundamentals
http://www.microsoft.com/mspress/books/2534.asp

4) Making the Most of the TreeView Control
http://www.devx.com/vb/free/articles/ep052102/ep052102-1.asp

5) Office Resource Kit
http://www.microsoft.com/office/ork/xp/default.htm

6) Microsoft Office 97 Visual Basic Programmer's Guide
http://www.amazon.com/exec/obidos/ASIN/1572313404/qid%3D1028656836/sr%3D11-1/ref%3Dsr%5F11%5F1/103-6219182-8987810

7) Microsoft Office 2000/Visual Basic : Programmer's
http://www.amazon.com/exec/obidos/ASIN/1572319526/qid%3D1028656804/sr%3D11-1/ref%3Dsr%5F11%5F1/103-6219182-8987810

Common Problems in Tool Adoption

Karen S. King
Quality Improvement Solutions

Abstract

Many organizations acquire new tools, and six months later, find they have shelfware rather than a tool that improves productivity. This paper describes eight common problems, along with examples based on experiences from real-life organizations. Examples will be from a variety of different tool acquisitions, such as defect tracking systems, configuration management, code coverage tools, testing tools, and project management tools. It continues to provide multiple solutions for each problem in a format readers can integrate into their own organizations. Finally, it provides a high-level process for introducing tools and describes where you'd apply each best practice.

Karen S. King, founder and principle consultant for Quality Improvement Solutions, has a special interest in effectively implementing tools to improve the software process. With twenty years experience in software quality improvement, she is widely known for her work in using defect metrics t o direct process improvement initiatives. She has participated in the software development process in a variety of roles, including: systems programmer, QA lead, applications engineer, supplier engineer, quality manager, IV&V lead, and SEPG chair. Ms. King can be reached at karenk@alumni.rice.edu

How many software tools do you have on the shelf that you do not use? How much money has your organization wasted, hoping to improve productivity? If you are like the typical company, the answer is more than you can measure. If you consider tools you use, but not to the extent intended, the number soars. Someone in your organization spent significant effort justifying the purchase of each of those tools, as well as integrating them into your organization. Why don't you use them? Here are stories from eight projects at Hilanderas International, a fictitious company experiencing problems common in real-life organizations. The stories explain how tools become shelfware, along with solutions.

1. Organization not Ready

The Project Jupiter managers heard about a tool to measure code quality. They were concerned about funds spent on maintaining their code, and thought they could decrease their maintenance costs by purchasing a code quality tool. Management established goals for each quality metric, and developers slavishly modified code to meet the numbers. Project Jupiter abandoned the tool when developers complained they could not meet schedule because they were modifying code to meet the goals. The problem was that no one had researched the metrics to understand how to effectively use the measurements.

Solutions:

A) Decide why you want the tool. Explicitly document the benefit to the organization. All users must understand "what's in it for me". If they see no benefit, they will not be interested in using the product.

B) Acquire tools to automate existing processes. If you already perform the function manually, you will simply improve an existing process and the acquisition will more likely be successful.

C) When acquiring tools for new practices, simulate the tool manually to better understand your needs. I once simulated new defect tracking fields by interviewing developers to manually collect data. By collecting the information, I identified useful data fields, and was able to manually calculate metrics that measured progress against our goals. Without collecting actual data, our team would have relied on the literature for data fields and metrics, and the tool would not have met our organizational needs.

D) Consider your organization's ability to assimilate new practices, and the current capability in the practice. Many organizations purchase tools with more capabilities than necessary, figuring they will "grow into it". It would be similar to buying an Indy racecar for a beginning driver, figuring that someday he might want the additional features.

2. Not All Stakeholders Involved in Introducing Tool

Tom has been studying the Capability Maturity Model, and believes a configuration management tool would solve many of the problems on Project Neptune. He found a tool, purchased it, and worked weekends to put the project's code under configuration management. Tom believed everyone would be eager to use the new system because it would provide a rigorous method for tracking past code releases. After the tool was in

place, Tom could not convince his colleagues to use it because they said it violated their personal development processes.

Solutions:

A) List job functions that could be affected by the tool. For a code coverage tool, developers might need to change their coding style, management might be involved in interpreting the metrics, system administrators must maintain the tool, and all testers must enable the tool when running tests. Ensure that you represent all functions in your tool adoption team.
B) Determine opinion leaders who understand the impact of the tool on their communities. These people are the stakeholders. If you involve opinion leaders, other people within the groups will be more likely to accept the new tool.
C) Work with stakeholders to collect requirements you have not anticipated. If you are selecting a defect tracking system, service might require customers be able to submit defect reports and receive limited information on the status of their report. If you only collect requirements from engineering, this higher-level requirement would never surface.
D) Consider the politics in your organization. If there are factions that could oppose your effort, talk to them about their concerns in adopting the new tool. Understand what it would take for them to support your effort. In Tom's case, he did not understand that his colleagues viewed their personal "code stashes" as power, and that they were resentful about Tom's efforts to consolidate the code.

3. Wrong Tool Selected

Jean was tasked to select a defect tracking system for project Apollo. She assembled a team of stakeholders and collected seventy requirements from the team. After researching products, Jean found a tool that met fifty of the requirements, the highest number of all available products. She purchased that tool, but later found the tool would not support custom business rules, and users could not get email when new defects were assigned. Users would not accept the tool because it did not support their business processes.

Solutions:

A) Conduct complete research on the strengths and weaknesses of potential products. One major source of shelfware is acquiring tools at the end of the year, and making a quick decision on a tool because otherwise you'll lose funding. If you acquire a tool without doing the homework, it is almost guaranteed the tool will become shelfware. It would be better to lose funding for a risky tool, rather than have to justify buying a second tool.
B) Rank your requirements. Rather than producing a list of equally weighted requirements, work with your stakeholders to distinguish the most critical, "must-have" features from the "nice to haves". A prioritized list will prevent your selecting a product that meets 90 percent of the requirements, but does not address a critical need.

C) Meet with representatives of the tool vendors and have them demonstrate their tool in your environment. For a code coverage tool, a demonstration might include running some tests in your standard work environment, and verifying the product produces the expected results.
D) Get a demo copy of the tool or a short-term license. Before bringing the tool in-house, have a plan in place on how you'll evaluate the tool, and have resources available to do the evaluation. Determine your minimum criteria to acquire the product, and do not be pressured into purchasing the product until you are satisfied the tool meets the minimum criteria.
E) If no tool meets all your critical requirements, consider customization. You might be able to negotiate with the vendor to customize the tool or customize the tool yourself. It's less expensive in the long run to acquire a customized tool that meets your requirements, rather than acquire an off-the-shelf tool that does not meet your needs, and will likely become shelfware.

4. Tool not Integrated into the Process

Al acquired a tool to collect code metrics. He always ran the metrics tool before he integrated his code. Mary ran metrics after she'd completed unit testing. Project Saturn management noticed the highly divergent data, and decided the code metrics were meaningless for managing projects. No one understood that differing processes caused the data disparity, rather than quality of the tool.

Solutions:

A) If you have process documentation, determine how the tool will best fit within your established development and quality processes. If the tool is complex, you might include a separate work instruction for using the tool and modify the main process to reference the new work instruction. If the tool is straightforward, you could simply modify the current process documentation to include the new tasks associated with the tool. Each function should examine their processes and coordinate changes.
B) If you do not have process documentation, at a minimum, draw a high-level flowchart of your development process. Mark where the tool will be used, and who will be using it. Then describe steps for using the tool. Even without other written process documentation, rudimentary documentation on using the tool increases successful tool adoption. Additionally, after several tools have been introduced and documented, your organization might see the benefit of documenting other development processes.

C) Make the tool documentation easily accessible. Consider providing a quick reference card or online process help, so users can easily find information about using the tool in the context of your process.

5. Inadequate Training

Joe acquired an estimation tool for the Minerva division. He trained all project managers at once so they would use the tool consistently. Joe tracked when the tool was used, and found project managers were not using the tool consistently. The problem became more

pronounced over time. The data in the estimation database became unusable because there was too much variance.

Solutions:

A) Train users "just in time". If they are trained before they need to use the tool, they might not see the relevance of the training (it's not what I'm doing now), and they will not effectively learn. Even if they do learn, they will probably forget their training by the time they need to use it.
B) Provide hands-on training. People need to try new ideas before they can accept them. Create an instance of the tool where users can safely experiment without corrupting live data. For example, when implementing a defect database, have a test database where users can enter defect data and route defects among users without affecting project metrics.
C) Develop training for new employees. It is not uncommon to give new employees documentation and tell them to figure out the tool themselves. Slightly better is when a current user demonstrates the tool to the new employee. Often only less experienced users have time to demonstrate the tools, and the training is inconsistent.
D) Have tool experts available on call to help users with any problems. Users will be more likely to use the tool if they know there are resources available to help.

