## 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 some of 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/cdrom.html 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 3.01 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

To order a copy of the Proceedings, please send a check in the amount of $35.00 each to:

PNSQC/Pacific Agenda
PO Box 10142
Portland, OR 97296-0142

Name _____

Affiliate _____

Mailing
Address _____

City _____

State _____

Zip _____ Phone _____

# EIGHTEENTH ANNUAL PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

## OCTOBER 17 - 18, 2000

### Oregon Convention Center
### Portland, Oregon

# TABLE OF CONTENTS

**Preface** ..... **iii**

**Conference Officers/Committee Chairs** ..... **iv**

**Conference Planning Committee** ..... **v**

**Keynote Address – October 17**

***Simple Steps to Achieve Results with Process Improvement*** ..... **1**
George Yamamura

**Keynote Address – October 18**

***Risk Management is Project Management*** ..... **7**
Tim Lister, Atlantic Systems Guild

**Testing Track – October 17**

***Methods for Effective C/C++ Unit Testing*** ..... **29**
Adam Kolawa, Parasoft

***Empathetic Listening: The Overlooked Testing Tool*** ..... **45**
Julie Fleischer, Intel Corporation

***Measurement of the Extent of Testing*** ..... **51**
Cem Kaner, Kaner.com

***Software Testing 101*** ..... **93**
Rick Clements

***When Will We Be Done Testing? Software Defect Arrival*** ..... **115**
Erik Simmons, Intel Corporation

**Management Track – October 17**

***Meeting Tight Schedules Through Cycle Time Reduction*** ..... **133**
Dennis J. Frailey, Raytheon, SMU & Mary Sakry, The Process Group

***Conflict Management in Software Development Environments*** ..... **149**
Lisa Burk, Communication Works & Jean Richardson, BJR Communications, Inc.

***Managing Software Product Quality*** ..... **195**
Manny Gatlin & Greg Hannon, Intel Corporation

***A Goal-Problem Approach for Scoping a Software Process Improvement Program*** ..... **205**
Mary Sakry, Neil Potter, The Process Group

## Quality Assurance/Metrics Track – October 17

***A is for "Assurance" – A Broad View of SQA*** ..... **217**
Alan S. Koch, Independent Consultant

***Practical Lessons Learned in the Coordination of Multi-site SQA Teams*** ..... **237**
John K. Suzuki, JKS & Associates & George McKewen, Xerox Corporation

***Using Reading Techniques to Support Inspection Quality*** ..... **253**
Stefan Biffl, Thomas Grechenig, Michael Halling, Technische Universität, Wien

***The Darker Side of Metrics*** ..... **265**
Douglas Hoffman, Software Quality Methods, LLC.

***A Universal Metrics Repository*** ..... **273**
Warren Harrison, Portland State University

## Internet Quality Track – October 18

***Testing Web-Applications – Effective Techniques for Analyzing & Reproducing Errors*** ... **289**
Hung Q Nguyen, LogiGear Corporation

***Solving the Software Quality Management Problem in Internet Startups*** ..... **297**
James Mater & Bala Subramanian, QualityLogic, Inc.

***A Pragmatic Software Configuration Management Model for the E-World*** ..... **307**
Michael K. Jones, Integrated Software Systems, WIU

## Improvement Models Track – October 18

***Software Six Sigma*** ..... **319**
Mike Palmer & Tom Lienhard, Honeywell

***A Practical Road Map for CMM Implementation*** ..... **333**
Dr. M.S. Ramkumar, HCI Technologies, Ltd.

***Self-Assessment Using the CMM – An Empirical Study*** ..... **349**
Robert F. Roggio, University of North Florida, Pratibha Kashyap, RSINET Consulting
& Ava S. Honan, Auburn university, Montgomery

***Software Process Improvement in Small Companies*** ..... **363**
Ita Richardson & Kevin Ryan, University of Limerick, Ireland

## Testing Requirements Track – October 18

***Using XML as a QA Foundation Technology*** ..... **375**
Richard Vireday & Steven B. Augustine, Intel Corporation

***Trials & Tribulations of Testing a Java/C++ Hybrid Application*** ..... **393**
Steve Whitchurch, Mentor Graphics, Inc.

***What Do You Mean By That?*** ..... **403**
Christopher Simmons & Erik Simmons, Intel Corporation

***Requirements-Based UML*** ..... **411**
Joseph D. Schulz, Technology Builders, Inc.

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

## Preface

Welcome to the 18th Annual Pacific Northwest Software Quality Conference. Our mission is to increase awareness of the importance of software quality and to promote software quality by providing education and opportunities for information exchange within the software community.

This year, our keynote speakers are talking about risk management and process management. Tim Lister's keynote presentation "Software Risk Management IS Software Project Management". Putting together a schedule is easy, but what are the risks that will prevent the schedule from being followed? When evaluating project risks, many companies ignore the risks their own processes create, as if those risks can't be addressed. George Yamamura's keynote presentation "Simple Steps to Achieve Results with Process Management" looks at managing processes to reduce project risks.

I look at the conference schedule and see more presentations I'd like to see than there is time to see them. The many interesting presentations include "Meeting Tight Schedules Though Cycle Time Reduction", "When will the testing be done", "Measuring the Extent of Testing" and "Trials & Tribulations of Testing a Java/C++ Hybrid Application".

I enjoy the wide range of presenters that are scheduled. There are experienced speakers I've enjoyed hearing and reading before. People like Tim Lister, Cem Kaner and Mary Sakry. From the academic world, the speakers include Warren Harrison, Ita Richardson and Stefan Biffl. There are engineers from companies including Cisco Systems, Intel, Mentor Graphics, Cypress Semiconductor and Technology Builders. The variety of experience and opinions is exceptional.

Our attendees also have a large set of experiences. At lunch, we have Birds of a Feather tables set up for attendees to share those experiences. You have experiences worth sharing! Look for the call for papers and submit an abstract. We have people from industry and academia presenting topics that range from basic issues to case studies of new techniques and issues. I accepted the challenge and I'm presenting a paper this year. Think seriously about presenting a paper next year.

Our spring workshops will be May 14-15, 2001 in Portland and Seattle. Our annual conference and fall workshops will be October 15-17, 2001 right here at the Portland Convention Center. We look forward to seeing you again at these events.

Rick Clements
PNSQC President and Conference Chair

## CONFERENCE OFFICERS/COMMITTEE CHAIRS

**Rick Clements – PNSQC President/Chair**
*Cypress Semiconductor*

**Ian Savage – PNSQC Vice President**
*SunGard Banking Systems -Tiger Systems Inc.*

**Howard Mercier – PNSQC Secretary, Program Co-Chair**
*STEP Technology, Inc.*

**Doug Vorwaller – PNSQC Treasurer**
*Willimette Industries*

**Sue Bartlett – Keynote 2000, 2001 Chair**
*STEP Technology, Inc.*

**Paul Dittman – Program Co-Chair**
*Intel Corporation.*

**Dennis Ganoe – Web Master**
*Professional Data Exchange*

**Shauna Gonzales – Birds of a Feather**
*ImageBuilder Software*

**Bhushan Gupta – Workshop Chair**
*Hewlett Packard*

**Sandy Raddue – Publicity Chair**
*Cypress Semiconductor*

**Piet van Weel – PNSQC Software Excellence Award**

**Don White – Exhibits Chair**
*BidTek*

## CONFERENCE PLANNING COMMITTEE

**Hilly Alexander**
*Cotelligent*

**Bob Brown**
*Intel Corporation*

**Kit Bradley**
*PSC Scanning, Inc*

**Mark Carver**
*Intel Corporation*

**Prabhala Ganesh**
*Intel Corporation*

**Manny Gatlin**
*Intel Corporation*

**Kathi Harris**
*Intel Corporation*

**Mark Johnson**
*OrCAD Corp.*

**Greg Jones**
*Nations Bank*

**Charles Knutson**
*Brigham Young University*

**Mike Kress**
*Boeing*

**Akiko McNair**
*Timberline Software*

**Allen Sampson**
*Intel Corporation*

**Chris Simmons**
*Intel Corporation*

**Erik Simmons**
*Intel Corporation*

**Vicki Shinneman**
*Hewlett Packard*

**Bryan Stickel**
*Flight Dynamics*

**Richard Vireday**
*Intel Corporation*

# Reflections on the Journey to World-Class Software Quality

George Yamamura

We can all remember when software development was a major problem area for many large integrated systems. Terms such as tiger teams, fire fighting and audits were common everyday activities. Programs faced reschedules, overruns, poor interfaces, frustrated employees and dissatisfied customers.

Within our organization, we have institutionalized a smooth running software development team using well-defined processes and trained employees. Our development capability and results are very predictable in producing high quality products with exceptional performance. Employees have a sense of pride, and desire coming to work in the mornings. This can be applied in any organization.

I would like to share some thoughts and insights that helped bring about this transition. There are two key items that come to mind. First, we approached process improvement in response to our business goals. Secondly, we acknowledged that process improvement is a human issue. Hear how we developed a management goal framework as a road map for the business case. Also, hear how we implemented improvements that addressed the human aspects of cultural change, a very simple four-step formula for success. This formula, applied to software development resulted in dramatic improvements and excellent performance. This is presented from a practitioner's viewpoint with how-to examples.

Following these major improvements and further development of deployment processes, we mapped our framework to the Software Engineering Institute (SEI) Capability Maturity Model (CMM) framework. These frameworks showed striking similarities, each validating the other, and demonstrating that the organization was operating at SEI CMM Level 5, and for the right reasons.

**George Yamamura** is a Software Engineering Process Manager. Formerly, he worked for Boeing in Seattle, Washington, supporting the Information, Space and Defense programs at sites throughout the United States. He managed the Space Transportation Systems (STS) software development organization and its efforts leading to a Software Engineering Institute (SEI) Capability Maturity Model (CMM) Level 5 rating, one of the few organizations in the world to attain that achievement.

Mr. Yamamura has thirty years of software experience in the space application field. His background covers orbital mechanics, data analysis, numerical analysis, statistics, business management, software engineering and processes. He has combined his knowledge in these areas and applied them to several software intensive programs. He has developed a proven formula for success that has achieved exceptional performances. He is a true practitioner who has implemented process improvement within his organization. He has written papers and given briefings at key software conferences, as well as, to International Executives in Europe and Asia. Articles have been written for the DoD CrossTalk Journal and third edition of the Handbook on Software Quality Assurance. He had been awarded the 1997 AIAA Technical Management Award and the 1997 Pacific NW Software Excellence Award.

He has a BS and a MS in Aeronautics and Astronautics from the University of Washington, and a MS in Applied Mathematics from the University of Santa Clara.

# SIMPLE STEPS TO ACHIEVE RESULTS WITH PROCESS IMPROVEMENT

by George Yamamura

# Topics

- Start the Change
- Overcome Barriers
- Implement with Results

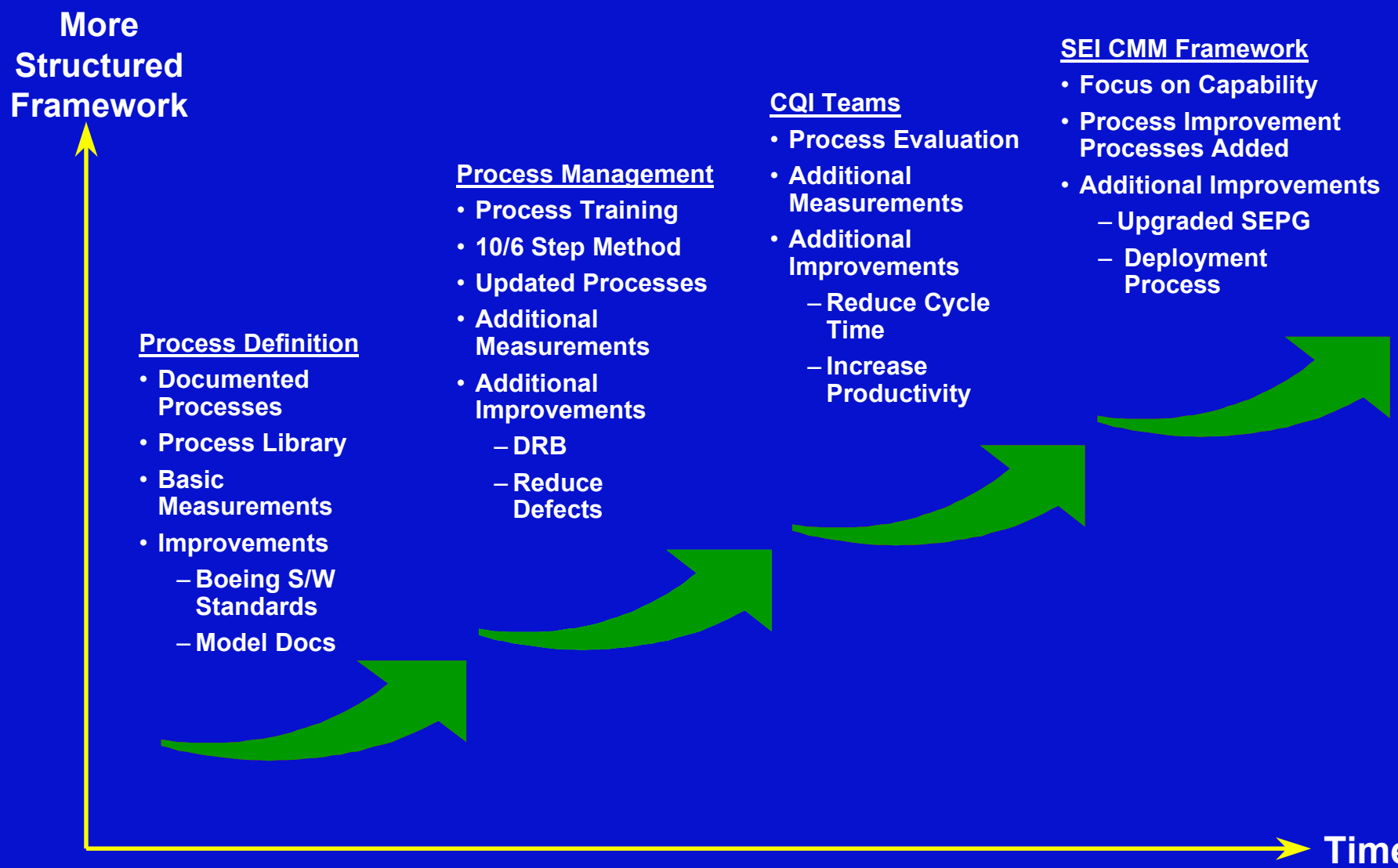
# Dark Ages of S/W Development

- Jump in and start programming
- Test your own programs
- Little or no documentation
- Little or no processes
- Small staff, turnover critical
- Work LONG, HARD hours
- Results:  High risk, many failures

# Evolution of Program Focus


**Process optimization allows enhanced technology with minimal cost and schedule impact.**

# Process Improvement History



# Ad Hoc Environment

- Many interface problems, misunderstandings, repeated mistakes
- Processes thought-up on the fly, reinventing the wheel
- Too busy with project work, no time to work improvements, just get the product out
- Little senior management sponsorship for process improvement, in tiger-team mode
- No extra budget for process improvement, did not want to pay for next project benefits

# Workshop Survey

1. What is your current job satisfaction level?

1. 10 Extremely satisfied
   9 Highly satisfied
   8 Very satisfied
   7 Satisfied
   6 Not quite satisfied
   5 Neutral/ don't care
   4 Not very excited
   3 Dissatisfied
   2 Very dissatisfied
   1 Highly dissatisfied

2. What is most important to you about your job?

2. _ Achievement & Recognition -(driven by accomplishment)
   ___ Advancement & Growth -(desire growth potential)
   ___ Relationships -(team dynamics is important)
   ___ Salary -(only pay matters)
   ___ Security -(regular income is most critical)
   ___ Supervision -(work for someone I respect)
   ___ Work Assignment & Responsibility -(must love my work)
   ___ Work Environment -(need nice work area)
   ___

3. What are the biggest issues or greatest barriers to improving your organization?

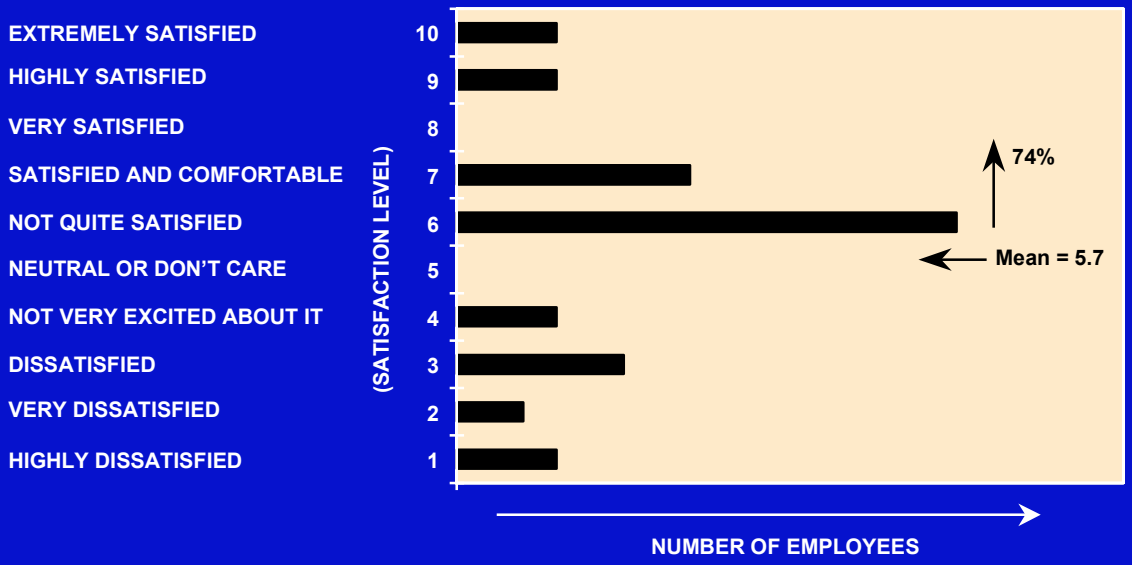
3. a._
_____

b._
_____

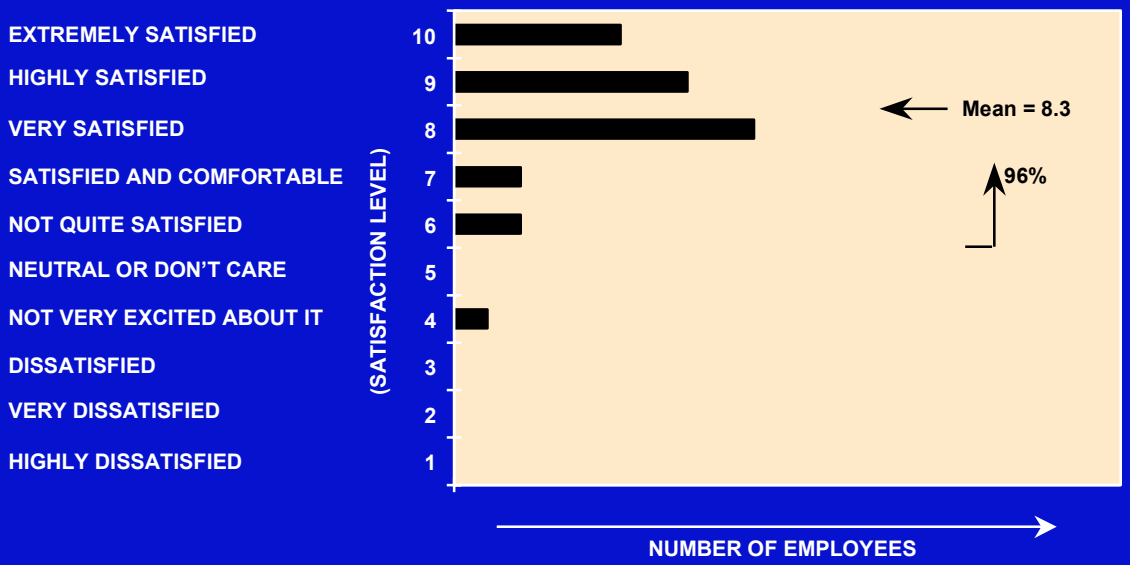
c._____

_____

# Employee Satisfaction




**Before Process Improvement Activities**




**After Process Improvement Activities**

# Employee Needs

- Achievement & Recognition
- Work Assignment & Responsibility
- Advancement & Growth

*Internal*

- Security
- Salary

*Financial*

- Work Environment
- Relationships
- Supervision

*External*

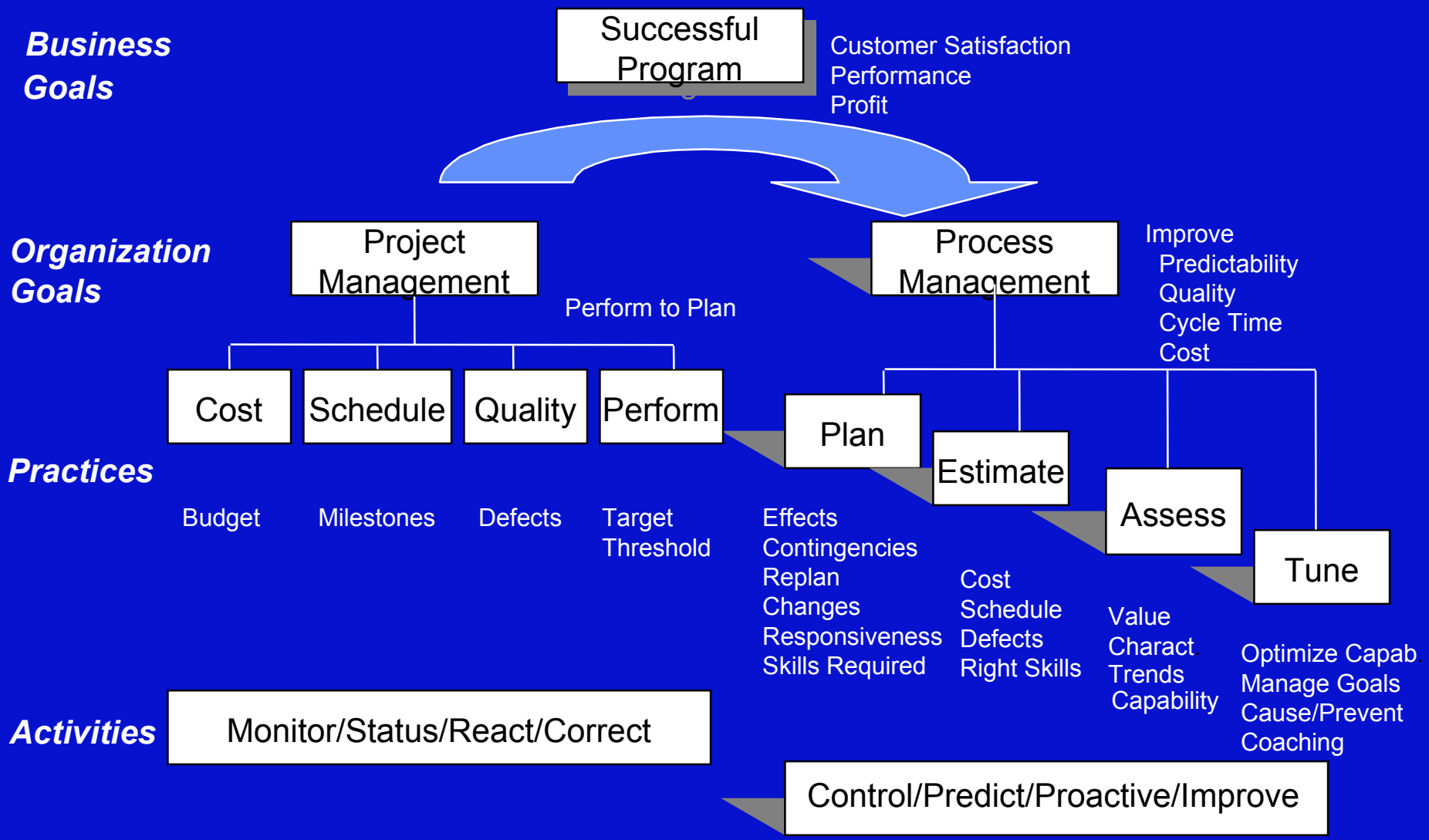
# Initial Steps for Change

- Gather facts and data
  - What is important to the workforce
  - What should be improved
  - What is the current satisfaction level
- Analyze the situation
  - Ad hoc environment
  - Employee inputs
- Implement effective decisions

# Effective Decisions to Reduce Defects

- Data analysis performed
  - Non-uniform peer reviews
  - Correlated to multitasking periods
    - Individuals work 3 or more concurrent tasks
  - Correlated to personnel turnover = .918
- Effective decisions
  - Inspection:  use checklists and accountable reviews
  - Training:  4 hours per week for 12 weeks
  - Human issues:  reduce multitasking and turnover
  - Ownership:  establish commitment and accountability

# Process Management Evolution



# STS Deployment Process

## Comprehensive Software Development for Space Transportation Systems Programs

Common processes across all STS programs reduce cost and risk. Programs start with mature processes in-place. Trained engineers provide domain expertise.

### Mature Processes

Space Transportation Systems takes advantage of many years of software process development and improvements to provide a highly successful and proven engineering methodology. Processes are the key to a well defined and disciplined approach for predictable performance, high quality products and excellence in customer satisfaction.

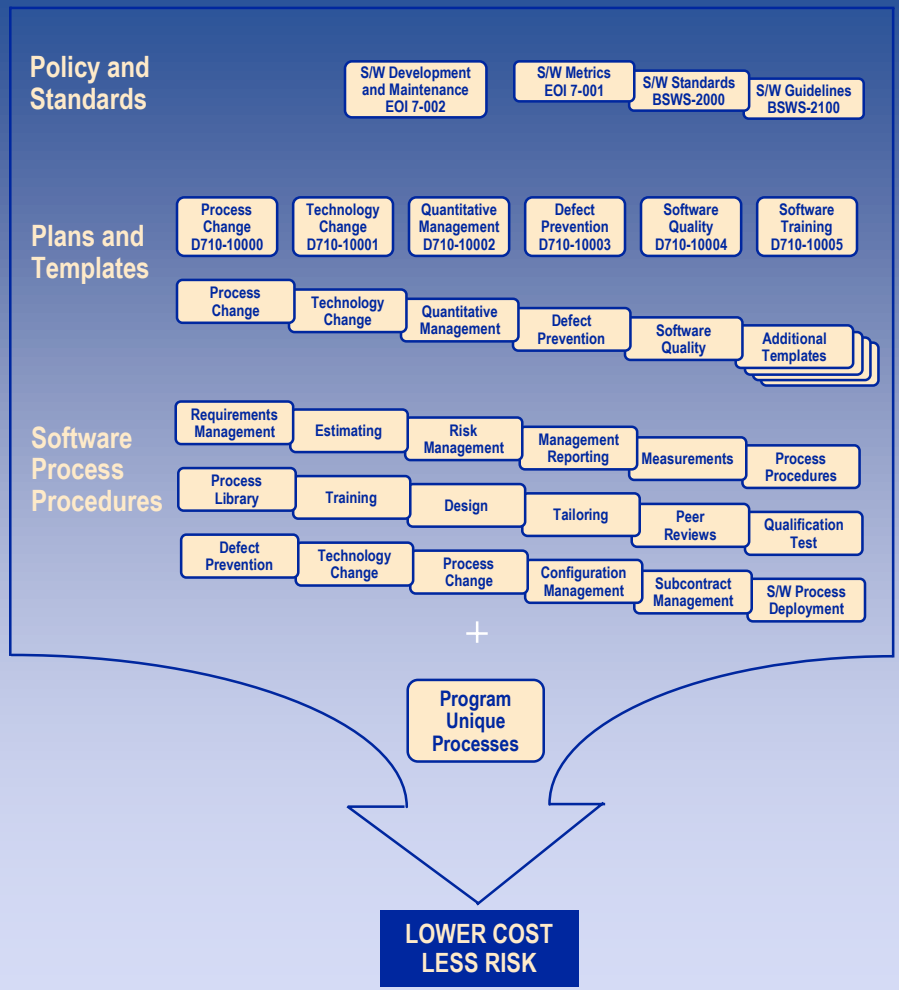

Mature processes define a predictable organization capability for the "big four": cost, schedule, quality and performance

### Benefits New Start Programs

Approximately 70% of policies, standards, plans and process procedures are in-place for managing a successful software program. Programs start with mature processes, minimizing cost and risk.

### Software Process Library

The STS Software Process Library, along with other existing program libraries, provide a complete set of documents to benefit existing and new start programs.

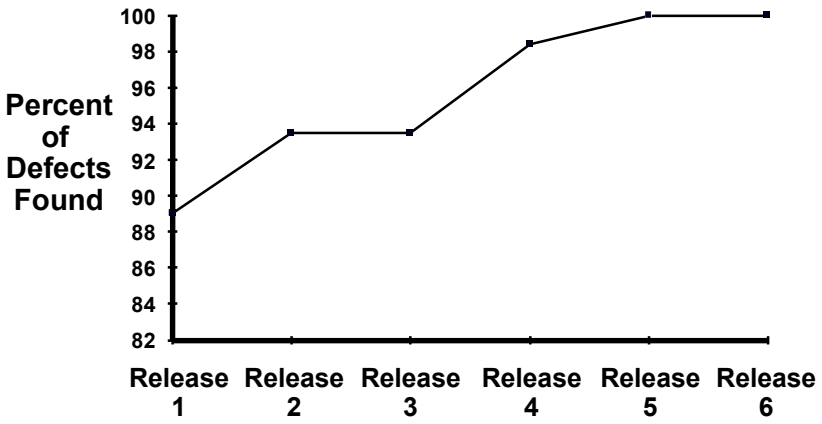



### Software Center of Excellence

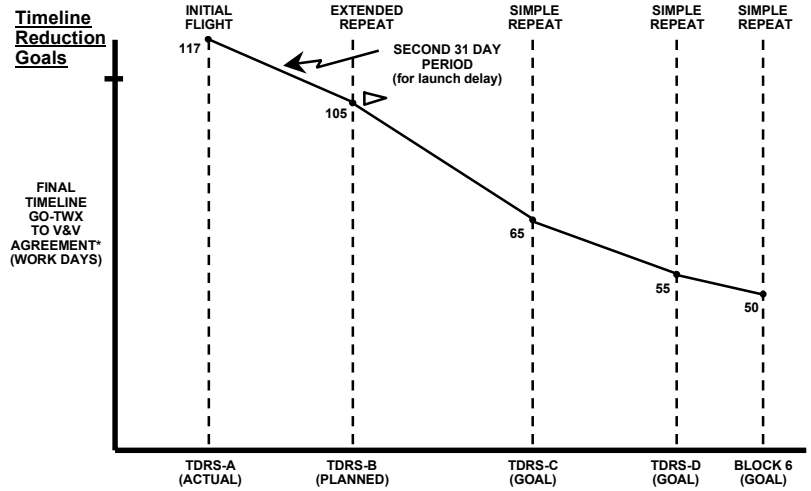
A team, trained in the software processes, provide the expertise for deployment. Engineers are experienced in multiple program applications. Orientation and training programs are applied for all engineers. The organization has attained a record of excellent performance.

# STS Significant Results

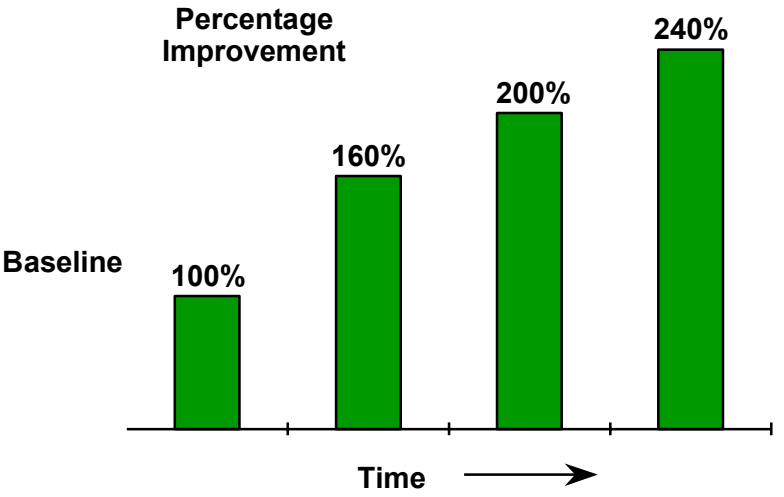
## Quality



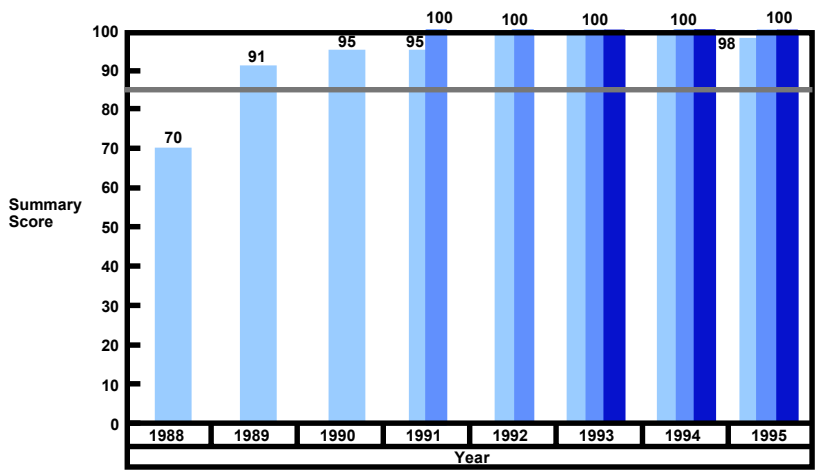

## Cycle Time Reduction



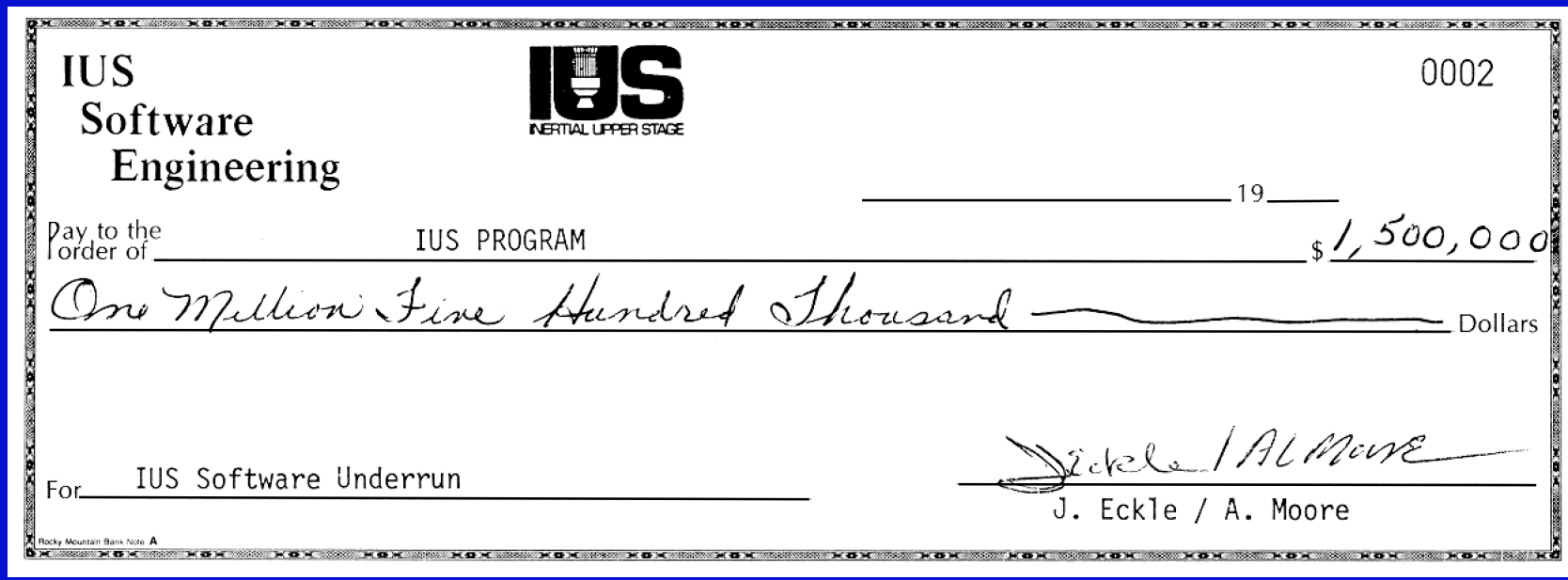

## Productivity




## Customer Satisfaction



# IUS Software Award



IUS
Software
Engineering

IUS
INERTIAL UPPER STAGE

0002

_____ 19____

Pay to the order of IUS PROGRAM $1,500,000

One Million Five Hundred Thousand ——— Dollars

For IUS Software Underrun

J. Eckle / A. Moore

Rocky Mountain Bank Note A

# Issues/Barriers Survey

## Data from SPI Forums/Workshops

- Gain consistent, constant commitment from all levels of leadership. Senior managers need to be champions.
- Reward the behavior you want.
- Keep funding for SPI above the budget line.

# Management Issues

- Decisions from facts and data
- Be part of the team
- Commit and accountable

# Commitment & Accountability

- **Document commitments**
- **Enforce accountability**

## Example of Resource Management/Accountability

| Name | Level of Effort | Overtime | IPT Lead/Manager |
|---|---|---|---|
| xxx | 1.0 | .3 | G.Y. |
| yyy | 0.8 | .1 | S.A. |
| zzz | 0.5 | .2 | C.M. |
| -- | 0.2 | | D.R. |
| -- | 0.1 | | R.B. |

Signatures:

G.Y. S.A. C.M. D.R. R.B.

# Overcoming Barriers to Process Improvement

- There was no barrier, but one was put in by focusing on an SEI level as the goal
- Start with business goals and concentrate on fixing the real problems, not achievement of a level
- Interpret what a level goal means to your organization. Look at what CMM feature will help your solution. After consulting with organization, 46% of Level 2 was covered.
- Getting started is a barrier itself, need a champion
- Top management sponsorship, need a business case
- Top management says to at least do it for awhile
- Metrics can be a barrier, right metrics are a motivator
- Share the success

# Challenges

- Senior Management has minimal understanding of SEI CMM or how long it takes to achieve each level.
- Some project S/W managers do not really believe SEI CMM benefits them. They think process improvement is just an extra overhead task and cost.
- Engineers are too busy with project direct work. They are working overtime already.
- No one really believes we can we prepared to be formally assessed in 6 months.