6. No Follow-up on Tool Deployment

Project Arachne purchased a GUI test tool. The quality assurance engineers wrote tests and developed automated test scripts. About a year later, one of the quality assurance engineers was out during a verification cycle. Their manager assumed the test cases were run automatically, and asked to see the test report. There was no test report because tests had been run manually over the past year. When questioned, the QA staff said the tests were run manually because it took more time to analyze the results from the automated scripts than running the tests manually. No one had questioned the effectiveness of the tool, and the tool became shelfware without management's knowledge.

Solutions:

A) Schedule periodic times to talk with users about how the tool is working. Collect information on positives, negatives, and opportunities for improvement. Implement suggestions that will improve the system.
B) Develop metrics to monitor tool usage. For a configuration management tool, you might want to monitor number of check-ins, average size of check-ins, number of code branches, etc. for significant changes. If the metrics vary from past history (positively or negatively), talk with the users to understand why. Do not use the metrics to make judgements about people.
C) Set expectations on the results you expect in the tool adoption. In the short term, the tool will probably not improve productivity. You will not see results until the organization is able to use the tool effectively, which will take multiple experiences. Many companies have abandoned tools before the organization mastered the new tool, and the result is extreme frustration for everyone.

7. No Resources for Adopting Tool

Bob was working on a critical project on a tight schedule. He was concerned about keeping the project on track, and convinced Hilandares to acquire a project tracking tool. Bob asked his team to use the tool to track their progress on the project. At the same time, there was extreme pressure on the team to complete the project as soon as possible. The team was overwhelmed, and figured they would enter status information when they had time available. As a result, Bob was not able to use the tool effectively because status information was not up to date. Bob soon became frustrated, and stopped using the tool.

Solutions:

A) Rather than using "volunteer time", work with a sponsor to get a time commitment, preferably from an improvement resource, as well as the people on the tool adoption team. If possible, try to get dedicated resources. If that isn't possible (project is small or organization is small), at a minimum, get project relief for team members. Set realistic expectations about the amount of time required to work on the tool adoption project. In Bob's situation, his highest priority was getting the product out on time. Although he had strong incentives to adopt the project tracking tool, it was a "spare time" activity. If Bob had help from an outside improvement resource, there would have been more effort available to deal with the user resource issues.

B) Before acquiring the tool, ensure all users will have time to learn the new tool. If users are already overloaded, look for ways to offload some of their work to ensure they can use the tool.

8. No Plan for Maintaining Tool

Susie worked with customer service to acquire a customer complaint system. Through detailed interviews with customers, Susie found the most critical aspect was usability, and Hilanderas' customers wanted a tool with pulldowns to easily select answers. Susie and her team implemented the system, and the customers were very pleased. Shortly afterwards, the number of complaints logged in the system decreased drastically. Hilanderas management believed they were now more responsive to their customers' complaints, which resulted in happier customers. Instead, the salespeople reported the customers were frustrated because they could not report their actual complaints because the selections on the pulldowns were out of date. Although the product offerings had changed, the product names in the pulldowns had not been updated. No one had been assigned to maintain the system after roll out, and what had been designed as a user-friendly system quickly became unusable.

Solutions:

A) Develop a maintenance plan at the beginning of the project. Be sure to provide enough resources to answer user questions, as well as make necessary changes to the system. Update the maintenance plan as the team learns more about the final solution.

B) Include extra maintenance time throughout roll out and into production. The new system is almost never perfect, and users will need help in successfully transitioning to the new system. Also, users always have suggestions on ways to improve the

system. It is important to be responsive to new ideas to help users accept the new system.

C) If your system is heavily customized to your environment, you will need more maintenance resources to manage those customization. If your work environment changes frequently, your tool might also require frequent changes to remain current.

Putting it All Together

Now you have ideas about improving the way you adopt tools, but how do you put it all together? It's helpful to have a collection of best practices, but the next step is to determine how the practices can be incorporated into your process. Do you currently have a process for adopting tools? If so, do you follow it? This section includes a high-level process for tool acquisition you can use as an outline to define your own specific process.

Typical Tool Adoption Process

Figure A shows a basic tool adoption process that you can tailor for your organization. Customize this process to include your specific steps. When developing the process, include guidelines for determining ways to tailor the process for different tools. For example, you might not want to follow all steps in the process when adopting a simple tool that all users accept. When customizing processes, consider the consequences of failure. If critical business processes are dependent on successfully adopting the tool, it is more important to follow the complete process.

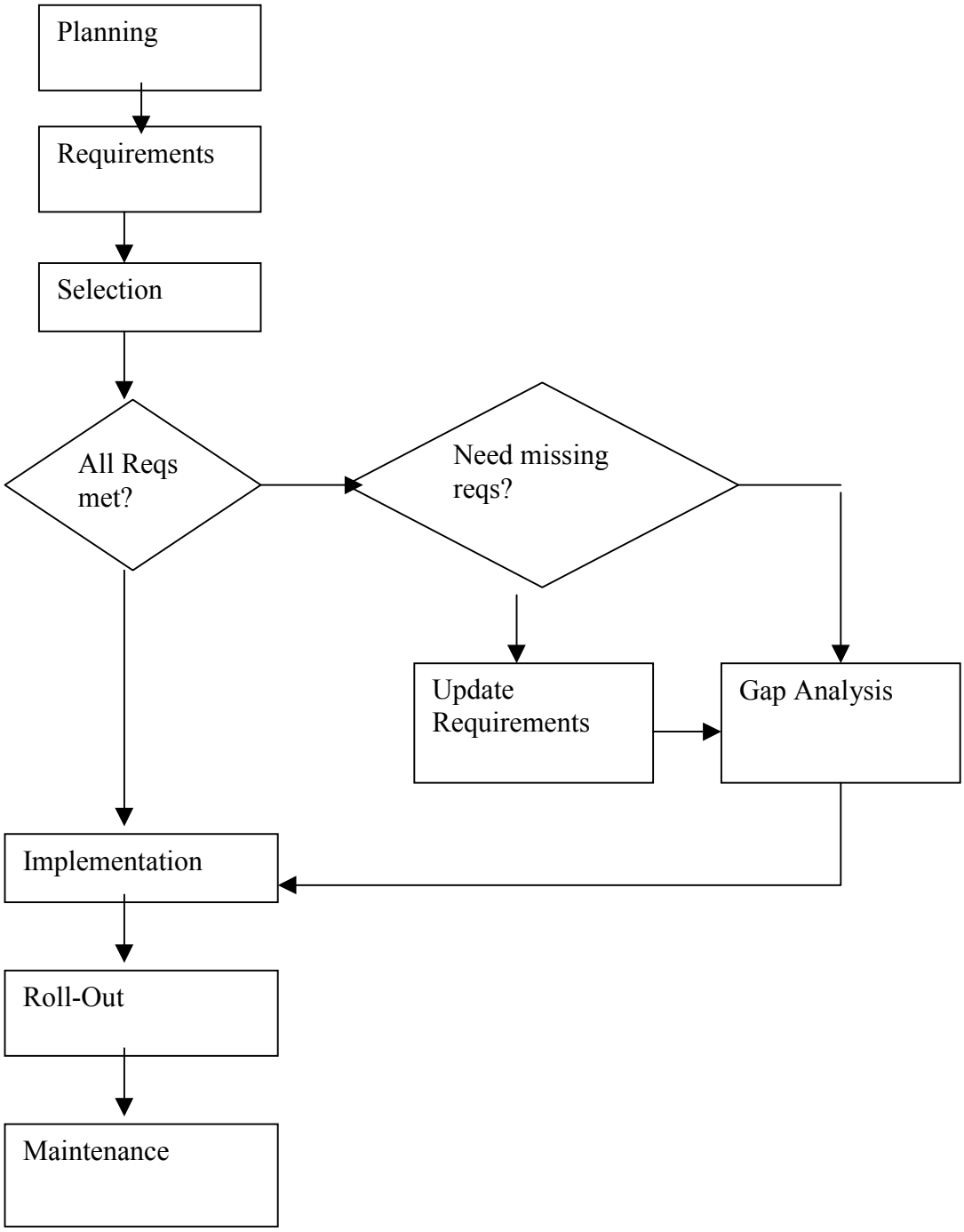


Figure A: Basic Tool Adoption Process

Planning – Prepare for the project by developing an improvement plan covering all aspects of the project. The plan should be documented, agreed to by all major stakeholders, and updated when there are changes to the project. (1A,1B,1C,1D,2A,2B,2D,6C,7A,7B,8A,8B)

Requirements – Collect requirements from all stakeholders. As a group, reach consensus on the priorities. Document how each requirement will be verified. Maintain all requirements under configuration control to keep track of requirements changes. Document an owner for each requirement. The owner will be the expert on the requirement. If there are any questions or issues related to the requirement, there will be a single focus. (2C,3B)