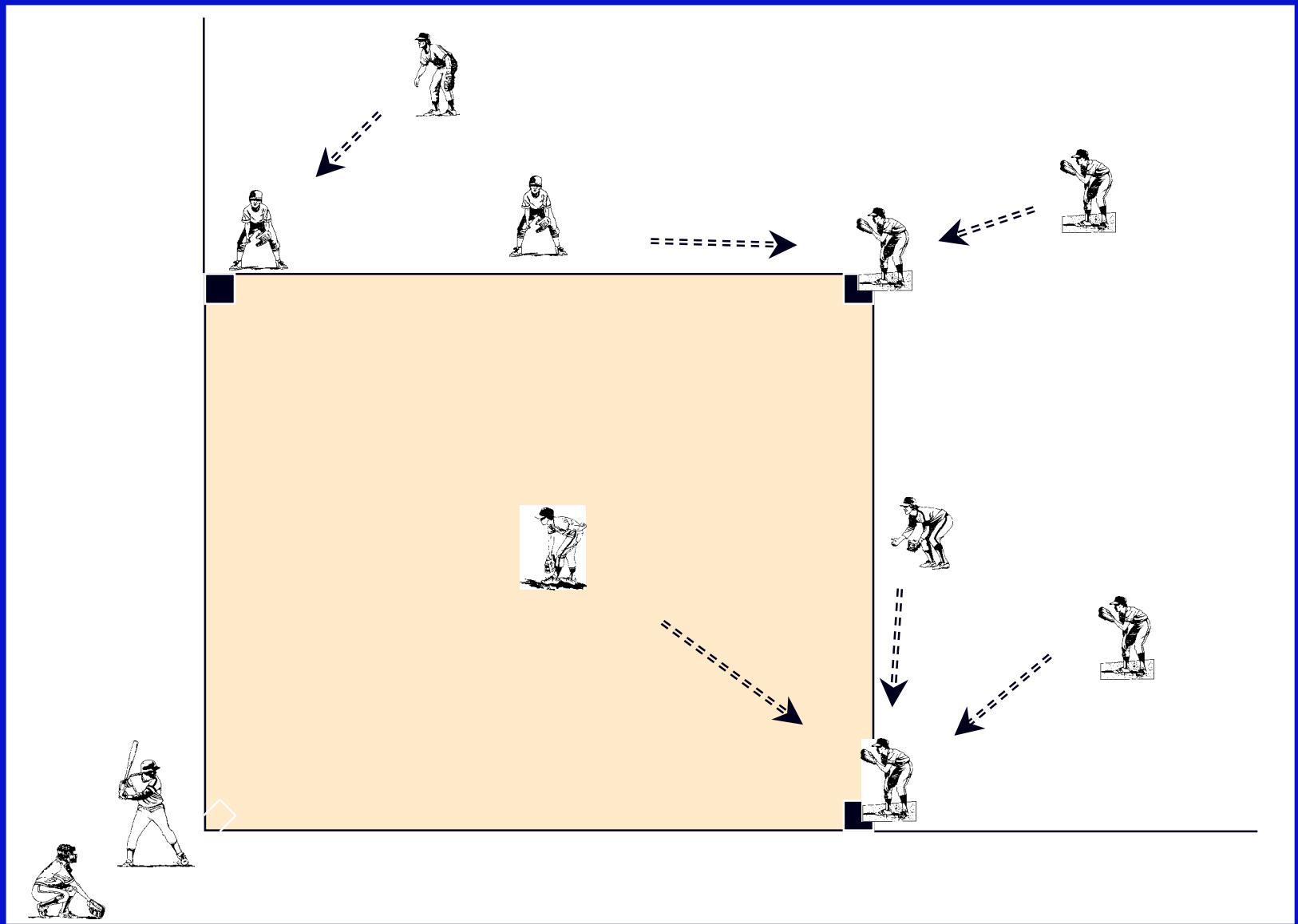
# Approach

- Sold to senior management
  - New business potential
  - Reduce cost and risk
  - Total commitment, awareness, part of a team
- Sold to the workforce
  - Achievement and recognition
  - Motivated
  - Growth potential
- Manage as  a project, not a side job
  - Planning
  - Budget
  - Schedules
  - Resources
  - Reports to management and customer
- Make visible what you do right
  - Market your work, make it easy to see
  - Develop the positive atmosphere

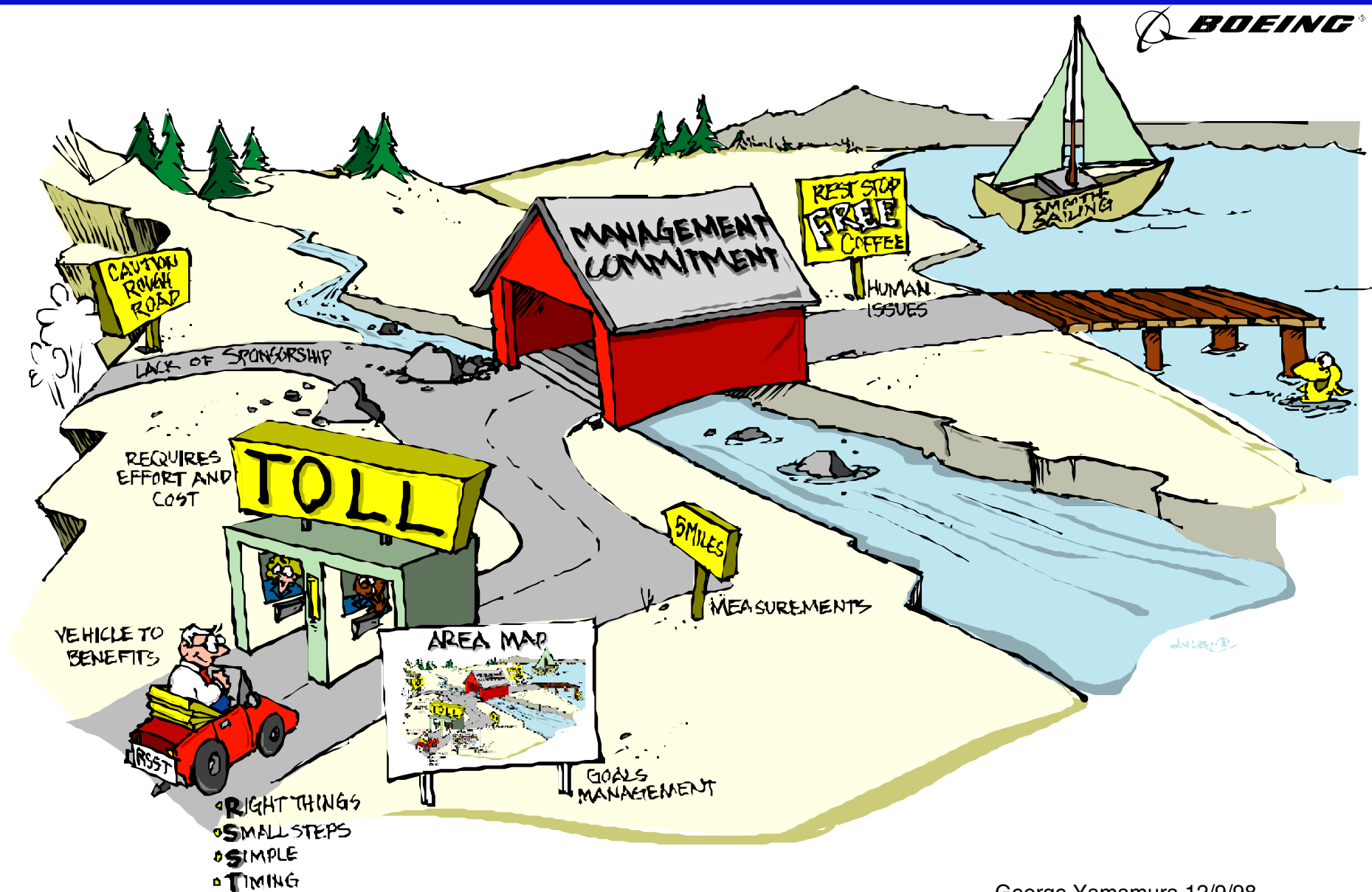
# Formula for Success

- Apply *RSST*:
  - Right Thing - Do the right thing for the situation, understand the problem/capability
  - Small Steps - Take small, do-able steps, get a success, look for highest value item to change
  - Simple - Keep it simple, look for the simple solution, use common sense, don't just follow the trend
  - Timing - Right timing is critical, when environment is right, able to apply right strategy

# Minimize Decisions Example



# The Journey




George Yamamura 12/9/98

# Effective Leadership

- **Did you make your decision from facts and data?**
  - State the:
    1. Fact
    2. Data
    3. Goal
    4. Decision
- **Did you apply goal management?**
  - Map the:
    1. Goals
    2. Practices
    3. Activities
    4. Measures
- **Did you recognize the human issues?**
  - Remember to:
    1. Ask
    2. Listen
    3. Act
    4. Feedback
- **Did you apply RSST to make a difference?**
  - Do:
    1. Right thing
    2. Small steps
    3. Simple
    4. Right timing
- **Did you keep senior management accountable?**
  - Have:
    1. Documented commitment
    2. Real ownership
    3. Continued awareness
    4. Visible recognition

# A Level 5 Dream

**Imagine what it would be like to work in an environment rich in well-coordinated and tested processes: everyone is trained and you could accurately estimate a job, finish it on time, within budget, and with exceptionally high quality. Nobody argues over who is at fault for a problem or who is responsible for a particular task. While you are dreaming, toss in some well-pleased managers and many proud employees. This is not a dream --- I am describing (our organization)...**

**Kimsey M. Fowler Jr.**
***CrossTalk, The Journal of Defense Software Engineering***
**Sep 1997**

# Inspired Performance

George,
I just returned from a Boeing banquet honoring my promotion to Technical Fellow, one of the key highlights of my career. I can attribute this accomplishment directly to working under your effective leadership. **You always focused on defining goals that were a stretch yet realistic, ensuring that there was visible employee growth, giving credit for accomplishments, and motivating us as a group and as individuals.** You did this through effective career counseling, by providing a vision for the group, by working the problems that kept me from being productive, and by allowing me to make decisions with you standing behind them. **You reminded me that my work was my own, and provided personal recognition for my accomplishments far beyond anything I had experienced before. It made me want to perform the best work that I could to make you and our organization look good. This is what makes a win-win situation!** I feel great satisfaction about our successes, and it could not have happened without you providing the environment and opportunities to excel. Thanks again, and I hope to experience other managers like you during the remainder of my career.

Gary Wigle
Technical Fellow

# RISK MANAGEMENT IS PROJECT MANAGEMENT!

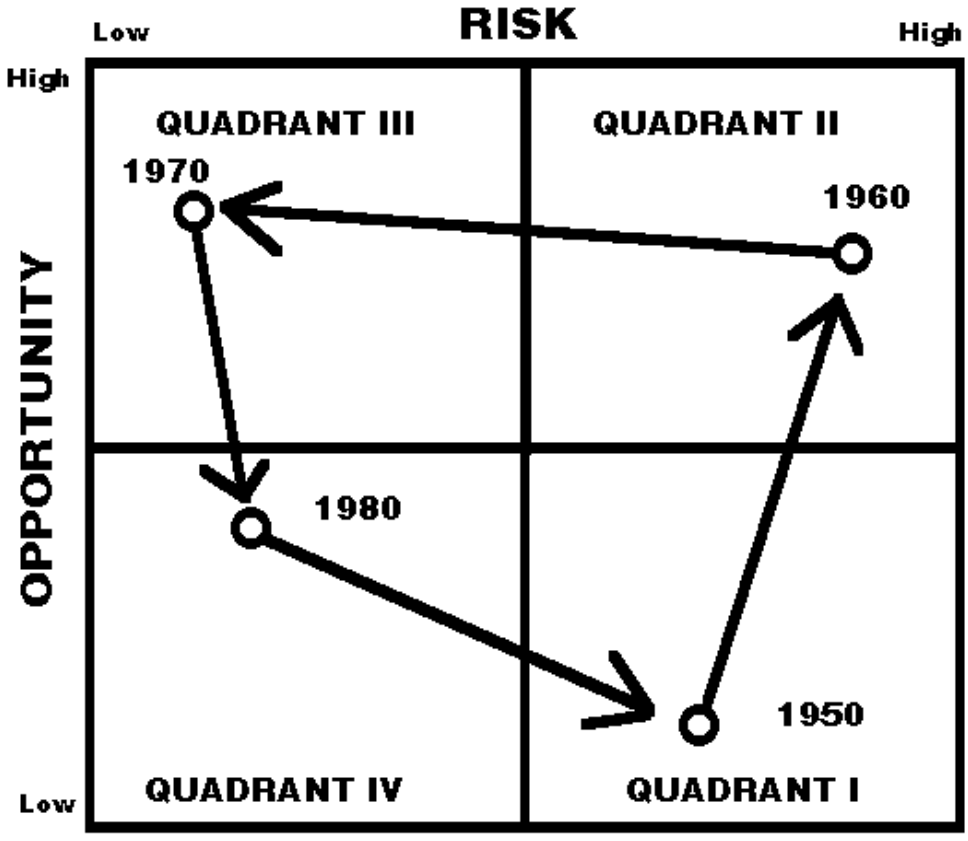
**Tim Lister**
**The Atlantic Systems Guild, Inc.**
**lister@acm.org**

**PNSQC**
**Fall, 2000**

"Risk Management decriminalizes risk."

- Paul Rook

# RISK AND OPPORTUNITY




**FROM CHARETTE, R.N., 'MANAGEMENT BY DESIGN,'' *SOFTWARE MANAGEMENT,* OCTOBER, 1993**

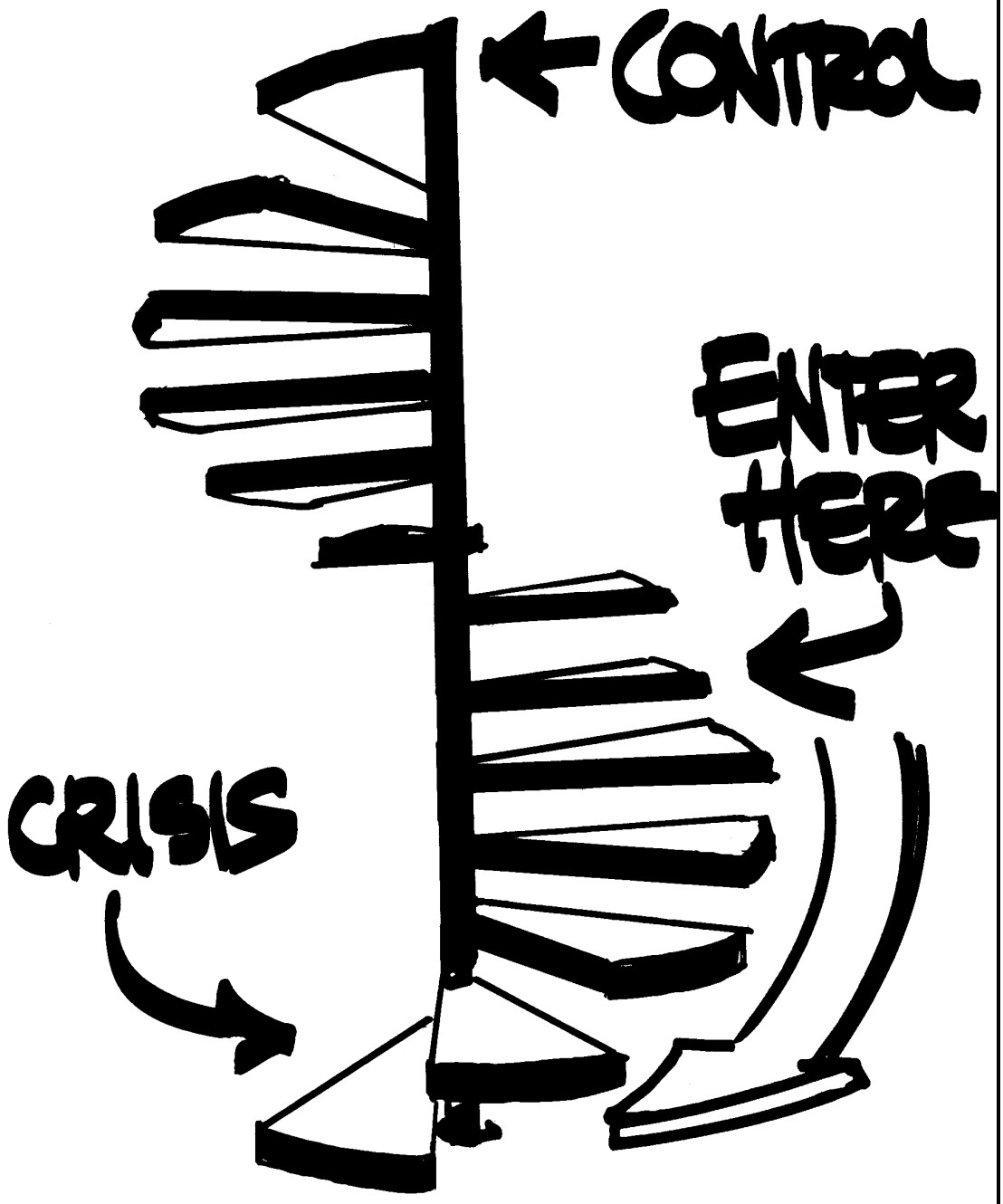
# RISK AND OPPORTUNITY

Risk Management is not the same as risk minimization.

> "All the projects with benefits and no risks were done years ago."
>
> - Tim Lister

We need to learn to *court* risks. That can't be done without some form of sensible risk management.

# CHARETTE'S RISK HELIX



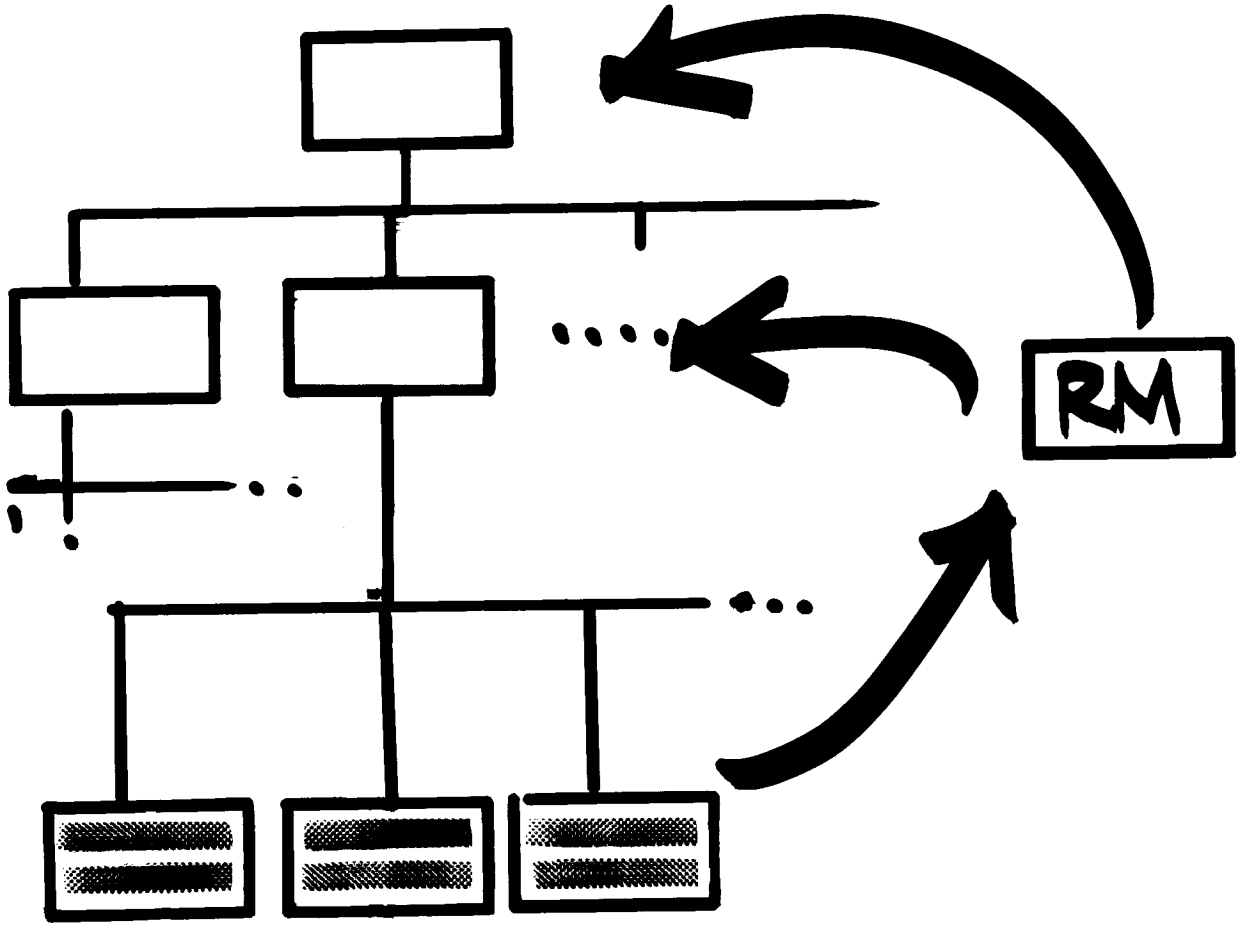
# CHARETTE'S RISK HELIX

The risk helix symbolizes the dynamic forced upon us by our competition:

- ❏ It takes forward progress to stay even.
- ❏ Standing still is a sure way to end up in crisis.
- ❏ Getting ahead (top of helix) means you can control the rate at which your competitor's helix spins.

Sliding down to the crisis level means that <u>new competition always enters above you</u>.

# WHO KNOWS THE BAD NEWS?



# CAN-DO MANAGEMENT . . .

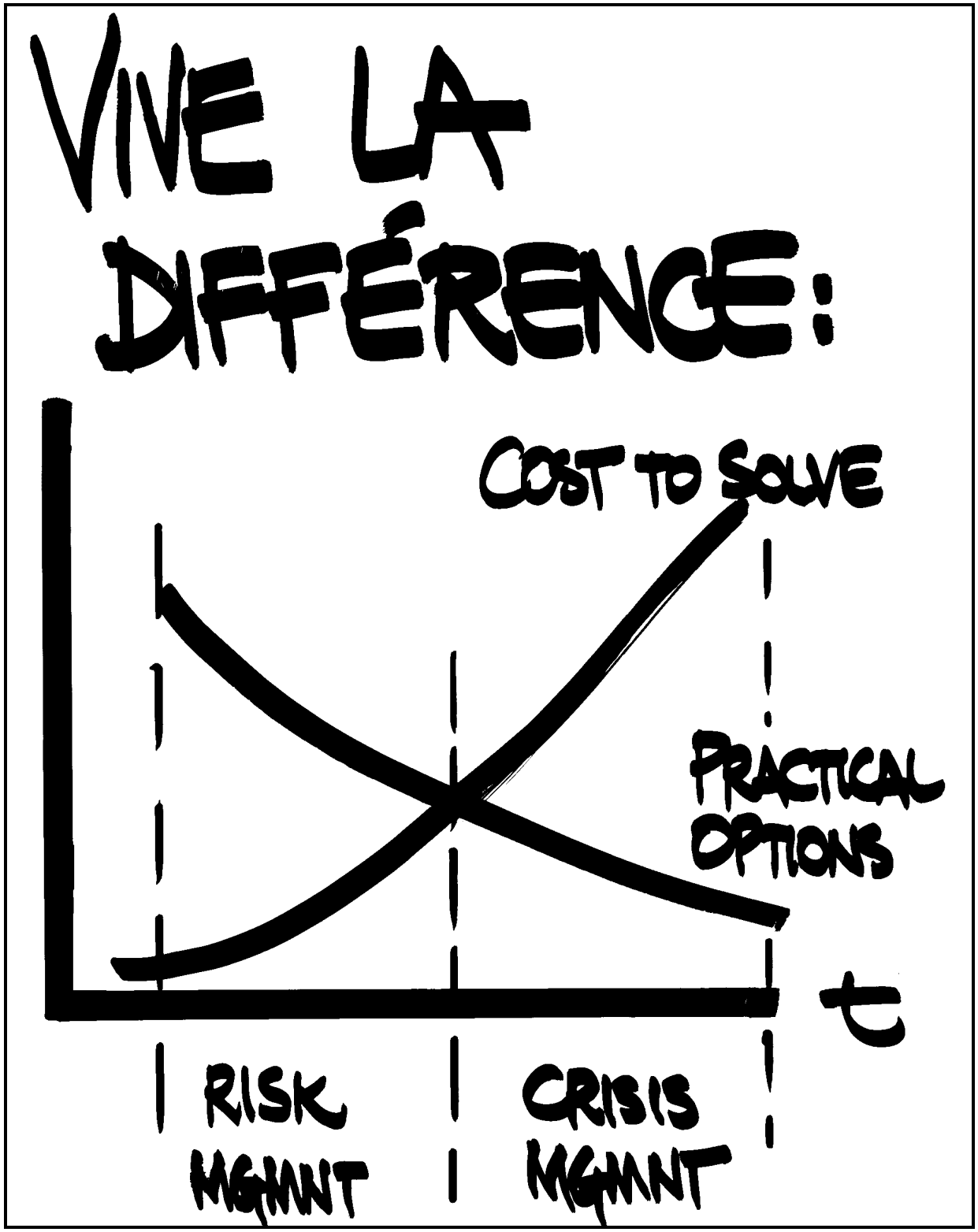
is a good thing, but:

- does it stop bad news from percolating up the hierarchy?
- does it focus us only on *curable* problems?

“Unless we do X right away, we’re not going to make the September delivery.”

as opposed to

“We’re not going to make September no matter what we do.”




Adapted from George Friedman: Address to the 3rd International Conference on Software Risk, Pittsburgh, 1994.

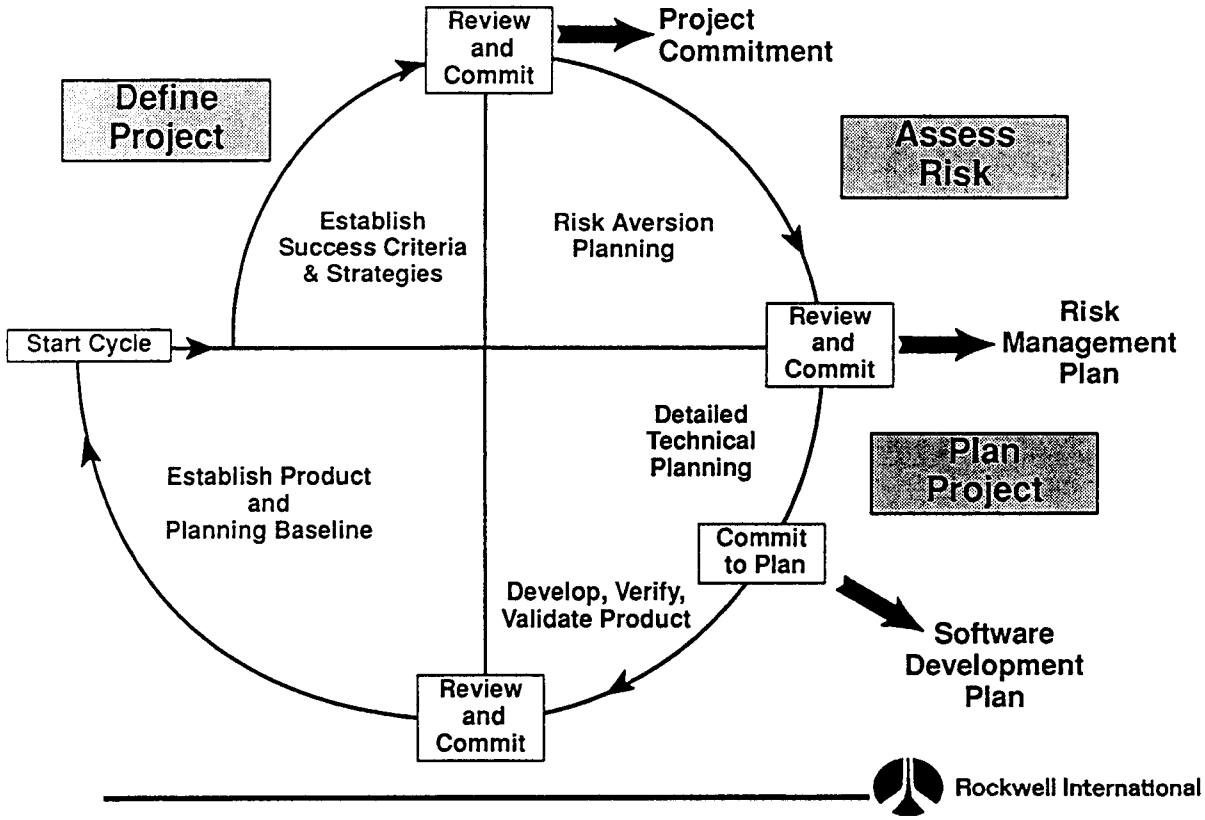
# RISK MANAGEMENT IS . . .

A methodical process for:

- risk identification
- risk evaluation
- risk ➔ problem transition* detection
- risk mitigation

* a *risk* is a potential problem
  a *problem* is a risk that has materialized

# ROCKWELL RISK MGMNT

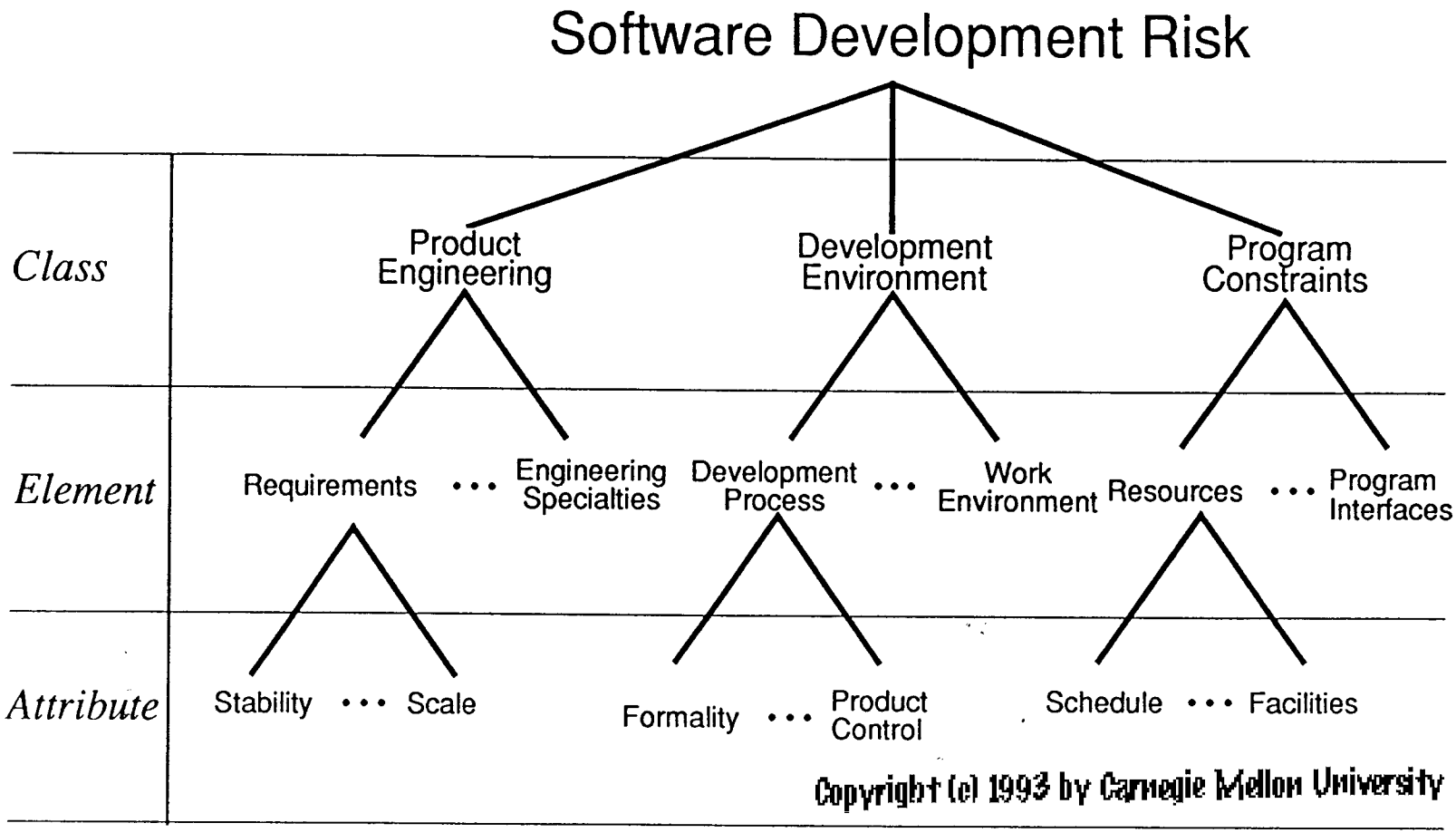



Gemmer A. and P. Koch, "Rockwell Case Studies in Risk Management," Proceedings 3rd International Conference on Risk Management, Pittsburgh, 1994.

# HOW-TO'S OF R.M.

1. create a census of risks
2. assess probabilities and effects of each risk
3. identify earliest expected symptom of each risk
4. agree on mitigation plans before first symptoms occur
5. monitor continually for symptoms (both foreseen and unforeseen)
6. keep the process going

# S.E.I.'s Taxonomy of Risks:



# CENSUS OF RISKS

Don't start off with a blank sheet. There are excellent risk checklists you can use. For example:

M.L. Carr et al: "Taxonomy-Based Risk Identification" (SEI-93-TR-006)

From SEI's taxonomy-based questionnaire:

[177] Is there any problem with getting schedules or interface data from associate contractors?

(No) (177.a) Are they accurate?

***If there are subcontractors***

c. Subcontractors
*[Is the program dependent on subcontractors for any critical areas?]*

[178] Are there any ambiguities in subcontractor task definitions?

[179] Is the subcontractor reporting and monitoring procedure different from the program's reporting requirements?

[180] Is subcontractor administration and technical management done by a separate organization?

Go to the SEI website and download a copy!

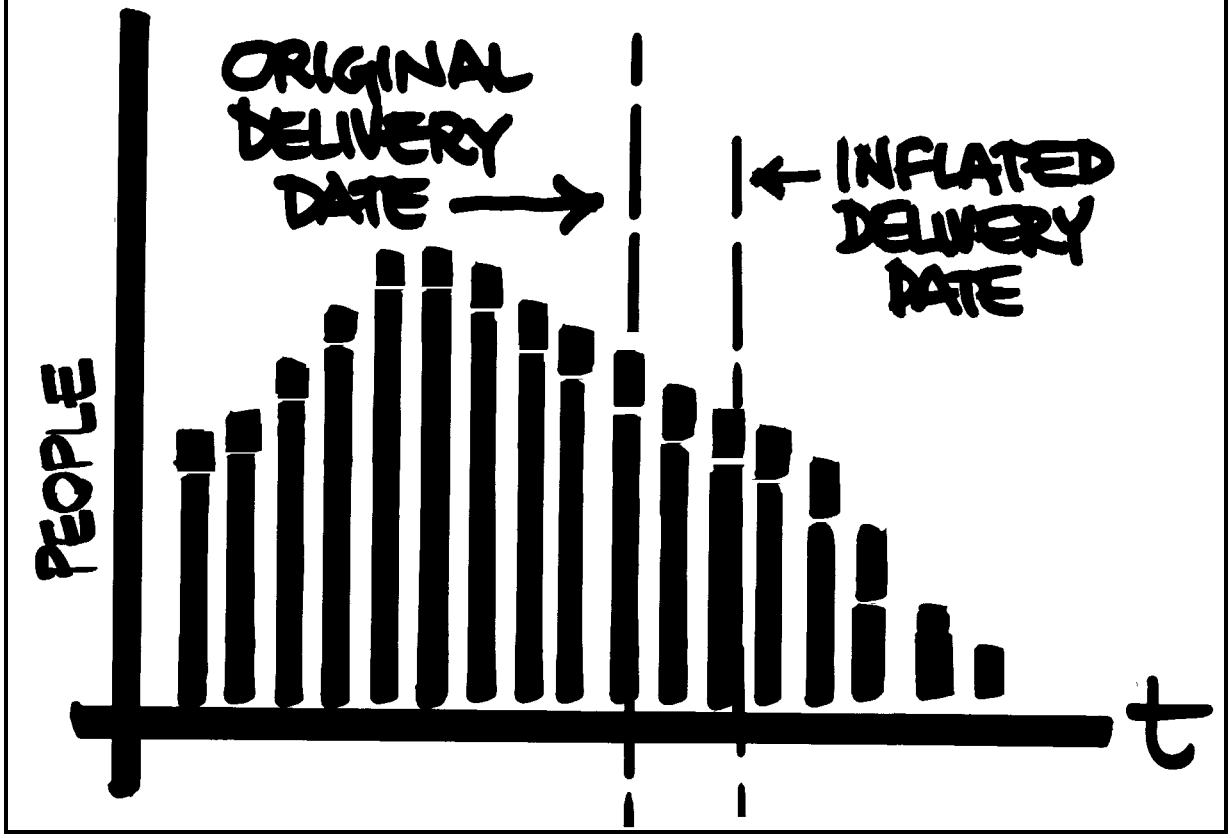
# RISK ASSESSMENT

For each risk:

- estimate cost impact
- estimate schedule impact
- estimate probability

Effective risk assessment requires at least some *political isolation*.

# RISK INFLATES BUDGET & SCHEDULE



# RISK ASSESSMENT . . .

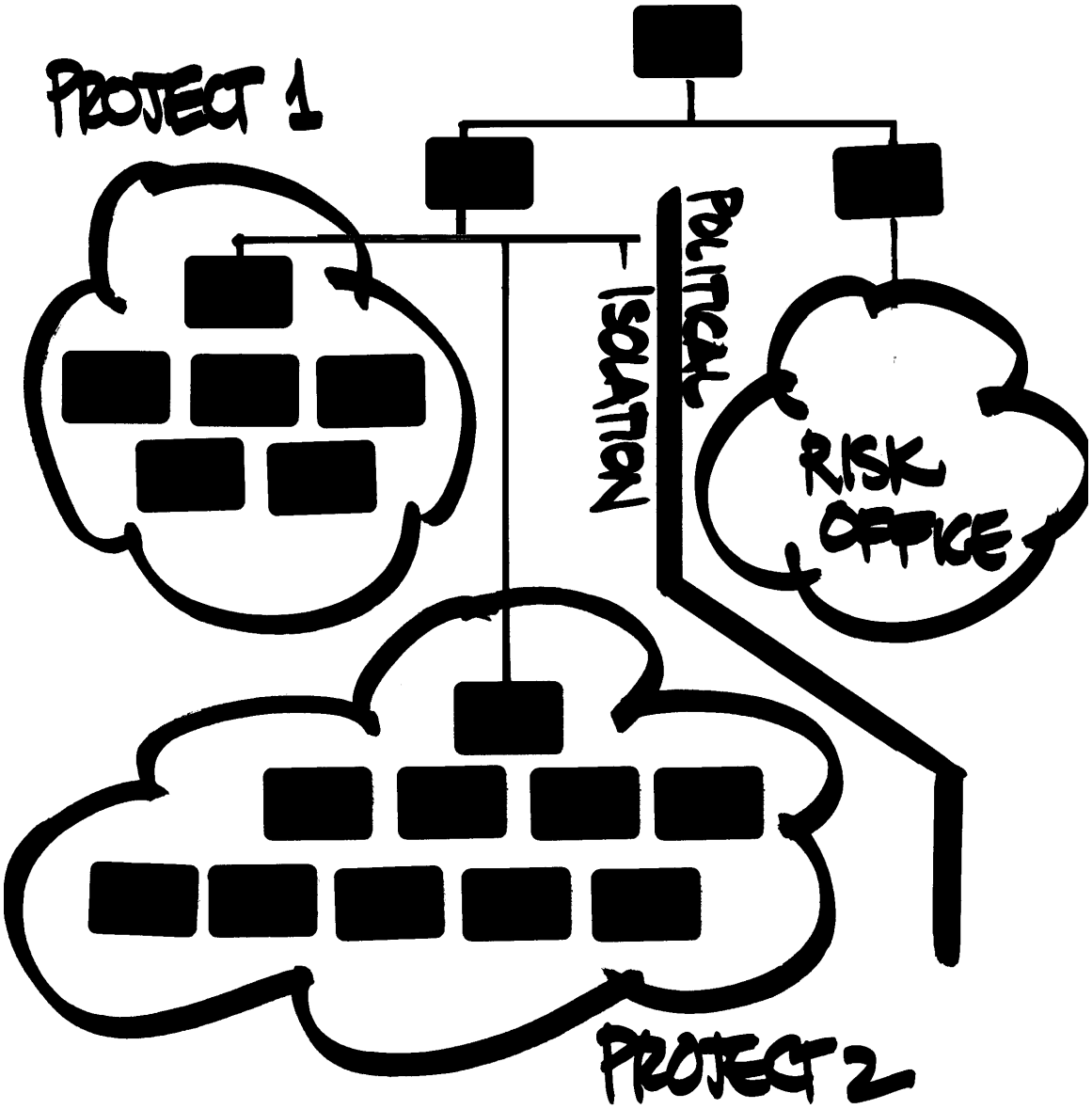
What does it mean to have a risk of probability P with expected cost C?

Consider an example:

| ITEM | R25 |
|---|---|
| RISK FACTOR: | May need to write specialty printer driver |
| EXTENT: | 500 function points |
| PROBABILITY | 20% |
| COST IMPACT: | $150,000 |
| SCHEDULE IMPACT: | +5 months |
| FIRST INDICATION | Failure of September 1 graphics print test sequence using standard driver |

Weighted impact of R25 is $30,000 and one schedule month.

# RM IN CONTEXT



**Your Folks Don't Mind Being on Risky Projects…**

***But..does your client***
***or***
***your management***

**have a hard time talking about anything but sunshine and clear roads ahead?**

---

**Maybe you can only do risk management**

**between consenting adults…**

# Good Luck on your Projects…

# JUST DON”T EXPECT IT!!!

# Methods For Effective C/C++ Unit Testing

A. Kolawa, C. Dunlop

## Abstract

Unit testing is an incredible technology that lets you reduce cost by preventing errors and reducing the number of errors in the code. Unit testing has traditionally been neglected because it has been perceived as being difficult and not cost-effective. However, recent developments in software technology enabled the automation of unit testing; these technologies are now available. This paper will explain why and how these automatic unit testing technologies can be integrated into virtually any development process. We will begin by defining unit testing, explaining its benefits, and explaining how to perform unit testing without automatic unit testing technologies. We will then describe how these technologies should be designed, demonstrate how they work, and explain how they can be integrated into the development process.

## What is Unit Testing?

Often, developers hear about unit testing and think the term refers to module testing. In other words, developers think they are performing unit testing when they take a module, or a sub-program that is part of a larger application, and test it. Module testing is important and should certainly be performed, but it is not the technique that we want to concentrate on in this paper. When we use the term “unit testing”, we are talking about an even lower-level testing--testing the smallest possible unit of an application; in terms of object-oriented languages, unit testing involves testing a class as soon as it is compiled.

Unit testing can dramatically improve software quality. Unit testing facilitates error detection by bringing you closer to the errors. Figures 1 and 2 demonstrate how unit testing does this.

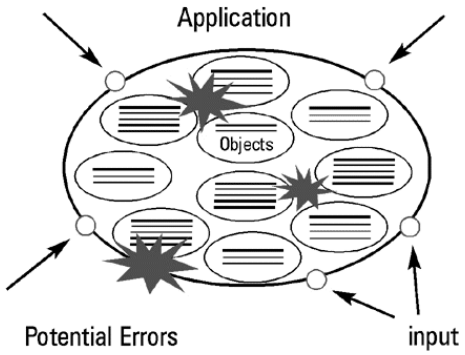



**Figure 1: Application Testing**

Figure 1 shows a model of testing an application containing many instances of multiple objects. The application is represented by the large oval, and the objects it contains are represented by the smaller ovals. External arrows indicate inputs. Starred regions show potential errors.

To find errors in this model, you need to modify inputs so interactions between objects will force the object to hit the potential errors. This is incredibly difficult. Imagine standing at a pool table with a set of billiard balls in a triangle at the middle of the table, and having to use a cue ball to move the triangle's center ball into a particular pocket-- with one stroke. This is how difficult it can be to design an input that finds an error within an application. As a result, developers that rely only on application testing may never reach many of the classes, let alone uncover the errors that they contain.

Testing at the unit level offers a more effective way to find errors. This is demonstrated by Figure 2.

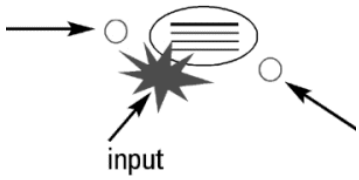



**Figure 2: Unit Testing**

As Figure 2 illustrates, when you test one object apart from all other objects, you can reach potential errors much easier because you are much closer to the errors. The difficulty of reaching the potential errors when the class is tested as an isolated unit is comparable to the difficulty of hitting one billiard ball into a particular pocket with a single stroke.

For the same reason, it is also easiest to achieve complete coverage at the unit level. While 100% coverage at the application is an unrealistic goal, 100% coverage is indeed possible at the unit level because it is so much easier to design inputs that reach each of the class's methods when you are testing at the unit level.

The second way that unit testing facilitates error detection is by relieving you from having to wade through problem after problem to remedy what began as a single, simple error. Because bugs build upon and interact with one another, pinpointing and remedying one error at higher levels often involves finding one problem after another. When you test at a higher level, your original error is like the innermost layer of an onion, and the additional errors are like the many layers that enclose that innermost layer: you can't even see the innermost layer until you peel away every layer that envelops it. When you test every class as soon as it is compiled, errors have little chance to build upon one another and interact to cause strange behavior. To extend the onion analogy, reaching an error at the unit level is as easy as reaching an innermost onion layer *that is not enveloped by any additional layers*.

The most significant effect of this easier error detection is its ability to slash development time and cost. There are several reasons why unit testing can reduce development time and cost at the same time that it improves application quality. First, because errors are easier to find, you consume less time and resources finding them. Second, because you are detecting and fixing errors as soon as you have written a class, you do not need to waste time relearning the class, as you would if you had delayed testing until later in the development process. Finally, the most important reason: because classes interact with and build upon one another, fixing one error at the unit level involves changing only the original class, while fixing one error at a higher level might mean changing the design and functionality of multiple program components. The later the problem is discovered, the more code must usually be changed in order to repair that error. And as the amount of code changed increases, two other factors also increase:

- The amount of time and money required to fix each error.
- The chance of introducing new errors into the code.

Study after study confirms that the time and cost involved in finding software errors rises dramatically the later a problem is detected. Consider the following data reported by Watts Humphrey (Humphrey 1995):

- IBM: An unpublished IBM rule of thumb for the relative costs to identify software defects: during design, 1.5; prior to coding, 1; during coding, 1.5; prior to test, 10; during test, 60; in field use, 100.
- TRW: The relative times to identify defects: during requirements, 1; during design, 3 to 6; during coding, 10; in development test, 15 to 40; in acceptance test, 30 to 70; during operation, 40 to 1000 [Boehm 81].
- IBM: The relative time to identify defects: during design review, 1; during code inspections, 20; during machine test, 82 [Remus].
- JPL: Bush reports an average cost per defect: $90 to $120 in inspections and $10,000 in test [Bush].
- Freedman and Weinberg: They report that projects that used review and inspections had a ten-fold reduction in the number of defects found in test and 50% to 8% reductions in test costs, including the costs of the reviews and inspections [Freedman].

Moreover, research has also confirmed that coding errors-- the types of problems that unit testing exposes-- account for a significant portion of the total errors found and of the total cost of fixing errors. Consider this data from Steve McConnell's *Code Complete* (McConnell 1993):

- Construction errors account for 45-75 percent of all errors on even the largest projects.
- In one study of coding errors on a small project (1000 lines of code), 75 percent of the defects resulted from coding, compared to 10 percent from analysis and 15 percent from design. (Jones 1986a). This error breakdown appears to be representative of many small projects.

- A study of two very large projects at Hewlett-Packard found that the average construction defect cost 25 to 50 percent as much to fix as the average design error (Grady 1987). When the greater number of construction defects was figured into the overall equation, the total cost to fix construction defects was one to two times as much as the cost attributed to design defects.

Unit testing can be divided into at least two distinct processes. The first process is black-box testing, the process of discovering functionality problems. When performed at the unit level, black-box testing checks a class's functionality by determining whether or not the class's public interface performs according to specification; this type of testing is performed without knowledge of implementation details. Performing black-box testing at the unit level lets you ensure that the methods in your class function properly-- before one minor functionality problem spurs a myriad of difficult to find and fix problems.

The second process is white-box testing, the process of discovering construction problems. When performed at the unit level, white-box testing validates that unexpected inputs to a class will not cause the program to crash; this type of testing must be performed by someone with full knowledge of the class's implementation details. By performing white-box testing, you can prevent crash-causing errors and ensure that your classes are robust (i.e., your classes perform well even when faced with unexpected inputs).

You can use these two processes as the basis for a third process: regression testing. If you save your black-box and white-box test cases, you can re-run them to perform unit-level regression testing and monitor your existing code's integrity as you modify it. The concept of performing regression testing at this level is novel. When you perform unit-level regression testing, you can immediately determine if your modifications introduced problems. This way, you can fix problems immediately after you introduce them, and you do not have to wade through layers of code in order to fix each error that you introduced.

## Why Automate Unit Testing?

Based on the above information, unit testing sounds like a panacea. If so, why doesn't every developer perform unit testing on every class as soon as he or she compiles it? When performed without automatic unit testing technologies, unit testing is difficult, tedious, and time-consuming. A brief look at what is involved in unit testing reveals why it is difficult-- if not impossible-- to integrate it into today's development cycles without an automatic unit testing tool.

The first step in performing unit testing is making the class testable. This requires two main actions:

- Designing scaffolding that will run the class.
- Designing stubs that return values for any external resources that are referenced by the class under test, but that are not available or accessible.

Creating scaffolding involves creating a new class that cannot be used for anything other than testing the original class. According to Hunt and Thomas (Hunt, Thomas 2000), scaffolding should include the following features:

- A standard way to specify setup and cleanup.
- A method for selecting individual tests or all available tests.
- A means of analyzing output for expected (or unexpected) results.
- A standard form of failure reporting.

In order to test the class thoroughly and accurately, you need to design scaffolding that fully exercises the class under test; several modifications or rewrites may be required to create such scaffolding. Once the scaffolding is created, you must examine it carefully to ensure that it does not contain any errors. An error in the scaffolding can sabotage the test, but because you cannot test a class in isolation (the original problem), you cannot test the scaffolding either.

To help illustrate the process of creating scaffolding, let's look at some very simple C++ examples. Suppose we want to test a simple `LinkedList` class that is only linked in one direction. This `LinkedList` contains the variable `value`, which stores the value of the LinkedList, and the variable `next`, which represents the next element. Each element in the list has a container that contains a value, and a link to the next element. You can see that we also create and print the `LinkedList`, and that we have created a method called `test` that is sensitive to one element in the `LinkedList`:

```
class LinkedList
{
public:

    LinkedList(int value, LinkedList * next):
        _value(value), _next(next)
        {}

    static void print(LinkedList * list) {
        while (list) {
            std::cout << list->_value << std::endl;
            list = list->_next;
        }

        return;
    }


    static void test(LinkedList * list) {
        if (!list->_next) {
            return;
        }

        if (!list->_next->_next) {
            return;
```

```
        }

        if (list->_value == 21342
            && list->_next->_value == 8765
            && list->_next->_next->_value == -1000) {

            int * null = 0;
            std::cout << *null << std::endl;
        }

        return;
    }

private:

    int _value;
    LinkedList * _next;
};
```

To test our class, we must create and print the LinkedList, then call the test. We can jump from one container to the next, in much the same manner as one would jump from one car to the next on a train, and we have specified that each element must be linked to the element that is one integer greater (represented by i+1 in the code):

```
#include <iostream>

#include "LinkedList.h"

int main()
{
// Create
    LinkedList * list = new LinkedList(0, 0);
    for (int i = 0; i < 10; ++i) {
        list = new LinkedList(i+1, list);
    }

// Test
    LinkedList::test(list);
    LinkedList::print(list);

    return 0;
}
```

When we run this test case, we get the following output:

```
10
9
8
7
6
```

```
5
4
3
2
1
0
```

We expected to receive this output because we have linked our integers in ascending order. Our scaffolding does not throw an exception. So we're done...right? Wrong. We still haven't had a chance to get into our code because one of our values is never what it is supposed to be. It just so happens that there are hidden errors in our class.

These hidden errors raise the question, "How do we know when we're finished with unit testing?" The goal in unit testing is 100% coverage. We need to dig into each method, find the exceptions, repair them, and then start the process over again. Only when each class comes up clean will we be finished.

To that end, we must test our `LinkedList` again. We need to have a very specific set of elements in the `LinkedList` in order to find a mistake, and we need to create a set of `LinkedLists` that contain these elements. We have intentionally written the following example in a peculiar manner to illustrate the difficulty of finding the error here. To find the hidden error, we need to write some new scaffolding that will look for the specific error we want to find:

```
#include <iostream>

#include "LinkedList.h"

int main()
{
// Create
    LinkedList * list;

    list = new LinkedList(0, 0);
    list = new LinkedList(-1000, list);
    list = new LinkedList(8765, list);
    list = new LinkedList(21342, list);

// Test
    LinkedList::test(list);
    LinkedList::print(list);

    return 0;
}
```

There was a piece of code we hadn't tested, but by writing the correct piece of scaffolding above, we finally got the exception we were looking for. Actually, you can see just by glancing at the original code that we call for an element [7], but there is no 7th element. If we had never bothered

to execute this segment of the code, we would have seen the error turn up later in the larger application, where it would have been more costly to fix.

If your class references any external resources (such as external files, databases, and CORBA objects) that are not yet available or accessible, you must then create stubs that return values similar to those that the actual external resources could return. When creating these stubs, you need to choose stub return values that will redirect the program's instruction pointer. . .

- in whatever directions must be executed in order to test the class's functionality, and
- in enough directions to provide thorough coverage of the class.

For example, imagine that we wanted to create a stub for the following simple C++ class that is going to talk to a database which is not yet constructed:

```
class db {
public:
    db (string s) {
        data_base my_file(s);
    }
    void read_data() {
        int i;
        i = my_file.read('k');
        cout << "Value for k = " << i << endl;


        i = my_file.read('o');
        cout << "Value for o = " << i << endl;


        i = my_file.read('a');
        cout << "Value for a = " << i << endl;
    }


public:
    int MAX;
    data_base my_file;
};
```

If our class is going to talk to a database but the database is not available yet, we will need a class called `database` for our class to talk to during testing. Our simple database is going to be a list of letters, and for each letter, the database will return the position of that letter in the alphabet, with "a" being 0 and "z" being 25.

In the coding example above, we want to find out the position of the letters "k", "o", and "a". Though this example may seem simplistic, it operates on the same principle as a database that returns phone numbers or identification numbers for a company's employees. We could now go and write a scaffolding class that would return the values of all the letters from "a" to "z", but

such a class would give us too much unnecessary information. What we really need is a class that returns the value “10” for “k”, “14” for “o”, and “0” for “a”. We don’t care about the other values because we are not using them in our class:

```
void main() {
    db m = db("junk");
    m.read_data();
}
```

The scaffolding creates a class called `db` and simply reads it, so it must have some sort of constructor. The constructor is taking the string, so it should connect to the database with a specific name. As for now, however, we do not care what the name of the database is because when the database is eventually available, we will make sure that our code will find the database. For now, we just want our code to create the database for testing purposes:

```
#include <iostream>
#include <string>


using namespace std;


class data_base {
public:
    data_base(){};
    data_base(string s){};
    int read(char val){
        switch (val) {
        case 'k':
            return 10;
        case 'o':
            return 14;
        case 'a':
            return 0;
        default:
            return -1;
        }
    }
};
```

In writing the stub, we want to create code that will mimic the actions of the actual database, but we want to write as little code as possible so that we do not waste time performing extra testing. The stub has a constructor; it takes a string but does nothing with it. We instruct the stub to return the values we want for “k”, “o”, and “a”. The stub will return -1 for any other input, so if someone else tries to use this class for testing something else, the values returned will be wrong.

We would have to extend the stub if we wanted to perform more testing, but we have written the stub to perform exactly the testing we need right now. A stub is typically a table, which essentially says, “For this input, return this value, and for that input, return that value.” We can write the stub as a switch statement that will do exactly what we need it to do. Rather than trying to re-write the entire class in the stub, which would defeat the purpose of the stub, we merely write the specific cases we need to test our piece of code.

The next step is designing and building appropriate test cases. In order to thoroughly test the class’s construction and functionality, you should design two types of test cases: black-box and white-box.

Black-box test cases should be based on the specification document. Specifically, at least one test case should be created for each entry in the specification document; preferably, these test cases should test the various boundary conditions for each entry. You should not just verify that a simple input produces the expected outcome; rather, you should consider what range of input/outcome relationships you need to verify in order to prove that the specified functionality is implemented correctly, then write test cases that will fully verify that functionality. You may want to test for omissions as well as faults of commission.

White-box test cases should uncover defects by fully exercising the class’s methods with a wide variety of inputs. These test cases should try to do two things:

- Aim for 100% coverage of the class As we explained earlier, this degree of coverage is possible at the unit level because it is so much easier to design inputs that reach all of the methods when you test a class apart from an application. 100% coverage may not be possible in all situations, but it is the goal that you should strive for.
- Make the class crash. For example, if you are testing a Java class, the test cases should aim to uncover as many undocumented, uncaught runtime exceptions as possible.

However, it is incredibly difficult to create such test cases on your own, without a technology to create them for you. To create effective white-box test cases, you must examine the class’s internal structure, then write test cases that will cover all of the class’s methods as fully as possible, and uncover inputs that will cause the class to crash. Achieving the scope of coverage required for effective white-box testing mandates that a significant number of paths are executed. For example, in a typical 10,000 line program, there are approximately 100 million possible paths; manually generating input that would exercise all of those paths is infeasible.

After these test cases are created, you should execute the entire test suite and analyze the results to determine where errors, crashes, and weaknesses occur. You should have a way to run all of these test cases and easily determine which test cases had problems.You should also gauge coverage to determine how thoroughly the class was tested and to determine what additional test cases are necessary.

Any time that a class is modified, you should perform regression testing to ensure that no new errors were introduced and/or that previous errors were corrected. There are two ways you can perform regression testing. The first way is to have a developer or tester examine every test case

and determine which test cases are affected by the modified code. This approach uses human effort and time to save the computer work.

A more efficient approach is to have a computer automatically run all test cases every time the code is modified. When you take this approach, you can preserve precious and costly developer time because you do not need to have a developer examine the entire test suite to determine which test cases need to be run and which do not. Even if you need to add additional systems to run all of your test cases in a reasonable period of time, you will save money: it is always cheaper to add systems than to pay developers good money to perform menial tasks.

# How to Automate Unit Testing

As you can see, unit testing is very beneficial to software quality; however, without automatic unit testing technologies, it is so complicated that it is virtually impossible to integrate it into the development process.

There have been multiple attempts to build schemas for unit testing. Most of these schemas are based on some type of container classes or templates. One example of such a schema is the framework entailed in Chuck Allison's recent article in *C/C++ Users Journal* (Allison 2000). Allison's technique is to set up templates that automate such mundane yet time-consuming tasks as running the tests and reporting failures. Allison is attempting to create "TheSimplestThingThatCouldPossiblyWork". His proposed test framework is GUI-less and customizable; it consists of two classes: one class which keeps track of successes and failures and prints failure information, one class that collects and runs test cases. While this approach does save some time, it does not automate enough of the process to allow developers to feasibly integrate it into their development cycle. Developers still need to build scaffolding, call the functions to be tested, and design all of the test cases.

Another solution is to use GUI-based tools such as Beck and Gamma's JUnit and CppUnit. These tools offer a more user-friendly way to automate the same processes as Allison's framework, but-- like Allison's framework, only automate a small percentage of the unit testing process.

The problem with these approaches is that they require a lot of effort from the developer, and developers tend to resist tools that require significant amounts of user intervention or that do not integrate seamlessly into their development process. The previously mentioned approaches may be adopted by high-level developers that are very determined to achieve the highest quality possible; however, the average developer may likely resist them.

Our company strived to fully automate unit testing so that the entire process could be performed with just one click. We wanted a completely automatic solution: a solution that would let developers test their classes with one click of a button-- without requiring them to create any scaffolding, test cases, or stubs. We decided to build technologies that would allow our own and other developers to easily perform this beneficial type of testing. Currently, we have developed automatic unit testing technologies for Java and C/C++. We will now describe what features such a technology should have and demonstrate how it could be used.

## What Should An Automatic Unit Testing Technology Do?

As we were designing our unit testing technologies, we researched what types of tasks such technologies should perform. We decided that the following functionality was the most critical:

- **Automatically create a test harness:** First, the technology should read and analyze the code under test. After it understands the code, it should determine how to create scaffolding and stubs, then automatically develop scaffolding and stubs to create an infrastructure that lets it write tests. This infrastructure should create enough hooks so that it can call each method multiple times without requiring recompilation, and it should relieve developers from having to add any function calls.
- **Automatically generate test cases**: The technology should automatically determine what test cases are needed to achieve maximum coverage of the function or method under test. It should try to achieve 100% test coverage: it should monitor coverage with each test, and continue creating tests until it has covered all of the possible paths.
- **Allow user defined test cases:** The technology should let users create their own test cases in order to verify whether the unit performs according to specification. It should offer an easy way for users to enter both simple and object-type inputs. Ideally, users should not need to enter outcomes; the technology should automatically determine the actual outcomes, then let the user correct any incorrect outcomes.
  By allowing users to enter their own test cases, the technology can help developers achieve close to 100% total coverage.
- **Run the test and report errors:** The technology should run all of the test cases (or selected test cases) against the class under test and report only test cases that failed or resulted in errors. It should be able to run the test cases automatically, with no human supervision.
- **Automate regression testing:** The technology should be able to save all automatically-generated and user defined test cases and test parameters, then re-run all or selected test cases and report any new problems. As long as the modifications did not intentionally change class functionality, this will detect if the modifications caused any undesired functionality changes or introduced new construction problems. The technology should provide a batch mode for regression testing, so that the tests can run in the background every night and developers can instantly see errors when they arrive in the morning.

To give you an idea of how these technologies actually work, we will describe what happened when we used them to test samples of well-known pieces of code: we tested Netscape 5, Milestone 10 Source Code (Released 09/28/99, Updated 10/08/99) files with our C/C++ technology, and we tested JDK 1.2.2 Package Source Files with our Java technology. Here are the results we received when we tested these classes without any user intervention other than telling the technology what to test and clicking the Start button:

**Test 1: C++ Code**
**Netscape 5, Milestone 10 Source Files (Released 09/28/99, Updated 10/08/99):**

nsAbCardProperty.cpp

Total Coverage: 29.0% [321/1089]

nsStreamTransfer.cpp

***Total Coverage: 66.0% [280/420]***

nsCThreadLoop.cpp

Total Coverage: 76.0% [84/110]

nsMailboxService.cpp

Total Coverage: 67.0% [636/940]

nsQuickSort.cpp

Total Coverage: 22.0% [20/87]

** All of the above coverage is the number of lines covered versus the total number of lines in a function **

**Test 2: Java Code**
**Common JDK 1.2.2 Package Source Files:**

BitSet.java

***Total Coverage: 56.0% [56/100]***
Multi-condition branch: 46.2% [37/80]
Method: 94.4% [17/18]
Constructor: 100.0% [2/2]

Hashtable.java

***Total Coverage: 52.3% [66/126]***

Multi-condition branch: 42.0% [42/100]
Method: 90.9% [20/22]
Constructor: 100.0% [4/4]

RandomAccessFile.java

***Total Coverage: 5.3% [6/112]***
Multi-condition branch: 5.8% [4/68]
Method: 0.0% [0/42]
Constructor: 100.0% [2/2]

Calendar.java

***Total Coverage: 52.8% [65/123]***
Multi-condition branch: 30.0% [24/80]

Method: 95.1% [39/41]
Constructor: 100.0% [2/2]

Vector.java

***Total Coverage: 80.4% [107/133]***
Multi-condition branch: 69.7% [60/86]
Method: 100.0% [43/43]
Constructor: 100.0% [4/4]

***** All of the above is branch coverage *****

These results were achieved when developers simply told the technology which classes to test and clicked Start. Developers did not write a single test case, scaffolding, or stub to achieve this degree of coverage. It is important to note that these technologies were able to achieve, on average, 50% coverage in the fully-automatic testing mode. This percentage is striking, considering that only 30-40% of code is tested at the application level, and that the only effort needed to achieve this degree of coverage was a click of a button. This coverage can be increased when developers add their own test cases.

### How Can Automatic Unit Testing Technologies Be Integrated Into Your Development Cycle?

Automatic unit testing technologies can be used to find construction errors while developers are implementing the code. These technologies can be used in two modes: interactive mode and batch mode. As soon as a developer completes and compiles a unit, he can test the unit with the technology in interactive mode and fix all errors found before checking the code into the source code repository. The technology should also have a batch mode interface so that it can be integrated into nightly builds. Once the technology is integrated into nightly builds, all appropriate tests can automatically be re-run every night so that developers can immediately determine if modifications caused any undesired functionality changes or introduced any new construction problems.

## Conclusion

Unit testing allows developers to find and fix a large number of errors very early in the development-- when it is easiest, cheapest, and fastest to do so. By doing so, it affords developers a tremendous opportunity to improve their software quality as well as the efficiency of their development cycle. Unit testing is incredibly difficult to perform by hand, but technologies are currently available that automate unit testing. By using these technologies, developers can reduce their development time and cost at the same time that they improve the quality of the final product.

# References

Allison, C, “The Simplest Automated Unit Test Framework That Could Possibly Work” *C/C++ Users Journal* (September 2000): 48-61.

Humphrey, W. *A Discipline for Software Engineering* (Reading, MA: Addison-Wesley 1995).

Hunt, A and Thomas, D. *The Pragmatic Programmer* (Reading, MA: Addison-Wesley 2000).

McConnell, S. *Code Complete* (Redmond, WA: Microsoft Press 1993).

# Methods for Effective Unit Testing

Adam Kolawa

*Para*Soft

ParaSoft

# What is Unit Testing

- Testing the smallest possible unit of an application
  - OOP: Testing a class or function as soon as it is compiled
- Not the same as module testing (testing a module or a sub-program that is part of a larger application)
  - Module testing is important, but is not our focus

ParaSoft

# Coding Errors

- Construction errors account for 45-75% of all errors on even the largest projects
- Small projects: 75% of defects result from coding
- 2 large HP projects: the average construction defect cost 25 to 50% as much to fix as the average design error; total cost to fix construction defects was one to two times greater than the cost of fixing design defects

*--Steve McConnell, Code Complete (p. 610-611)*

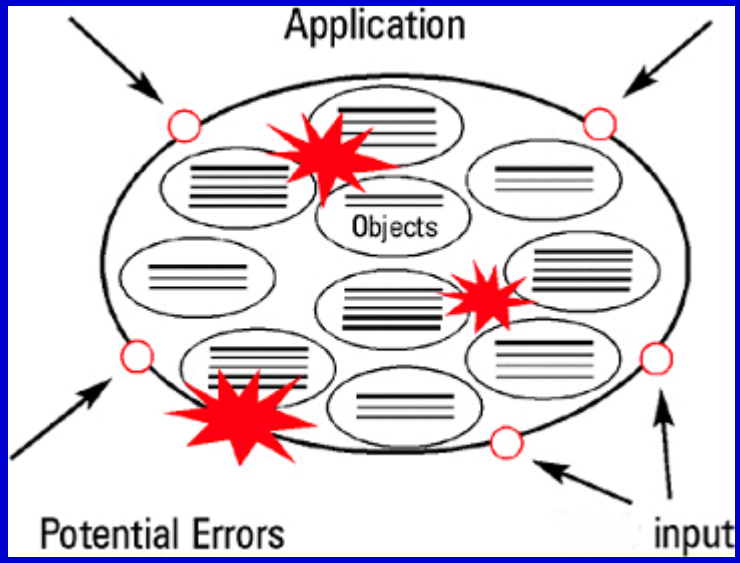
ParaSoft

# The Problem

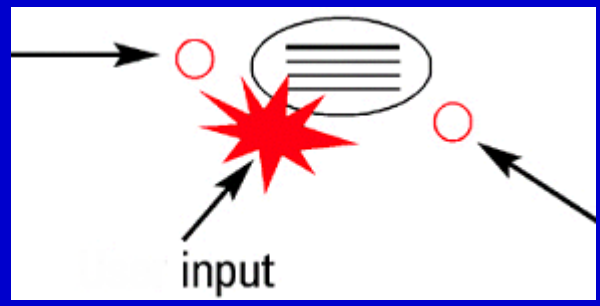
- Unit testing lets you find the largest proportion of all software errors– at the point where it is easiest, fastest, and cheapest to do so
- Unit testing is difficult– if not impossible– to perform without a way to automate the unit testing process

ParaSoft

# Benefits




**External Testing**




**Unit Testing**

When you test a class apart from all other objects, it becomes significantly easier to reach potential errors because you are much closer to the errors

It is also easier to achieve 100% coverage at this level

ParaSoft

# Benefits

- Don’t have to wade through layer after layer of errors to fix one problem
- Don’t have to change as much code to fix problems
- Easier error detection = less time and money spent finding errors

ParaSoft

# Benefits

- Find certain types of errors at the stage where it is easiest, fastest, and cheapest to do so
  - Relative cost to identify defects: during design 1.5; prior to coding 1; during coding 1.5; prior to test 10; during test 60; in field use 100
  - Relative time to identify defects: during requirements 1; during design 3 to 6; during coding 10; in development test 15 to 40; in acceptance test 30 to 70; during operation, 40 to 1000
  - Relative time to identify defects: during design review 1; during code inspections 20; during machine test 82

*-- Watts Humphreys, A Discipline for Software Engineering (p. 275)*

ParaSoft

# Errors Found

- Functionality errors: Class does not behave according to specification
  - Exposed with black-box testing
    - Compare input/outcome relationships
    - Benefit: Ensures methods function properly before one minor functionality problem spurs many difficult to find and fix problems

ParaSoft

# Errors Found

- Construction errors: Class crashes when it encounters unexpected inputs
  - Exposed with white-box testing
    - Test as many methods as possible with a variety of inputs
    - Benefit: Prevents crash-causing errors

ParaSoft

# Errors Found

- Regression errors: Class modifications introduced problems
  - Exposed with regression testing
    - Run all or selected black-box and white-box test cases
    - Benefit: Maintains class integrity

ParaSoft

# Performing Unit Testing

- Unit testing is difficult, time-consuming, and expensive without automatic unit testing technologies
- A brief look at how to perform simple unit testing without automatic unit testing tools reveals why it is so impractical

ParaSoft

# Making the Class Testable

- First step in testing: make class testable
  - Build scaffolding to run the class
  - Build stubs that return values for unavailable external resources

ParaSoft

# Scaffolding

- Runs the class under test
- Can only be used to test original class
- Should fully exercise class under test
- May require several modifications or rewrites

ParaSoft

# Simple BankAccount Class

```
#include "BankException.h"

class BankAccount
{
public:
    BankAccount() : _balance(0) {}

    int balance() { return _balance; }

    void deposit (int amount) {
        if (amount < 0) {
            throw BankException("invalid deposit
   amount");
        }
        _balance += amount;
        return;
    }
```

ParaSoft

```
void withdraw (int amount) {
        if (amount < 0) {
            throw BankException("invalid withdraw
   amount");
        }
if (_balance - amount < 0) {
            throw BankException("maximum withdraw
   amount is exceeded");
        }
        _balance -= amount;
        return;
    }
private:
    int _balance;
};
class BankException {
private:
    const char *_message;
public:
    BankException(const char *message) :
   _message(message) {}
    const char *message() { return _message; }
};
```

ParaSoft

# Scaffolding

```
#include "BankAccount.h"
#include <iostream>
using namespace std;
class BankAccountTest {
public:
    static void test() {
        BankAccount account;
        account.deposit(20);
        account.deposit(30);
        account.withdraw(7);
        if (account.balance() == 43) {
            cout << "Test passed" << endl;
        } else {
            cout << "Test failed" << endl;
        }
    }
};
int main()
{
    BankAccountTest::test();
    return 0;
}
```

ParaSoft

# Stubs

- Used when classes reference unavailable external resources (external files, databases, CORBA objects)
- Should return the types of values the external resource would return
- Need to redirect program's instruction pointer:
  - In whatever directions are necessary to test the class's functionality
  - In enough directions to provide thorough coverage

ParaSoft

# Simple Class

```
class db {
public:
    db (string s) {data_base my_file(s);
    }
    void read_data() {
        int i;
        i = my_file.read('k');
        cout << "Value for k = " << i << endl;

        i = my_file.read('o');
        cout << "Value for o = " << i << endl;

        i = my_file.read('a');
        cout << "Value for a = " << i << endl;
    }
public:
    int MAX;
    data_base my_file;
};
```

ParaSoft

# Scaffolding

```
void main() {
    db m = db("junk");
    m.read_data();
}
```

ParaSoft

# Stub

```
#include <iostream>
#include <string>
using namespace std;
class data_base {
public:
    data_base(){};
    data_base(string s){};
    int read(char val){
        switch (val) {
        case 'k':
            return 10;
        case 'o':
            return 14;
        case 'a':
            return 0;
        default:
            return -1;
        }
    }
};
```

ParaSoft

# Creating Test Cases

- You should create both black-box and white-box test cases to test functionality and construction
- Test special cases
  - BankAccount example:
    - Current balance: $43
    - Withdraw $43, then another $3 to see how the class handles a negative balance
    - The class should not allow withdrawals when the balance is less than or equal to 0

ParaSoft

# Simple Test Case

```
#include "BankAccount.h"

#include <iostream>
using namespace std;

class BankAccountTest {
public:
    static void test() {
        BankAccount account;

        account.deposit(20);
        account.deposit(30);
        account.withdraw(7);
        account.withdraw(43);

        if (account.balance() == 0) {
            cout << "Your balance is 0" << endl;
        } else {
            cout << "Error!" << endl;
        }
```

ParaSoft

```
try {
            cout << "withdrawing $3" << endl;
            account.withdraw(3);
            cout << "return from withdraw" << endl;
        }
        catch(BankException) {
            cout << "catching BankException" << endl;
        }
        cout << "done " << endl;
    }
};

int main()
{
    BankAccountTest::test();
    return 0;
}
```

ParaSoft

# Automating Unit Testing

- Unit testing needs to be automated in order for it to be integrated into the development process
- Several approaches to automation:
  - Semi-automation
  - Complete automation

ParaSoft

# Automating Unit Testing

- Semi-automation
  - Automate running test cases, error-reporting
  - Simple 2 class, GUI-less framework (Allison)
  - GUI-based tools like JUnit and CppUnit
  - Drawback: Developers still need to create scaffolding and stubs, call functions, and design all test cases
- Total automation
  - Automate everything from building scaffolding and stubs to creating and running test cases
  - Our company wanted this type of solution; we couldn't find one, so we built one
  - Currently have automatic unit testing technologies for C/C++ and Java

ParaSoft

# Complete Automation

- Automate everything from building scaffolding and stubs to creating and running test cases
- Our company wanted this type of solution. We couldn't find one, so we built one.
- Currently have automatic unit testing technologies for C/C++ and Java

ParaSoft

# Critical Features

- Automatically create a test harness
- Automatically generate test cases
- Allow user defined test cases
- Execute tests and report errors
- Automate regression testing

ParaSoft

# Integrating Unit Testing Into Your Development Process

- Developers should use tool immediately after compiling a class
- Use in 2 modes:
  - Interactive (GUI-based): Use this mode after class is compiled; fix all errors found before checking code into repository
  - Batch: Integrate batch-mode regression testing into nightly builds to immediately determine if modifications caused problems

ParaSoft

# Conclusion

- Technologies that automate unit testing are now available
- Developers serious about quality should review their development process and determine how to integrate automatic unit testing into that process

ParaSoft

# Empathetic Listening: The Overlooked Testing Tool

**Julie Fleischer**
**Home Products Group**
**Intel Corporation**
MS JF2-70
2111 NE 25th Ave.
Hillsboro, OR 97124
Julie.N.Fleischer@intel.com

**Abstract**
In the fast-paced world of technology, individuals can find themselves thrown into the testing and QA role before they have learned all the skills necessary to succeed. While there are many resources available for improving one's technical skills, the necessary training for communication skills does not always exist. As a result, testers can end up frustrated and angry, feeling misunderstood and not listened to. Listening - a major building block for effective communication - is often overlooked as a skill a tester should develop. **Empathetic Listening: The Overlooked Testing Tool** defines empathetic listening and details why a tester will find it beneficial to develop this skill as part of their tester's tool box.

Julie Fleischer is a Software Quality Engineer at Intel Corporation. She received her M.S. and B.S. in Computer Science from Case Western Reserve University.

**Introduction**

Let's face it; a tester's job is tough. Technically, a tester needs to have problem-solving skills, attention to detail, tolerance of disorder, at least some programming experience, and creativity. On the interpersonal skills side, a software tester needs to have excellent verbal and written communication skills and an ability to deal with others (Beizer 318). The good news is that, on the technical side, the resources available to the software tester are abundant, and there are many universities that teach and help cultivate the skills to make one a technically strong software tester. However, the important communication skills sometimes seem to be downgraded in a technical field such as software testing (Steil et al 71). A tester not skilled in this area could find frustration and disappointment with the high-pressure and demanding environment in which testers work. In order to communicate well with others, a tester must be well trained in the art of listening - truly empathetically listening - to others, as listening is the basis for effective communication. This is not a skill that can be willed into existence or developed naturally over time; it is a skill that requires training (Steil et al 41). This paper attempts to present the basics of empathetic listening, what it is and what it isn't, as well as illustrate some of the benefits that testers can see when they practice empathetic listening as part of their on-the-job skills.

**Learning to Listen**

> Imagine that Sally, a tester, and Darren, a developer, are discussing one of the bugs that Sally has written up. Darren claims that this bug is not actually a bug; the program is supposed to have this behavior.
>
> "But it can't be a 'feature,'" Sally moans. "This 'feature' will drive the customers crazy!"
>
> "Well, we have to assume that the customer has a little common sense," retorts Darren.
>
> "Common sense? Our customers have never used a program like this before! We can't assume they will see this as a 'feature!'"
>
> Then, Darren's boss walks in and asks what the two of them are arguing about. Darren explains that Sally wants him to fix a bug in the software that is not really a bug. He quickly shows his boss the issue, and, before Sally can get a word in edgewise, Darren's boss gives her decision.
>
> "This is as designed. Sally, you need to remove that bug from the bug list."
>
> "But, he didn't really show it to you!" cries Sally. "The customers won't see it like that!"
>
> Darren's boss tightens her lips and takes a deep breath. "The bug needs to be removed," she says loudly. "End of story."

As a tester, can you imagine how Sally would feel? Misunderstood? Not listened to? Ignored? Imagine how she would have felt if Darren and his boss had first tried to understand the issue before stating that it was a feature. If you can imagine the mutual satisfaction and respect that would have come if there had been more listening and attempts to understand first, then you can understand the motivation behind learning to listen.

**Empathetic Listening Explained**

It is easy to see that the world would be a better place for us if others would listen to us. However, in the position we are placed as testers, where we must frequently bring unwanted news to other people, we sometimes overlook the fact that there are things we can do to make people more receptive to what we have to say. It lies, however, not in staunchly supporting our opinions until we wear everyone down; instead, it lies in reaching out to others, to understand

their frame of references before we make our move. *A tester's best defense is truly a good offense.*

> Imagine that when Sally left Darren, her coworker Scott, who was a close friend of Darren's explained the situation. "You know when they had the schedule push out a few weeks ago? Apparently, they told all the developers that they weren't working hard enough. Darren and his boss have been working here every weekend and until midnight for the past two weeks to try to bring the project back on schedule. They're trying to fix all the bugs they can, but we keep finding more. Instead of pushing out the schedule again, they just have to work more hours to get everything fixed."
>
> "Wow," Sally muses. "I didn't know that. I wonder why he didn't tell me." Then she remembers that Darren had mentioned something about this bug "not being as important as the other ones he **had** to fix," and she remembers that he had seemed pretty tired lately when she went to talk with him. "Imagine what could have happened if I had known this initially!" she thinks.

If Sally had spent more time on her offense, or really trying to understand Darren's frame of reference, her defense would not have needed to be so strong when Darren told her he did not want to fix the bug she had just found. This is the basis behind empathetic listening. Empathetic listening is listening to really understand someone and have a sense of his or her frame of reference (Covey 240). In order to understand truly someone else's frame of reference, we need to listen with a goal to understand, not to judge. We cannot presuppose that we already know a person's thoughts or feelings ("Darren is a developer. He doesn't understand the user, which is why he will always reject my bugs."). Empathetic listening is really an openness to discover what others are thinking and feeling ("I am unsure why Darren would feel this way, but I respect him enough to find out why.") (Nichols 43).

*Empathetically listening is temporarily suspending our frame of reference to attempt to understand the other person's.* When we are truly listening to another, we are temporarily suspending our own needs and are existing just for the sake of the other person (Nichols 64).

> Imagine Sally had instead approached Darren with the attitude to first empathetically listen and suspend her frame of reference in mind.
> "I think this is an important bug," Sally thinks to herself. "But I will try to listen to understand how Darren feels about it."
> "Darren," Sally begins. "Can we talk about a bug I just wrote?" Sally explains the bug to Darren.
> "That's not a bug!" Darren exclaims. "That's how we coded it. It would just be stupid to call it a bug."
> "I don't think I understand yet." Sally asks patiently. "Can you explain to me what you mean?"
> "It means," Darren is a little calmer now. "That we can't fix it right now. Actually," Darren pauses. "I guess it is a little awkward, but the problem is I have to fix that list of fifty bugs that you wrote by tomorrow. I'm just thinking this one isn't as important as those others."

"Oh! Fifty bugs?" Sally exclaims. "You're right about this one! It's not a system crash or anything. It's just something that might not be as intuitive to the user." Sally thinks for a moment. "Maybe I can look over that list of fifty bugs. I know some in there aren't so important that they need to get fixed by tomorrow either."

"Sounds great! Thanks, Sally."

Suspending her own needs at first, even when Darren snapped at her allowed Sally to truly listen to Darren. In that process, she gained the valuable information that Darren was really looking to fix the highest priority bugs, and she could help him achieve this goal.

*Empathetic listening is suspending judgement.* When we are truly listening to understand someone else, we are not listening to judge him or her (Nichols 64). This is often quite difficult to do when we have been accustomed to doing the opposite for so long; however, when we judge we lose our chance at really hearing what the true nature of the problem is. Judging is imposing our own viewpoints on the situation, and doing this when we are supposed to be listening defeats the purpose of understanding the other person's viewpoint.