Selection – Choose the supplier that best meets your requirements. Note that the best supplier might not meet all requirements. If not, revisit your requirements to reevaluate the importance of missing requirements. (3A,3C,3D)

Gap Analysis – If no product meets all requirements, consider either developing your own customizations, or negotiate with the tool vendor to have customizations developed. If you need customizations, treat the development as a miniature development project, including project planning, requirements, project tracking, etc. Note that it might not always be cost effective to continue the project if the gap is too great. At this point, evaluate whether the project is still cost effective. Update the plan to reflect changes from the gap analysis. (3E)

Implementation – Develop the new process, training, documentation, etc. If there is any product development required, do it during implementation. Track progress against the plan throughout the implementation phase. (4A,4B,4C,5A,5B,5C,6B)

Rollout – When the implementation is complete, roll out the new tool based on the rollout plan. Rollout might include test runs with a pilot group. The rate of rollout should be dependent on the number of people who will be using the new tool, number of resources available for rollout, project schedules, as well as the complexity of the tool. (5D,6A)

Maintenance – Once the tool is rolled-out, the improvement project proceeds to a maintenance phase. Like development projects, improvement projects are almost never perfect. Users will provide feedback on areas to change, as well as ideas for improvement. It is critical to have resources available to monitor the success of the improvement project and make any necessary changes. (6A,6B)

Conclusion

There are many reasons tools become shelfware, many of which you can mitigate by implementing best practices and good project planning. Try spending more effort on each tool adoption to ensure your organization will get maximum benefit from the tool. In addition to saving money on wasted tools, you will help your organization achieve its goals by using tools effectively.

Real Testing of Real Software in the Real World

Rick Clements
cle@cypress.com
Cypress Semiconductor
9125 SW Gemini Dr., Suite 200
Beaverton, OR 97008

Abstract

With embedded software it's important to not only know how the customer will use the product but what environment the product will perform in. Unlike web and host based systems, there are few off the shelf test automation tools. Automation has to be designed into the product if it's going to be available. This paper looks at several case studies to see how these issues were addressed on real products.

Biography

Rick Clements is a staff software engineer at Cypress semiconductor. He has over 25 years of experience developing and testing software. Most of it involves embedded software. The software includes drivers for high-speed communications devices, infrared imagers, software controlled gyro stabilized gimbals, color printers and array processors.

Introduction

Embedded software in many cases actually does something. It reads sensors. It causes something to move. It doesn't run on a desktop PC. This results in four major differences between embedded and desktop software. Timing is important and needs to be included in the tests. The whole product must be tested and not just software. The environment the product will be used in is important. Many of the tools to automate tests need to be created.

Much embedded software is soft real time (it must perform to timing constraints most of the time) or hard real time (it must perform to timing constraints every time). The testing must verify this requirement is met. This may be by directly measuring that an event occurs in the proper time limit or it may be an indirect measurement by verifying the system doesn't fail.

The product working involves the whole system working. If the software responds properly but a mechanical part doesn't respond, the product still failed. Some tests may involve simulating the hardware, but some tests need to be done with real hardware. The tests need to be run in the environments the product will be used in. This environment may be multiple stimuli, or cold or hot temperatures.

Commercial tools exist to connect into the operating systems of programs running under Windows, X-Windows and MacOS. Those same tools don't exist for embedded products.

Testability must be designed into the product up front if the tests will be automated. Not building in testability may mean hours of toggling switches. A few products exist for automating tests in embedded systems. Testability must still be designed in. An interface still must be build. Whether a commercial tool or a custom tool is the right decision for the product, creating a test framework will be a significant effort that is a project of its own. Time, resources and planning are required if it will be successful.

This paper discusses how these issues were addressed on six real projects that resulted in product being used by real customers. These are projects from multiple companies.

Case #1 - Testing the Retrofit of Control Software

The Product Under Test

The product was a camera that's mounted on a helicopter. It was developed at a previous employer before I was hired. It was already in the field and customers were having trouble with the focus and field of view stepper motors loosing torque and control. This produced the need to modify the product and to test the changes more extensively than the original product.

The project involved two parts. The first step was rewriting the control software to provide closed loop control of the stepper motors. The second phase was replacing the magnets with stronger magnets to pull the filed of view slide into position if it got close.

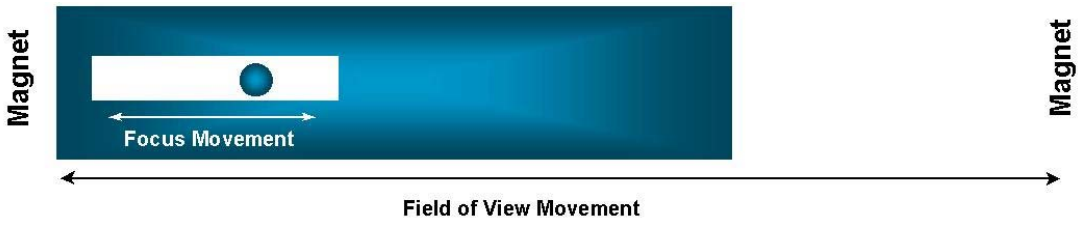


Figure 1 - Focus & Field of View Assembly

Figure 1 shows the assembly with two moving parts. Both of the parts are under software control. The field of view slider is held at either end by a magnet while a stepper motor moved the slider between the two positions. A second stepper motor moved the lens back and forth within the slot to focus the image. An FPGA was used to count pulses to track the position of the slider and focus lens.

The Tests

There were two cases that were causing the failures in the field. The first condition was cold (-20^{o} C). The grease became thick and the motor was stepped to quickly causing it to loose torque. The second condition, centrifugal force is created when the helicopter turned. Again, the old stepper motor code running open loop would try to step the motor too quickly and loose torque.

The software was rewritten for closed loop control. The stronger magnets hadn't arrived yet, so development went ahead without them. The tests were manual tests because the product was

already designed and it wasn't practical to make enough changes to facilitate automation.

The mechanism is mounted in a gimbaled system that would be mounted on a helicopter. The unit could be oriented in any position. The tests need to simulate the worst-case conditions. Vertical and horizontal should cover all of the cases, but to confirm that there weren't any surprises at other angles a set of tests were run at 45^{o}. Those tests confirmed our belief and those no more of those tests were run.

When the helicopter turns, centrifugal force is applied to the mechanism. To simulate the centrifugal force we added weights to the mechanism. We ran tests without the weight because much of the time the helicopter isn't turning.

The entire system is in a sealed gimbal that is exposed to the wind and cold air while the helicopter is flying. Since, it's a sealed system it can get very warm when the system is stationary. (This case happens most often in a booth at a trade show, but it needs to work in this environment.) Many tests were run at room temperature because they were easier to run than tests in the thermal chamber. Some of the tests were run at both minimum and maximum temperatures.

Since the tests were manual, we limited each configuration to an hour of testing. We also limited the number of configurations. The test matrix in table 1 shows the configurations that we tested.

	Horizontal	**45^{o}**	**Vertical**
Max. temperature	No weight		
Room temperature	Both	Both	Both
Min temperature	Weight		Weight

Table 1 - Test Matrix

Results

Changing the software to provide closed loop control eliminated loosing control of focus and field of view complaints the customers had about this product. The two most troublesome cases were weighted at minimum temperate and unweighted at maximum temperature. For the configuration mounted vertically with cold temperature and weight, the stepper motor had to overcome both the weight and cold, stiff weight. For the configuration mounted horizontally, with hot temperature and no weight, there was the greatest acceleration.

Not only weren't the stronger magnets needed, they could have decreased the reliability of the system. As the mechanism pulled away, the magnets pulled back. When it broke free from the magnet, there was a sling shot effect. This caused the mechanism to move fast enough the FPGA couldn't count the pulses fast enough to track the position of the mechanism. The stronger magnets would have increased the affect. They were also more expensive.

A simple change to the code that configured the FPGA was made to solve the problem of it loosing count. However, electrical engineers couldn't be bothered by silly software engineering practices like configuration management did the FPGA code. This resulted in a week exercise to locate the source code to the FPGA.

Case #2 - Non-Mission Critical Equipment

The Product Under Test

This case involves a new product meant to replace the product in case #1. There are two customers with different requirements. Neither of the customer groups is using this in a mission critical environment. The FAA defines mission critical as does the aircraft need the equipment to fly or land safely. Since the FAA didn't defined it as being mission critical, the need for many of their required procedures and testing was eliminated.

The first set of customers is broadcasters who wanted a stable image. This isn't simple because the helicopter isn't a stable platform. The helicopter introduces a vibration below 30 Hz that is very objectionable to the viewer.

The second set of customers is law enforcement. They want quick response when they want to look in a particular direction. Because they are searching, they are seldom looking in the same direction for a long period of time.

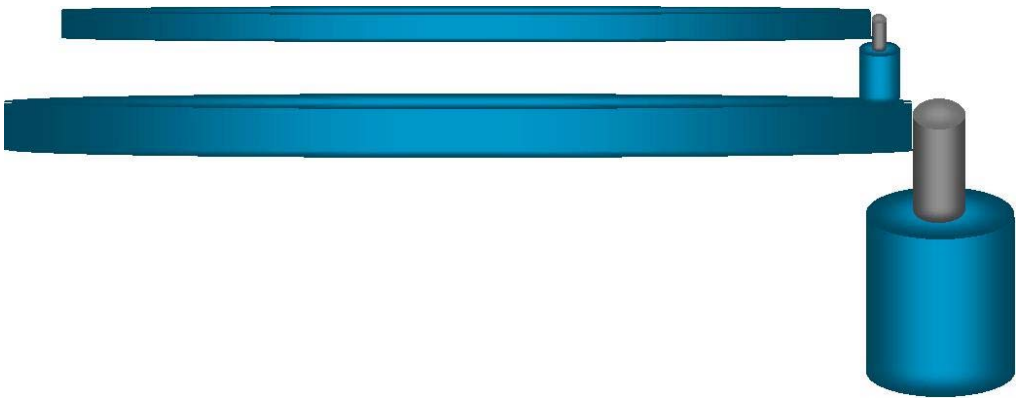


Figure 2 - Course & Fine Positioning Control

Figure 2 shows the X-Y plane of positioning and stabilization. The Z plane is similar. For each axis the product has two positioning motors. The largest motor provides course positioning. The smaller motor provides the fine positioning. Positioning in the X-Y plane requires 360^{o} movement. Tilting the camera does positioning in the Z plane. The position of the camera is determined by a set of gyroscopes.

Test #1 - Image Stability

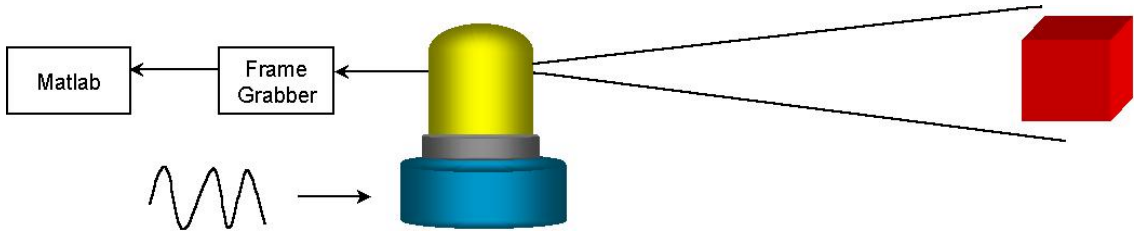


Figure 3 - Image Stability Test

The first test measured image stability. The unit was mounted on a shaker table that was driven with frequencies common from different helicopters. The unit was aimed at a fixed image. The video image from the unit was fed into a frame grabber in a PC. The image was analyzed in Matlab to determine that we met the stability figures we advertised.

Test #2 - Responsiveness

The second test tested the responsiveness of the unit. It was run over the range of voltages that might be seen in the helicopter. The system voltage affected the responsiveness of the system. Table 2 shows the range of voltages the unit operates in. In contrast to test #1, these were subjective tests.

Condition	Voltage
Maximum	29.5 V
Normal	28 V
Minimum	22 V
Emergency	18 V

Table 2 – Helicopter Voltages

The responsiveness felt crisp at maximum and normal voltages. At minimum voltages, there was some overshoot but the unit was still usable for the law enforcement customers. The degradation was do to lack of power to the motors and not an error in the software. The unit was unusable at emergency voltages. This was acceptable because it was non-mission critical. The only time the unit would operate at extreme low voltage was during maintenance. The solution for this case was to recharge the battery.

Case #3 - System Loading

The Product Under Test

In the small volume production environment, it's often necessary to make changes in the software to make a sale. In this case, a new GPS interface was being added to an existing

product. The GPS communicates with the system via one of the aircraft buses. The data rate was faster than the system could process, so the system had to sample packets from the GPS.

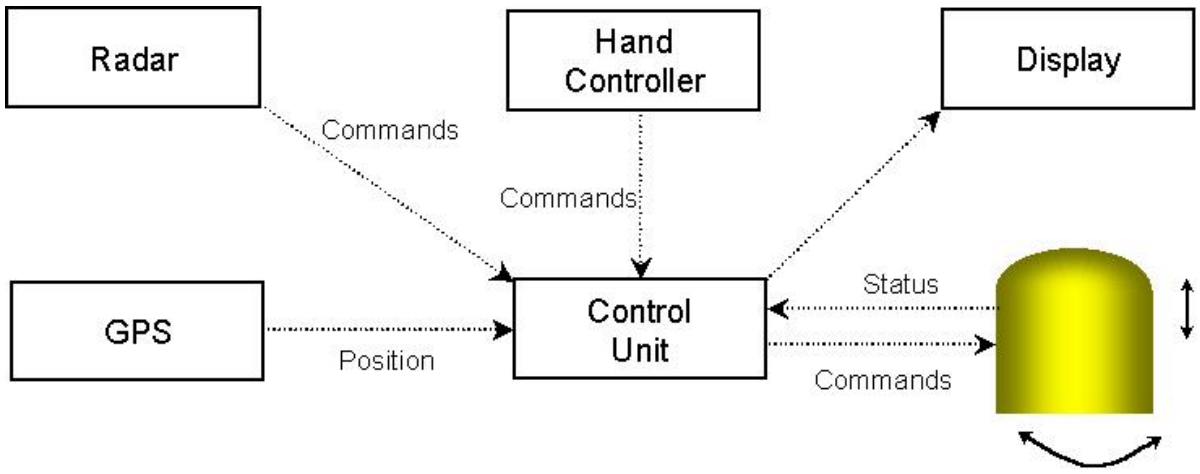


Figure 4 - Inter Component Communication

Figure 4 shows system configuration. The GPS sends positions to the system via one of the aircraft data busses. The radar sends pointing commands when the system is commanded to track the radar. The hand controller can provide position information and configures the system though a set of menus. Video plus overlaid text and symbols are sent as video data to display.

The gimbal is responsible for its own positioning. Where to point comes from commands from the control unit. The gimbal returned status as to where it is pointing.

Test - Maximum System Communication Traffic

Maximum System Loading	
Control Unit	Gimbal
Commands from radar	Positioning
Update display overlay	System communications
User menu selection or focus	
Gimbal communications	

Table 3 – Component Loading

This test achieved the best result with a mixture of automation and manual operation. This test automated the GPS and radar data because they couldn't be generated by hand. A person using the menus, focusing the system and observing the update of the overlays best was the best way to determine what appeared normal.

The product was set to track the radar. A PC simulated both the radar and GPS data with cards to communicate on the two aircraft busses. The radar was continuously moving in a figure eight pattern. This required the system to continually point the gimbal in a new position. This required the gimbal to position the gimbal using all axes. The position was updating at a steady rate. The display was a video signal.

The test was short enough that there wasn't a benefit to using a frame grabber and tracking the changes. The display updates needed to appear normal. A person watching the display best determined what appeared normal.

The user needed to be able to access the menus and focus the image. Like the display updates, these needed to look unaffected by the system being busy. Again, appearing normal was best determined by the user accessing the menus and focusing the system.

Case #4 - Test Automation with Software Simulation

The Product Under Test

Unlike the previous products, this product was being used in a mission critical situation. It was worn by firefighters in a burning building. The smoke produced zero visibility in the visible spectrum, but not in the infrared spectrum.

The firefighters are trained to work in that in zero visibility, but it increases the chance that a victim could be missed. Victims have been missed because the firefighter put their hand next to the victim and didn't touch them.

The unit allowed them to move faster because they weren't on their hands and knees feeling their way. This means they might get farther into the building than they could get out before their air ran out without the unit. The unit would run to failure but it needed to warn the firefighter that it might fail due to temperature or low battery.

Test Automation Build In

A serial port was built in for testing during development and manufacturing. This port wasn't accessible to the customer. The serial port allowed the software to be configured though a set of remote procedure calls. These are both operating constants and test parameters. The test PC makes a procedure call which appears to be on the PC. The function is carried out on the unit and the results are returned to the calling program.