*Empathetic listening means putting aside our roles during communication.* If we perceive our role as "software tester" or "quality assurer," this influences how we listen to someone else (Sund, Carin 131). True empathetic listening is putting aside these roles while we are listening to ensure we have all of the data. As a tester, it is easy to simply hear the developer say that they will not fix our bug and overlook what they are saying about the difficulty of fixing our bug or the priority. After all, those are developer details and don't appear to be as important to a tester. However, if we put aside our role as software tester and look at the situation for what it is, it is much easier to listen to the other side. Then, afterwards, we can go back to our role of software tester and decide our correct response, armed with more data on the big picture.

*Empathy is not the same as sympathy.* There is a tendency to equate empathy with the feeling of agreement with someone else or a feeling of pity for their situation; however, this is not what empathy is (Covey 240). This is along the lines of the fact that empathy is not judgmental; sympathy is just a positive judgement on, or an agreement with, the other person's views. An empathetic listener is listening to understand the other person's frame of reference, not necessarily to agree with it.

For example, the conversation with Sally and Darren could have gone like this.

After Sally asks Darren to explain why he does not think the bug is truly a bug, Darren replies, "It's just too hard to fix. I mean, I see that it's awkward, but the customer has to learn that they can't double click on that box, or the system will crash. I think that after the system crashes once on them, they'll probably learn that they have to single click for things to work."

"Like a cat who jumped onto a hot stove," Sally jokes. "They won't try that again…."

"Exactly," Darren says. "One crash and they should learn. I mean, I'd like to fix it, but I have this list of fifty bugs that I have to fix by tomorrow, and I don't think I can fit this one in."

"Could I look over that list?" Sally thinks aloud. "I'm wondering if some of those bugs that I put on that list aren't as important as this one. I mean, I'd agree with you that our customers **should** have common sense, but I'm thinking that one crash may make them

think badly of the software. Maybe we could replace fixing this one with two or three others on that list."

"Yeah, that would help," says Darren. "If I didn't have to do all fifty of those, I think I could find time to fix this one. Yeah. I think it's just a matter of adding another switch statement…. Yeah…."

If Sally had thought that she had to agree with Darren in order to empathetically listen, she may have felt frustrated that she was agreeing to something she did not want to be agreeing with and could have abandoned the thought of empathetic listening at all. It is important to remember that judgements are best saved for after listening, when the listener has all of the data to consider.

*A tester can use his or her imagination to try to help empathetic listening.* By their nature, testers have strong imaginations that they use to envision scenarios that users could perform to break the product they are testing. Using one's imagination is a widely recommended technique for preparing one's response (Covey 103) or developing the motivation to empathetically listen to another (Dalai Lama, Cutler 88). When someone is stuck or is having a hard time wanting to put themselves in another's shoes, it helps to digress by playing children's games and imagining you are the other person. This technique can very quickly put you in their frame of reference and help you understand potential problems they may be having. Imagination also helps you prepare to talk with someone else. Just as we should have the software design before the software development, there should also be a mental creation of a situation before there is a physical creation (Covey 99). We can imagine ourselves empathetically listening to developers or managers, and truly losing ourselves and suspending judgement before it even happens. And just as a software design guards against sloppily written, error-prone code, imagining a situation before it happens guards against making mistakes or errors when the time comes to communicate with others.

**So, Why Should We Do This?: The Benefits of Empathetic Listening**

The biggest benefit to us is that *empathetic listening gives us accurate data with which to work.* Instead of projecting our own thoughts and feelings onto others, we are analyzing the situation more closely and seeing it for what it truly is (Covey 241). This will allow for better decision making because we have more accurate data to work with (Steil et al 44). As a tester, it is vital that we make informed decisions on the quality of the software, and the only way we can get this information is by truly trying to understand another's opinion before we make our final decision.

When we listen empathetically, we also convey an interest in and a respect and value for the other person (Sund, Carin 127). These feelings *reduce fear on both sides and allow for more openness to happen*; they create a positive atmosphere and allow for the other person to respond with respect and value in return (Dalai Lama, Cutler 69). The other person no longer has to worry about not being respected or understood and can instead concentrate on understanding your point of view or trying to find a mutually beneficial solution (Covey 241). The other person will be more likely to cooperate when he feels you have a genuine interest in him and his problems, feelings, and thoughts (Steil et al 44).

*Empathetic listening reduces tension.* Sometimes the act of listening to someone vent can allow them to relieve tension they have built up and make the air clean of tension and hostility (Steil

et al 43). After this, the person may be much more willing to listen to your point of view, since they have purged some of the negative emotions out of their immediate thoughts.

*Empathetic listening can prevent trouble.* Frequently, when we get into a dispute with someone else, we say things that we later wish we hadn't said. Listening first, then speaking reduces the risk that we will fly off the handle and say something we later want to retract (Steil et al 44).

*Empathetic listening can give the listener confidence.* When we are truly listening to the other person and know that we have more accurate data, we can be more confident that what we finally do say is appropriate and valid (Steil et al 45).

**Tying it All Together: Empathetic Listening is an Effective Tester Tool**

By watching Sally and Darren, we have seen that empathetic listening, when used correctly, can be an effective tool for a tester to have in her toolbox. We have also seen that this tool is something a tester develops through practice and training; it does not come naturally with experience. Empathetic listening occurs when a tester makes the conscious decision to suspend his or her needs, thoughts, and judgements for a moment and focus on the other person's point of view. If a tester can do this, he can open the door for further communication and openness with the people he works with.

Bibliography


Beizer, B. (1984*). Software system testing and quality assurance*. New York, NY: Van Nostrand Reinhold Company, Inc.

Covey, S. R. (1989). *The seven habits of highly effective people: Powerful lessons in personal change.* New York, NY: Fireside.

Dalai Lama, Cutler, H. C. (1998). *The art of happiness: A handbook for living*. New York, NY: Riverhead Books.

Nichols, M. (1995). *The lost art of listening*. New York, NY: The Guildford Press.

Steil, L. et al. (1983). *Listening – It can change your life*. NY: John Wiley & Sons, Inc.

Sund, R. B., Carin, A. (1978). *Creative questioning and sensitive listening techniques: A self-concept approach, 2nd edition.* Columbus, OH: Charles E. Merrill Publishing Company.

# Measurement of the Extent of Testing

**CEM KANER, J.D., Ph.D.**
**Professor**
**Department of Computer Sciences**
**Florida Institute of Technology**
**150 West University Blvd., Melbourne, FL 32901**
**kaner@kaner.com www.kaner.com**

*Invited Address*

**Pacific Northwest Software Quality Conference**

**Portland, Oregon**

**October 17, 2000**

# Biography

Cem Kaner is Professor of Software Engineering at the Florida Institute of Technology. He is senior author of Testing Computer Software and of Bad Software: What To Do When Software Fails.

Kaner has worked with computers since 1976, doing and managing programming, user interface design, testing, and user documentation, teaching courses on software testing, and consulting to software publishers on software testing, documentation, and development management issues. He is also co-founder and co-host of the Los Altos Workshop on Software Testing and the Software Test Managers' Round Table.

Kaner also practices law, focusing on the law of software quality. He has been active (as an advocate for customers, authors, and small development shops) in several legislative drafting efforts involving software licensing, software quality regulation, and electronic commerce.

## Acknowledgment

Much of the material in this paper was presented or developed by the participants of the *Software Test Managers Roundtable* (STMR) and the *Los Altos Workshop on Software Testing* (LAWST).

- STMR 1 (October 3, November 1, 1999) focused on the question, *How to deal with too many projects and not enough staff?* Participants included Jim Bampos, Sue Bartlett, Jennifer Brock, David Gelperin, Payson Hall, George Hamblen, Mark Harding, Elisabeth Hendrickson, Kathy Iberle, Herb Isenberg, Jim Kandler, Cem Kaner, Brian Lawrence, Fran McKain, Steve Tolman and Jim Williams.

- STMR 2 (April 30, May 1, 2000) focused on the topic, *Measuring the extent of testing.* Participants included James Bach, Jim Bampos, Bernie Berger, Jennifer Brock, Dorothy Graham, George Hamblen, Kathy Iberle, Jim Kandler, Cem Kaner, Brian Lawrence, Fran McKain, and Steve Tolman.

- LAWST 8 (December 4-5, 1999) focused on *Measurement*. Participants included Chris Agruss, James Bach, Jaya Carl, Rochelle Grober, Payson Hall, Elisabeth Hendrickson, Doug Hoffman, III, Bob Johnson, Mark Johnson, Cem Kaner, Brian Lawrence, Brian Marick, Hung Nguyen, Bret Pettichord, Melora Svoboda, and Scott Vernon.

Facilities and other support for STMR were provided by Software Quality Engineering, which hosts these meetings in conjunction with the STAR conferences. Facilities for LAWST were provided by the University of California (Extension) Santa Cruz.


# Background

At PNSQC a year ago, I spoke with you about problems inherent in software measurement, and described a framework for developing and evaluating measures (Kaner, 1999). Similar approaches have been laid out by other authors on software measurement (Kitchenham, Pfleeger, & Fenton, 1995; Zuse, 1997) and in other fields, such as physics (*see, e.g.* Sydenham, Hancock & Thorn, 1989) and psychometrics (*see, e.g.* Allen & Yen, 1979).

My point of departure from the traditional software measurement literature is in the extent to which I ask (a) whether commonly used measures are valid, (b) how we can tell, and (c) what kinds of side effects are we likely to encounter if we use a measure? Even though many authors mention these issues, I don't think they've been explored in enough depth and I don't think that many of us are thoughtfully enough facing the risks associated with using a poor software measure (Austin, 1996; Hoffman, 2000).

Developing a valid, useful measure is not easy. It takes time, combined with theoretical and empirical work. In Kaner (1999), I provided detailed examples from the history of psychophysics to illustrate the development of a field's measures over a 100-year period.

The most common question that has come to me in response to that paper (Kaner, 1999) is what measures I think are valid and useful. I don't have a simple answer to that, but I can report on some work in progress regarding measurement of the extent of testing of a product. This paper is a progress report,

based primarily on the work of colleagues who manage software test groups or consult to test managers. It describes some of the data they collect and how they report it to others.

Some of the material in this paper will be immediately useful to some readers, but the point of the paper is not to present finished work. I'm still at the stage of collecting and sifting ideas, looking for common threads and themes, trying to get a better idea of the question that we intend when we ask how to measure how much testing has been done, and to understand the possible dimensions of an answer to such a question.

There's an enormous amount of detail in this paper and you might get lost. Here's the one thing that I would most like you to take away from the paper:

***Bug count metrics reflect only a small part of the work and progress of the testing group. Many alternatives look more closely at what has to be done and what has been done. These will often be more useful and less prone to side effects than bug count metrics.***

# The Measurement Framework

To evaluate any proposed measure (or metric), I propose that we ask the following ten questions. For additional details, see Kaner (1999):

1. ***What is the purpose of this measure?*** Some measures are used on a confidential basis between friends. The goal is to spot trends, and perhaps to follow those up with additional investigation, coaching, or exploration of new techniques. A measure used only for this purpose can be useful and safe even if it is relatively weak and indirect. Higher standards must apply as the measurements become more public or as consequences (rewards or punishments) become attached to them.

2. ***What is the scope of this measure?*** Circumstances differ across groups, projects, managers, companies, countries. The wider the range of situations and people you want to cover with the method, the wider the range of issues that can invalidate or be impacted by the measure.

3. ***What attribute are we trying to measure?*** If you only have a fuzzy idea of what you are trying to measure, your measure will probably bear only a fuzzy relationship to whatever you had in mind.

4. ***What is the natural scale of the attribute?*** We might measure a table's length in inches, but what units should we use for extent of testing?

5. ***What is the natural variability of the attribute?*** If you measure two supposedly identical tables, their lengths are probably slightly different. Similarly, your weight varies a little bit from day to day. What are the inherent sources of variation of "extent of testing"?

6. ***What instrument are we using to measure the attribute and what reading do we take from the instrument?*** You might measure length with a ruler and come up with a reading (a measurement) of 6 inches.

7. ***What is the natural scale of the instrument?*** Fenton & Pfleeger (1997) discuss this is detail.

8. ***What is the natural variability of the readings?*** This is normally studied in terms of "measurement error."

9. ***What is the relationship of the attribute to the instrument?*** What mechanism causes an increase in the reading as a function of an increase in the attribute? If we increase the attribute by 20%, what will show up in the next measurement? Will we see a 20% increase? Any increase?

10. ***What are the natural and foreseeable side effects of using this instrument?*** If we change our circumstances or behavior in order to improve the measured result, what impact are we going to have on the attribute? Will a 20% increase in our measurement imply a 20% improvement in the underlying attribute? Can we change our behavior in a way that optimizes the measured result but without improving the underlying attribute at all? What else will we affect when we do what we do to raise the measured result? Austin (1996) explores the dangers of measurement—the unintended side effects that result—in detail, across industries. Hoffman (2000) describes several specific side effects that he has seen during his consultations to software companies.

***The phrase "directness of measurement" is often bandied about in the literature. If "directness of measurement" means anything, it must imply something about the monotonicity (preferably linearity?) of the relationship between attribute and instrument as considered in both #9 and #10.***

***The phrases "hard measure" and "soft measure" are also bandied about a lot. Let me suggest that a measure is "hard" to the extent that we can describe and experimentally investigate a mechanism that underlies the relationship between an attribute and its measure and that accurately predicts the rate of change in one variable as a function of the other.***

# Exploring the Question

When someone asks us for a report on the amount of testing that we've completed, what do they mean? The question is ambiguous.

At an early point in a class he teaches, James Bach has students (experienced testers) do some testing of a simple program. At the end of the exercise, James asks them (on a scale from 0 to 100, where 100 means completely tested) how much testing they've done, or how much they would get done if they tested for another 8 hours. Different students give strikingly different answers based on essentially the same data. They also justify their answers quite differently. We found the same broad variation at STMR (a collection of experienced test managers and test management consultants).

If someone asks you how much of the testing you (or your group) has done, and you say 30%, you might be basing your answer on any combination of the following dimensions (or on some others that I've missed):

- ***Product coverage:*** In principle, this compares the testing done with the total amount of testing of the product that could be done. But that can't be right because the amount of testing possible is infinite (Kaner, 1997) and 30% of infinity is unachievable. Instead, this percentage is offered with reference to a model (which might be implicit or explicit) of what would be good enough testing. Of the population of tests needed for sufficient testing, 30% have been done. ***This measure is inherently dynamic.*** At the start of testing, you might decide that certain tests need not be done. During testing, you'll probably discover that some of your assumptions were wrong. If someone finds bugs that you didn't expect or you learn something new about market expectations, you might decide to add new tests. If you do, then what you used to call 30% coverage might now be better estimated as 20%.

- ***Agreement-based:*** Compare to what has been promised (or planned and committed to). Given a list of tasks that you've agreed to do (for example, see Kaner, 1996), what percentage have you finished? However, if you add tasks or deepen your work in some areas, your percentage "complete" will need revision.

- ***Project-history based:*** Compare this project with some others that are similar or that can bracket your expectations for this one.

- ***Risk-based:*** One form of risk-based estimation translates into agreement-based estimation. You do a risk analysis of the product, derive a set of tests (or of areas to be tested, with time boxes for the work), and then report what you've done against what you intended or agreed. (For example, see Amland, 1999.) A different estimator is less formal. When someone asks you how much you have gotten done, rather than saying 30%, you tell the person about the most critical areas of risk remaining in the product. Some areas are untested, others are undertested, others are awaiting difficult bug fixes. Assuming that nothing else bad comes up, you say, it will take about X weeks to deal with these, and then the product will probably be ready to release.

- ***Evaluation (of testing).*** This is another form of risk evaluation. How has the testing been? Are you finding surprises that indicate that your testing is weaker than expected? Should you reconsider your plans for the testing effort?

- ***Results:*** How much testing have you gotten done? For example, you might reports bug statistics or statistics on tests completed.

- ***Results and obstacles:*** When someone asks you how much testing you've actually gotten done, what is their purpose for collecting that information? Perhaps it is predictive (she wants an estimate of when the product can be shipped) or perhaps it is diagnostic or disciplinary. If the hidden question is not just "How far are you?" but is also "Why have you only gotten this far?" then your report of 30% might mean "30% of what was humanly possible to do during this period" and it might be accompanied by a list of obstacles that wasted your time or blocked some testing tasks.

- ***Effort:*** You report that you've tested for 400 hours or for 4 calendar weeks. Again, depending on the purpose of the question, this might be the best thing to report. Perhaps the underlying question is whether the you (or the testing group) are doing much work (or are able to focus your time on testing tasks). Or perhaps you work under a fixed budget or schedule, and you are reporting that you have consumed 30% of your resources. (If you are testing 10 projects at the same time, such measures can be particularly useful.)

## Another Look at the Question

When we ask, "*What is a good measure of the extent of testing?"* what are we looking for? There are several related questions to consider:

- ***The measurement question:*** What are we trying to measure (what is the attribute) and how can we measure it?

- ***The communication question:*** How should we tell management how much testing is done and how much is left?

- ***The control question:*** Can we use measures of the extent of testing to help us control the testing project? Which measure(s) will be effective?

- ***The side effect question.*** What risks are inherent in measuring the extent of testing? In what ways might the process of measuring (and of managing on the basis of the measures) have results that we didn't intend or anticipate?

- ***The organizational effects question.*** In what ways might the use of this type of measure affect the organization?

- ***The requirements question:*** How does this type of measure serve our testing requirements? How do our requirements push us to choose measures?

These questions overlap with the measurement framework analysis but in discussions, they seem to stimulate different answers and different ideas. Therefore, for now, I'm keeping them as an independently useful list.

# The Rest of this Paper

The material that follows lists and organizes some of the ideas and examples that we (see the Acknowledgement, above) collected or developed over the last year. I have filtered out many suggestions but the lists that remain are still very broad. My intent is to show a range of thinking, to provide you with a collection of ideas from many sources, not (yet) to recommend that you use a particular measure or combination of them. The ordering and grouping of ideas here are for convenience. The material could be reorganized in several other ways, and probably in some better ways. Suggestions are welcome.

Let me stress that I do not endorse or recommend the listed measures. I think that some of them are likely to cause more harm than good. My objective is to list ideas that reasonable people in the field have found useful, even if reasonable people disagree over their value. If you are considering using one, or using a combination of them, I suggest that you evaluate the proposed measure using the measurement framework described above.

The sections below do not make a good enough connection with the software measurement literature. Several sources (for example, Zuse, 1997, and Ross Collard's software testing course notes) provide additional measures or details. As I suggested at the start, this paper reports work in progress. The next report will provide details that this one lacks.

# Coverage-Based Measures

"Coverage" is sometimes interpreted in terms of a specific measure, usually statement coverage or branch coverage. You achieve 100% statement coverage if you execute every statement (such as, every line) in the program. You achieve 100% branch coverage if you execute every statement and take every branch from one statement to another. (So, if there were four ways to reach a given statement, you would try all four.) I'll call this type of coverage (coverage based on statements, branches, perhaps also logical conditions), *code coverage.*

Code coverage is a tidy measure—it is easy to count, unambiguous, and easy to explain. Unfortunately, this measure carries risks (Marick, 1999). It is easy (and not uncommon) to write a set of relatively weak

tests that hit all of the statements and conditions but don't necessarily hit them very hard. Additionally, code coverage is incomplete. For examples of incompleteness:

- Testing the lines of code that are there does not necessarily reveal the problems arising from the code that is not there. Marick (2000) summarizes data from cases in which 22% to 54% of the errors found were faults of omission.

- You can achieve 100% code coverage while missing errors that would have been found by a simple data flow analysis. (Richard Bender provides a clear and simple example of this in his excellent course on Requirements Based Testing.)

- Code coverage doesn't address interrupts (there is an implicit branch from every statement in the program to the interrupt handler and back, but because it is implicit—wired into the processor rather than written into the program directly—it just doesn't show up in a test of every visible line of code) or other multi-tasking issues.

- Table-driven programming is puzzling for code coverage because much of the work done is in the table entries, which are neither lines of code nor branches.

- User interface errors, device incompatibilities, and other interactions with the environment are likely to be under-considered in a test suite based on code coverage.

- Errors that take time to make themselves visible, such as wild pointers or stack corruption or memory leaks, might not yield visible failures until the same sub-path has been executed many times. Repeatedly hitting the same lines of code and the same branches doesn't add any extent-of-testing credit to the code coverage measure.

Sometimes, the most important coverage measure has nothing to do with code coverage. For example, I worked on a product that had to print well. This was an essential benefit of the product. We selected 80 printers for detailed compatibility testing. We tracked the percentage of those 80 printers that the program could pass. This was a coverage measure, but it had nothing to do with lines of code. We might pass through exactly the same code (in our program) when testing two printers but fail with only one of them (because of problems in their driver or firmware).

You have a coverage measure if you can imagine any kind of testing that can be done, and a way to calculate what percent of that kind of testing you've done. Similarly, you have a coverage measure if you can calculate what percentage of testing you've done of some aspect of the program or its environment. As with the 80 printers, you might artificially restrict the population of tests (we decided that testing 80 printers was good enough under the circumstances) and compute the percentage of that population that you have run.

Here are some examples of coverage measures. Some of this list comes from Kaner (1995), which provides some additional context and discussion.

- ***Line coverage:*** Test every line of code (Or Statement coverage: test every statement).

- ***Coverage of non-dead code***, code that can be reached by a user.

- ***Branch coverage:*** Test every line, and every branch on multi-branch lines.

- ***N-length sub-path coverage:*** Test every sub-path through the program of length N. For example, in a 10,000 line program, test every possible 10-line sequence of execution.

- ***Path coverage:*** Test every path through the program, from entry to exit. The number of paths is impossibly large to test. (*See* Myers, 1979, and Chapter 2 of Kaner, Falk, and Nguyen, 1993).

- ***Multicondition or predicate coverage:*** Force every logical operand to take every possible value. Two different conditions within the same test may result in the same branch, and so branch coverage would only require the testing of one of them. (See Myers, 1979, for multiple condition coverage, and Beizer, 1990).

- ***Trigger every assertion check in the program:*** Use impossible data if necessary.

- ***Loop coverage:*** "Detect bugs that exhibit themselves only when a loop is executed more than once." (Marick, 1995, p. 146)

- ***Every module, object, component, tool, subsystem, etc.*** This seems obvious until you realize that many programs rely on off-the-shelf components. The programming staff doesn't have the source code to these components, so measuring line coverage is impossible. At a minimum (which is what is measured here), you need a list of all these components and test cases that exercise each one at least once.

- ***Fuzzy decision coverage.*** If the program makes heuristically-based or similarity-based decisions, and uses comparison rules or data sets that evolve over time, check every rule several times over the course of training.

- ***Relational coverage.*** "Checks whether the subsystem has been exercised in a way that tends to detect off-by-one errors" such as errors caused by using < instead of <=. (Marick, 1995, p. 147) This coverage includes:

    - *Every boundary on every input variable.* (Boundaries are classically described in numeric terms, but any change-point in a program can be a boundary. If the program works one way on one side of the change-point and differently on the other side, what does it matter whether the change-point is a number, a state variable, an amount of disk space or available memory, or a change in a document from one typeface to another, etc.? Kaner, Falk, & Nguyen, 1993, p. 399-401.)

    - *Every boundary on every output variable.*

    - *Every boundary on every variable used in intermediate calculations.*

- ***Data coverage.*** At least one test case for each data item / variable / field in the program.

- ***Constraints among variables:*** Let X and Y be two variables in the program. X and Y constrain each other if the value of one restricts the values the other can take. For example, if X is a transaction date and Y is the transaction's confirmation date, Y can't occur before X.

- ***Each appearance of a variable.*** Suppose that you can enter a value for X on three different data entry screens, the value of X is displayed on another two screens, and it is printed in five reports. Change X at each data entry screen and check the effect everywhere else X appears.

- ***Every type of data sent to every object.*** A key characteristic of object-oriented programming is that each object can handle any type of data (integer, real, string, etc.) that you pass to it. So, pass every conceivable type of data to every object.

- ***Handling of every potential data conflict.*** For example, in an appointment calendaring program, what happens if the user tries to schedule two appointments at the same date and time?

- ***Handling of every error state.*** Put the program into the error state, check for effects on the stack, available memory, handling of keyboard input. Failure to handle user errors well is an important problem, partially because about 90% of industrial accidents are blamed on human error or risk-taking. (Dhillon, 1986, p. 153) Under the legal doctrine of *forseeable misuse* (this doctrine is cleanly explained in Brown, 1991), the manufacturer is liable in negligence if it fails to protect the customer from the consequences of a reasonably forseeable misuse of the product.

- ***Every complexity / maintainability metric against every module, object, subsystem, etc.*** There are many such measures. Jones (1991, p. 238-341) lists 20 of them. Beizer (1990) provides a sympathetic introduction to these measures. Glass (1992) and Grady & Caswell, (1987) provide valuable perspective. People sometimes ask whether any of these statistics are grounded in a theory of measurement or have practical value. (*For example*, Kaner, Falk, & Nguyen, 1993, p. 47-48; *also* Glass, 1992, p. 303, "Software metrics to date have not produced any software quality results which are useful in practice.") However, it is clear that, in practice, some organizations find them an effective tool for highlighting code that needs further investigation and might need redesign. (*For example, see* Grady & Caswell, 1987, and Grady, 1992, p. 87-90.)

- ***Conformity of every module, subsystem, etc. against every corporate coding standard.*** Several companies believe that it is useful to measure characteristics of the code, such as total lines per module, ratio of lines of comments to lines of code, frequency of occurrence of certain types of statements, etc. A module that doesn't fall within the "normal" range might be summarily rejected (bad idea) or re-examined to see if there's a better way to design this part of the program.

- ***Table-driven code.*** The table is a list of addresses or pointers or names of modules. In a traditional CASE statement, the program branches to one of several places depending on the value of an expression. In the table-driven equivalent, the program would branch to the place specified in, say, location 23 of the table. The table is probably in a separate data file that can vary from day to day or from installation to installation. By modifying the table, you can radically change the control flow of the program without recompiling or relinking the code. Some programs drive a great deal of their control flow this way, using several tables. Coverage measures? Some examples:

    - *check that every expression selects the correct table element*

    - *check that the program correctly jumps or calls through every table element*

    - *check that every address or pointer that is available to be loaded into these tables is valid (no jumps to impossible places in memory, or to a routine whose starting address has changed)*

    - *check the validity of every table that is loaded at any customer site.*

- ***Every interrupt.*** An interrupt is a special signal that causes the computer to stop the program in progress and branch to an interrupt handling routine. Later, the program restarts from where it

was interrupted. Interrupts might be triggered by hardware events (I/O or signals from the clock that a specified interval has elapsed) or software (such as error traps). Generate every type of interrupt in every way possible to trigger that interrupt.

- ***Every interrupt at every task, module, object, or even every line.*** The interrupt handling routine might change state variables, load data, use or shut down a peripheral device, or affect memory in ways that could be visible to the rest of the program. The interrupt can happen at any time—between any two lines, or when any module is being executed. The program may fail if the interrupt is handled at a specific time. (Example: what if the program branches to handle an interrupt while it's in the middle of writing to the disk drive?) The number of test cases here is huge, but that doesn't mean you don't have to think about this type of testing. This is path testing through the eyes of the processor (which asks, "What instruction do I execute next?" and doesn't care whether the instruction comes from the mainline code or from an interrupt handler) rather than path testing through the eyes of the reader of the mainline code. Especially in programs that have global state variables, interrupts at unexpected times can lead to very odd results.

- ***Every anticipated or potential race.*** **(**Here as in many other areas, see Appendix 1 of Kaner, Falk, & Nguyen, 1993 for a list and discussion of several hundred types of bugs, including interrupt-related, race-condition-related, etc.) Imagine two events, A and B. Both will occur, but the program is designed under the assumption that A will always precede B. This sets up a race between A and B—if B ever precedes A, the program will probably fail. To achieve race coverage, you must identify every potential race condition and then find ways, using random data or systematic test case selection, to attempt to drive B to precede A in each case. Races can be subtle. Suppose that you can enter a value for a data item on two different data entry screens. User 1 begins to edit a record, through the first screen. In the process, the program locks the record in Table 1. User 2 opens the second screen, which calls up a record in a different table, Table 2. The program is written to automatically update the corresponding record in the Table 1 when User 2 finishes data entry. Now, suppose that User 2 finishes before User 1. Table 2 has been updated, but the attempt to synchronize Table 1 and Table 2 fails. What happens at the time of failure, or later if the corresponding records in Table 1 and 2 stay out of synch?

- ***Every time-slice setting.*** In some systems, you can control the grain of switching between tasks or processes. The size of the time quantum that you choose can make race bugs, time-outs, interrupt-related problems, and other time-related problems more or less likely. Of course, coverage is an difficult problem here because you aren't just varying time-slice settings through every possible value. You also have to decide which tests to run under each setting. Given a planned set of test cases per setting, the coverage measure looks at the number of settings you've covered.

- ***Varied levels of background activity.*** In a multiprocessing system, tie up the processor with competing, irrelevant background tasks. Look for effects on races and interrupt handling. Similar to time-slices, your coverage analysis must specify:

    - *categories of levels of background activity (figure out something that makes sense) and*

    - *all timing-sensitive testing opportunities (races, interrupts, etc.).*

- ***Each processor type and speed***. Which processor chips do you test under? What tests do you run under each processor? You are looking for:

    - *speed effects, like the ones you look for with background activity testing, and*

- *consequences of processors' different memory management rules, and*
- *floating point operations, and*
- *any processor-version-dependent problems that you can learn about.*

- ***Every opportunity for file / record / field locking***.
- ***Every dependency on the locked (or unlocked) state of a file, record or field.***
- ***Every opportunity for contention for devices or resources.***
- ***Performance of every module / task / object:*** Test the performance of a module then retest it during the next cycle of testing. If the performance has changed significantly, you are either looking at the effect of a performance-significant redesign or at a symptom of a new bug.
- ***Free memory / available resources / available stack space at every line or on entry into and exit out of every module or object.***
- ***Execute every line (branch, etc.) under the debug version of the operating system:*** This shows illegal or problematic calls to the operating system.
- ***Vary the location of every file.*** What happens if you install or move one of the program's component, control, initialization or data files to a different directory or drive or to another computer on the network?
- ***Check the release disks for the presence of every file.*** It's amazing how often a file vanishes. If you ship the product on different media, check for all files on all media.
- ***Every embedded string in the program.*** Use a utility to locate embedded strings. Then find a way to make the program display each string.
- ***Every menu option.***
- ***Every dialogue or other data entry screen.***

## *Operation of every function / feature / data handling operation under:*

- ***Every program preference setting.***
- ***Every character set, code page setting, or country code setting.***
- ***The presence of every memory resident utility (inits, TSRs).***
- ***Each operating system version.***
- ***Each distinct level of multi-user operation.***
- ***Each network type and version.***

- ***Each level of available RAM.***

- ***Each type / setting of virtual memory management.***

## Compatibility

- ***Compatibility with every previous version of the program.***

- ***Ability to read every type of data available in every readable input file format.*** If a file format is subject to subtle variations (e.g. CGM) or has several sub-types (e.g. TIFF) or versions (e.g. dBASE), ***test each one.***

- ***Write every type of data to every available output file format.*** Again, beware of subtle variations in file formats—if you're writing a CGM file, full coverage would require you to test your program's output's readability by ***every one*** of the main programs that read CGM files.

- ***Every typeface supplied with the product.*** Check all characters in all sizes and styles. If your program adds typefaces to a collection of fonts that are available to several other programs, check compatibility with the other programs (nonstandard typefaces will crash some programs).

- ***Every type of typeface compatible with the program.*** For example, you might test the program with (many different) TrueType and Postscript typefaces, and fixed-sized bitmap fonts.

- ***Every piece of clip art in the product***. Test each with this program. Test each with other programs that should be able to read this type of art.

- ***Every sound / animation provided with the product.*** Play them all under different device (e.g. sound) drivers / devices. Check compatibility with other programs that should be able to play this clip-content.

- ***Every supplied (or constructible) script*** to drive other machines / software (e.g. macros) / BBS's and information services (communications scripts).

- ***All commands available in a supplied communications protocol.***

- ***Recognized characteristics.*** For example, every speaker's voice characteristics (for voice recognition software) or writer's handwriting characteristics (handwriting recognition software) or every typeface (OCR software).

- ***Every type of keyboard and keyboard driver.***

- ***Every type of pointing device and driver at every resolution level and ballistic setting.***

- ***Every output feature with every sound card and associated drivers.***

- ***Every output feature with every type of printer and associated drivers at every resolution level.***

- ***Every output feature with every type of video card and associated drivers at every resolution level.***

- ***Every output feature with every type of terminal and associated protocols.***

- ***Every output feature with every type of video monitor and monitor-specific drivers at every resolution level.***

- ***Every color shade displayed or printed to every color output device (video card / monitor / printer / etc.) and associated drivers at every resolution level.*** And check the conversion to grey scale or black and white.

- ***Every color shade readable or scannable from each type of color input device at every resolution level.***

- ***Every possible feature interaction between video card type and resolution, pointing device type and resolution, printer type and resolution, and memory level.*** This may seem excessively complex, but I've seen crash bugs that occur only under the pairing of specific printer and video drivers at a high resolution setting. Other crashes required pairing of a specific mouse and printer driver, pairing of mouse and video driver, and a combination of mouse driver plus video driver plus ballistic setting.

- ***Every type of CD-ROM drive, connected to every type of port (serial / parallel / SCSI) and associated drivers.***

- ***Every type of writable disk drive / port / associated driver.*** Don't forget the fun you can have with removable drives or disks.

- ***Compatibility with every type of disk compression software.*** Check error handling for every type of disk error, such as full disk.

- ***Every voltage level from analog input devices.***

- ***Every voltage level to analog output devices.***

- ***Every type of modem and associated drivers.***

- ***Every FAX command (send and receive operations) for every type of FAX card under every protocol and driver.***

- ***Every type of connection of the computer to the telephone line (direct, via PBX, etc.; digital vs. analog connection and signalling); test every phone control command under every telephone control driver.***

- ***Tolerance of every type of telephone line noise and regional variation (including variations that are out of spec) in telephone signaling (intensity, frequency, timing, other characteristics of ring / busy / etc. tones).***

- ***Every variation in telephone dialing plans.***

- ***Every possible keyboard combination.*** Sometimes you'll find trap doors that the programmer used as hotkeys to call up debugging tools; these hotkeys may crash a debuggerless program. Other times, you'll discover an Easter Egg (an undocumented, probably unauthorized, and possibly embarrassing feature). ***The broader coverage measure is every possible keyboard***

***combination at every error message and every data entry point.*** You'll often find different bugs when checking different keys in response to different error messages.

- ***Recovery from every potential type of equipment failure.*** Full coverage includes each type of equipment, each driver, and each error state. For example, test the program's ability to recover from full disk errors on writable disks. Include floppies, hard drives, cartridge drives, optical drives, etc. Include the various connections to the drive, such as IDE, SCSI, MFM, parallel port, and serial connections, because these will probably involve different drivers.

## Computation

- ***Function equivalence.*** For each mathematical function, check the output against a known good implementation of the function in a different program. Complete coverage involves equivalence testing of all testable functions across all possible input values.

- ***Zero handling.*** For each mathematical function, test when every input value, intermediate variable, or output variable is zero or near-zero. Look for severe rounding errors or divide-by-zero errors.

- ***Accuracy of every graph,*** across the full range of graphable values. Include values that force shifts in the scale.

## Information Content

- ***Accuracy of every report.*** Look at the correctness of every value, the formatting of every page, and the correctness of the selection of records used in each report.

- ***Accuracy of every message.***

- ***Accuracy of every screen.***

- ***Accuracy of every word and illustration in the manual.***

- ***Accuracy of every fact or statement in every data file provided with the product.***

- ***Accuracy of every word and illustration in the on-line help.***

- ***Every jump, search term, or other means of navigation through the on-line help.***

## Threats, Legal Risks

- ***Check for every type of virus / worm that could ship with the program.***

- ***Every possible kind of security violation of the program, or of the system while using the program.***

- ***Check for copyright permissions for every statement, picture, sound clip, or other creation provided with the program.***

## *Usability tests of:*

- ***Every feature / function of the program.***
- ***Every part of the manual.***
- ***Every error message.***
- ***Every on-line help topic.***
- ***Every graph or report provided by the program.***

## *Localizability / localization tests:*

- ***Every string.*** Check program's ability to display and use this string if it is modified by changing the length, using high or low ASCII characters, different capitalization rules, etc.
- ***Compatibility with text handling algorithms under other languages (sorting, spell checking, hyphenating, etc.)***
- ***Every date, number and measure in the program.***
- ***Hardware and drivers, operating system versions, and memory-resident programs that are popular in other countries.***
- ***Every input format, import format, output format, or export format that would be commonly used in programs that are popular in other countries.***
- ***Cross-cultural appraisal of the meaning and propriety of every string and graphic shipped with the program.***

## *Verifications*

- ***Verification of the program against every program requirement and published specification.*** (How well are we checking these requirements? Are we just hitting line items or getting to the essence of them?)
- ***Verify against every business objectives associated with the program*** (these may or may not be listed in the requirements).
- ***Verification of the program against user scenarios.*** Use the program to do real tasks that are challenging and well-specified. For example, create key reports, pictures, page layouts, or other documents events to match ones that have been featured by competitive programs as interesting output or applications.
- ***Verification against every regulation (IRS, SEC, FDA, etc.) that applies to the data or procedures of the program.***

## Coverage of specific types of tests:

- ***Automation coverage:*** Percent of code (lines, branches, paths, etc.) covered by the pool of automated tests for this product.
- ***Regression coverage:*** Percent of code that is covered by the standard suite of regression tests used with this program.
- ***Scenario (or soap opera) coverage:*** Percent of code that is covered by the set of scenario tests developed for this program.
- ***Coverage associated with the set of planned tests.***
    - Percent that the planned tests cover non-dead code (code that can be reached by a customer)
    - Extent to which tests of the *important* lines, branches, paths, conditions (etc.—important in the eyes of the tester) cover the total population of lines, branches, etc.
    - Extent to which the planned tests cover the *important* lines, branches, paths, conditions, etc.
    - Extent to which the scenario tests cover the *important* lines, branches, paths, conditions, or other variables of interest.

## Inspection coverage

- ***How much code has been inspected.***
- ***How many of the requirements inspected/reviewed.***
- ***How many of the design documents inspected.***
- ***How many of the unit tests inspected.***
- ***How many of the black box tests inspected.***
- ***How many of the automated tests inspected.***
- ***How many test artifacts reviewed by developers.***

## User-focused testing

- ***How many of the types of users have been covered or simulated.***
- ***How many of the possible use cases have been covered.***

## Agreement-Based Measures

Agreement-based measures start from an agreement about what testing will and will not be done. The agreement might be recorded in a formal test plan or in a memo cosigned by different stakeholders. Or it might simply be a work list that the test group has settled on. The list might fully detailed, spelling out every test case. Or it might list more general areas of work and describe the depth of testing (and the time budget) for each one. The essence of agreement-based measures is progress against a plan.

All of these measures are proportions: amount of work done divided by the amount of work planned. We can convert these to effort reports easily enough just by reporting the amount of work done.

- ***Percent of test requirements turned into tests.***
- ***Percent of specification items turned into tests.***
- ***Percent of planned tests developed.***
- ***Percent of tests planned that have been executed.***
- ***Percent of tests executed that have passed.***
- ***Percent of tests passing, but that are old (haven't been rerun recently).***
- ***Percent (and list) of tests not yet executed.***
- ***Percent (list, whatever) of deliverables given to the client vs. total expected, where deliverables include bugs, scripts, reports and documentation (e.g. the matrix).***
- ***Test hours (or number of tests executed) per priority level (compared to the proportional levels of work intended for different priorities).***
- ***Test hours (or number of tests executed) per feature area (compared to the time intended for different areas).***
- ***Pattern of progress of testing (how much planned, how much done, how much left to do) across different test types across feature (or activity) areas.***
- ***Timesheet report of how tester hours were spent this week, against a pre-existing set of categories of tasks (time consumption types).***
- ***All test cases written / run.***
- ***Compare current results / status to product release criteria.***
- ***It's Tuesday – time to ship.***

## Effort-Based Measures

These can be turned into agreement-based measures if we have an expected level of effort for comparison to the actual level completed.