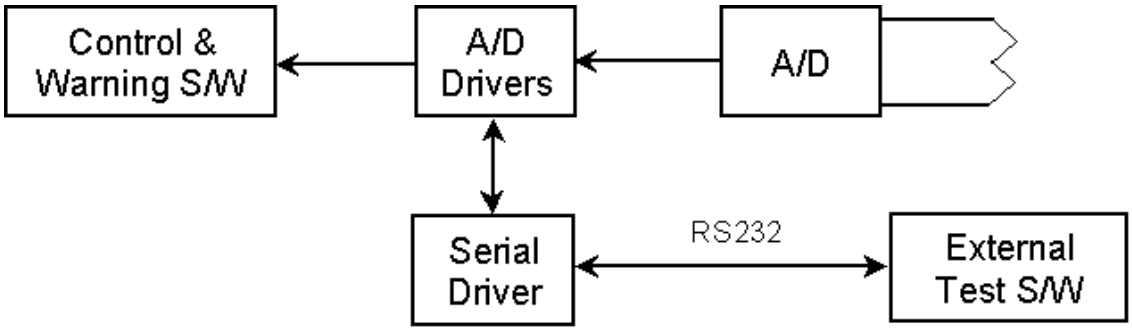


Figure 5 - External Control for Software Simulation

The object-oriented nature of the product software allowed software to simulate different conditions though remote procedure calls. Calls were added to instruct the driver to return specified values instead of the value of the sensor allowing conditions to be simulated. This

allowed tests to be simulated in much less time than would be required if the unit was placed a thermal chamber. The thermal chamber requires time to stabilize and it's difficult to have the temperature changes at the same time the battery status is changing.

Test #1 - Manual Test Verify Reading Sensors

Manual tests verified that the temperature could be correctly. These tests created temperatures and voltages at two different values then the values were returned by remote procedure calls.

These tests created confidence that the values were correct, but they weren't fool proof. The tests were run on a prototype. A change in the A/D converter was made in the prototype but the change wasn't made in the bill of materials. Manufacturing tests caught this error when the software didn't match the hardware. This was an electrical error, but it was fixed in software.

Test #2 - Temperature and Battery Warnings

The software measured the temperatures and battery. It displayed warning messages if the internal temperature was too high or the battery was nearly discharged. It also displayed the battery status independent of the warnings. The software had hysteresis when displaying temperature warnings. The messages were all prioritized so the software could display the most important warning.

Because the temperatures and battery voltages were simulated, it was possible to check the exact temperatures the different temperature warnings were displayed and removed. The battery status was verified at normal temperatures and elevated battery temperatures because a different set of icons was used.

The simulation allowed error conditions to be created and removed in desired sequences allowing the entire warning priority table to be verified. It also allowed testing that lower priority conditions were displayed after higher priority conditions were removed because we had more independent control than we would have in a thermal chamber.

Test #3 - Mechanical Reliability

The mechanical reliability tests were added because mechanical testing didn't have a way to perform the tests. Since software had hooks built into the system, we were able to do the tests with the software tools. In addition to testing different material for life, an electrical failure was found that was fixed in software.

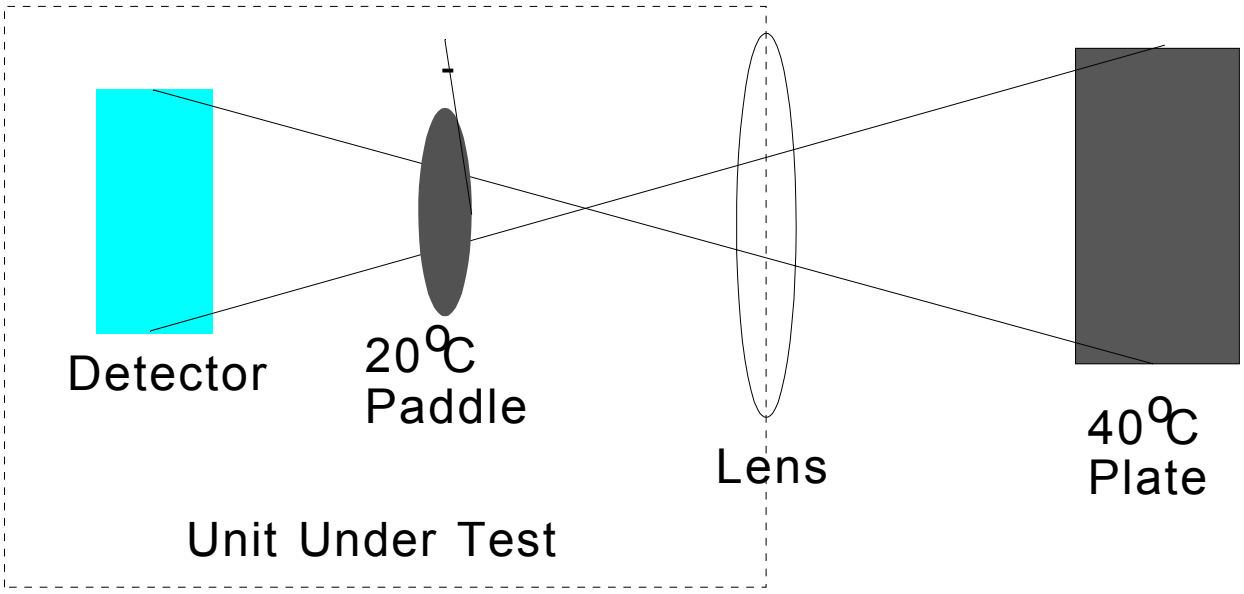


Figure 6 – Non-uniformity Correction Reliability Test

Unlike visible detector arrays, the array of thermal detectors requires non-uniformity correction (NUC). Testing needed to be done to verify the life the paddle. To do this, we aimed the unit at a black body (a plate of uniform temperature) at an elevated temperature. Though the remote procedure calls, the average scene temperature could be measured with the paddle covering and not covering the detector. About a week of testing could simulate the lifetime of the unit by instructing the unit to move the paddle more frequently than it's used for the NUC.

Measuring the results wasn't strait forward, since the units weren't calibrated; the ambient temperature affected the temperature returned. The data also showed the cycle time of the building's air conditioner. The two temperatures formed parallel sinusoids. It was possible to remove the sinusoidal component mathematically, but it would have been difficult to detect some of the failures.

The various failures were easy to see visually when the data was plotted. Any vertical lines represented a failure. The test software wrote the values to a log in pairs. The lists were split into segments that were small enough for Excel to handle. The data was graphed and visually inspected.

The primary purpose of the tests was to detect mechanical failures; so several units were tested on different PC's. When the graph went from two lines to one, the paddle had failed. Several iterations were necessary to find a combination of materials to produce the desired life span of the part.

The test showed other important results. When there was a spike extending part way into the image, the paddle had bound and partly covered the detector. In the field, this would cause part of the image to be unusable until the next time it did a correction five minutes later. Five minutes is a long time in a burning building. This required another mechanical change to fix.

In some cases, the paddle would miss a cycle. In the field, this would have caused the entire image to be unusable for five minutes. This was caused by the pulse to drive the paddle being too short. The pulse was less than a software instruction, so it wasn't a software problem. The processor that was used had a main CPU and a communications processor that handled the ports. The mostly cause of the problem was a race condition between the two processors internal to the chip. Having the driver read the port status after it had been written then write the port again if the value wasn't as expected solved this problem.

Case #5 - Extending Product Code

The Product Under Test

The product has a serial interface that's used by manufacturing during the manufacturing process. The system would be subject to electrical noise for other equipment in the manufacturing environment. Also, units were added and removed from the fixture from at each station as the product moved though the manufacturing process. The manufacturing line couldn't go down because the software locked up if a malfunctioning unit was connected or because of noise on it's communication bus.

The communications class was part of the common code library and would be used by multiple projects. This product was the first use that software. The lowest level of the communications class supported making a connection between the host and the product, creating a packet, sending a packet, receiving a packet and disassembling a packet. It also supported resynchronizing if communications were interrupted do to data errors.

Extending Product Code for Test

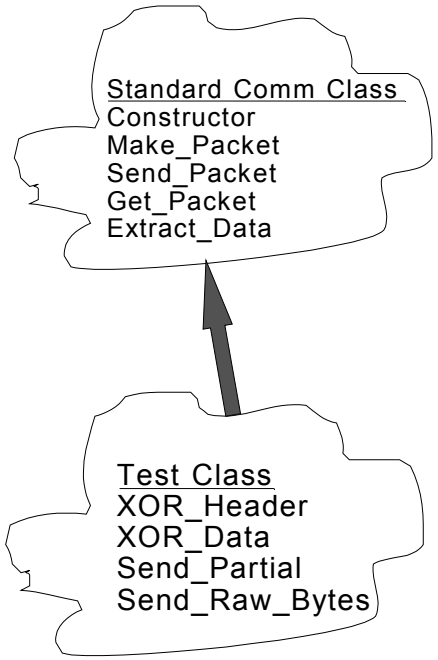


Figure 7 – Parent and Child Classes

The test code used basically the same serial driver that was in the product. This had the benefit of not having to develop the code twice. Using the code for test exercised parts of the code that would be used on later products. Having code place into the common code library completely tested even if the features weren't used on the original product was important. The majority of errors found because of this were in handling big endian vs. little endian. (Endian is which end of the number is considered most significant.) This would be a benefit to the next product using the common code because it was the same endian as test software. The risk was the test code might compensate for an error in the product. This wasn't seen in manufacturing.