- ***How much code must be tested.***
- ***How many spec pages / manual pages are associated with the material to be tested.***
- ***How many changes have been specified for the new version of an old product? How changes many are not specified but are being made?***
- ***How different is the new test plan from the old test plan.***
- ***Number of hours spent testing.***
- ***Amount of overtime per tester.***
- ***Compare the amount of overtime worked by testers and developers.***
- ***Number of test cycles completed.***
- ***Number of tests performed in each cycle.***
- ***Duration of each test cycle.***
- ***Number of tests executed on production hardware.***
- ***Number of test sessions (or some other unit of exploratory testing effort).*** See Bach (2000) for a discussion of exploratory testing sessions.
- ***Number of functional areas in which we're doing exploratory testing on a weekly basis.***
- ***Average cost of finding and/or fixing a bug.***
- ***Time spent maintaining automated tests {total, to date, per test cycle}.***
- ***Time to add new tests.*** How long it takes to add manual or automated tests to your population of tests. Include the time it takes to document the test, to file it in your source control system (if you have one), to plan the test, including checking whether it is redundant with current tests, etc.
- ***Time spent debugging the tests.***
- ***Time spent checking failures to see if you are reporting duplicates.***
- ***Time spent by testers writing defect reports.***

## Project-History Based Measures

If you've run several projects, you can compare today's project with historical results. When someone claims that the project has reached "beta", you might compare current status with the state of the other products when they reached the beta milestone. For example, some groups add features until the last minute. At those companies, "all code complete" is not a criterion for alpha or beta milestones. But you might discover that previous projects were 90% code complete 10 weeks before their ship date.

Many of the measures flagged as based on risk, effort, or agreement can be used as project-history based if you have measured those variables previously, under circumstances that allow for meaningful comparison, and have the data available for comparison.

- ***How much time spent, compared to time expected from prior projects***

- ***Counts of defects discovered, fixed, open compared to previous project patterns.*** For example, number (or proportion) of defects open at alpha or beta milestone. These reports are widely used and widely abused. Significant side effects (impacts on the test planning, bug hunting, bug reporting, and bug triage processes) seem to be common, even though they are not often discussed in print. For a recent discussion that focuses on bug counts, see Hoffman (2000). For an excellent but more general discussion, see Austin (1996)..

- ***Arrival rates of bugs (how many bugs opened) per week, compared to rates on previous projects.*** I have the same concerns about the curves that show bugs per week as the single point (e.g. beta milestone) comparisons.

- ***How many (or percent of) modules or components coded compared to previous releases at this point in the project.***

## Risk Based Measures

As I am using the term, the risk measures focus on risk remaining in the project. Imagine this question: *If we were to release the product today, without further testing, what problems would we anticipate? Similarly, if we were to release the product on its intended release date, what problems would we anticipate?*

- ***Of the test cases planned using hazard or risk analysis, how many have been written, run, or passed?***

- ***In a group that creates new tests throughout the testing project, have we reached a point in which we don't have any more questions?*** This might tell us that there is a problem with the test group, or it might tell us that we are close to being finished. (There might be plenty of errors left in the product, but if we're out of ideas, we're not going to find those errors.)

- ***How many bugs have been found in third party software that you include with or run with your products?***

- ***How many failures found per unit (per test case, per thousand pages printed, per thousand lines of code, per report, per transaction, per hour, etc.)***

- ***Mean time (transactions, etc.) between failures?***

- ***Length of time before system failure (or other detectable failures) in a stochastic test or a load or stress test.***

- ***Rate at which you are finding new bugs.***

- ***Rate at which previously fixed bugs are reappearing.***

- ***Number of complaints from beta testers.***

- ***Number of bugs found in the first manufactured copy of the product.*** If you release a product to manufacturing, you should test the first few manufactured copies, before sending the rest to customers. Manufacturing errors happen. The disks (books, collaterals) that you sent out for duplication may or may not match what you get back.

- ***Number of bugs found in a random selection of manufactured copies.*** In most products, all of the manufactured copies are identical. Some products, however, vary from copy to copy. Even if the only difference across products is in the serial number, problems might show up on one copy that don't show up on others. The more extensively customized each copy is, the more relevant is this type of testing.

- ***After the product is released, we obtain data from the field that tell us whether we should reopen the product, such as:***

    - ***Angry letters to the CEO.***

    - ***Published criticism of the product.***

    - ***Rate of customer complaints for technical support.***

    - ***Number of "surprise" defects*** (problems found by customers that were not found pre-release by testers).

- ***Number of "surprisingly serious" defects*** (deferred problems that have generated more calls or angrier calls than anticipated.)

- ***Perceived risk.*** For example, the group might provide a rough estimate of risk for each area of the product on a 5-point scale (from low risk to very high risk), in order to think through the pattern of work left to be done in a short time available.

- ***Desperation level of triage team.*** The triage team is the group that decides whether or not to fix each defect. Toward the end of the project (especially if this "end" is a long time after the initially scheduled ship date), this team might be pushed very hard to stop fixing and just ship it. The more desperate the team is to ship the product *soon*, the higher the risk that the product will leave in bad shape.

- ***Number of defects resolved without being fixed.***

- ***Frequency of tester outbursts.*** This measure is informal but can be extremely important. As a poor quality product marches to release, many testers or test groups feel increasing pressure to

find defects that will block the shipment. Frustration mounts as perfectly "good" bugs are deferred. Indication of high stress among the testers is a signal of danger in the product or the review process.

- ***Level of confusion or dissent among the development team as a whole or among sub-groups.*** (You measure this by talking with people, asking them how they feel about the product and the project, and its projected release date.

## Obstacle Reports

Obstacles are *risks to the testing project* (or to the development project as a whole). These are the things that make it hard to do the testing or fixing well. This is not intended as a list of everything that can go wrong on a project. Instead, it is a list of common problems that make the testing less efficient.

- ***Turnover of development staff (programmers, testers, writers, etc.)***
- ***Number of marketing VPs per release.*** (Less flippantly, what is the rate of turnover among executives who influence the design of the product?).
- ***Layoffs of testing (or other development) staff.***
- ***Number of organizational changes over the life of the project.***
- ***Number of people who influence product release, and level of consensus about the product among them.***
- ***Appropriateness of the tester to programmer ratio.*** (Note: I've seen successful ratios ranging from 1:5 through 5:1. It depends on the balance of work split between the groups and the extent to which the programmers are able to get most of the code right the first time.)
- ***Number of testers who speak English*** (if you do your work in English).
- ***Number of testers who speak the same language as the programmers.***
- ***How many bugs are found by isolated (such as remote or offsite) testers compared to testers who are co-located with or in good communication with the programmers?***
- ***Number of tests blocked by defects.***
- ***List of defects blocking tests.***
- ***Defect fix percentage (if low).***
- ***Slow defect fix rate compared to find rate.***
- ***Average age of open bugs.***
- ***Average time from initial report of defect until fix verification.***
- ***Number of times a bug is reopened (if high).***

- ***Number of promotions and demotions of defect priority***
- ***Number of failed attempts to get builds to pass smoke tests.***
- ***Number of changes to (specifications or) requirements during testing.***
- ***Percentage of tests changed by modified requirements.***
- ***Time lost to development issues*** (such as lack of specifications or features not yet coded).
- ***Time required to test a typical emergency fix.***
- ***Percentage of time spent on testing emergency fixes.***
- ***Percentage of time spent providing technical support for pre-release users (such as beta testers).***
- ***How many billable hours per week (for a consulting firm) or the equivalent task-focused hours (for in-house work) are required of the testers and how does this influence or interfere with their work?***
- ***Ability of test environment team (or information systems support team) to build the system to be tested as specified by the programming team.***
- ***Time lost to environment issues*** (such as difficulties obtaining test equipment or configuring test systems, defects in the operating system, device drivers, file system or other 3rd party, system software).

## Evaluation-of-Testing Based Measures

These help you assess the testing effort. How hard are the testers doing, how well are they doing it, what could they improve? A high number for one of these measures might be good for one group and bad for another. For example, in a company that relies on an outside test lab to design a specialized set of tests for a very technical area, we'd expect a high bug find rate from third party test cases. In a company that thinks of its testing as more self-contained, a high rate from third parties is a warning flag.

- ***Number of crises involving test tools.***
- ***Number of defects related to test environment.***
- ***Number of bugs found by boundary or negative tests vs. feature tests.***
- ***Number of faults found in localized versions compared to base code.***
- ***Number of faults found in inspected vs. uninspected areas.***
- ***Number of defects found by inspection of the testing artifacts.***
- ***Number of defects discovered during test design (rather than in later testing).***

- ***Number of defects found by 3rd party test cases.***
- ***Number of defects found by 3rd party test group vs. your group.***
- ***Number of defects found by developers.***
- ***Number of defects found by developers after unit testing.***
- ***Backlog indicators.*** For example, how many unverified bug fixes are there (perhaps as a ratio of new bug fixes submitted) or what is the mean time to verify a fix?
- ***How many phone calls were generated during beta.***
- ***Number of surprise bugs (bugs you didn't know about) reported by beta testers.*** (the content of these calls indicate holes in testing or, possibly, weaknesses in the risk analysis that allowed a particular bug to be deferred)**.**
- ***After the product is released,***
    - *Number of "surprise" defects* (problems found by customers that were not found pre-release by testers).
    - *Number of "surprisingly serious" defects* (deferred problems that have generated more calls or angrier calls than anticipated.).
    - *Angry letters to the CEO*. (Did testers mis-estimate severity?)
    - *Published criticism of the product.* (Did testers mis-estimate visibility?)
    - *Rate of customer complaints for technical support*? (Did testers mis-estimate customer impact?)
- ***Number of direct vs. indirect bug finds (were you looking for that bug or did you stumble on it as a side effect of some other test?)***
- ***Number of irreproducible bugs (perhaps as a percentage of total bugs found).***
- ***Number of noise bugs (issues that did not reflect software errors).***
- ***Number of duplicate bugs being reported.***
- ***Rumors of off-the-record defects*** (defects discovered but not formally reported or tracked. The discoveries might be by programmers or by testers who are choosing not to enter bugs into the tracking system—a common problem in companies that pay special attention to bug counts.)
- ***Number of test cases that can be automated.***
- ***Cyclomatic complexity of automated test scripts.***
- ***Size (e.g. lines of code) of test automation code. Appraisal of the code's maintainability and modularity.***

- ***Existence of requirements analyses, requirements documents, specifications, test plans and other software engineering processes and artifacts generated for the software test automation effort.***

- ***Percentage of time spent by testers writing defect reports.***

- ***Percentage of time spent by testers searching for bugs, writing bug reports, or doing focused test planning on this project.*** (Sometimes, you're getting a lot less testing on a project than you think. Your staff may be so overcommitted that they have almost no time to focus and make real progress on anything. Other times, the deallocation of resources is intentional. For example, one company has an aggressive quality control group who publish bug curves and comparisons across projects. To deal with the political hassles posed by this external group, the software development team sends the testers to the movies whenever the open bug counts get too high or the fix rates get too low.)

- ***Percentage of bugs found by planned vs. exploratory vs. unplanned methods, compared to the percentage that you intended.***

- ***What test techniques were applied compared to the population of techniques that you think are applicable compared to the techniques the testers actually know.***

- ***Comparative effectiveness:*** what problems were found by one method of testing or one source vs. another.

- ***Comparative effectiveness over time:*** compare the effectiveness of different test methods over the life cycle of the product. We might expect simple function tests to yield more bugs early in testing and complex scenario tests to be more effective later.

- ***Complexity of causes over time:*** is there a trend that bugs found later have more complex causes (for example, require a more complex set of conditions) than bugs found earlier? Are there particular patterns of causes that should be tested for earlier?

- ***Differential bug rates across predictors:*** defect rates tracked across different predictors. For example, we might predict that applications with high McCabe-complexity numbers would have more bugs. Or we might predict that applications that were heavily changed or that were rated by programmers as more fragile, etc., would have more bugs.

- ***Delta between the planned and actual test effort.***

- ***Ability of testers to articulate the test strategy.***

- ***Ability of the programmers and other developers to articulate the test strategy.***

- ***Approval of the test effort by an experienced, respected tester.***

- ***Estimate the prospective effectiveness of the planned tests.*** (How good do you think these are?)

- ***Estimate the effectiveness of the tests run to date*** (subjective evaluation by testers, other developers).

- ***Estimate confidence that the right quality product will ship on time*** (subjective evaluation by testers, other developers).

## Results Reports

So what has the test group accomplished? Most of the bug count metrics belong here.

- ***Build acceptance test criteria and results.***
- ***Benchmark performance results.***
- ***Number of builds submitted for testing.***
- ***Number of builds tested.***
- ***Number of revisions (drafts) of the test plan***
- ***Number of test cases created / coded / documented.***
- ***Number of reusable test cases created.***
- ***Percent or number of reusable test artifacts in this project.***
- ***Number of times reusable artifacts are reused.***
- ***Number of defects in test cases.***
- ***List of failing (defective) tests.***
- ***Number of promotions and demotions of defect priority.***
- ***Tester to developer ratio.***
- ***Defects per 1000 lines of code.***
- ***Total bugs found / fixed / deferred / rejected.***
- ***Find and close rate by severity.***
- ***Total number of open bugs.***
- ***Number of open bugs by {feature, subsystem, build, database, supplier}.***
- ***Number of open bugs assigned to {programmer, tester, someone else, total}.***
- ***Number of open bugs by platform.***
- ***Defect find rate per week (open defects/week).***
- ***Cumulative defects each week (Total defects found/time.)***

- ***Defect find rate normalized by the amount of time spent testing.***
- ***Number of bugs found from regression testing.***
- ***Number of bugs fixed or failed from regression testing.***
- ***Number of bugs found in beta/real customer testing over time.***
- ***Percent of tests per build {passing, failing, not yet analyzed}.***
- ***Number of defects found in requirements / specifications / other development planning documents.***
- ***Number of defects found in unit testing.***
- ***Number of bugs found after a ship candidate has been declared***
- ***Number of defects found in LAN vs. modem testing. (Similarly for other configuration variations.)***
- ***Percentage of bugs found per project phase.***
- ***Number or percentage of bugs found by programmers.***
- ***Number of bugs found by programmers after unit testing.***
- ***Number of bugs found in first article (first item manufactured by manufacturing).***
- ***Proportion of unreviewed bugs that become critical after review. (Some companies don't have a review process. Others have each bug report reviewed by (depending on the company) another tester, a programmer, or the triage team as a whole. )***

# Progress Reporting Examples

The following examples are based on ideas presented at LAWST and STMR, but I've made some changes to simplify the description, or because additional ideas came up at the meeting (or since) that seem to me to extend the report in useful ways. Occasionally, the difference between my version and the original occurs because (oops) I misunderstood the original presentation.

## *Feature Map*

This description is based on a presentation by Jim Bampos and an ensuing discussion. This illustrates an approach that is primarily focused on agreement-based measures.

For each feature, list different types of testing that would be appropriate (and that you intend to do). Different features might involve different test types. Then, for each feature / testing type pair, determine when the feature will be ready for that type of testing, when you plan to do that testing, how much you plan to do, and what you've achieved.

A spreadsheet that tracked this might look like:

| Feature | Test Type | Ready | Primary Testing (week of. . .) | Time Budget | Time Spent | Notes |
|---|---|---|---|---|---|---|
| *Feature 1* | *Basic functionality* | *12/1/00* | *12/1/00* | *½ day* | | |
| | *Domain* | *12/1/00* | *12/1/00* | *½ day* | | |
| | *Load / stress* | *1/5/01* | *1/15/01* | *1.5 days* | | |
| | *Scenario* | *12/20/00* | *1/20/01* | *4 days* | | |

The chart shows "time spent" but not "how well tested." Supplementing the chart is a list of deliverables and these are reviewed for quality and coverage. Examples of primary deliverables are:

- Bug reports
- Test plans (test case design matrices, feature coverage matrices, etc.)
- Test scripts
- Summary reports (describing what was done / found for each area of testing)
- Recommendation memos.

## *Component Map*

This description is based on a presentation by Elisabeth Hendrickson and an ensuing discussion. It primarily focuses on agreement-based measures.

For each component list the appropriate types of testing. (Examples: functionality, install/uninstall, load/stress, import/export, engine, API, performance, customization.) These vary for different components. Add columns for the tester assigned to do the testing, the estimated total time and elapsed time to date, total tests, number passed, failed or blocked and corresponding percentages, and the projected testing time for this build. As Elisabeth uses it, this chart covers testing for a single build, which usually lasts a couple weeks. Later builds get separate charts. In addition to these detail rows, a summary is made by tester and by component.

| Component | Test Type | Tester | Total Tests Planned / Created | Tests Passed / Failed / Blocked | Time Budget | Time Spent | Projected for Next Build | Notes |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| | | | | | | | | |

## *Area / Feature Summary Status Chart*

Gary Halstead and I used a chart like this to manage a fairly complex testing project. This was one of the successful projects that led to my report in Kaner (1996) rather than one of the failures that led to my suggesting in that report that this approach doesn't always work.

For our purposes, think of the output from testing project planning as a chart that spans many pages.

- An "area" represents a significant block of expenses. For example, "printer configuration testing" might be an area. An area might include one or more features or one type of testing or it might include a pure block of expense like "vacations and training." If you will spend a lot of staff time on something, you can call it an area. I prefer to work with 10-25 areas.

- Within an area, there are sub-areas, such as individual features, or sub-features. Alternatively, you might break an area down into a series of different types of tests that you would apply within that area. Ideally, you'll break the area down until you have a list of items that won't take more than a day apiece. In some cases, you might not understand the task well enough to break it down this finely, or the natural breakpoint might involve more time. The goal is to end up with a list with tasks that are well enough broken down that you can make reasonable time estimates..

- For each sub-area, estimate the depth of testing to be done. We used labels like "mainstream", "guerilla" (which means time-boxed exploratory testing), "formal planned testing," and "planned regression testing" to identify different depths, but you can use any categories that work. "Small", "medium" and "large" might be good enough.

- Next, estimate time. How long will it take to test this feature using this technique (do this sub-area task) at this level of depth of testing?

- The chart has two more columns, one to record how much time was actually spent and the other to record an estimate of how much work in the sub-area task was actually completed.

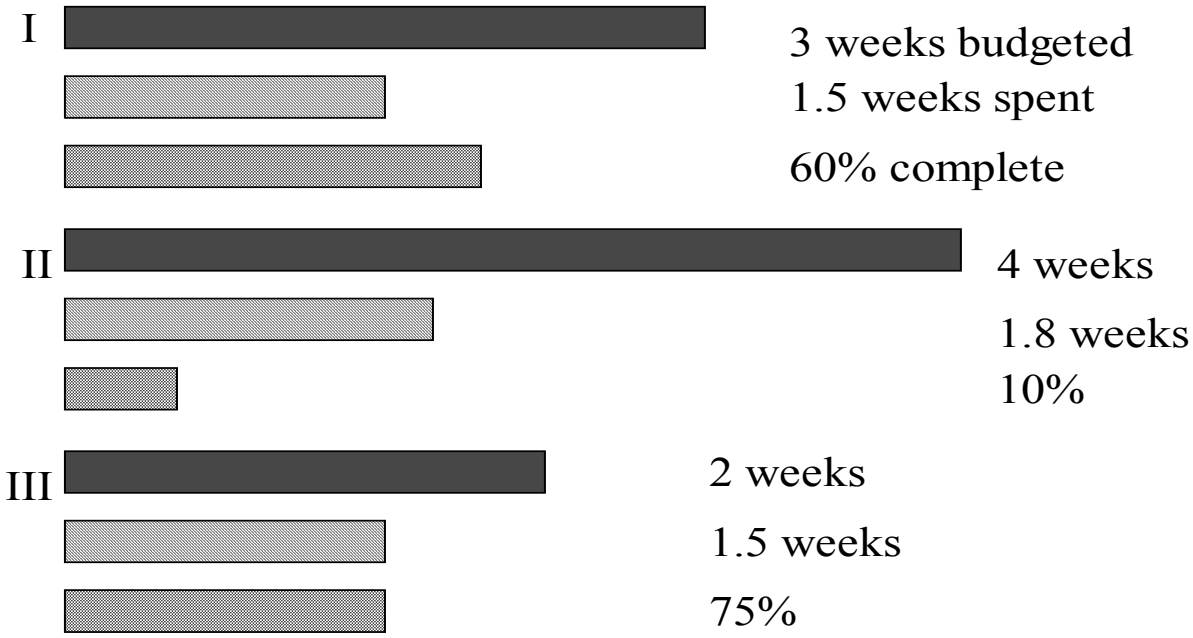
This chart might run as many as 50 pages (one row per sub-area task). No one will want to review it at a status meeting, but it is a useful data collection worksheet. There are various ways to figure out, each week, how much got done on each task. This administrative work is not inexpensive. However, it can pay for itself quickly by providing relatively early notice that some tasks are not being done or are out of control.

A *summary* of the chart is useful and well received in status reports. For each area, you can determine from the worksheets the total amount of time budgeted for the area (just add up the times from the individual tasks), how much time has actually been spent, and what percentage of work is actually getting

done. (Use a weighted average—a 4 week task should have 4 times as much effect on the calculation of total percent complete as a 1 week task.)

The figure below illustrates the layout of the summary chart. The chart shows every area. The areas are labeled I, II, and III in the figure. You might prefer meaningful names. For each area, there are three bars. The first shows total time budgeted, the second shows time spent on this area, the third shows percentage of this area's work that has been completed. Once you get used to the chart, you can tell at a glance whether the rate of getting work done is higher or lower than your rate of spending your budgeted time.




## *Project Status Memo*

Some test groups submit a report on the status of each testing project every week. I think this is a useful practice.

The memo can include different types of information. Borrowing an analogy from newspapers, I put the bug counts (software equivalent of sports statistics) (the most read part of local newspapers) back several pages. People will fish through the report to find them.

The front page covers issues that might need or benefit from management attention, such as lists of deliverables due (and when they are due) from other groups, decisions needed, bugs that are blocking testing, and identification of other unexpected obstacles, risks, or problems. I might also brag about significant staff accomplishments.

The second page includes some kind of chart that shows progress against plan. Any of the charts described so far (the feature map, the component map, the summary status chart) would be useful on the second page.

The third page features bug counts. The fourth page typically lists the recently deferred bugs, which we'll talk about at the weekly status meeting.

Across the four pages (or four sections), you can fit information about effort, obstacles, risk, agreement-based status, bug counts, and anything else you want to bring to the project team's or management's attention.

## *Effort Report*

When you assign someone to work on a project, you might not get much of their time for that project. They have other things to do. Further, even if they were spending 100% of their time on that project, they might only be running tests a few hours per week. The rest of their time might be spent on test planning, meetings, writing bug reports, and so on.

The following chart is a way of capturing how your staff spend their time.

Rather than using percentages, you might prefer to work in hours. This might help you discover that the only people who are getting to the testing tasks are doing them in their overtime. (Yes, they are spending 20% of their time on task, but that 20% is 10 hours of a 50 hour week.)

Effort-based reporting is useful when people are trying to add new projects or new tasks to your group's workload, or when they ask you to change your focus. You show the chart and ask which things to cut back on, in order to make room for the new stuff.

| | 0-10% | 10%-20% | 20%-30% | 30%-40% | 40%-60% |
|---|---|---|---|---|---|
| **Coordination** | 8 | 2 | 0 | 2 | 0 |
| **Status Reporting** | 12 | 0 | 0 | 0 | 0 |
| **Setup Environment** | 4 | 3 | 3 | 0 | 1 |
| **Scoping** | 5 | 5 | 2 | 0 | 0 |
| **Ad Hoc** | 7 | 2 | 1 | 2 | 0 |
| **Design/Doc** | 1 | 3 | 5 | 1 | 2 |
| **Execution** | 1 | 3 | 5 | 3 | 0 |
| **Inspections** | 7 | 5 | 0 | 0 | 0 |
| **Maintenance** | 9 | 3 | 0 | 0 | 0 |
| **Review** | 10 | 2 | 0 | 0 | 0 |
| **Bug Advocacy** | 5 | 5 | 2 | 0 | 0 |

## *A Set of Charts for Requirements-Driven Testing*

Jim Kandler manages testing in an FDA-regulated context. He tracks/reports the progress of the testing effort using a set of charts similar to the following four:

- Traceability matrix that traces test cases to requirements
- Test case status chart
- Testing effort charts
- Bug status charts

### Traceability Matrix

In the traceability matrix, you list specification items or requirements items across the top (one per column). Each row is a test case. A cell in the matrix pairs a test case with a requirement. Check it off if that test case tests that requirement.

| | Requirement 1 | Requirement 2 | Requirement 3 | Requirement 4 | Requirement 5 | Etc. |
|---|---|---|---|---|---|---|
| Test 1 | * | | * | | | |
| Test 2 | | * | * | | * | |
| Test 3 | | | * | | | |
| Totals | 1 | 1 | 3 | 0 | 1 | |

The chart shows several things:

- If Requirement 5 changes, then you know that you need to update test case 2 and that you probably don't need to update the others.
- Requirement 4 is untested
- Requirement 3 is tested in every test. The matrix is *unbalanced*—there appears to be too much testing of some features compared to others.

Notice that if we don't write down a requirement, we won't know from this chart what is the level of coverage against that requirement. This is less of an issue in an FDA-regulated shop because all requirements (claims) must be listed, and there must be tests that trace back to them.

## Test Case Status Chart

For each test case, this chart tracks whether it is written, has been reviewed, has been used, and what bugs have been reported on the basis of it. If the test is blocked (you can't complete it because of a bug), the chart shows when the blocking bug was discovered.

Many companies operate with a relatively small number of complex regression tests (such as 500). A chart that shows the status of each test is manageable in size and can be quite informative.

| Test Case ID | Written | Reviewed | Executed | Blocked | Software Change Request |
|---|---|---|---|---|---|
| 1 | 1 | 1 | | | |
| 2 | 1 | 1 | 1 | | 118 |
| 3 | 1 | 1 | 1 | 10/15/00 | 113 |
| 4 | 1 | | | | |
| 5 | | | | | |
| 6 | | | | | |
| 7 | | | | | |
| **Totals 7** | **4** | **3** | **2** | | |
| **100%** | **57%** | **42%** | **29%** | | |

## Testing Effort Chart

For each tester, this shows how many hours the person spent testing that week, how many bugs (issues) she found and therefore how many hours per issue. The numbers are rolled up across testers for a summary report to management that shows the hours and issues of the group as a whole.

| Week of | Hours | Issues | Hr/Issue |
|---|---|---|---|
| **20 Aug** | 19.5 | 19 | 1.03 |
| **27 Aug** | 40.0 | 35 | |
| **3 Sept** | 62 | 37 | |
| **10 Sept** | 15 | 10 | |

A chart like this helps the manager discover that individual staff members have conflicting commitments (not enough hours available for testing) or that they are taking remarkably short or long times per bug. Trends (over a long series of weeks) in availability and in time per failure can be useful for highlighting problems with individuals, the product, or the project staffing plan.

As a manager, I would not share the individuals' chart with anyone outside of my group. There are too many risks of side effects. Even the group totals invite side effects, but the risk might be more manageable.

### Bug Status

Bug handling varies enough across companies that I'll skip the chart. For each bug, it is interesting to know when it was opened, how long it's been opened, who has to deal with it next, and its current resolution status (under investigation, fixed, returned for more info, deferred, etc.). Plenty of other information might be interesting. If you put too much in one chart, no one will understand it.

### Summary

Jim shared some lessons learned with these four charts (traceability, test case status, effort, and bug statistics) that are generally applicable for project status reporting. In summary, he said:

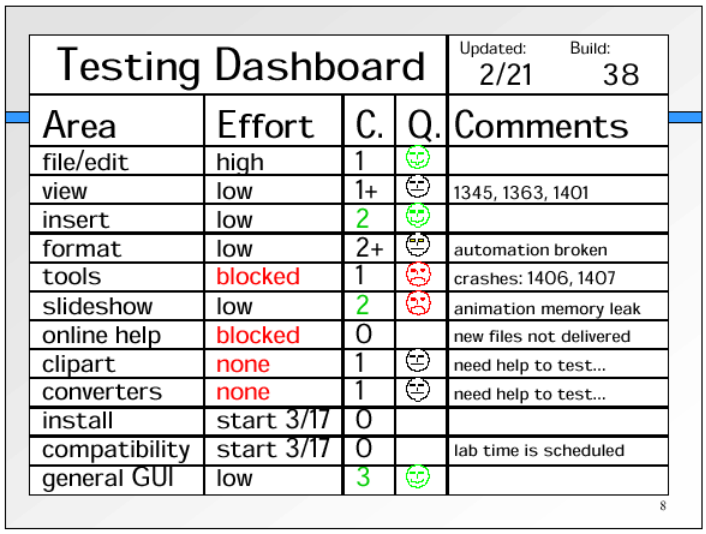
- It doesn't have to be complicated to be valuable. Simple charts are valuable.
- It's important to understand your universe (or scope) of tests.
- It's important to have very clean tracking of progress against the universe of tests.
- Under the right circumstances, repeated use of the same tests can be very valuable.
- You learn a lot of project status issues by reporting multi-week statistics instead of a snapshot.
- It's valuable to track human hours against a specific activity. The process of tracking this leads you to questions that help you manage the project more effectively.
- You can pinpoint high risk areas once you've established scope.
- You can use this tool as a guide to the chronology of the project.
- This approach to tracking is time effective and easy to delegate.
- It's instructive to note that not all tests are created equal—the percentages in the chart may be misleading.

These lessons apply well in highly regulated industries. In mass-market software, some of them will be less applicable.

## *Project Dashboard*

James Bach has spoken several times about his project status dashboard. See, for example, Bach (1999), which includes the following picture as an example. This is a simple tool, but a very useful one.



Testing Dashboard — Updated: 2/21 — Build: 38

| Area | Effort | C. | Q. | Comments |
|---|---|---|---|---|
| file/edit | high | 1 | ☺ | |
| view | low | 1+ | 😐 | 1345, 1363, 1401 |
| insert | low | 2 | ☺ | |
| format | low | 2+ | 😐 | automation broken |
| tools | blocked | 1 | ☹ | crashes: 1406, 1407 |
| slideshow | low | 2 | ☹ | animation memory leak |
| online help | blocked | 0 | | new files not delivered |
| clipart | none | 1 | 😐 | need help to test... |
| converters | none | 1 | 😐 | need help to test... |
| install | start 3/17 | 0 | | |
| compatibility | start 3/17 | 0 | | lab time is scheduled |
| general GUI | low | 3 | ☺ | |

8

The dashboard is created on a large whiteboard, typically in the main project conference room.

Different groups vary the columns and the headings. Bach prefers to keep the design as simple and uncluttered as possible. Here's the variation that I prefer (which breaks one of James' columns in two).

- ***Area:*** Break the testing project down into not more than 25 or 30 areas (at most).
- ***Effort***: How much work / time are we currently spending on this area? ("Blocked" means that we want to test in this area but we can't because the code isn't testable, perhaps because of missing code or a blocking bug.)
- ***Coverage planned***: However we define coverage, how much coverage do we intend to achieve in this area? (0 = untested, 3 = very high coverage).
- ***Coverage achieved:*** How much have we gotten done?
- ***Quality:*** Blank = no data. Frowny-face = bad quality. Neutral-face = not great yet but not a crisis. Smiley-face = high quality.
- **Comments:** Includes notes and bug report numbers. Whatever is needed to quickly flag issues.
- **Green means things go well. Red indicates a serious problem. Yellow indicates a problem that needs attention.**

A manager can read the testing project status at a glance from this board. If there are lots of frowny faces and red rows, the project has trouble. If planned and achieved coverage don't match, the product needs more testing. Update the dashboard frequently (every day or few days) and it becomes a focal point for project status discussions.

Bach (1999) provides more information about the dashboard in practice. His live presentation is worth attending.

## Summing Up

Many test groups focus their progress reports on bug counts. When you ask how much testing they've done, they say, 322 bugs worth. Or they speak in terms of a product coverage measure, like code coverage.

There are many other ways to answer the questions:

- How much testing have you gotten done?
- How much do you have left to do?
- How are you going to tell management about it?

This paper has provided a survey of suggested answers, a framework for evaluating them, and samples of presentation approaches used by several successful test managers and consultants.

This paper doesn't provide what we might think that we really need—a small set of measures that are known to be useful and valid, and a standardized reporting style that everyone will anticipate and learn to understand.

It takes time to develop and validate a good set of metrics. I'm not there yet. I don't think the field is there yet. I don't expect us to be there next year or the year after, but we'll make progress. Unless we have a breakthrough, progress is incremental.

## References


Amland (1999), "Risk Based Testing and Metrics" *16th International Conference on Testing Computer Software.*

Austin, R.D. (1996) *Measuring and Managing Performance in Organizations*.

Bach, James (1999) "A Low Tech Testing Dashboard", *STAR Conference East*, Available at www.satisfice.com/presentations/dashboard.pdf.

Bach, Jonathan. (2000) "Measuring Ad Hoc Testing", *STAR Conference West*.

Beizer, B. (1990) *Software Testing Techniques* (2nd Ed.)

S. Brown (Ed., 1991) *The Product Liability Handbook: Prevention, Risk, Consequence, and Forensics of Product Failure*.

Cornett, S., *Code Coverage Analysis*, www.bullseye.com/coverage.html

Dhillon, B.S. (1986) *Human Reliability With Human Factors*.

Fenton, N. & Pfleeger S. (1997) *Software Metrics: A Rigorous & Practical Approach*.

Glass, R.L. (1992) *Building Quality Software*.

Grady, R.B. (1992) *Practical Software Metrics for Project Management and Process Improvement*.

Grady, R.B. & D.L. Caswell, (1987) *Software Metrics: Establishing a Company-Wide Program*.

Hoffman, D. (2000) "The darker side of metrics", *Pacific Northwest Software Quality Conference*.

Johnson, M.A. (1996) *Effective and Appropriate Use of Controlled Experimentation in Software Development Research*, Master's Thesis (Computer Science), Portland State University.

Jones, C. (1991) *Applied Software Measurement,* 1991, p. 238-341.

Kaner, C. (1995) "Software negligence and testing coverage", *Software QA Quarterly*, Volume 2, #2, 18. Available at www.kaner.com.

Kaner, C. (1996) "Negotiating testing resources: A collaborative approach", *Ninth International Software Quality Week Conference.* Available at www.kaner.com.

Kaner, C. (1997) "The impossibility of complete testing", *Software QA,* Volume 4, #4, 28. Available at www.kaner.com.

Kaner, C. (1999) "Yes, but what are we measuring?", *Pacific Northwest Software Quality Conference,* available from the author at kaner@kaner.com.

Kaner, C., J. Falk, & H.Q. Nguyen (1993, reprinted 1999) *Testing Computer Software* (2nd. Ed.)

Kitchenham, Pfleeger, & Fenton (1995) "Towards a framework for software measurement and validation." *IEEE Transactions on Software Engineering*, vol. 21, December, 929.

Marick, B. (1995) *The Craft of Software Testing*.

Marick, B. (1999), "How to Misuse Code Coverage", *16th International Conference on Testing Computer Software*. Available at www.testing.com/writings.html.

Marick, B. (2000), "Faults of Omission", *Software Testing & Quality Engineering,* January issue. Available at www.testing.com/writings.html.

Myers, G. (1979) *The Art of Software Testing*.

Schneidewind, N. (1994) "Methodology for Validating Software Metrics," *Encyclopedia of Software Engineering* (Marciniak, ed.) *see also* IEEE 1061, *Standard for a Software Quality Metrics Methodology*.

Weinberg, G.M. (1993) *Quality Software Management, Volume 2, First-Order Measurement*.

Weinberg, G.M. & E.L. Schulman (1974) "Goals and performance in computer programming," *Human Factors*, 16(1), 70-77.

Zuse, H. (1997) *A Framework of Software Measurement*.

# *Measuring the Extent of Testing*

Cem Kaner

Pacific Northwest Software Quality Conference

October 17, 2000

# Question

- Imagine being on the job. Your local PBH (pointy-haired boss) drops in and asks

  *"So, tell me.*

  *How much testing have you gotten done?"*

- Please write down an answer. Feel free to use fictitious numbers but (except for the numbers) try to be realistic in terms of the type of information that you would provide.

# *The Question is Remarkably Ambiguous*

Common answers are based on the:

**Product** ➢ We've tested 80% of the lines of code.

**Plan** ➢ We've run 80% of the test cases.

**Results** ➢ We've discovered 593 bugs.

**Effort** ➢ We've worked 80 hours a week on this for 4 months.
We've run 7,243 tests.

# *The Question is Remarkably Ambiguous*

Common answers are based on the:

**Obstacles** ➢ We've been plugging away but we can't be efficient until X, Y, and Z are dealt with.

**Risks** ➢ We're getting a lot of complaints from beta testers and we have 400 bugs open. The product *can't be* ready to ship in three days.

**Quality of Testing** ➢ Beta testers have found 30 bugs that we missed. Our regression tests seem ineffective.

**History across projects** ➢ At this milestone on previous projects, we had fewer than 12.3712% of the bugs found still open. We should be at that percentage on this product too.

# *What Are We Measuring?*

- Before we can measure something, we need some sense of what we're measuring. It's easy to come up with "measurements" but we have to understand the relationship between the thing we want to measure and the statistic that we calculate to "measure" it.
- **If we want to <u>measure</u> the "extent of testing", we have to start by understanding what we mean by "extent of testing."**

# *What is measurement?*

- Is measurement really “the assignment of numbers to objects or events according to a clear cut rule”?
  - No, it can’t be. If it was, then many inappropriate rules would do.
- **Measurement is the assignment of numbers to objects or events according to a rule derived from a model or theory.**

# *Surrogate measures*

- "Many of the attributes we wish to study do not have generally agreed methods of measurement. To overcome the lack of a measure for an attribute, some factor which can be measured is used instead. This alternate measure is presumed to be related to the actual attribute with which the study is concerned. These alternate measures are called surrogate measures."
  - Mark Johnson's MA Thesis
- "Surrogates" provide unambiguous assignments of numbers according to rules, but they don't provide an underlying theory or model that relates the measure to the attribute allegedly being measured.

# *Definitions From Other Fields*

"Measurement is the assigning of numbers to individuals in a systematic way as a means of representing properties of individuals."

"Measurement theory is a branch of applied statistics that attempts to describe, categorize, and evaluate the quality of measurements, improve the usefulness, accuracy and meaningfulness of measurements and propose methods for developing new and better measurement instruments."

Allen & Yen, *Introduction to Measurement Theory.*

"Measurement is the process of gathering information from the physical world. . . . Measurement is achieved by means of sensors, also called transducers, and involves the gathering of data and its comparison with an agreed standard. . . . A signal is . . . a symbolic representation of some attribute (or combination of attributes) of the system under observation. . . . Data are the numbers associated with a signal. . . . Information is the representation of attributes of the event or object being measured by a known and agreed symbolism."

"Measurement science is the systematic study and organization of the methods by which information is gathered from the physical world."

Sydenham, Hancock & Thorn, *Introduction to Measurement Science and Engineering.*

# *A Framework for Measurement*

- A measurement involves at least 10 factors:
  - **Attribute to be measured**
    - appropriate scale for the attribute
    - variation of the attribute
  - **Instrument that measures the attribute**
    - scale of the instrument
    - variation of measurements made with this instrument
  - **Relationship between the attribute and the instrument**
  - **Likely side effects of using this instrument to measure this attribute**
  - **Purpose**
  - **Scope**

# *Framework for Measurement*

**Attribute** ➢ Extent of testing – *What does that mean?*

**Instrument** ➢ What should we count? *Lines? Bugs? Test cases? Hours? Temper tantrums?*

**Mechanism** ➢ If we increase our "extent of testing", how will that affect our reading (the measure) on the instrument?

**Side Effect** ➢ If we do something that makes the measured result look better, will that mean that we've actually increased the extent of testing?

**Purpose** ➢ Why are we measuring this? What will we do with the number?

**Scope** ➢ Are we measuring the work of one tester? One team on one project? Is this a cross-project metrics effort? Cross-departmental research?