For testing, it was necessary to generate bad packets, which isn't a feature that's desirable in the product. Figure 7 shows the child class used by the test software in relationship to the communication class used in the product. The child class added the ability to XOR a value with any byte in the header, XOR a value with any byte in the data allowing packets to be built with the correct CRC then change the data to create a CRC error. The child class also allowed part of a packet and a raw stream of bytes to be sent. These simulated the different cases of dropped bytes and allowed all of the cases of resynchronizing to be simulated.

Case #6 - Test Automation with Hardware Simulation

The Product Under Test

The product was a print engine controller that controls the motion of the paper, the print head, heaters and transport of the ink. The testing is done on the print engine controller before the imaging processor is added to the system. The testing needs to be automated including simulating signals because it's faster, it doesn't consume supplies and some conditions are hard to generate but must be tested.

Hooks for Testing Designed In

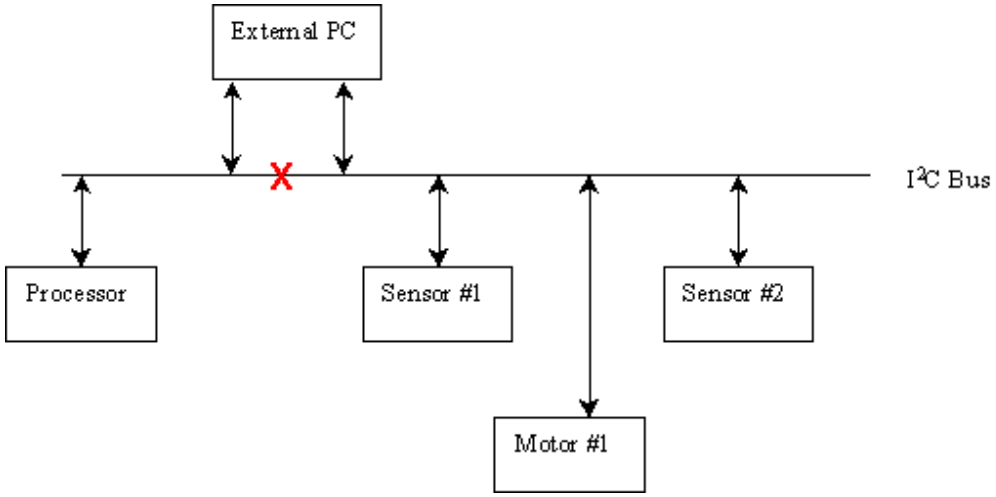


Figure 8 - External Control for Hardware Simulation

This is similar to case #4 except the conditions were simulated in hardware instead of in software. The test system was inserted into the bus between the processor, and the sensors &

motors allowing signals to be simulated and control signals read. It also allows controls signals to be blocked, since the state of the system was being simulated. The signals needed to be blocked so overfilling the ink chamber or the heaters driving too hard didn't damage unit.

Summary

Testing the Customers' Environment

In all testing, it's important to know how the customer will use the system. With embedded systems, it's also important to know what environment it will be used in. Case study #1 involved redesigning a product that was in the field because the original project hadn't taken cold temperatures or centrifugal forced into account.

In case study #2, there were two sets of customer with separate requirements. In order to identify the different requirements each set of customers had, it was necessary to know what environment the customer would be in when the system was being used. In one case, the customer needed a stable image. In the other case, the customer was scanning looking for potential danger other team members might face. Very different tests were designed for the requirements of the two sets of customers.

The system was tested for the maximum stimulus from the user and the environment in case study #3.

In case study #4, the unit was being used in a hostile environment. The product was being used where it could impact the safety of the user. Tests to verify warning messages were correctly displayed and tests for system reliability were important.

Test Automation

Automating testing requires the necessary hooks to be designed in up front. The tests in case study #1 would have been more effective if then had been automated. Testing consumed valuable resources that could have been doing something other than pushing buttons for hours. The tests had to be manual because the necessary hooks weren't designed into the product.

Case study #4 and #6 showed two different approaches to automating tests. In case #4, software hooks into the driver simulated test conditions. In case #6, placing the test system between the processor, and hardware sensors and controls stimulated test conditions. The method of automation depends on what makes sense for the architecture of the product. But in both cases, the hooks were designed into the hardware or software.

In case study #5, the test fixture used and extended the code that was in the product. This gave more used of the product code and allowed for testing that there otherwise wouldn't have been the resources for. The risk that a defect would be hidden because the product code and test code were the same was never materialized.

It doesn't make sense to automate all tests. In case study #2, the "does it feel right" tests were manual while the "does it meet spec" tests were automated. In case study #4, manual tests were

performed to verify the results returned were valid. The other tests with more complex setup and interactions could then be automated. In case study #3, the test was a mixture of automation and manual intervention. When designing the automation, we must look at what makes sense and what do we have the resources to automate.

In case study #2, a frame grabber was used because the only system output was video. Capturing the video frame by frame and analyzing it, greatly adds to the complexity of automation. But, it does make sense for some testing.

Working Smarter
By Applying a Knowledge Management Framework
to Software Testing

Kersti Nogeste

Principal Consultant

I.T. Expertise Pty Ltd, Melbourne, Australia

(PO Box 464, Collins Street West VIC 8007 Australia)

Candidate in Doctorate of Project Management (DPM) program

RMIT University, Melbourne, Australia

ABSTRACT

Organisations are increasingly recognising that their competitive advantage can be attributed in part to intangible assets such as knowledge. Therefore, there is potential for an organisation to be well served by managing their intangible assets, including knowledge. This paper describes how an IT Production Support group applied the APQC (American Productivity and Quality Center) knowledge management framework to revise an existing repository of regression test cases by converting tacit business knowledge into explicit (documented) test cases.

The paper describes the business environment and business need, the APQC KM framework, how the KM framework was used to address the business need, implementation feedback and lessons learned. The paper concludes with remarks about how this experience relates to other situations where there is a need to create/revise/maintain a knowledge repository for ongoing use and reference.

ABOUT THE AUTHOR

Kersti Nogeste (knogeste@itexpertise.com.au) is a self-employed contract project manager with industry experience in Australia and North America, especially in the practical application of software quality processes. Kersti has spent the past seven years working in the Australian telecommunications and utility industries.

Kersti has an undergraduate degree in Computer Science, a Master's degree in Business and is currently working towards a Doctorate in Project Management. This paper is an outcome of her doctorate level research into the practical use of knowledge management on projects.

Kersti is a past PNSQC speaker, a member of the American Society for Quality (ASQ), the Project Management Institute (PMI) and the Australian Institute of Management (AIM).

INTRODUCTION

The utilities industry in Australia is becoming increasingly deregulated and therefore also increasingly competitive. Utility companies are well aware that their competitive advantage is somewhat reliant on the performance of their Customer Service business operations, including customer care, billing and credit management. Therefore it is very important that the Customer Information System (CIS) computer systems supporting Customer Service business operations function correctly. From the perspective of the CIS systems, the changes occurring in the utilities industry are implemented as a series of software releases, with each release corresponding to one or more changes. So, whilst it is important to ensure that *new* business operations, including CIS functions operate correctly, it is also important to ensure that *existing* business operations and CIS functions continue to operate correctly i.e. that the business operations and/or CIS functions do not regress.

A common means used to ensure against this regression is regression testing, where a repository of test cases is used to verify that pre-existing business operations and CIS functions continue to operate as expected and have not been adversely affected by newly introduced changes. Therefore, it is very important for an organisation to conduct regression testing before a new version of software is released. A repository of regression test cases must adequately test high-risk business functions especially those with the potential to affect many customers (and degrade customer service levels) or the potential to have major financial impacts upon either the utility or its customers.

Upon reviewing the business risks associated with the next planned software releases, the CIS Production Support management team at a particular Australian utility company recognised that Business Analysts' test planning skills and the repository of test cases both required improvement. An independent consultant was engaged to address these needs. The first need, to improve business analysts' test planning skills was addressed by two workshops reviewing software testing concepts and test planning skills. The second need, to improve the repository of test cases was addressed by a third workshop during which a knowledge management (KM) based testing process was developed, to revise the repository of test cases.