# Attributes and Instruments

| | |
|---|---|
| Length | Ruler |
| Duration | Stopwatch |
| Speed | Ruler / Stopwatch |
| Sound energy | Sound level meter |
| Loudness | Sound level comparisons by humans |
| Extent of testing??? | ??? |
| ----Product coverage | ??? Count statements / branches tested ??? |
| ----Proportion of bugs that we've found | ??? Count bug reports or graph bug curves??? |

# *Example: Statement/Branch Coverage*

**Attribute** ➢ Extent of testing – ***How much of the product have we tested?***

**Instrument** ➢ Count statements and branches tested

**Mechanism** ➢ If we do more testing and find more bugs, does that mean that our line count will increase? ***Not necessarily. Example—configuration tests.***

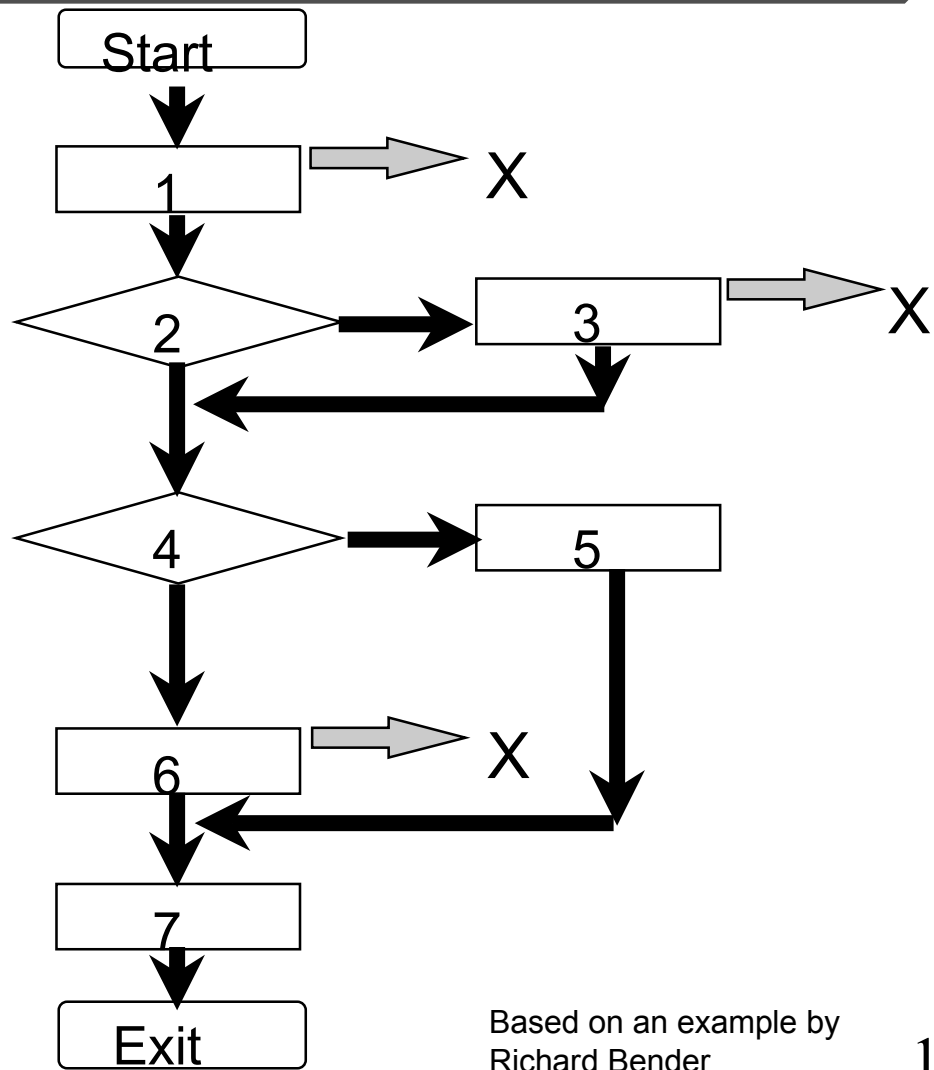
**Side Effect** ➢ If we design our tests to make sure we hit more lines, does that mean we'll have done more extensive testing? ***Maybe in a trivial sense, but we can achieve this with weaker tests that find fewer bugs.***

**Purpose** ➢ Not specified

**Scope** ➢ Not specified

# *Statement / Branch Coverage and Data Flows*

X
means this routine
changes variable X




**1(x) 2 3(x) 4 5 7**
**1(x) 2 4 6(x) 7**
***Now we have 100% branch coverage, but where is 1(x) 7?***
**1(x) 2 4 5 7**



Based on an example by
Richard Bender

# *Statement / Branch Coverage Just Test the Flowchart*

**You're not testing:**

- » data flow
- » tables that determine control flow in table-driven code
- » side effects of interrupts, or interaction with background tasks
- » special values, such as boundary cases. These might or might not be tested.
- » unexpected values (e.g. divide by zero)
- » user interface errors
- » timing-related bugs
- » compliance with contracts, regulations, or other requirements
- » configuration/compatibility failures
- » volume, load, hardware faults

# *Side Effects*

- Without a mechanism that ties changes in the attribute being measured to changes in the reading we get from the instrument, we have a “measure” that is ripe for abuse.

- People will optimize what is tracked. If you track “coverage”, the coverage number will go up, but (as Marick has often pointed out) the quality of testing might go down.

# *Example: Bug Counts*

**Attribute** ➢ Not sure. Maybe we're thinking of percentage found of the total population of bugs in this product.

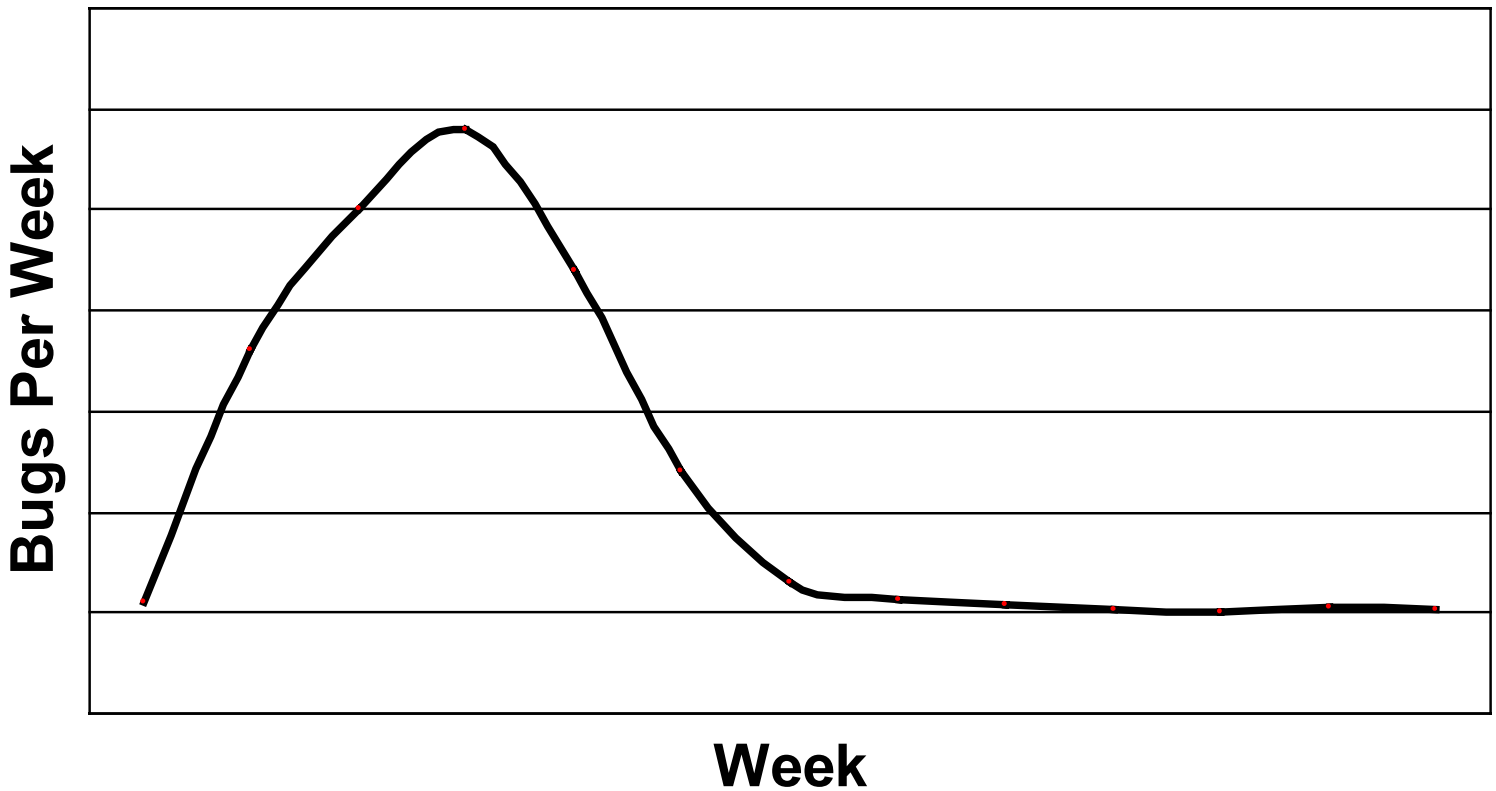
**Instrument** ➢ Bugs found. (Variations: bugs found this week, etc., various numbers based on bug count.)

**Mechanism** ➢ If we increase the extent of testing, does that result in more bug reports? *Not necessarily.*

**Side Effect** ➢ If we change testing to maximize the bug count, does that mean we've achieved more of the testing? *Maybe in a trivial sense, but what if we're finding lots of simple bugs at the expense of testing for a smaller number of harder-to-find serious bugs.*

# The Bug Curve

## What Is This Curve?



# *Example: Bug Curves*

**Attribute** ➢ We have a model of the rate at which new bugs will be found over the life of the project.

**Instrument** ➢ Bugs per week. A key thing that we look at is the agreement between the predictive curve and the actual bug counts.

**Mechanism** ➢ As we increase the extent of testing, will our bug numbers conform to the curve? *Not necessarily. It depends on the bugs that are left in the product.*

**Side Effect** ➢ If we do something that makes the measured result look better, will that mean that we've actually increased the extent of testing? *No, no, no. See side effect discussion.*

# *Side Effects of Bug Curves*

Earlier in testing: (Pressure is to increase bug counts)

- Run tests of features known to be broken or incomplete.
- Run multiple related tests to find multiple related bugs.
- Look for easy bugs in high quantities rather than hard bugs.
- Less emphasis on infrastructure, automation architecture, tools and more emphasis of bug finding. (Short term payoff but long term inefficiency.)

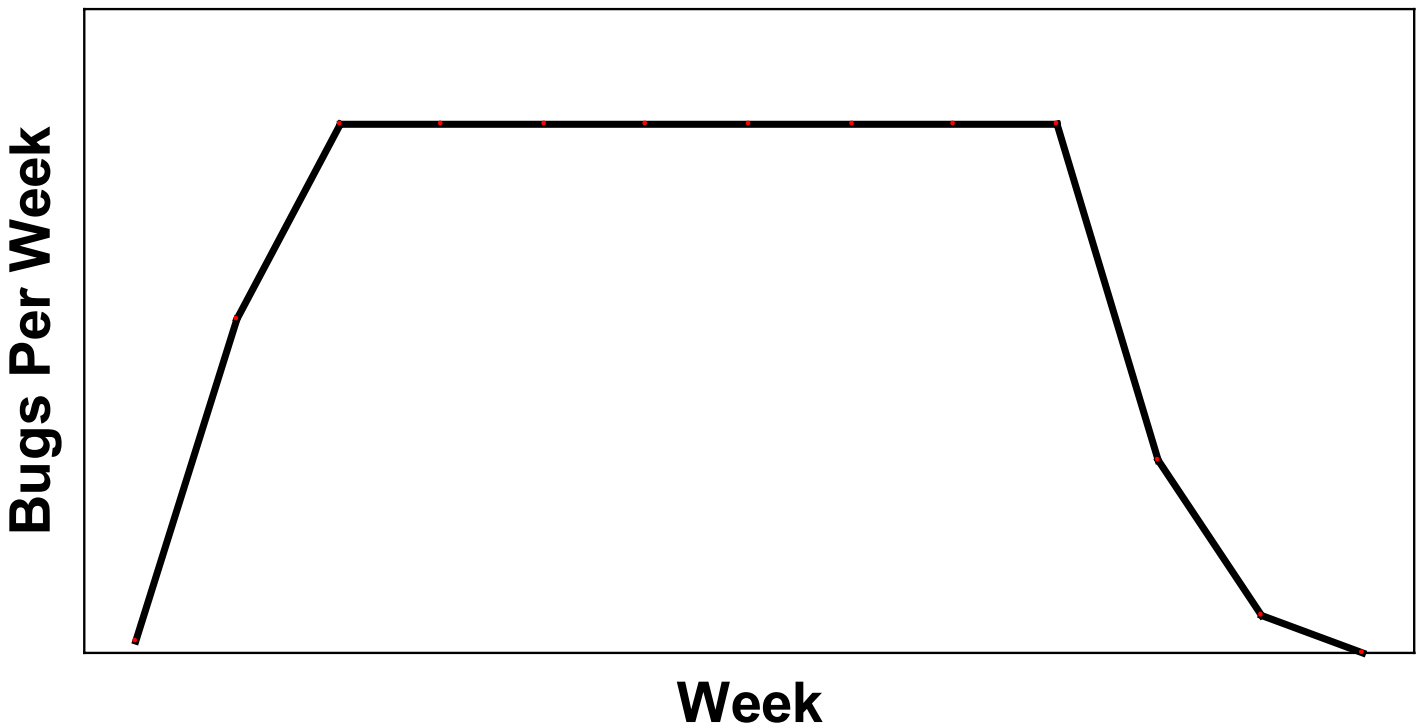
# *Some Side Effects of Bug Curves*

**Later in testing: (Pressure is to decrease new bug rate)**

- Run lots of already-run regression tests
- Don't look as hard for new bugs.
- Shift focus to appraisal, status reporting.
- Unrelated bugs are classified as duplicates
- Related bugs are classed as duplicates (and closed), hiding key data about the symptoms / causes of the problem.
- Bug reporting is postponed until after the measurement checkpoint (milestone). (Some bugs are lost.)
- Bugs are reported informally and kept out of bug tracking system
- Testers get sent to the movies before the measurement checkpoints.
- Programmers ignore bugs they find until/unless the testers report them.
- Bugs are taken personally.
- More bugs are rejected.

# Bug Curve Counterproductive?

## *Shouldn't We Strive For <u>This</u> ?*



# *Having Framed the Problem . . .*

- Do I have a valid, useful measure of the extent of testing?
  - Nope, not yet.
  - The development and validation of a field's measures takes time.
- In the meantime, what do we do?
  - We still have to manage projects, monitor progress, make decisions based on what's going on - - - so ignoring the measurement question is not an option.
  - Let's look at strategies of some seasoned test managers and testers.

# *Thanks to . . .*

Much material is from participants of *Software Test Managers Roundtable* (STMR) and the *Los Altos Workshop on Software Testing* (LAWST).

- STMR 1 (October 3, November 1, 1999) focused on the question, ***How to deal with too many projects and not enough staff?*** Participants: Jim Bampos, Sue Bartlett, Jennifer Brock, David Gelperin, Payson Hall, George Hamblen, Mark Harding, Elisabeth Hendrickson, Kathy Iberle, Herb Isenberg, Jim Kandler, Cem Kaner, Brian Lawrence, Fran McKain, Steve Tolman and Jim Williams.
- STMR 2 (April 30, May 1, 2000) focused on the topic, ***Measuring the extent of testing***. Participants: James Bach, Jim Bampos, Bernie Berger, Jennifer Brock, Dorothy Graham, George Hamblen, Kathy Iberle, Jim Kandler, Cem Kaner, Brian Lawrence, Fran McKain, and Steve Tolman.
- LAWST 8 (December 4-5, 1999) focused on ***Measurement***. Participants: Chris Agruss, James Bach, Jaya Carl, Rochelle Grober, Payson Hall, Elisabeth Hendrickson, Doug Hoffman, III, Bob Johnson, Mark Johnson, Cem Kaner, Brian Lawrence, Brian Marick, Hung Nguyen, Bret Pettichord, Melora Svoboda, and Scott Vernon.

# *Suggestion: Treat "Extent" as a Multidimensional Problem*

- We developed the 8 aspects (or dimensions) of "extent of testing" by looking at the types of measures of extent of testing that we were reporting.
- The accompanying paper reports a few hundred "measures" that fit in the 8 categories
  - product coverage
  - effort
  - Obstacles
  - quality of testing
  - plan / agreement
  - results
  - risks
  - project history
- **ALL of them have problems**

# *Suggestion: Treat "Extent" as a Multidimensional Problem*

- So, look at testing progress reports. Do we see focus on these individual measures?
  - Often, NOT.
  - Instead, we see reports that show a pattern of information of different types, to give the reader / listener a sense of the overall flow of the testing project.
  - Several examples in the paper, here are two of them

# *Project Report / Component Map (Hendrickson)*

Page 1 --- Issues that need management attention
Page 2 --- Component map
Page 3 --- Bug statistics

| Component | Test Type | Tester | Total Tests Planned / Created | Tests Passed / Failed / Blocked | Time Budget | Time Spent | Projected for Next Build | Notes |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| | | | | | | | | |

We see in this report:

- Progress against plan
- Effort
- Obstacles / Risks
- Results

# *Bach's Dashboard*

| Testing Dashboard | | | | Updated 11/1/00 | Build 32 |
|---|---|---|---|---|---|
| Area | Effort | Coverage Planned | Coverage Achieved | Quality | Comments |
| File/edit | High | High | Low | 😐 | 1345, 1410 |
| View | Low | Med | Med | ☺ | |
| Insert | **Blocked** | Med | Low | ☹ | **1621** |
| | | | | | |
| | | | | | |

We see coverage of areas, progress against plan, current effort, key results and risks, and obstacles.

# *Suggested Lessons*

- Bug count metrics cover only a narrow slice of testing progress. There are LOTS of alternatives.
- Simple charts can carry a lot of useful information and lead you to a lot of useful questions.
- Report multidimensional patterns, rather than single measures or a few measures along the same line.
- Think carefully about the potential side effects of your measures.
- Listen critically to reports (case studies) of success with simple metrics. If you can't see the data and don't know how the data were actually collected, you might well be looking at results that were sanitized by working staff (as a side effect of the imposition of the measurement process).

# Software Testing 101

Rick Clements
cle@cypress.com
Cypress Semiconductors
9125 SW Gemini Dr, Suite 200
Beaver ton, OR 97008

## Abstract

This paper covers the basics of software validation. It's aimed at someone who is new to software validation regardless of his or her experience developing software. It covers knowing what you are testing (requirements and configuration control) and how you will test it (selecting the test cases and test procedures). It covers the basics of a number of areas. It isn't a detailed workshop in any of the areas.

The validation of the software is the testing it against the system requirements and requirements derived from the system requirements. Validation of the software occurs after the software has been debugged, unit tested and integrated.

This paper covers what you need before you can design your tests. Unfortunately, since we live in the real world, it will also talk about some of the things you can do when you don't get everything you need.

This paper doesn't cover QA, verification or test automation. Some QA topics (requirement management and configuration control) are discussed only as they are needed to do effective validation. Verification of software is a cost-effective method for finding defects before testing begins and reducing the overall development cycle. Test automation can decrease the people require to run tests. These are all important topics, but they are outside the scope of this paper.

## Biography

Rick Clements is a senior software quality assurance engineer at Cypress Semiconductor where his duties include the common release of software from different divisions. He has 20 years of software experience including built in test software, embedded software design, and testing, and process improvement.

# Glossary

| | |
|---|---|
| Integration testing or Interface testing | The testing of pre-tested modules and hardware to determine that they work together. |
| Black box testing | Testing by looking at the requirements to develop test cases. System level testing is often black box testing. |
| Quality assurance (QA) | The management of quality. The encouragement of best practices. QA is about preventing defects. (The phrase "QA the software" is a misuse of the term QA to mean simply test the software.) |
| System testing | The testing of the entire product to see that it meets its requirements. This occurs after the product has been integrated. |
| Quality control (QC) | Validation and verification of the product. This includes reviews, inspections and testing. |
| Test cases | Specific data used to test the product. |
| Test plan (or validation plan) | The plan describing the general approach, the people required, equipment required and schedule for testing and other validation activities |
| Test procedures | The procedures to follow in testing the product. These may be manual or automated. |
| Unit testing | The process of running small parts of the software. The design teams often hand unit testing. |
| Validation | The process of evaluating software at the end of the software development process to ensure compliance with software requirements.[1] |
| Verification | The process of process of determining whether or not the products of a given phase of the software development cycle fulfil the requirements established during the previous phase.[2] |
| White box testing | Testing by looking at the structure of the program to develop test cases. White box testing often occurs in unit testing. |

# Documentation for Software Testing

The level of documentation depends on the scope of the software, the type of software, possibly government regulations and possibly customer requirements. A large project with mission critical software requires more documentation than small changes to software in office equipment. The amount of documentation needs to fit the task at hand.

The minimum requirements for the documentation are: needs to be sufficient for project planning, allow for planning what cases will be used to test the product features, clear enough for the people doing the testing, and provide a record of what was tested and what the results were.

The appendices at the end of this paper provide templates that can be used. The templates provide a checklist of what needs to be included in the document. They also eliminate the need to create a new format and allow people to focus on producing good tests for the product. The IEEE has collected a set of ANSI/IEEE standards that make good templates. These can be found in *Software Engineering Standards*. Any of these templates should be tailored to fit the needs of your company.

# Requirements

A complete coverage of requirements can fill an entire course. This section covers requirements as they are important to testing.

**Why are requirements important to testing?** If you don't know what the requirements are, how do you know what to test? How do you know if the product is working correctly? The system level validation tests are based on the requirements.

**How are the requirements documented?** The requirements may be in different levels of detail and take different forms. Companies use a requirement specification, functional specification, a requirements database and user's guide to document requirements. If the software is being developed under contract, that contract should spell out the system requirements.

Requirement documents will often start off being general and become more defined as part of the project development. The requirements start out at the level of a data sheet. This allows the company a chance to decide if the project is worth spending the time to develop real requirements. There may be more detail requirements that define specific requirements for modules or what requirements the software has to support the system requirements. However, it's the system level requirements that are important for the validation of the system.

The requirements may take the form of a requirement specification, a functional specification, a database or be managed by a requirements tool. Each form has its own benefits. However, the most important things about requirements are:

- They exist
  - System level requirements are required for system level tests
  - Interface specifications are required for integration testing
- They are unambiguous and testable
- They cover requirements from all of the customers not just the final customer
- They are under some form of configuration control

Assuming you have requirements in some form they need to be unambiguous and testable. A requirements document can be produced and everyone begins developing software and tests. It's not until testing begins that it's discovered that not everyone had the same understanding of what the requirements meant. If you are reviewing a requirements document, look at each

requirement and ask, “How will I test it?” If you can’t answer that question, the requirement isn’t testable.

It’s important to have all the requirements from all of the stake holders included. For example, the need for manufacturablity and testability may result in a remote procedure call interface (a method for external software to access internal features in the software under test). This interface needs to be designed into the product up front. It directly affects the level of testing required and the level of testing that’s possible. If only the final customer is focused on, these requirements will probably be missed.

Requirements will change during the product cycle. Everyone — including testing — needs to know there is a proposed change to the requirement. If management knows the full cost of the change in time and resources, they can make an informed choice about whether to make the change or not. Otherwise, they may ask a developer and be told it’s a 30 minute change. They may never find out until the change is made that it require two more man weeks of test preparation and one more week of testing time. Once the change is made, everyone needs to know about it. This may be a requirements data base, or e-mail telling people that a change was made and to check “The Spec” on the network. “The Spec” should have a version or date to help keep everyone current.

**What if you don’t have any requirements?** How do the developers know what they are building? If they “just know”, it’s time to be concerned. If you can’t convince anyone that the requirements need to be recorded, you have no choice but to find them yourself. Find out what the manager, the developers, the marketing department (or other customer representative) and any domain experts you have in the company think the requirements are. Once you find out what they are, write them down in some fashion.

Let people know that is what you are testing to. It’s better to have the discussion now then after you have written your tests, made the first pass though your tests, you are arguing about what’s a bug and it’s three weeks until the trade show where the product will be unveiled. By this time, fixing the software (and possibly the hardware) is difficult and expensive. With a large amount of changes to make and a deadline approaching, there is a strong temptation put a Band-Aid on a major problem and ship it. Many projects fail because you can't test quality in at the end of the project.

If you are creating a requirements document in an environment where none has existed before, start simple, build support and experience. As you show the benefit of requirements, additional detail and tools can be added. It will allow you to determine which features are most beneficial in your environment. There are tools that help track requirements and requirement changes. However, they won’t provide any benefit if the practices don’t exist to make use of them.

# Configuration Control

Configuration control is a part of quality assurance that directly affect testing. Good configuration control will prevent the following problems:

- Testing one version of the product and shipping an other

- Modules of the software randomly reverting to an old revision
- Debug code accidentally being linked in
- Chasing a problem that only occurs on one system in the test lab only to find it has a different version of software.
- The ability to re-create the software and tests of a previous version. (It may become necessary to fix a bug in an old version of software.)

What is the minimum level of configuration control you need? Before you start testing, the software should be checked into the revision control system. Then, it should be checked out onto a clean directory (a clean system is better) then built. That final candidate needs to have a unique version number that can be displayed by the system in an initial splash screen or an about box. The software needs to be stored in a secure location. If after testing, that final candidate is released, that binary file needs to be released. If your software will be run in the Windows environment, the install procedures are complex enough that this or an other person needs to create the install scripts.

If the software is rebuilt after testing, there is a chance that new bugs will be introduced. A new file may have been checked into the revision control system and accidentally included. Some people want to change the version number after the software has been tested. Just changing the version number (specifically making it longer) has introduced errors. Use an extra digit for the build or an additional build number, but don't change the software once it has been tested.

A simple system needs to exist for creating the version and build number. Some systems use "*majorNumber.minorNumber.buildNumber*" or "*majorNumber.minorNumber* build *buildNumber*". Other systems use the build date as the version number. It's preferable to have a single number. However, marketing may require their own number for advertising reasons. The number just needs to be:

- Simple to generate
- Unique for each build
- Readily visible and validated for correctness

The application code is under configuration control, but what about the test procedures, test scripts and test programs? If the tests are changed for a new feature, then that feature is found to be buggy to ship on schedule, can the earlier tests be recovered? If a fix needs to be added to an older version of software (or a feature added to old software for a big customer), can the earlier tests be recovered and updated. The fact that this will never happen is of no consequence when it does. In addition to placing the tests under revision control, mark them with the same label as that version of software.

# Test Plan

The test plan needs to cover what areas you will test and any areas you won't test. This gives the developers and manager a chance to assess the level of testing. The developer's feedback is important because a feature that appears to be a minor variation of a feature that's being tested

toughly may turn out to be a completely different algorithm. The manager needs to know where the testing will be focused so s/he can manage the risks.

Different features require different approaches. Automated tests require more resources up front to create the tests. Manual tests require more people at the end of the project to run the tests. Can all the tests be functional tests? Does performance need to be measured also? Does there need to be some form of stress testing? Once the type of the tests are known, the type of people and when they are needed can be scheduled.

It's important to identify any areas that won't be tested. The project manager needs to know what the risks are of not doing the testing. For example, the test plan may say, "the temperature tests are not being run because there aren't any changes to temperature sensing code so there is little risk in not running them and they require several days of time in a thermal chamber." The design engineers may know of a change that affects that area. Continuing the example, the review of the plan may turn up the fact that a change to non-uniformity correction algorithm will make it dependent on the ambient temperature.

If the tests require hooks into the software or hardware, identifying them in the planning stage allows the hooks to be designed in. A hook that is easy to provide when the product is being designed may be difficult to provide later. This is the time to balance test coverage against the cost of providing the required hooks.

Schedule planning needs to take into account the following needs: equipment than need to be built, products to be purchased and tools that needs to be created.

# Test Cases

Bugs have been compared to land mines. Testing clears a path though those land mines. Test cases need to be chosen well so the user will be likely to stay on the path that has been cleared for them. It's not possible to test everything[3].

**What will the users do most often?** Testing needs to focus on what the user will use. "If there's in the code and no one finds it, is it a bug?" If we can't test everything, we want to cover what the user will use We can't ignore lesser-used areas. I once heard someone who worked on a tape backup system say they focused most of their testing on writing the tapes because that's what the customer does most often. I think about that statement every time I can't get a file restored. While we heavily test the areas the user will most use often, we can't ignore the others.

Common system usage will help you to catch an entire case of errors. For example, a printer might correctly detect an empty paper tray when it's empty. It may correctly detect the tray has been refilled and continue printing. What happens if a non-empty tray is removed? Why would someone ever do that? The user may start a long print job then check the paper tray before going off to lunch.

**What is the most serious if it fails?** If something is done incorrectly, could it cause a major financial impact or hurt someone? Multiplying the subtotal by the tax rate then using that value

for the total instead of adding it to the subtotal will have a big impact on the bottom line. If the product has moving parts, does the software stop all the motion when cover is opened?

**Test boundary conditions.** If you have the requirement to display a temperature, what are the minimum and maximum temperatures? Select an invalid value one below the minimum, the minimum value, minus one, zero, one, a typical value, the maximum value and one above the maximum value. These cases check the handling of valid and invalid values. They check the both the boundaries created by the requirements and boundaries that require the software to do something different.

**What are the system interfaces?** Anywhere there is an interface in the system, data has to be transferred correctly and unexpected values must be handled. It may be between the user and the product, your product and a different product or components within your own product.

To the user, the user interface is the product. It is how the user interacts with the product. Because errors in the user interface will be noticed every time the user uses the product, they need to be tested.

The user interface's error handling needs to be well tested for two reasons. A keyboard has been referred to as a device for entering errors into the computer. Second, error handling is often missed by developers. The developer is focusing on making the system work. They often miss the handling of unexpected input.

If your program has to exchange data with another program, there is possibility for error. If your product receives data from another program, what is the correct response if the data doesn't conform to the standard? Unfortunately, if the other program is popular enough with users, your product may need to work with it anyway. This may result in a set of tests to see if your product works with popular programs not just that it works with proper input.

The Mars Climate Orbiter spacecraft was lost because of interface errors between different modules. One module was using distance measurements in miles while the other was using kilometers[4]. The interfaces between different modules need to be verified. This is more important if products can be configured with different modules using this common interface.

**Where have other errors been found?** There are likely to be errors that have occurred in the past. If there are errors of a specific type, there are likely to be more errors of the same type.

If do what you've always done, you will get what you've always gotten. Unless the design team has done some sort of root cause analysis, they are likely to continue with the same procedures that produced the original errors. Root cause analysis is outside of the scope of this paper. It's a good idea to prepare tests for the problems you see often. For example, if one of the text files on the UNIX disk has PC carriage returns and one of the text files on the Macintosh disk has UNIX carriage returns, check the PC disk for the proper carriage returns.

Developers often don't enter defects into the bug tracking system. A useful way of determining which modules have the most errors is to look at the configuration control system. These systems track the number of versions that have been checked in and the number of changes in each version.

Complex modules will tend to have more errors. Talk to the developers. Find out what modules were most challenging. Another indication of complexity is size. The longer modules tend to be more complex.

**Usability -** The testers are the first people after the developers to see the software. They are in the best position to advocate for better usability.

The user interface may highlight features the developers are proud of. This may make the product harder to use. For example, a when inserting a picture into a popular word processor, the picture floats on top of the text by default. Advanced placement features can be use to flow the text around the picture. However, most users don't use these features. If the picture was placed simply on the page, it would be easier for most users.

The interface may be easy to use if you know the internal structure and possibly the internal state of the software. For example, the original caller ID feature had a method to toggle sending of the caller ID. It had no way to tell what the current state was.

The test engineer can provide valuable feedback on the usability. However, it does require that you develop a feeling for the typical user of your system. In many cases, you won't be a typical user.

This is also an area where you may find the greatest resistance from the design engineers. It is an interface that makes sense to them. You may need to enlist the aid of your marketing or customer representative. They are the people who should have the final word about how the typical user will react to the interface.

**What is unique to your product's environment?** Each environment has a unique set of problems that must be dealt with.

With a web-based application, you have to deal with browsers from different companies, browsers on different platforms and different versions of those browsers. The surveys supporting a page and its links may be running from different operating systems. The combinations can explode very quickly. You need to determine what is important to the users of your product.

If you are dealing with a data based application, you likely have an existing database the program needs to be compatible with. Getting a recent copy of the database before you go live can prevent a number of unexpected errors.

If you are dealing with embedded software, the system needs to be stress tested. Does the control loop software compensate for the hardware being in extreme environmental conditions? The system will be more sluggish at -30$^{o}$C. Does the system correctly handle stimulus from all of the inputs while the system is busy doing something else?

Does the software correctly handle a failure of the system hardware? Depending on your environment, the software may shut the system down to prevent further damage or it may warn the user but operate to complete system failure. For this type of testing it's desirable to be able to simulate input values.

Figure (1) shows a fault may be simulated by replacing the sensor, by having a hardware switch that allows simulating the value or by a remote procedure call that instructs the driver to return a simulated value. If you need to simulate values, these are requirements you need to get into the system early in design.

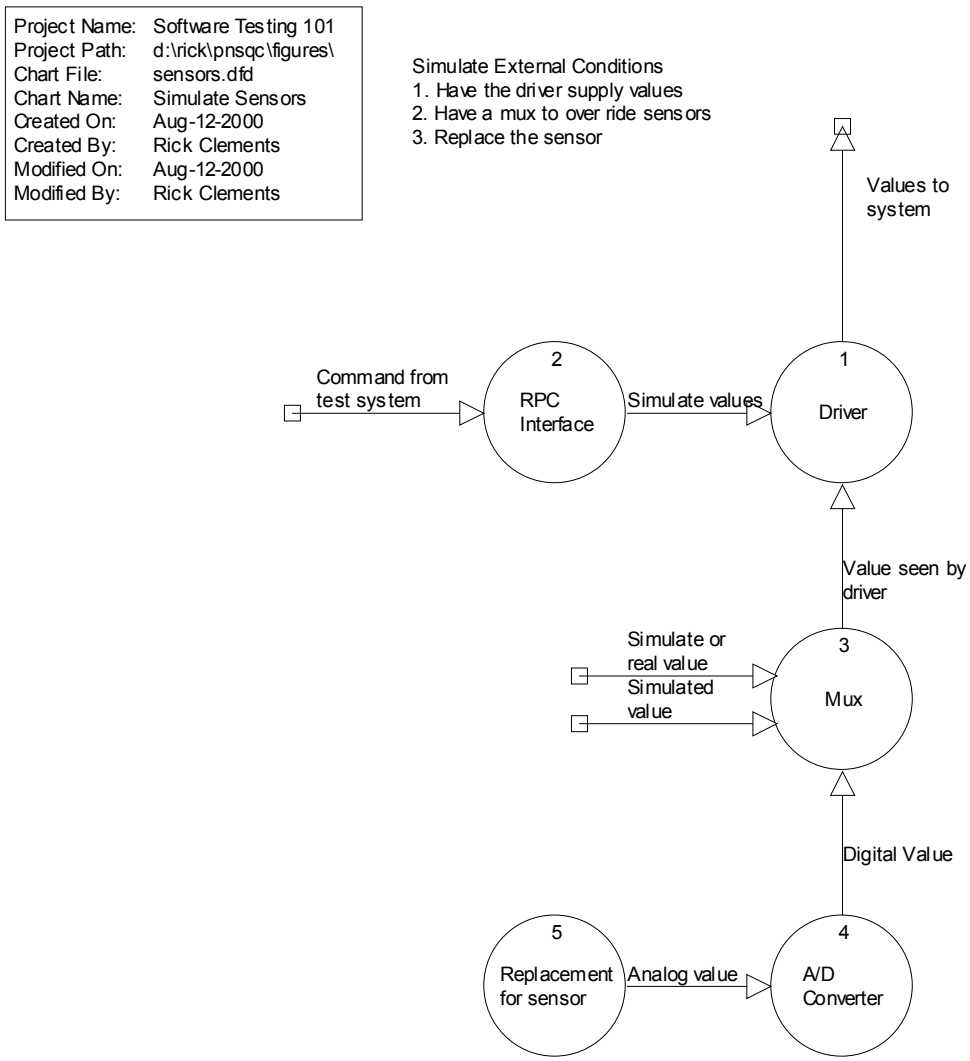



Figure (1) - Simulating Fault Conditions

If the product is running on a multi-use computer instead of an embedded system, there are different failures that need to be tested. Does the software correctly handle a shortage of disk space or memory? Does it correctly handle an inaccessible server? If you need tools to create these conditions, resources needed to be considered in the decision to test those cases. If you need to buy or rent equipment, your manager is needs to know the risks if this testing isn't done and the cost of the equipment.

**Why document the test cases?** The test cases are what will be tested. It's beneficial to review the test cases before starting on the test. Reviewing the test cases first allow the focus to be adjusted before the work of creating the test procedures and test scripts. It's useful to include at least one other test engineer and one design engineer on the review team. The test engineer will

tend to catch common errors and standard patterns that are missed by the test cases. The design engineer will tend to catch internal limits and conditions that are missed by the test cases.

# Test Procedures

**How the test procedures are documented depend on the environment.** The type of software, the experience of the testers, the tools being used, customer requirements and government requirements all affect the way tests are documented.

Performance tests will go into detail on how tests are set up and measurements taken. Functional tests may be a list of test cases. Tests for combinations of features may give instructions for choosing the combination to test and provide a location for recording what was tested. On the next test pass, a different combination of features may be tested to increase coverage.

The contract or government regulations may call out the detail needed to document tests.

The amount of detail that's required for a beginning test operator will frustrate an experienced tester or someone with domain experience. Too little detail will cause the inexperienced tester to continually ask you questions. Experienced people don't need a lot of detail on how to setup the test, they need to know what to test.

**Can the testing be automated?** Test automation is often used in testing. It's useful for tests that will be run often. They excel in testing detail that is mind numbing for a person.

If automation is being used, the test tools and test scripts need to be solid before the testing begins. If there isn't any confidence in the tests, the task of isolating failures is magnified. If test software and hardware need to be created, this is a development task of its own. It needs to be planned, managed and tested just like the application software. It needs to be well designed for the same reasons the product software does. It needs to be given adequate time and resources. If there isn't time to do it right, it won't be complete or reliable.

If the tests will be run in the Windows, UNIX or Internet environments, there are a number of tools available. If the tests are for the embedded environment, there are few tools and they need to be customized to work with the system.

Regardless of the environment, start simple. Simple may mean doing it by hand the first time. As you start to automate tests, you will gain experience at what is important. For example, automated tests can generate a lot of data. You need an automated way of checking and reporting the results.

# Bug Tracking and the Ship Decision

Bug tracking is best done with a bug tracking program. Programs don't forget about bugs. They allow you to sort the bugs by assigned engineer, importantance and other useful fields. There are

a number of bug tracking programs. These programs will need to be customized for your company.

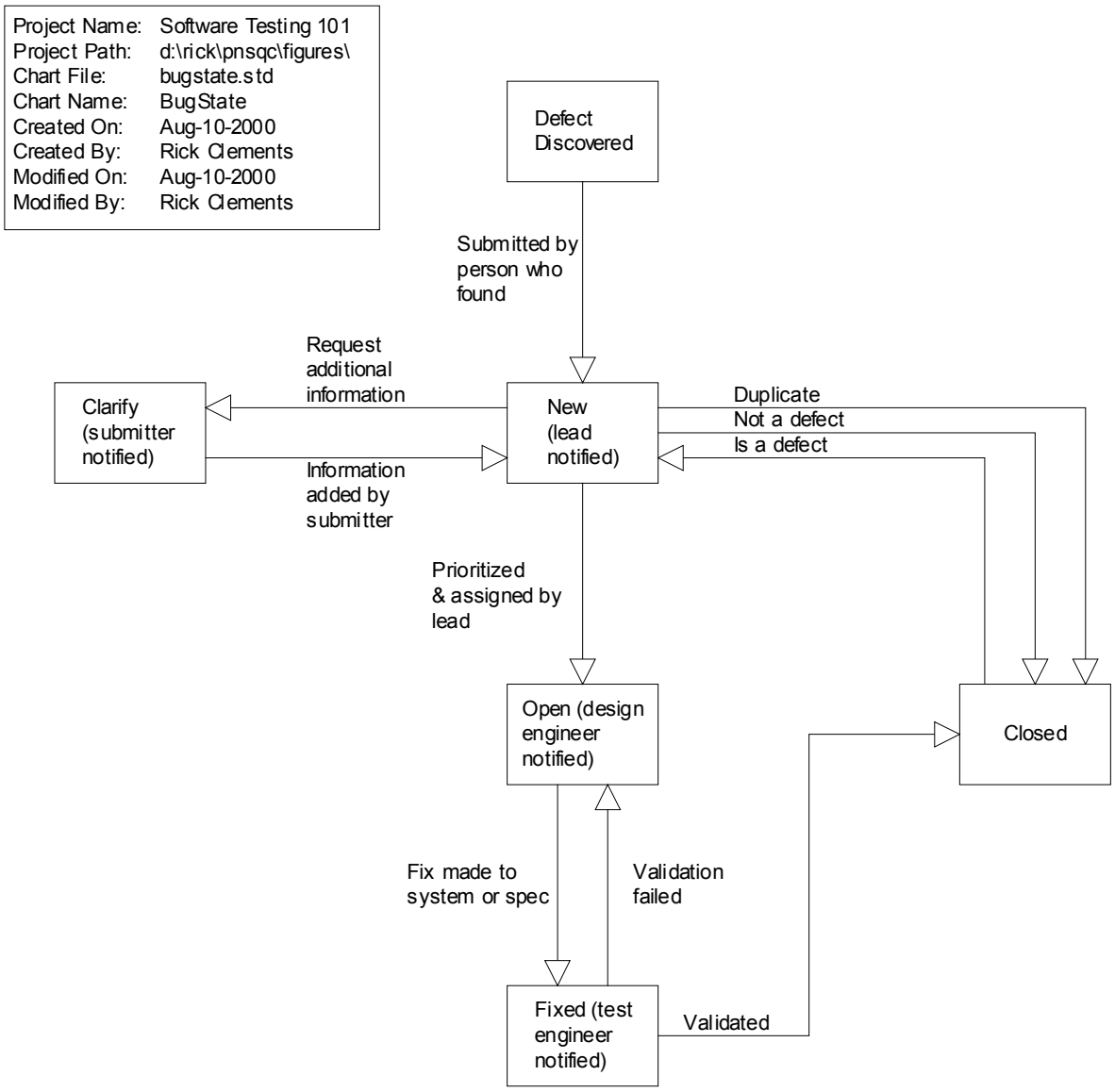



Figure (2) – Defect States

Figure (2) shows the states in a typical bug tracking system. When a bug is first found, the person who found it creates a ticket. This could be a test engineer, a design engineer, a manufacturing engineer using a preliminary version to write build procedures or a marketing representative. The record usually includes a one-line title and a description. The ticket is sent to a reviewer. This is often the lead design engineer. If more information is needed, the ticket is sent back to the submitter. Otherwise, it's give an initial priority and assigned to a design or test engineer. The assignment of who will do isolation will depend on who has the best tools and the current workloads.