The need to convert and convey tacit business knowledge into explicit (documented) test cases prompted the consultant to use a KM framework as the basis for developing the process to revise the test cases. This decision and the choice of the APQC KM framework in particular, were prompted by the consultant's prior study of the framework as part of formal postgraduate studies. The APQC KM framework had been chosen for study by the consultant because it was the result of significant development effort by the APQC and the Arthur Andersen consulting company, and information about the framework was readily and inexpensively available via the APQC web site (http://www.apqc.org) and APQC publications.

The remainder of this paper describes the APQC KM framework, how the KM framework was used to develop a process for revising the repository of test cases, implementation feedback and lessons learned. The paper concludes with remarks about how this experience relates to other situations where there is a need to create/revise/maintain a knowledge repository for ongoing use and reference.

THE APQC KM FRAMEWORK

The American Productivity and Quality Center (APQC)

The APQC was established in 1977 and is based in Houston Texas, USA. The APQC mission is to *work with people and organizations around the world to improve productivity and quality (*http:/www.apqc.org*)*. A non-profit organisation supported by nearly 500 companies, government organizations, and educational institutions, the APQC

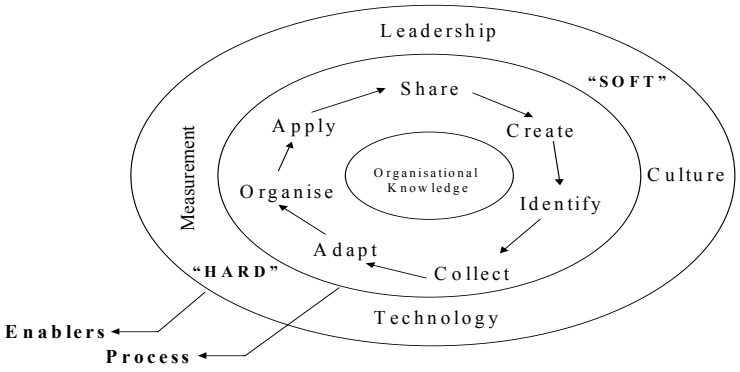
- discovers, researches, and understands emerging and effective methods of both individual and organisational improvement;
- broadly disseminates their findings through education, advisory, and information services; and
- connects individuals with one another and with the knowledge, resources, and tools they need to successfully manage improvement and change.

The APQC Knowledge Management Framework

According to the APQC *(*http:/www.apqc.org*)*

> *The age of knowledge management "early adopters" is over.*
>
> *Hundreds of organisations, large and small want to capitalise on what they know. They want to create, identify, capture, transfer and reuse their valuable knowledge - in other words "manage" it.*

Based on the premise that knowledge management (KM) had evolved into a systematic process, in 1995 the APQC developed a knowledge management framework with support from the Arthur Andersen consulting company. The APQC KM framework (Figure 1) is intended to illustrate the process and environmental enablers required to successfully transfer tacit and explicit organisational knowledge and best practices (O'Dell 2000). In this model, tacit knowledge is defined as *experience, know-how, skills and intuition most often embedded in the individual* and explicit knowledge is defined as *information you can easily put into words or pictures or that is easy to articulate and communicate. (O'Dell 2000, p1)*



American Productivity and Quality Center

Figure 1 — American Productivity and Quality Center KM Framework

For the purposes of revising the repository of test cases, the framework was applied as follows.

- The steps of the knowledge management process (in the inner circle) were used as a checklist for developing the process for revising the test cases.
- The four enablers of leadership, culture, technology and measurement (in the outer circle) were interpreted as defining the environment required for the knowledge management process to succeed. Therefore, for the purposes of this exercise, the enablers were considered as defining different categories of dependencies; what needed to be in place for the KM based process to succeed.

USING THE APQC KM FRAMEWORK TO DEVELOP A PROCESS FOR REVISING THE REPOSITORY OF TEST CASES

Workshop Participants

All of the workshops were led by an independent consultant and attended by experienced CIS Production Support Business Analysts and business subject matter experts temporarily assigned from business operations as new testers. Following the third workshop, where the repository of test cases was evaluated and revised, the new testers were responsible for implementing the KM based process that was developed collaboratively by all workshop participants.

Create, Identify and Collect Knowledge

Following the APQC framework, the workshop participants initially considered the activities of creating, identifying and collecting knowledge as discrete activities. After some discussion, however, it was decided to group these three activities together. The workshop participants generated the following list of knowledge management activities to create, identify and collect the knowledge required to revise the existing test cases.

1. Collect the complete set of existing regression test case documentation.
2. Talk to technical experts to develop a better understanding of the technical aspects of the CIS, including batch jobs and the system environment.
3. Contact the Training group to get electronic access/copies to/of business process and training documentation, to be used as reference material when planning and conducting the tests.
4. Identify business experts.
5. Standardise the naming/numbering of all resources, so that all information related to a particular test case can be readily found and updated as required.
6. Add new fields to the test case template to capture additional relevant information. e.g. business priority
7. Identify missing test cases.
8. Draft additional test cases, as required.

Adapt and Organise Knowledge

Again following the APQC framework, the workshop attendees initially considered the activities of adapting and organising knowledge as discrete activities. After some discussion, however, it was decided to group these two activities together.

The workshop participants generated the following list of activities to adapt and organise the knowledge required to revise the existing repository of test cases :

1. Cross-reference the business process and training documentation to test cases.
2. Schedule business experts to review test cases and annotate training/process documentation.
3. Update test case documentation with feedback from business experts.
4. File (organise) all information so that it can be readily found (both electronic and paper copies)

Apply Knowledge

Workshop attendees considered that the knowledge captured in the test cases would be applied by the regression testing of

- the next planned release with the revised test cases
- subsequent releases using the continually updated repository of test cases.

In addition, a key facilitator of the knowledge contained in the repository continuing to be applied was that the test cases and all associated documentation would be clearly indexed, named and filed on a "shared drive" accessible as reference material to all members of the CIS Production Support group. Given that the revised repository would contain test cases to test all high risk business functions, it was expected that people would refer to the repository more than they had before; potentially applying the knowledge contained in the test cases to more than their immediately intended purpose of regression testing. It was recognised that test case documentation could be used as a means of learning how a particular CIS function was used to support a Customer Service business process.

In addition, the workshop attendees recommended that the CIS Production Support management team contact CIS related projects and advise them of the existence and potential reference value of the regression test cases while providing them with read-only access to these files.

Share Knowledge

Upon discussing the sharing of knowledge in this process, the workshop attendees unanimously agreed that knowledge sharing was not a single step in the process of revising the regression test cases. Rather, knowledge sharing was included in all steps of the process; with technical and business experts reviewing draft test cases, annotating process/training documentation and supporting the ongoing regression testing of each subsequent release.

Additionally it was recognised that discussion and review of the status of regression testing at CIS Production Support team meetings would also contribute to the sharing of knowledge.

KM Framework Enablers

As described above, for the purposes of this exercise, the four enablers of leadership, culture, technology and measurement (in the outer circle of the APQC KM framework) were interpreted as dependencies; defining what needed to be in place for the KM based process to succeed.

Table 1 — KM Process Dependencies

KM Framework Enabler	KM Process Dependency
Leadership	CIS Production Support management to • Ensure new testers remained available to revise the repository of test cases; • Ensure business process and training materials were made available; • Ensure technical/business experts would make enough time available to help out; • Advise other groups in the company that test case documentation was available in a shared storage location.
Culture	All CIS Production Support group members to agree that the current regression test cases need to be revised prior to regression testing the next planned release. (The larger organisational culture was considered outside the scope of this exercise.)
Technology	All members of the CIS Production Support group to have ready access to standard office systems software and particular CIS system environments.
Measurement	The repository of test cases must be revised by a defined due date, prior to the next release of CIS software changes that require regression testing. (The workshop group did not set measures of success for the revised repository of test cases, since the scope and complexity of software releases varies greatly and there are no metrics in place to indicate the relative "size" of a release.)

IMPLEMENTATION FEEDBACK

Method

The consultant responsible for conducting the series of three workshops gathered feedback on the implementation of the KM based process periodically for two months. Feedback was provided by business and CIS Production Support management representatives, CIS Production Support Business Analysts and new testers via a combination of questionnaires and meetings. Questionnaire responses were provided according to a five-grade scale ranging from Poor to Excellent.

The Existing Repository of Test Cases

All respondents considered the existing repository of test cases to be outdated and providing inadequate coverage of key business operations. The existing test cases had ceased to be updated for nearly a year, the process for conducting the test cases was undocumented and no test cases existed for some business areas.

The more experienced Business Analysts considered the existing repository of test cases to be Good, whilst the less experienced Business Analysts and new testers considered it to be Poor (2 grades below Good). The difference in these assessments suggests that the more experienced Business Analysts may have taken their tacit knowledge into account, while the lesser experienced Business Analysts and new testers have based their assessment on a lack of tangible explicit knowledge.