A well-written bug report is important to provide smooth communication between the test and design engineers. Defects in different areas of the software require different amounts of information. It's important to understand what information is needed to be reported with the different defects. The description needs to include the environment, the configuration of the system, any symptoms, what was being done at the time of the failure and if the defect is repeatable.

Figure (3) shows an example defect report. Some of the details are include in specific fields. The rest of the information is in the description field.

> **Title:** Page fault when closing the about box
>
> **Description:** 3 of 4 times program Y crashed with a page fault when launched from program X when the about box was closed. The program crashed 0 of 3 times when launched from the start menu.
>
> Sequence:
> Launch from program X
> Open the about box from the "Help" menu
> Close the about box
>
> Impact: If this happens after the user has been editing, the user will lose edits.
>
> **Severity:** Critical
>
> **Version:** 6.0 build 3
>
> **System:** All Windows
>
> **State:** New

Figure (3) - Example Defect Report

On larger teams, it's useful to include a work log. This allows the work on the bug to be documented even if several people work on it. A test engineer may do some initial isolation. It may then be assigned to a software design engineer to fix. It may be determined that the problem is actually an electrical problem and assigned to an electrical design engineer to fix. It may be given back to the software engineer to work around the bug in the hardware.

After the bug has been "fixed", it's given to a test engineer to validate. If it is fixed, the bug is closed out. If it isn't, it's sent back to the design engineer to be worked on again.

**Can we ship it yet?** At the end of the project, the lead test engineer, the lead design engineer, the project manager, and the marketing or customer representative need to go though the open bugs. There needs to be a decision as to which bugs will be fixed and which ones are minor or enhancements. The bugs should have an initial priority already. It's useful to sort the list by priority so the most important bugs are at the top of the list. This way the important bugs get the most attention.

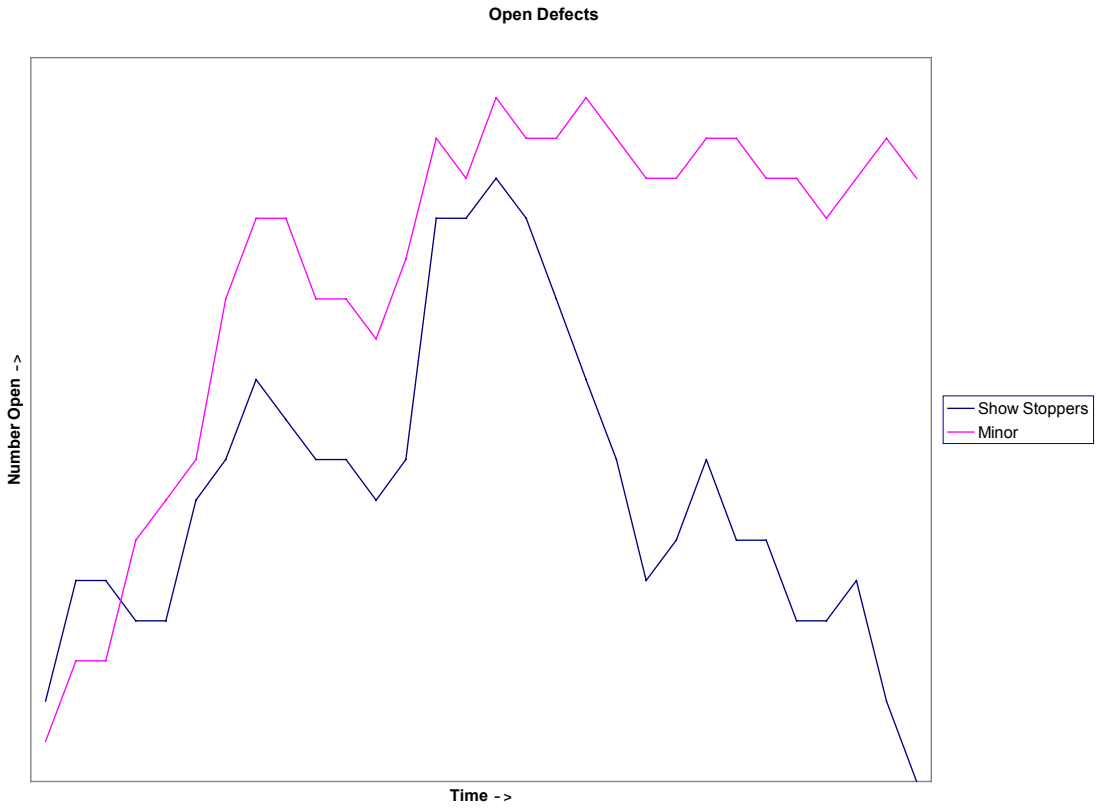



Figure (4) - Open Defects Over Time

Tom DeMarco said, if you can only collect one metric it should be defect count[5]. With not much more work, the defect rate can be graphed against time. This is simply the bugs that have been found but not fixed over the life of the project as shown in figure (4). Two lines can be plotted the showstoppers and the non-show stoppers. It's the show stopper line that will be used to make decisions. However, if there are too many non-show stoppers, there is reason to question the readiness for release. When the average slope of the show stopper line is positive, you aren't approaching the point where the software is ready to ship.

**Is customer service ready for the new release?** Companies have different ways of preparing their customer service and help desks for a new release. As a test engineer, you have some very valuable information for them. You have a list of known bugs. The list should be summarized. They don't need all of the symptoms and guesses that were added to the bug as it was isolated. A list of possible workarounds is useful. A good "Bugs and Workarounds" list will give them the benefit of the weeks or months the test engineers have spent learning the way the software works.

When assessing the risk of not fixing a defect, you need to know how likely the user is to come across it and what is the effect is when they see it. If you can express the chance of failure and the cost of failure then Risk = ChanceOfFailure * CostOfFailure. Many times the chance of failure and cost of failure are only measured in relative terms. A defect that can cause a loss of data, silently report incorrect results or create a dangerous condition has a high risk even though

it might not be seen often. A defect that requires the user to access a feature though a menu instead of a short cut would only have a large risk if users will experience it frequently.

# Test Reports

The test report is structured as a technical report. It starts with an introduction to provide a description of what the report covers. The next section describes recommendations and high level results. The rest of the report provides the supporting data for the recommendations and more detail regarding the results.

The first section details what has been tested, what the results were and if any part of the test plan couldn't be carried out for some reason. The report should include how many cases have major defects, how many minor defects, any cases that couldn't be run and how many total cases. There may also be a section for improvement to the tests.

The amount of detail depends on the audience for the test report. If the customer will see a copy of the test report, it should contain information that will describe proprietary algorithms. If a government agency will receive the test report, there may be requirements for specific levels of detail.

A test report should cover one subject for a single audience. For example, automatic functional tests may be reported separately form manual tests based on common user sequences. If a government agency is only interested in one aspect of the product, other areas of testing shouldn't be included in the same report.

# Appendices

These appendices include an example set of templates for common documents. Many of these documents started from the IEEE standards and have be tailored one or more times. If you don't have a standard template for these documents, they can serve as a starting point.

The example text in the templates is *italicized.*

## Testing Strategy Worksheet

This document is used for initially scoping the testing effort. It is a one page document that's done before the test plan. This document is modified from a worksheet presented by Randall Rice of Rice Consulting at the PNSQC workshop on "Effective Web-Based Testing Techniques".

| | |
|---|---|
| **Application Name:**<br>*Gimbal Software* | **Author:** *Bob Smith*<br>**Date**: *1 Jan. 2000* |
| **Type of Software:**<br>*Object oriented C++ code running on a CPU*<br>*C code running on a DSP* | **Development Methodology:**<br>*Waterfall* |
| **Scope of Testing:**<br>*Selection of features*<br>*Stability of gimbal* | |
| **Critical Success Factors:**<br>*Correct control of payloads*<br>*Image stability when subjected to vibration*<br>*Correct movement under temperature extremes* | **Tradeoffs:**<br>*Schedule - yes*<br>*Scope - no*<br>*Cost - yes*<br>*Performance - no*<br>*Quality - no* |
| **Testers:**<br>*Image stability - designers*<br>*Control of payloads - testers*<br>*Environmental extremes - testers*<br>*Flight Test - testers, designers* | **Timelines:**<br>*Requirements - Jan. 31*<br>*Software design - Feb. 28*<br>*Test cases - Feb. 28*<br>*Coding complete - Mar. 31*<br>*Hardware available - Mar. 31*<br>*Debug complete - Apr. 30*<br>*Test procedures - Apr. 30*<br>*Trade Show - 1 May*<br>*Testing complete - May 30* |
| **Risks:**<br>*Prototype shown at trade show before testing is complete.*<br>*The type of hand controller may affect the user's ability to control the system.* | |
| **Constraints:**<br>*Lack of customer input to provide feedback for feel of the controls. Hardware won't be available for integration testing before system tests begin.* | **Assumptions**:<br>*Type of hand controller won*'t affect system performance. |
| **Test Approach:**<br>*Debugging and unit testing will occur before formal testing. Formal testing will include both integration testing and system testing.* | |
| **Tools:**<br>*Automated scripts to run shaker profiles.*<br>*Automated scripts to simulate the hand controller.* | **Deliverables:**<br>*Test plan*<br>*Test cases & procedures*<br>*Bug list*<br>*Test Reports* |

# Test Plan

## 1. Identifier

*XYZ_TPL*

## 2. Introduction

### 2.1 Purpose

*This plan covers the software and system testing done on the XYZ project.*

### 2.2 Scope

*This plan covers the software and system functionality. It doesn't performance testing or flight testing of the system.*

## 3. Test Items

- *The software in the central electronic unit*
- *The software in the hand controller*

## 4. Features to be Tested

- *Commands sent by the hand controller*
- *Menu provided by the central electronics unit*
- *Selection of modes and features in the gimbal by the central electronics unit*

## 5. Features Not to be Tested

- *The software in the gimbal is being developed as a separate project. This software will be used to test the rest of the system software but not explicitly tested.*

## 6. Approach

### 6.1 Design and Code Reviews

- *Design and code reviews will be conducted on critical software for early detection of defects.*

### 6.2 Functional Testing

- *The selection of features by the hand controller*
- *The ability to select functions in the gimbal*
- *The menuing system in the central electronics unit*

### 6.3 Exception Testing

- *Invalid user input*
- *Error conditions reported by the gimbal*

### 6.4 Regression Testing

- *Existing tests for the previous product*

### 6.5 Qualification Testing

- *Safety related warnings will be tested by creating each condition and verifying the appropriate warning is displayed.*

## 7. Item pass/fail criteria

*Completion of the Spectrum Core CEU validation process as outlined in this test plan, acceptance of documented testing issues, and release notes entries will constitute a condition for the Spectrum Core CEU to pass.*

## 8. Suspension criteria and Resumption requirements

*Suspension of testing may exhibit best use of personal if testing results lead to a decision to make significant software correction and these corrections with a subsequent recompilation of the software would require retesting of features still to be tested.*

*When a new release of software is available and resumption of testing is indicated, the appropriate testing approach(s) will be used.*

## 9. Test Deliverables

- Test Plan
- Test Cases
- Test Procedures
- Test Scripts
- Test Logs
- Defect List
- Test Report

## 10. Testing Tasks

*List of tasks and who will be conducting the tests.*

## 11. Equipment Needs

- *A thermal chamber for testing over temperature and ice over conditions*
- *Both types of gimbals*

## 12. Schedule

*What resources are needed and when are they needed.*

**13. Risks / Contingencies**

| Risk | Contingency |
|---|---|
| *Because the memory access for the processor is controlled by the FPGA, it's possible to starve the processor if the FPGA becomes busy.* | *Load testing will be conducted where both the FPGA and processor are heavily loaded.* |

## Test Cases and Procedures

This template combines the test cases and procedures. This keeps them together and makes the easier to update. The test cases are reviewed before completing the later sections.

**1. Introduction**

**1.1 Scope**

*This document covers the user interface.*

**1.2 Purpose**

*Verify the proper function of the buttons and icons.*

**1.3 Overview**

**1.4 Applicable Documents**

- *The requirements document*
- *The test plan*

**1.5 Definitions & Acronyms**

*TBD - Too Bloody Difficult*
*TLA* - Three *Letter Acronym*

**2. Features and Test Cases**

*Temperature Icons*

- Simulate Room temperature
- Simulate 0.1°C below warning level
- Simulate 0.1°C above warning level
- Simulate 0.1°C below danger level
- Simulate 0.1°C above danger level

### 3. Features Not To Be Tested

Identify specific system features that will not be exercised in this test. Identify the reasons the features will not be tested and any potential risks or impacts this might have on the release. This section is more detailed than the test plan. The test plan will list entire requirements that aren't tested. This section lists cases that won't be tested for requirements that are partially tested.

### 4. Dependencies

Identify all dependencies that are required for testing to begin the tests with in this specification. The test plan will be the union of all the test case specifications it references.

- *Production level unit*
- *PC*
- *Serial connector*

### 5. Testing Instructions/Approach

Describe the general process for testing. Include instructions for recording test results and reporting test failures. This section is focused toward what the test operator needs to know as opposed to the test plan that is more of a test architecture.

### 6. Testing Procedures

| Procedure | Expected Results | Actual Results |
|---|---|---|
| *Run script to simulate temperature. Set the temperature to 60.1°C.* | *Yellow thermometer on the bottom of the screen. "Temperature Warning" displayed beside the thermometer.* | |

## Test Report

### 1. Introduction

#### 1.1 Purpose and Scope

The Test Results Report summarizes the software testing performed during the development of a release. It also points to the storage location of the test log files, input data, scripts and pertinent documents used to design and conduct the testing. It explains where the actual testing deviated from the planned activity, why this was necessary and the risk involved.

### 1.2 Applicable Documents

### 1.3 Definitions & Acronyms

*BOB - Best of the best*
*WOW - Worst of the worst*

## 2. Executive Summary

This section contains a recommendation for the ship decision. If the recommendation is to not ship, the list of open high severity problems, or problems identified by the project team as high priority for future attention.

## 3. Testing Synopsis

Describe the history of testing for this release. Include:

- Software version number(s),
- Version label used for software build.
- Specific information about the hardware (revision level, serial number) used for system testing.

## 4. New System Software Problems

Bug number, severity and single line description of the defect sorted by severity.

## 5. System Hardware Discrepancies

Include any hardware problems discovered during testing. Even if the scope of the testing doesn't include hardware, there may have been hardware issues that impacted testing.

## 6. Test Cases Not Run

List which test cases were not run (if any), the reason the test case was not run and the risk assessment.

## 7. Test Case Issues

### 7.1 Errors in Existing Cases

List any issues with the test cases discovered during testing. Some examples of test case issues are the test case turns out not to be meaningful, or a typo in the spec shows the expected outputs as X, Y when Y, X are returned, etc. Refer to the test case identifier in which the problem exists**.**

### 7.2 New Test Cases Needed

List areas of functionality that were found to be either not tested or inadequately tested in the current Test Case Specification. Be sure to include the items from Section 4, New System Software Problems.

## 8. Beta Testing

This is a section that may or may not be needed. If a formal beta testing period is defined and exercised before release of the product it may be included. If the release is for the purpose of an extended beta test in the field then a separate document to record the results better serves the purpose.

---

[1] The *IEEE Standard Glossary of Software Engineering Terminology* ANSI/IEEE Std 729-1983.

[2] The *IEEE Standard Glossary of Software Engineering Terminology* ANSI/IEEE Std 729-1983.

[3] "Impossibility of Complete Testing" by Cem Kaner located at http://www.kaner.com/imposs.htm.

[4] "NASA Says Human Error Caused Loss Of Mars Craft" by Michael Miller in the November 1999 edition of the *Testing Technique News* located at http://www.soft.com/News/TTN-Online/ttnnov99.html.

[5] *Why Does Software Cost So Much?* Tom DeMarco, 1995 Dorset House on page 15.

# When Will We Be Done Testing?

# Software Defect Arrival Modeling Using the Weibull Distribution

Erik Simmons
Intel Corporation
JF1-46
2111 NE 25th Ave.
Hillsboro, OR 97214-5961
erik.simmons@intel.com

Version 1.3, 8/7/00

**Prepared for the 18th Annual Pacific Northwest Software Quality Conference**

**Key Words**: Software Defect Arrival; Weibull; Software Testing

## Author Biography

Erik Simmons has 15 years experience in multiple aspects of software and quality engineering. He holds a Masters degree in mathematical modeling and a Bachelors degree in applied mathematics from Humboldt State University in California. Erik currently works as Platform Quality Engineer in the Platform Quality Methods group, part of the Corporate Quality Network at Intel Corporation.

## Abstract

One of the most common yet vexing questions asked of Software Quality Assurance managers and testers is "When will we be done testing?" Product engineering and marketing groups have a vested interest in knowing when the software under test will be at an acceptable level of quality. While this question is not at all easy to answer, modeling the arrival of defects during testing can provide clues such as predictions of when a given percentage of the estimated total defects will be found, or the time at which the rate of newly arriving defects will be below a given threshold. The Weibull distribution serves as an excellent model for software defect arrival. Three case studies based on actual projects are provided.

## Introduction

Software Quality Assurance managers, testers, and others involved in software system testing are often confronted with questions from product engineering and marketing groups concerning when the software will exit the test process at an acceptable level of quality. This information is essential to planning product launches, staffing and training support teams, and similar activities. An accurate prediction of the defect arrival rate over time is one part of the information required to get beyond the obvious answer "when enough of the defects are gone". Of course, your answer might be "when the release date arrives", but that's another problem.

The Weibull distribution was created in 1937 by Waloddi Weibull. Since the 1950s, it has been used to model phenomena in a wide range of disciplines. Weibull analysis has a large following today, where it is used for modeling hardware failures, analyzing radar clutter, predicting warranty and support costs, forecasting spare parts levels, and many other purposes [Abernethy98]. The Weibull distribution has tremendous flexibility. It can accommodate decreasing, constant, and increasing failure rates. Few if any other models can match its combination of breadth and simplicity.

The Weibull distribution has been shown to describe the defect arrival pattern of software projects [Kan95], [Lyu95], [Putnam92]. The process for modeling defect arrival with the Weibull distribution involves several steps. First, an estimate of the total number of defects in the software must be made. This estimate is then used to determine the cumulative proportion of defects that arrive each time period. Estimates of the parameters of the Weibull distribution are then obtained from a regression line fit to a mathematical transformation of this data.

In this paper, the parameters of the Weibull distribution were estimated using Microsoft* Excel*, though any spreadsheet or statistical software package could be used. The first parameter of the distribution is given by the slope of the fitted line, and the second is calculated with a simple formula. Once these parameter estimates are known, inferences about the future defect arrival patterns can be made.

The projected defect arrival rate can be used to predict the time at which a given percentage of the estimated total defects will be located, or when the number of newly discovered defects per time period is likely to be below a given threshold. These predictions can be used to validate the viability of a release date or to track progress towards a given quality goal. For example, if a release criterion for a software system states that 95% of the total estimated defects must be removed, the analysis described here can be used to predict the date at which that milestone will be met, given the project's current staffing and effort.

## The Weibull Distribution

The form of the Weibull distribution used for this analysis is the two-parameter Weibull. The cumulative distribution function (CDF) of the two-parameter Weibull is given by [Abernethy98] as:

$$F(t) = 1 - e^{-(t/\eta)^{\beta}}$$

In this formula, $\beta$ is called the shape parameter, and $\eta$ is called the characteristic life (the point at which 63.2% of the failures have occurred)[1]. Figure 1 shows examples of the Weibull probability density function (PDF) with $\eta$ = 10 and $\beta$ = .5, 1, 2, 3, and 4.

[1] To see this, substitute $t = \eta$ in the Weibull CDF to get $F(t) = 1 - 1/e = .632$, regardless of $\beta$.

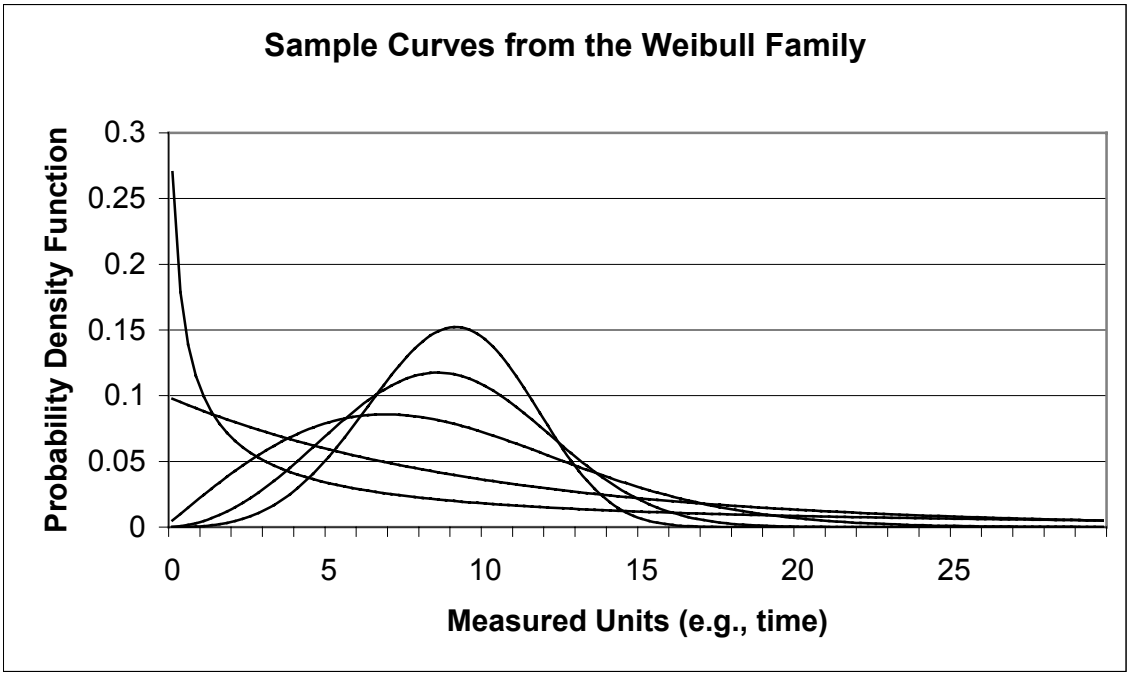



**Figure 1**

In reliability applications, the measured quantity is time, duty cycles, miles, or similar quantities associated with failure. When $\beta$ is less than one, reliability increases with time (known as infant mortality). When $\beta$ is equal to one, the Weibull distribution reduces to the Exponential distribution, implying random failure. When $\beta$ is greater than one, reliability decreases with time (wear-out).

## Modeling Defect Arrival Data

Defect arrival modeling is a subset of software reliability modeling. There are two general classifications for software reliability models [Lyu95], depending on the type of data used:

1. Failures per time period
2. Time between failures

Distinction between two types of time period is provided in [Musa87], where the authors note that software reliability can be modeled in an execution time domain or a calendar time domain. The execution time domain is generally accepted as superior to the calendar time domain. However, calendar domain data is more commonly collected.

In an execution time domain, time is measured in such a way that it represents the processor time used by the software under test. This measurement can be as fine as CPU seconds, or may be a coarse measurement such as the cumulative number of hours the system is under test by each tester. For example, 40 hours of accumulated time by the test team is equal to one tester-week, regardless of the elapsed calendar time required.

Modeling defect arrival in a calendar time domain ignores changes in test effort[2] and simply records the number of failures or the time between failures against elapsed calendar time.

[2] This could be caused by changing the number of testers, imposing automated test suites instead of manual testing, etc.

Since most managers think more easily in the calendar time domain, a means of translating between the execution time and calendar time domains is helpful. It is also important to understand the difference between time under test and time under normal use – that is, how many hours of typical field use are equivalent to one hour of test time[3].

The model presented in this paper uses failures per time period, measured in a calendar time domain as the number of failures per week. Although not as precise as more sophisticated measures of execution time like CPU seconds or even failures per 40 hours of testing, calendar time data is relatively easy to measure and provides acceptable results within the model as long as the test effort is reasonably consistent. The data in the case studies contains events such as the addition of a tester, occasional vacation and sick days, etc., but there are no large-scale changes like moving from manual to automated testing, doubling the number of testers, or an extended hiatus in testing.

## Weibull Model Assumptions

The assumptions of the Weibull model are (after [Lyu95]):

1. The rate of defect detection is proportional to the current defect content of the software
2. The rate of defect detection remains constant over the intervals between defect arrivals
3. Defects are corrected instantaneously, without introducing additional defects
4. Testing occurs in a way that is similar to the way the software will be operated
5. All defects are equally likely to be encountered
6. All defects are independent
7. There is a fixed, finite number of defects in the software at the start of testing
8. The time to arrival of a defect follows a Weibull distribution
9. The number of defects detected in a testing interval is independent of the number detected in other testing intervals for any finite collection of intervals

These assumptions are often violated in the realm of software testing. Despite such violations, the robustness of the Weibull distribution allows good results to be obtained under most circumstances.

Assumption 2 can be violated by changes in the test plan, changes to the number of testers, introduction of automated testing, and similar factors. Assumption 3 is violated 100% of the time in the case of instantaneous correction, and about 1/6 of the time for perfect correction [Jones97]. Assumption 4 depends on an accurate operational profile, which may be difficult to prove. Assumption 5 is violated when defects are masked by other defects, or more generally when one failure mode covers another. Assumption 6 is commonly violated in software, where dependent defects abound.

## Fitting the Weibull Distribution to Defect Arrival Data

The steps to fit the two-parameter Weibull distribution to defect arrival data are:

1. Obtain an estimate of the number of defects in the software
2. Calculate the cumulative proportion of total defects arriving each period
3. Transform the data to obtain a linear form
4. Fit a least-squares line to the data
5. If the fit is acceptable, use the line to obtain estimates of $\beta$ and $\eta$.
6. Plot the Weibull distribution versus the actual data

[3] For example, how many hours would a typical user need to accumulate to match the amount of use placed on a word processing program in one hour by an automated test suite?

The parameters can be re-estimated each period by adding the new data point and repeating steps 2 through 6. Each step of the modeling process is described below.

**Step 1: Obtain an Estimate of the Number of Defects in the Software**
There are several different methods for obtaining this estimate. Historical data can be used when it exists. Several commercial applications provide estimates of total defects given various inputs. In many cases, estimates are derived from the total lines of source code or the Function Point count of the application. In one approach, an estimated Function Point count is 'backfired' from the total logical statements in the source code. Tables containing the average logical statements per Function Point for most common languages are available [Jones97]. Once the estimated Function Point count is known, the number of defects present can be estimated using an assumed or known defect potential per Function Point. If desired, a range of estimates (high, expected, and low) for the total number of defects can be created and used in subsequent steps.

**Step 2: Calculate the Cumulative Proportion of Total Defects Arriving Each Period**
This is accomplished by dividing the cumulative total of located defects for each time period by the estimated total defects. For example, assuming an estimate of 1200 total defects measured week by week:

**Table 1**

| Week | New Defects | Cumulative Proportion F(t) |
|---|---|---|
| 1 | 5 | 0.004 |
| 2 | 10 | 0.013 |
| 3 | 27 | 0.035 |
| 4 | 82 | 0.103 |
| 5 | 94 | 0.182 |
| 6 | 61 | 0.233 |
| 7 | 77 | 0.297 |
| 8 | 111 | 0.389 |

**Step 3: Transform the Data to Obtain a Linear Form**
In order to fit a linear regression line, the CDF of the two-parameter Weibull must be rearranged to obtain a linear relation. After some algebraic manipulation, the Weibull can be written in the following form:

$$\ln\left(\ln\left(\frac{1}{1-F(t)}\right)\right) = \beta\ln(t) - \beta\ln(\eta)$$

So the cumulative proportion F(t) is transformed to ln(ln(1/(1-F(t))), and Weeks are transformed to ln(Weeks). The required transformations the data from Table 1 are given in Table 2.

**Table 2**

| ln(Week) | ln(ln(1/(1-F(t))) |
|---|---|
| 0 | -5.47855 |
| 0.693147 | -4.37574 |
| 1.098612 | -3.33465 |
| 1.386294 | -2.21576 |
| 1.609438 | -1.60701 |
| 1.791759 | -1.32947 |
| 1.94591 | -1.04434 |
| 2.079442 | -0.70739 |

For ease of plotting, the y-coordinate is typically rescaled to zero by subtracting the first y value from all y values as shown in Table 3. This step is optional, and is just done to have all the plotted y values be non-negative.

**Table 3**

| ln(Week) | ln(ln(1/(1-F(t))) + 5.47855 |
|---|---|
| 0 | 0 |
| 0.693147 | 1.102808 |
| 1.098612 | 2.143905 |
| 1.386294 | 3.262797 |
| 1.609438 | 3.871539 |
| 1.791759 | 4.149079 |
| 1.94591 | 4.434213 |
| 2.079442 | 4.771166 |

**Step 4: Fit a Least-Squares Line to the Data**
Now that the data has been transformed, plot the data and fit a least-squares regression line (see Figure 2).

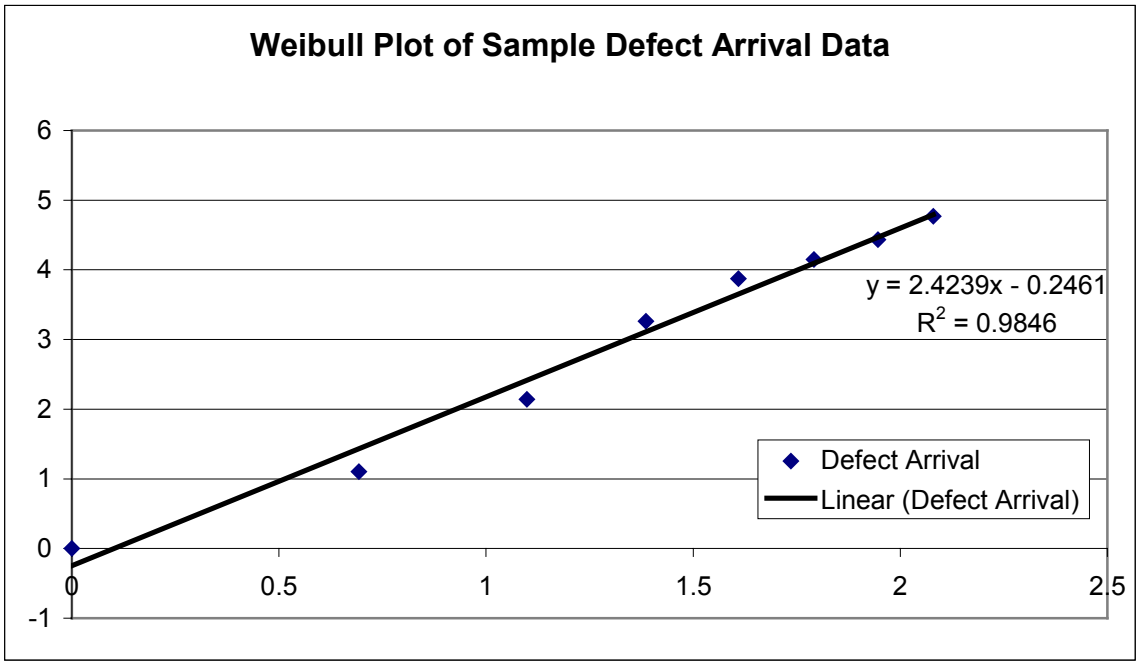



**Figure 2**

**Step 5: If the fit is acceptable, use the line to obtain estimates of $\beta$ and $\eta$**
The correlation coefficient of the fit is .9846, so this fit is acceptable as about 98% of the variability in the data is explained by the line. The estimate of $\beta$ can be read from the equation for the line as 2.4239. The value of $\eta$ can be calculated using the fact that $\eta$ is the 63.2 percentile of the distribution, that is, the time at which 63.2% of observations have occurred. This means that when $t = \eta$, (remember we added 5.47855 to rescale all y values in Step 3)

$$\ln\left(\ln\left(\frac{1}{1-.632}\right)\right)+5.47855=2.4239\ln(\eta)-.2461$$

Solving this equation, $\eta$ = 10.6 weeks. These calculations can be automated easily in Excel*.

**Step 6: Plot the Weibull Distribution versus the Actual Data**
This step is actually optional, as inferences can be made from the Weibull plot constructed in the previous step. However, because of the significant rescaling and transformation of the data, it is a good idea to return the plot to normal space before presenting it.

The plot makes use of the WEIBULL function in Excel*, which has the following arguments: WEIBULL(data, $\beta$, $\eta$, Cumulative?). In this function, the data values are the time periods (in this case, 1 through 8), and the 'Cumulative?' parameter determines whether the resulting plot is cumulative (the CDF) or not (the PDF). Figure 3 and Figure 4 show examples of both types for the sample data:

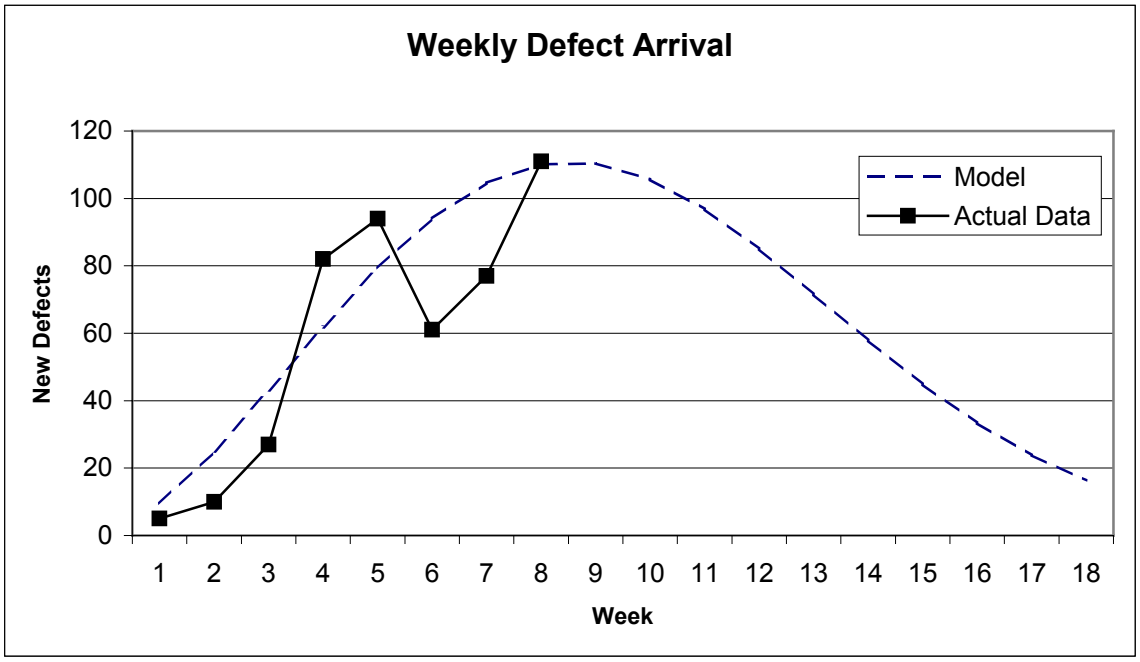



**Figure 3**

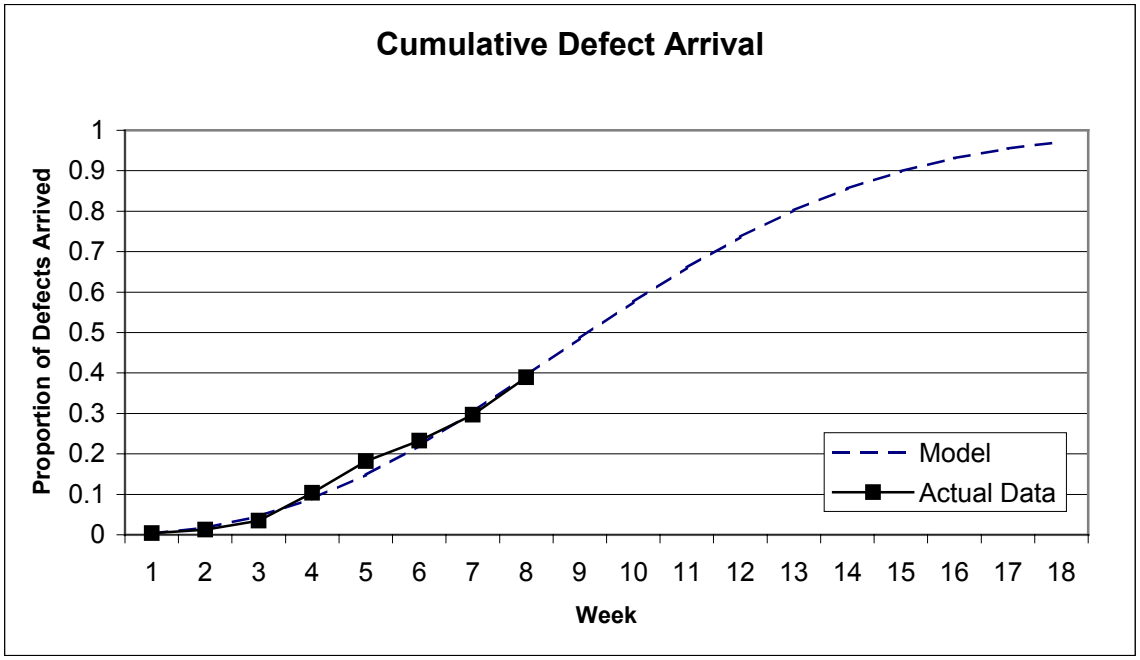



**Figure 4**

There is a danger in presenting cumulative measures. On projects with a large number of defects, cumulative charts can mask even reasonably big trends, precisely when they are of greatest interest. For this reason, the weekly arrival chart is normally a better indication of defect arrival trend. An example of this problem is given in the second case study.

## Results

Three case studies from actual projects completed over the last several years are given below. The fits are performed on non-cosmetic (Projects A and B) or High/Medium priority defects (Project C), depending on the type of data available. In all three cases, the fits including cosmetic and low priority defects were not as good. One possible reason for this is that most testers tend to focus more on the significant and severe defects first, and focus on less serious or cosmetic issues only when few remaining serious issues can be found.

**Project A**

Project A was a traditional two-tier client/server application, written by a team of four developers of varying experience and one tester. Testing was manual, according to a well-written test plan. The Weibull distribution was fit post-hoc using the 1008 non-cosmetic defects located during testing.

The model's fit is shown in Figure 5. There are two failure modes evident in the plot. Note that the decision to isolate distinct failure modes must be made based on knowledge of the process used to create the system, not just on anomalies in the data. If there is a solid process or engineering reason why a failure mode should be separated, it is appropriate as long as the new failure mode adds value and not just complexity to the overall model.

The first failure mode, lasting three weeks, reflects a period when integration defects caused blocking failures that prevented full system execution. This is 'infant mortality', since $\beta$ = .19, far less than one. The longer the system ran, the more reliable it became (that is, if an integration

defect did not cause a nearly immediate problem, the software could be executed much more fully).