The Revised Repository of Test Cases

Interestingly, CIS Production Support management representatives and all Business Analysts thought that the revised repository of test cases was Very Good, whilst the new testers assigned responsibility for revising the repository of test cases thought it to be Excellent. This difference of opinion could exist for a number of reasons, including existing staff members' reluctance to over-rate the new repository, loyalty to the former version of the repository and the people who had developed it, or the new testers' more detailed (closer) experience with the revised repository. Whilst a single point of difference, the consistent difference in opinion does suggest that various peoples' opinions should be sought when assessing the value of a new process.

A CIS Production Support management representative indicated that they were satisfied that regression testing was being planned and conducted in a better manner, that the knowledge management based process would serve as a good model for other Production Support areas, that the CIS Production Support group's reputation had been further enhanced and that the KM based process had provided a structured professional development opportunity for the new testers assigned responsibility for revising the test cases.

The business management representative's feedback indicated that they neither saw nor experienced any additional benefits arising from the revised repository apart from the personnel rotation and professional development outcomes described in more detail in following sections. From the business management representative's perspective, the main change they noted was the more relaxed attitude towards test management resulting from a change in the CIS Production Support management team. A technically astute and process-driven manager was replaced with another less technically minded and less process driven manager. So, from the business perspective there was a perception that an increasing number of problems were slipping through to the Production environment due to relaxed management controls.

It must be stressed that this is a perception, since the introduction of the new manager coincided with the testing of generally acknowledged "larger" and more complex software releases that included the delivery of relatively more undetected problems into the Production environment. Under the circumstances there is no way of directly relating the influence of the new manager to the increased number of problems detected in the Production environment, since the complexity of the software releases and the testing required must also be considered as major influencing factors.

Process Improvements

Shortly after starting to implement the KM based process, it was recognised that the value derived by having the new testers consulting with technical experts was greater than had been expected, and encouraged an increased level of ongoing interaction between the new testers and technical experts.

After two or three rounds of test case reviews, the new testers developed the skills and confidence to conduct more rigorous peer reviews themselves prior to handing test cases to

business experts for review. The new testers began looking for duplicate test cases, overlaps between test cases and mis-wordings in test case documentation.

The KM based process continued to be applied for the series of planned releases. When system functionality was added or fixed in a release, a selection of corresponding new test cases were added to the repository of test cases and then used to regression test each subsequent release. Therefore, the cyclical knowledge management process continued to be applied.

Personnel Rotation

During the course of using the revised repository of test cases to test the first planned release, one of the new testers asked to return to their former business operations role. This person was then replaced by a third person, also temporarily assigned from a business operations role.

According to the business manager responsible for assigning two of the three new testers (the unexpected returnee and their replacement), a formal selection process had been used to select both candidates. This selection process had taken into account the complexity of an individual's regular business role, the thoroughness and speed with which they generated work outcomes and their willingness to learn. The person who requested to return to their regular business role had been the best-qualified candidate according to the selection criteria. However, once confronted with the work required to revise the test cases, they decided that it was too difficult for them and stated a strong preference to be replaced. Interestingly, the replacement person, who was a relative newcomer to the same business, had a less complex business role but indicated a stronger willingness to learn, as evidenced by their recent completion of an undergraduate degree and intention to pursue post-graduate studies. In this case, the willingness and recent proven ability to "learn new things" was a better indicator of a person suited to rotation into the new tester role.

Professional Development

The new testers assigned to revise the repository of test cases indicated that over a period of weeks, they developed a greater appreciation and understanding of end-to-end business processes and system functionality that included an improved understanding of the impacts of their usual work upon others. They also felt that their broader business understanding would assist them upon their return to their business areas, both with their own work and their interaction with colleagues from other business areas.

The fact that the new testers were drawn from different business areas was also considered beneficial, as was the decision to assign *two* new testers, since the two testers were able to work together to revise the test cases.

During the course of testing the first few releases, the Business Analysts' confidence in the new testers grew to the point where the Business Analysts requested assistance from them, with tasks outside their immediate regression testing responsibilities. This peer recognition and demonstration of trust are important indications that the Business Analysts believed the new testers had developed sufficiently good skills and experience in the practice of testing and use of CIS functionality and business processes to assist the Business Analysts with their additional tasks. The Business Analysts also indicated an increased understanding of regression testing and its importance to the testing of each release.

LESSONS LEARNED

Upon review, the following lessons have been learned and are considered valuable for the reasons provided :

Table 2 — Lessons Learned

Lesson Learned	Value
The APQC KM framework can be used to develop a knowledge management based process for revising a repository of test cases.	Consider using the APQC knowledge management model when required to create/revise/maintain a knowledge repository for ongoing use and reference.
The knowledge management process enablers included in the APQC framework can be used as a basis for identifying knowledge management process dependencies.	Pay sufficient heed to both soft and hard environmental factors that can influence the outcomes of the knowledge management process.
When suitably introduced into a problem-solving context, the APQC KM framework is a relatively quick means of generating a knowledge management process.	Organisations and their personnel are under increasing pressure to work smarter and faster. This case study demonstrates that the APQC KM framework was able to provide a means of doing both.
Quantitative measurement enablers for a knowledge management process can only be identified in an environment where relevant quantitative measures already exist.	As an example of enablers, if a method for quantifying outcomes is currently not in place it will prove challenging for the group involved in developing the knowledge management based process to institute a formal measurement process without support from the larger operational context.
When defining leadership enablers, consider including a dependency upon continued/improved management of quality assurance processes.	Success of the knowledge management process will be dependent on existing/improved leadership.
Expect continuous improvement of the knowledge management based process during implementation.	Don't agonise unnecessarily over the development of the initial version of the KM process. Plan the process, implement it and learn, gaining knowledge about the implementation of KM.
Acknowledge the professional development value of rotating personnel through roles responsible for implementing a knowledge management based process.	Provide personnel with the opportunity to better understand the organisation's "business", including their and other peoples' place in it.
When selecting people to be rotated through roles, take into consideration their recent track record for learning new concepts and skills.	The process steps of create, identify, collect, adapt, organise, apply and share all depend on a person's willingness to be open-minded and have a willingness to learn, no matter how good they are at performing their present role.

Lesson Learned	Value
Management needs to ensure that subject matter experts are sufficiently available to participate in all steps of the knowledge management process and that both "givers" and "receivers" of assistance understand the time and timing of the business expert's expected participation.	Since a main purpose of applying a KM process is to convert tacit knowledge into explicit knowledge, management needs to ensure that subject matter experts possessing tacit knowledge are sufficiently available during the course of applying the KM process; to provide input and review deliverables.
When assessing a knowledge management process, seek feedback from people involved to different degrees in developing and implementing the process. Consider both quantitative and qualitative feedback.	Different people will have different views and opinions depending upon their personal and professional experiences. Make sure to take different perspective into account, since each person's perspective is based on a combination of explicit and tacit knowledge. Seeking feedback from a variety of people will continue to draw upon this mix of knowledge.

CONCLUSIONS

The APQC proposes that their KM framework illustrates the process and environmental factors required for the successful transfer of internal knowledge and best practices. (O'Dell 2000).

In this case study, the APQC knowledge framework was used by a group new to knowledge management as the means of transferring internal knowledge into documented test cases for a Customer Information System. The process for revising the repository of test cases was developed and implemented effectively and efficiently, most probably because "the APQC framework provides a simple, straightforward and efficient process for sharing and tapping into a knowledge pool" (PRNewswire 1996).

Whilst this experience is specific to regression testing the combination of a particular CIS and associated business processes, the need to improve a knowledge repository is shared by teams in situations similar to that of this Production Support team, where a team or organisation wants to

- capture and convey tacit knowledge;
- use an easily comprehensible and relatively quick method to define a KM based process;
- define the factors that can influence the success of a KM based process.

Therefore it is proposed that project managers, team leaders and line managers consider the APQC KM framework as a means of defining a knowledge management based process for creating/revising/maintaining a knowledge repository for ongoing use and reference.

REFERENCES

Arthur Andersen, (1995). Knowledge Report (1996). Knowledge Imperative Symposium, Houston, Texas, USA, Arthur Andersen, Chicago, Illinois, USA

de Jager, M. (1999). The KMAT : Benchmarking Knowledge Management. Library Management. 20: 367 - 372.

O'Dell, C. (2000). Identifying and Transferring Internal Best Practices. an APQC White Paper. Grayson, C. J., American Productivity and Quality Center, Houston, Texas, USA.

O'Dell, C. (2000). Knowledge Management - a Guide for Your Journey to Best-Practice Processes. American Productivity and Quality Center, Houston, Texas, USA.

PRNewswire (1996). Arthur Andersen Helps Define the Knowledge Manager's Role in the Knowledge Revolution. Chicago, Illinois, USA.