The second failure mode represents system testing once blocking was removed. For the second failure mode, $\beta$ = 1.8 and $\eta$ can be calculated to be 17.1. There are noticeable deviations from the model for this failure mode. Examining the data, one might wonder if there are really three failure modes contained within the period after week 4: a pattern of shallow slope, followed by a steeper slope, then shallow again. Is there a reason to separate the steep slope from the two shallower slopes?

In manufacturing environments, this shallow-steep-shallow pattern in Weibull plots is associated with batch defects where a known, stable failure mode suddenly changes due to defective materials, equipment out of tolerance, etc. A similar situation in software engineering is when a module of code is produced with an uncharacteristically high error content. This is precisely what occurred on Project A in week 6. After exploring this further, the model using four failure modes was not significantly better than the one with only two modes for purposes of estimates and projections, so the simpler two-failure-mode model was used.

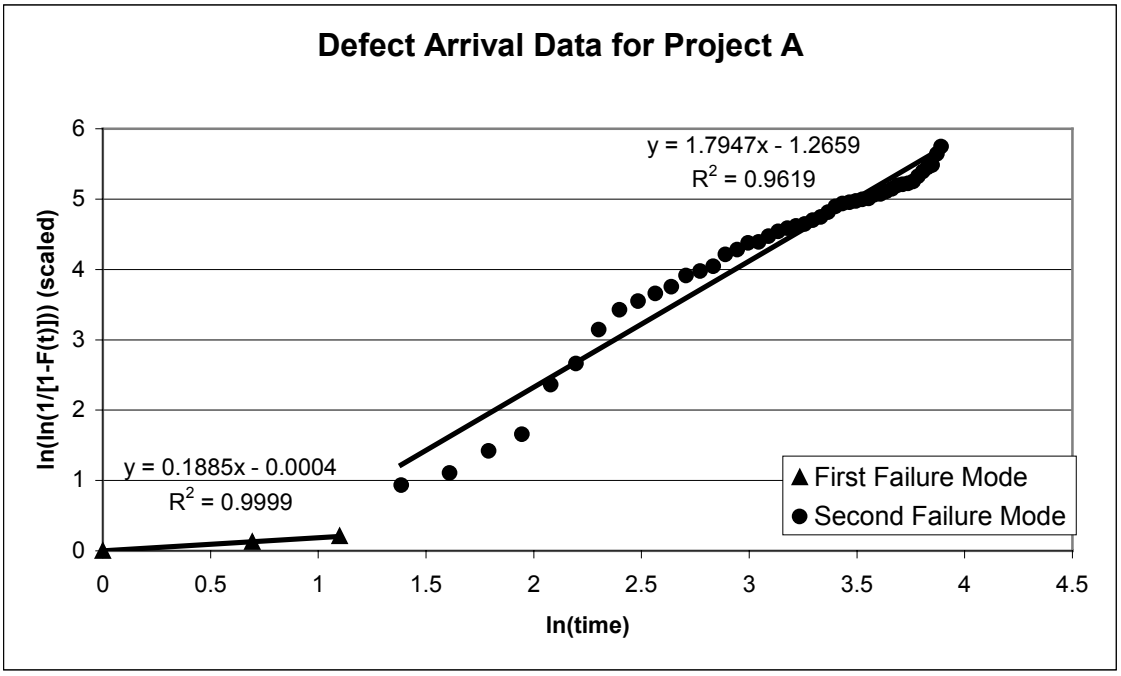



**Figure 5**

The actual weekly defect arrival versus two models for Project A is shown in Figure 6. In this figure, the effects of the integration and blocking defects can be seen, as can the large spike in defect arrival after the integration of a large and defect-prone module in week 6. The final model fits the data quite well after the spike in defects has past.

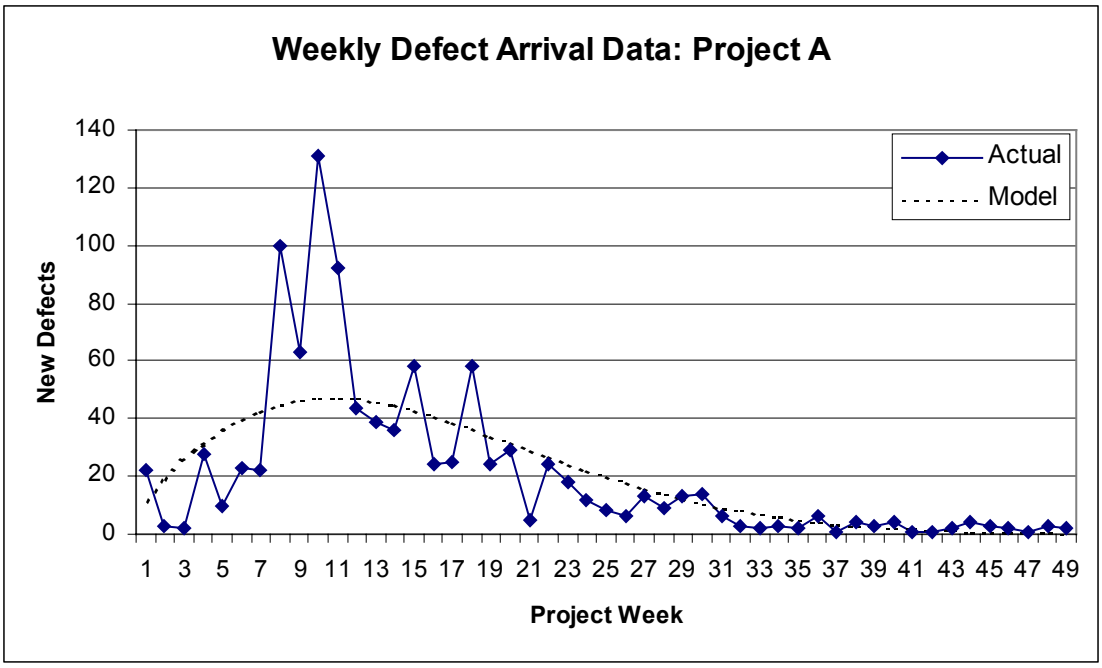



**Figure 6**

**Project B**
Project B was a Web-enabled three-tier application built using distributed objects. It was written by a team of 8 developers, and there were between 1 and 2.5 testers on the project at any given time. The development team was made up of moderately experienced to highly experienced individuals. Testing was manual, based on an extensive, formally inspected test plan.

The system was integrated in three approximately equal stages. The integration cycles were all relatively smooth, and there were few regressions.

The Weibull distribution was fit using an initial estimate of 1000 non-cosmetic defects, but the quality proved to be better than that estimate. The total defect estimate was adjusted to 800 near week 18 of the project. The linear fit given the actual 739 defects is shown in Figure 7. There are three failure modes present. The first failure mode represents integration defects blocking execution (as in Project A). The second failure mode represents system testing and shows the characteristic undulations of the 'build rhythm' of an iterative project. This failure mode has $\beta$ = 3.26 and $\eta$ about 24.5 weeks.

The final failure mode corresponds to customer acceptance testing. During this three-week period, the equivalent of several additional fulltime testers were evaluating the product at the customer site, increasing the number of defects located per week.

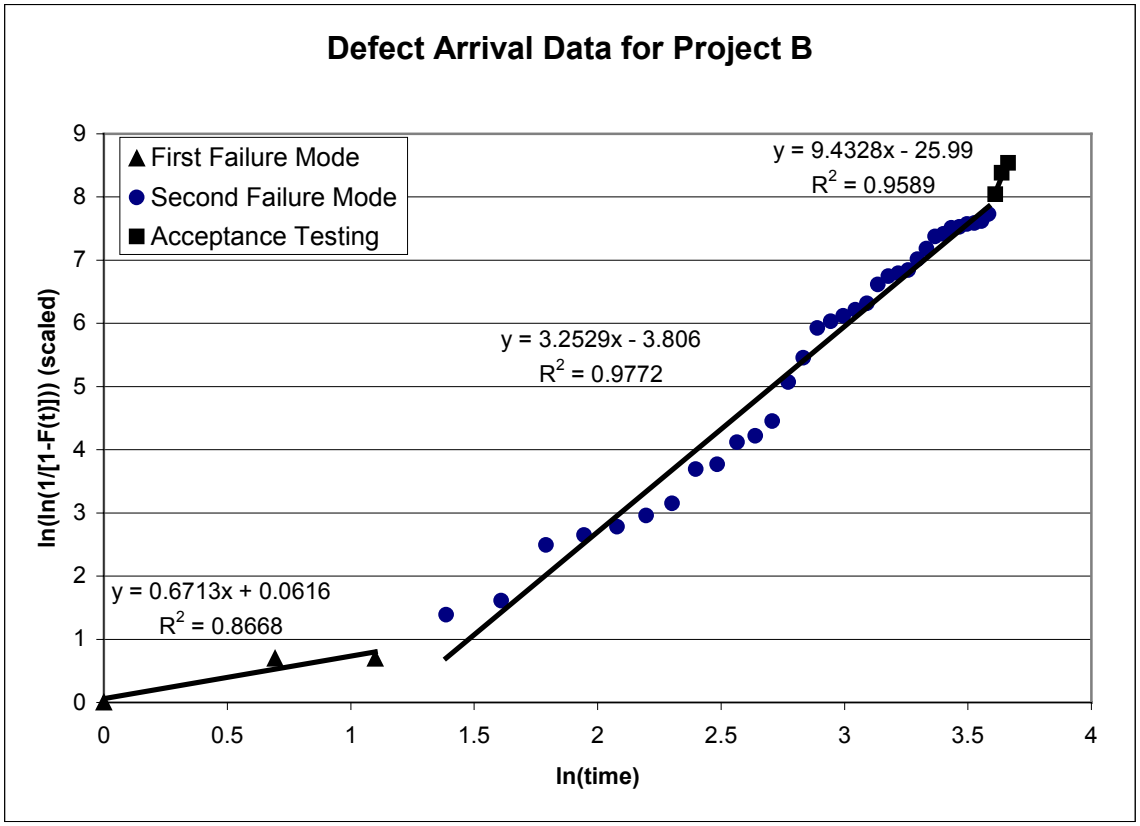



**Figure 7**

The actual weekly defect arrival versus the model for Project B is shown in Figure 8. The effects of initial integration and blocking defects can be seen on weekly defect counts through week 6. It is interesting to note that with each new integration, the height of the spike in defect arrival is reduced and the time between integrations is shorter – a comforting trend. The spike in defects represented by acceptance testing (weeks 36-38) is also apparent, but might be missed in casual observation without the benefit of the data in Figure 7.

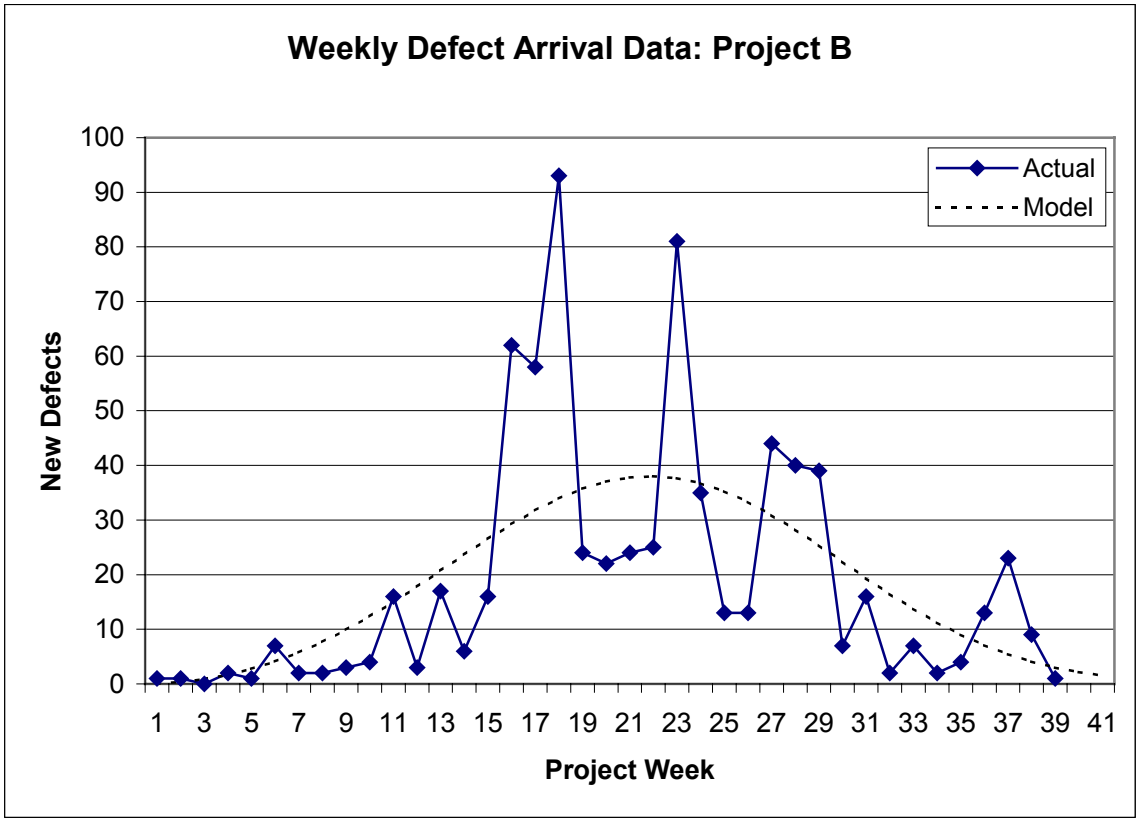



**Figure 8**

The cumulative defect arrival pattern for Project B is shown in Figure 9. While this view of the data has some value, it is also a good illustration of the way that cumulative statistics can hide local trends. The significant spike in defect arrival between weeks 36 and 38 looks no different than other periodic differences caused by the build rhythm, but Figure 7 shows that it is very different. Had this been a trend towards a much higher defect density (through increased defect injection, for example), the problem may not have been apparent for several more weeks using only the cumulative view of the data.

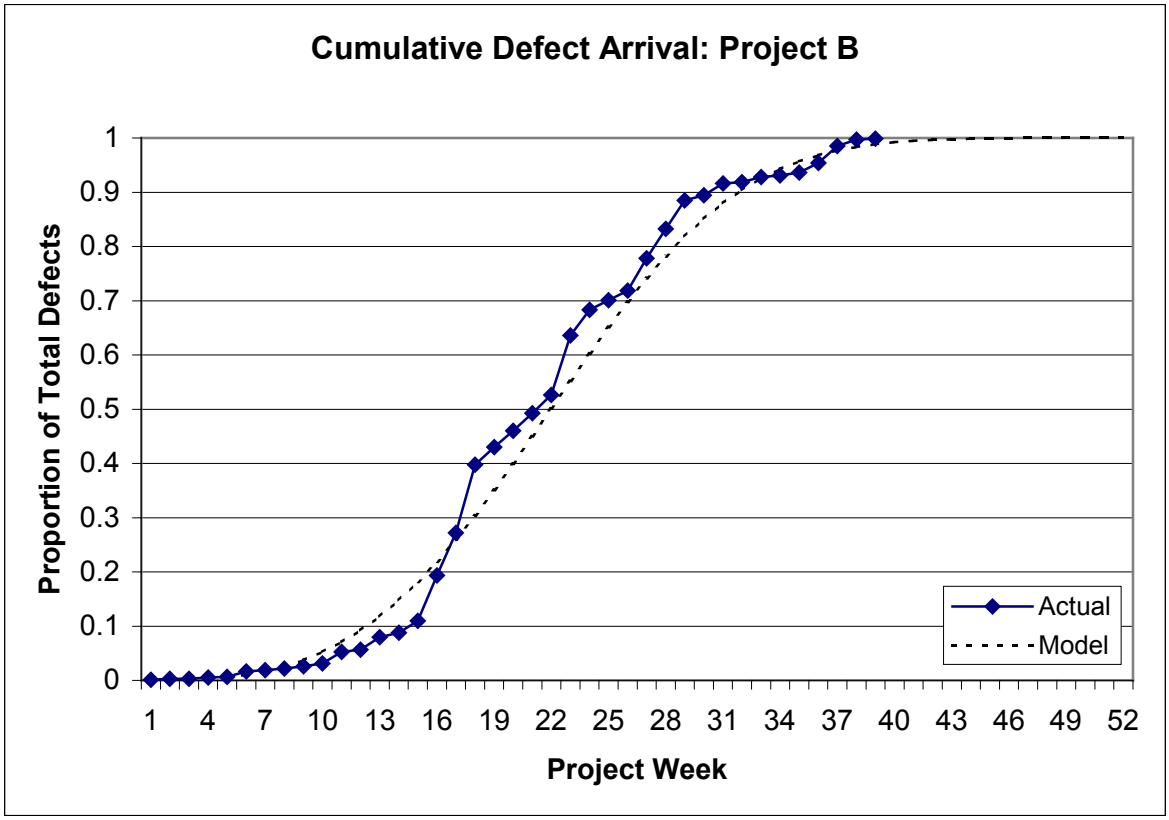



**Figure 9**

**Project C**

Project C was a traditional two-tier client/server application written by a team of 12 developers. The system was based on a core system that already existed, but there were extensive additions. There were between 4 and seven testers on the project at various times. Testing consisted of a mix of manual and automated testing, but was predominantly manual. Testing was based on extensive test plans and specifications.

The system was built through several small to moderate integration cycles, with one larger cycle near the end of the project.

The Weibull distribution was fit initially to an estimate of 2500 total defects (2000 assumed to be high or medium priority) obtained from commercial software. After examining the defect arrival trends about three-fourths of the way into the project, it became apparent that the total defect count was closer to 3000 (2500 assumed to be high or medium priority). This later estimate of 2500 defects is used in the figures that follow. The linear fit is shown in Figure 10. Three failure modes are present. The first mode containing weeks 1-3 represents mainly human factors and usability testing of the design using screen shots and design walkthroughs. Failures in this mode were approximately random (a $\beta$ of approximately 1). The second mode indicates infant mortality as found in the other two case studies (a $\beta$ of about .61). Once blocking was removed and functional testing began in week 10, the third failure mode appears with $\beta$ approximately 2.6.

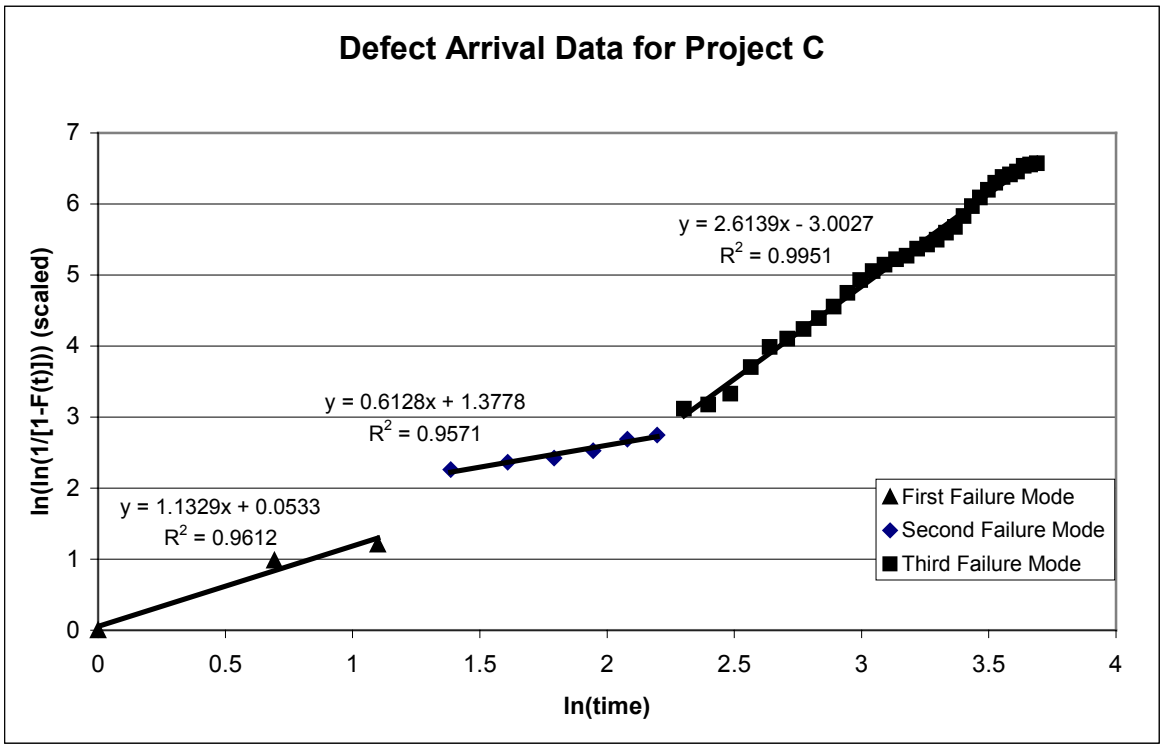



**Figure 10**

The weekly arrival pattern is shown in Figure 11. The pattern shows the characteristic build rhythm of an iterative project, and the blocking prior to week 10 is evident. The large integration that occurred in week 30 is followed by the largest spike in defect arrival (as might be expected). While this spike caused some concern during the weeks it arrived, examination of the failure mode data as shown in Figure 10 and the defect database gave some feeling that this was not a new failure mode, but more of a singular event. This turned out to be the case, as can be seen by defect arrival levels following week 35.

On this project, defect arrival modeling data was used as strong evidence to adjust the initial schedule estimate. It also was used to help decide when to stop functional testing of the current version of the system and to begin the next version with increased functionality. Based on having found about 90% of the estimated defects, functional testing was curtailed in week 40 and the new round of development was begun.

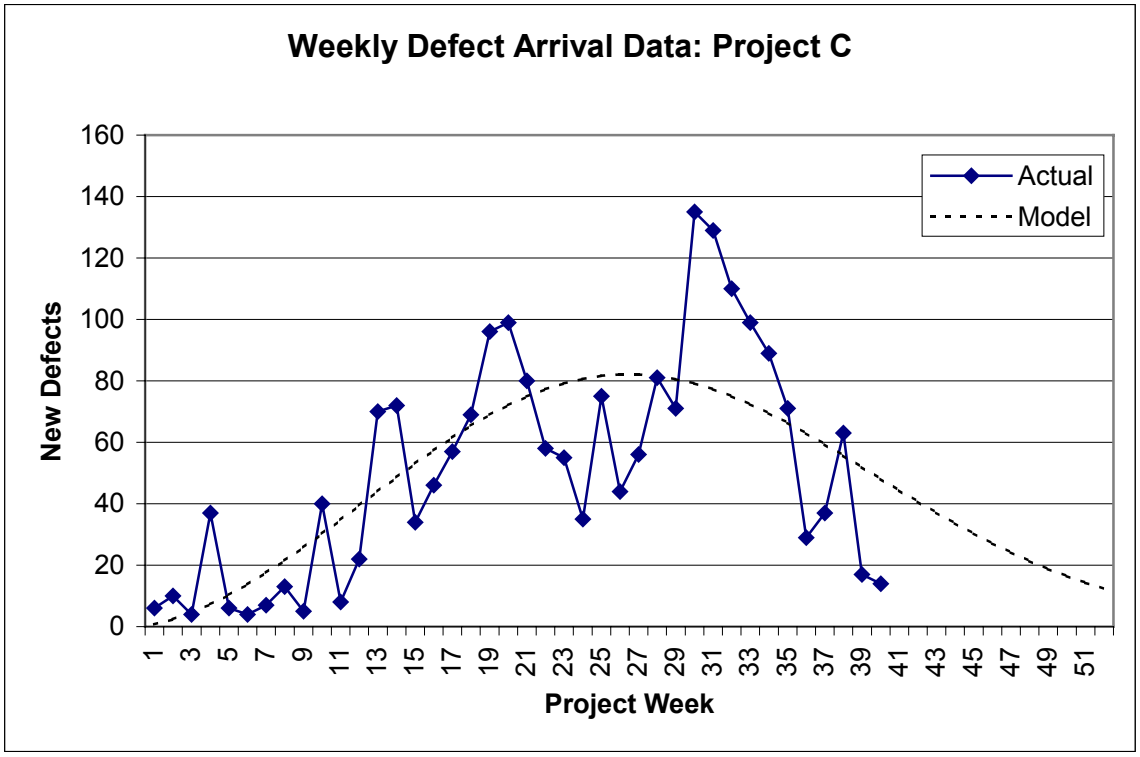



**Figure 11**

## Discussion

When fitting Weibull models to defect arrival data there are some things to consider based on these results:

- Fits appear to be better when the more serious (non-cosmetic, high/medium priority, etc.) defects are used. Fits using all defects are not as good.
- All three case studies had a period of 'infant mortality' at the start of testing. Once blocking and integration defects were removed, the main failure mode appeared and remained relatively constant thereafter. The length of the infant mortality failure mode appears to be proportional to the size of the system (consistent with intuition).
- The value of $\beta$ for a failure mode is influenced by where on the axes the transformed data are plotted, since both axes are measured in a log scale rather than a linear scale. When early (i.e., infant mortality) failure modes are included in the Weibull plot the value of $\beta$ for the main failure mode is larger than when it is omitted. This matters if two projects are to be compared for testing effectiveness by comparing their respective $\beta$ values. The best approach may be to plot the values obtained without the early failure modes included (see Figure 12). In this case, the first two projects each used manual testing and a single tester, and their dominant failure modes have very similar slopes. Project C used several testers and a mix of manual and automated testing, and the dominant failure mode shows a correspondingly steeper slope.
- Estimates of $\beta$ appear to be reliable enough for basic decision support soon after the appearance of the main failure mode. While the successive estimates to $\beta$ vary by week in the case study data, they are consistent enough that the basic conclusions drawn from the data do not change.

- The particular project lifecycle used for a project will influence the defect arrival pattern. Iterative lifecycles will have different patterns than those using continuous integration or a modified waterfall. In [Putnam92], the distribution is used to model defects located in all phases of development through many types of testing and static defect removal activities. In this paper, the data come from system testing only. In either case, the model fits the data well enough to make project management and engineering support decisions.
- This technique should work well on all types of applications, relying only on a good source of defect data. However, it has not been widely used so future work may uncover situations where it works too poorly to be useful.

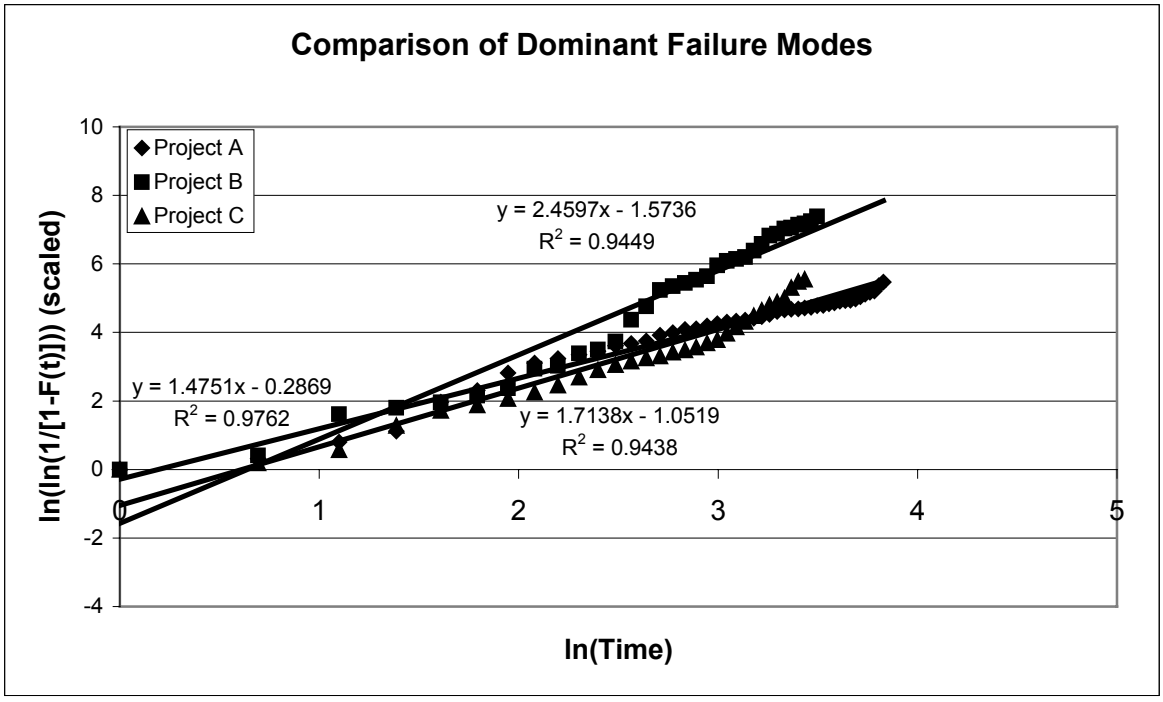



**Figure 12**

## Opportunities for Further Work

There are three principle areas for further work in this area. The first is an understanding of the relationship between factors such as the number of testers, type of testing (manual, automated, mixed), etc. and the resulting shape parameter $\beta$. This information can be used in a technique known as Weibayes (coined from a contraction of *Weibull* and *Baysian*) to make accurate inferences very early in the testing process when only a few defects have arrived [Abernethy98]. Many such libraries for hardware failure exist, but no one has investigated possible parallels in the software world.

Second, techniques that remove the need to estimate the total number of defects in the software might be explored, as this is in most cases a difficult, error-prone task. Specialized Weibull analysis techniques for interval data might be adapted and applied to software defect arrival given coarse data like that used in this paper. If more precise failure time data are available, the use of median rank plotting positions may be able to remove the need to know the total number of defects.

Last, a better understanding of how results vary for different types of data would be helpful in guiding the choice of what data to collect. The number of failures in calendar time is easy to collect and gives useful results. Other papers could outline comparative fits obtained using this type of data and execution time domain time between failures (generally regarded as the best for these purposes).

## Summary

Because of its flexibility and power, the Weibull distribution can be used to create useful models of software defect arrival even in a calendar time domain. The Weibull distribution is fit to defect arrival data via a six-step process that is easily automated in a PC spreadsheet. The resulting models are capable of locating different failure modes caused by changes in the test environment or application under test. Projections based on the models can help drive schedule and release decisions, and might allow a test manager to measure the affects of adding testers, moving to automated testing, and similar strategies on the defect arrival rate. It may be possible to establish libraries for the shape parameter $\beta$ so that given data on the number of testers, the type of testing, etc., accurate projections of the amount of time required to test the application can be made very early in the project. Better precision within failure time data and or more advanced Weibull analysis techniques may remove the need to estimate the total number of defects in the software.

## Acknowledgements

The author acknowledges the kind support of several colleagues who provided data and background for the case studies contained in this paper. Thanks are also due to the reviewers that helped improve the presentation and content.

## References

| | |
|---|---|
| Abernethy98 | Abernethy, Dr. Robert B., *The New Weibull Handbook, 3rd ed.*, Self-published 1998 |
| Jones97 | Jones, Capers, *Software Quality: Analysis and Guidelines for Success*, Thompson Computer Press 1997 |
| Kan95 | Kan, Stephen H., *Metrics and Models in Software Quality Engineering,* Addison Wesley 1995 |
| Lyu95 | Lyu, Michael (ed.), *Handbook of Software Reliability Engineering,* McGraw-Hill/IEEE Computer Press 1995 |
| Musa87 | Musa, John, *et al., Software Reliability: Measurement, Prediction, Application,* McGraw-Hill 1987 |
| Putnam92 | Putnam, Lawrence, and Myers, Ware, *Measures for Excellence – Reliable Software On Time, Within Budget*, Yourdon Press 1992 |

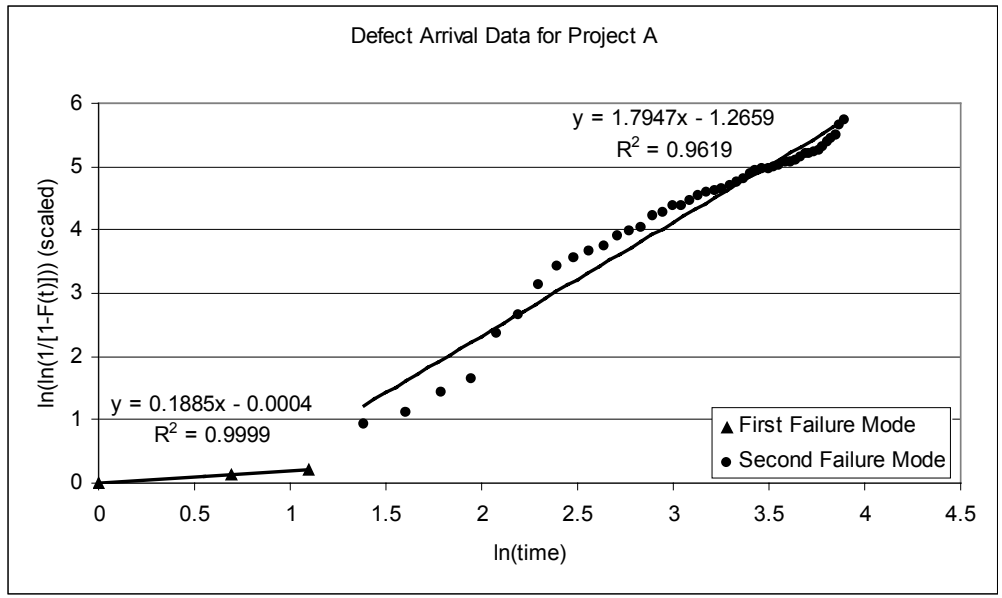
---

[*] Third-party brands and names are the property of their respective owners.

# When Will We be Done Testing?

## Software Defect Arrival Modeling With the Weibull Distribution

Erik Simmons, Intel Corporation




Version 0.1
8/00



For information, contact:
Erik.Simmons@Intel.com

# Contents

- The Weibull Distribution
- Software Defect Arrival Modeling – the Basics
- Fitting the Weibull Distribution to Defect Arrival Data
- Case Studies
- Sources for More Information

# The Weibull Distribution

Discovered in 1937 by Waloddi Weibull

Used since the 1950s to model diverse things:

- Hardware failures
- Radar clutter
- Warranty & support costs
- Spare parts levels
- And many more…

The two-parameter Weibull:

$$F(t) = 1 - e^{-(t/\eta)^{\beta}}$$

# The Weibull Distribution

The parameters of the Weibull distribution:

$\beta$ - the Shape parameter

$\eta$ - the Characteristic Life

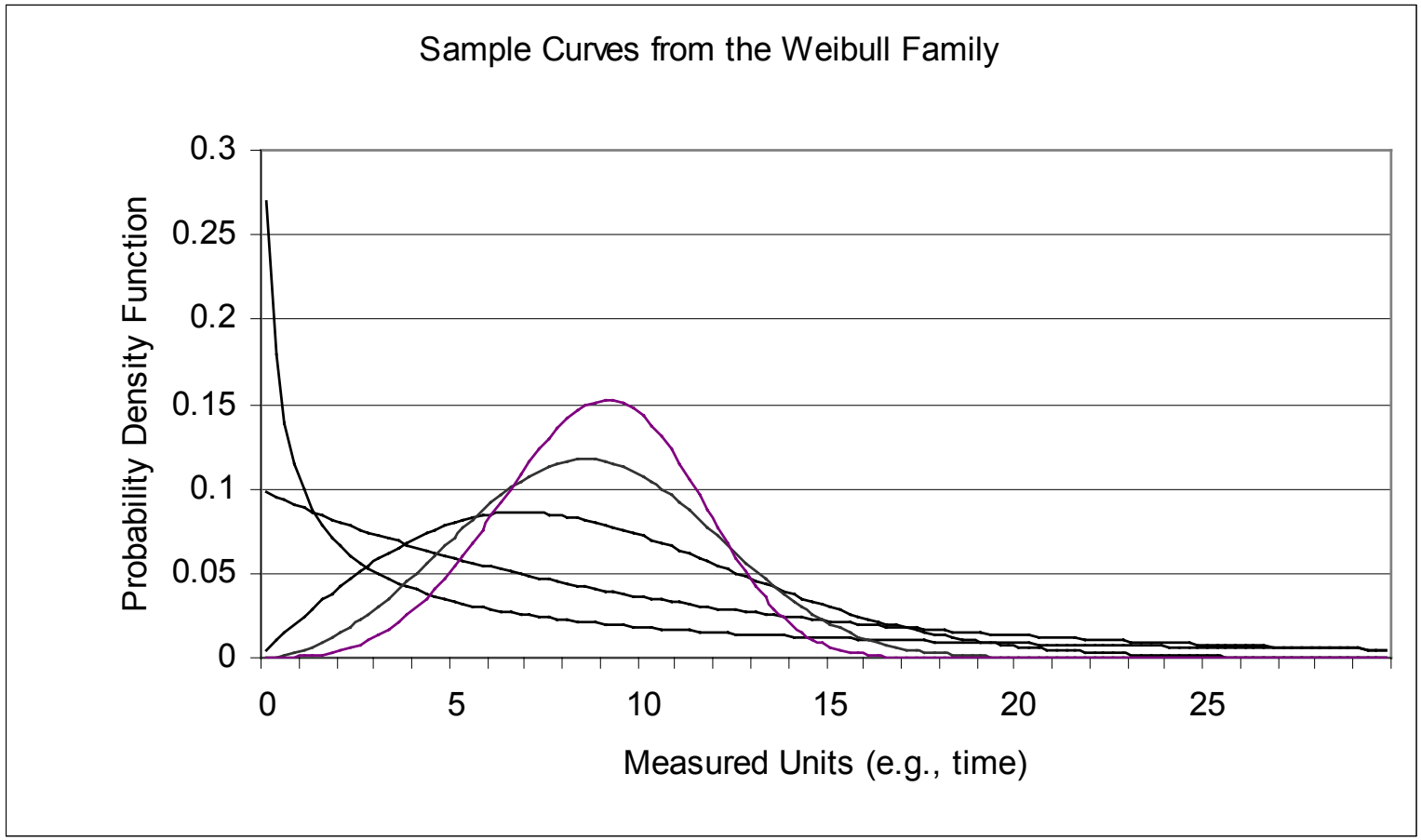
$\beta < 1$ indicates 'infant mortality, where the longer the system runs the less likely failure becomes

When $\beta = 1$, the Weibull reduces to the Exponential distribution, implying random failure

$\beta > 1$ indicates wear out, where the longer the system runs the more likely failure becomes

$\eta$ is the point at which 63.2% of the failures have occurred

# The Weibull Distribution



# Software Defect Arrival Modeling

Failure model classifications [Lyu95]:

- Failures per time period
- Time between failures

Time domains [Musa87]:

- Execution time
- Calendar time

The execution time domain is generally recognized as superior to calendar time, but can be harder to measure

This paper uses the number of failures per calendar week

# Wiebull Model Assumptions

The assumptions of the Weibull model are (after [Lyu95]):

- The rate of defect detection is proportional to the current defect content of the software
- The rate of defect detection remains constant over the intervals between defect arrivals
- Defects are corrected instantaneously, without introducing additional defects
- Testing occurs in a way that is similar to the way the software will be operated

# Weibull Model Assumptions

- All defects are equally likely to be encountered
- All defects are independent
- There is a fixed, finite number of defects in the software at the start of testing
- The time to arrival of a defect follows a Weibull distribution
- The number of defects detected in a testing interval is independent of the number detected in other testing intervals for any finite collection of intervals

Luckily, the Weibull is robust to most violations…

# Fitting the Weibull Distribution

The steps to fit the two-parameter Weibull distribution to defect arrival data are:

- Obtain an estimate of the number of defects in the software
- Calculate the cumulative proportion of total defects arriving each period
- Transform the data to obtain a linear form
- Fit a least-squares line to the data
- If the fit is acceptable, use the line to obtain estimates of $\beta$ and $\eta$.
- Plot the Weibull distribution versus the actual data

# Sample Data

| Week | New Defects |
|---|---|
| 1 | 5 |
| 2 | 10 |
| 3 | 27 |
| 4 | 82 |
| 5 | 94 |
| 6 | 61 |
| 7 | 77 |
| 8 | 111 |

# Estimate the Total Number of Defects

An estimate can be derived in several ways:

- Historical data
- Commercial software
- LOC → Function Points → Defects
- Etc…

If the estimate is off significantly, you will see it in the plots as time goes on

The estimate can be revised during the process

Sample data estimate: 1200 defects

# Calculate the Cumulative Proportions

| Week t | New Defects | Cumulative Proportion F(t) |
|---|---|---|
| 1 | 5 | 0.004 |
| 2 | 10 | 0.013 |
| 3 | 27 | 0.035 |
| 4 | 82 | 0.103 |
| 5 | 94 | 0.182 |
| 6 | 61 | 0.233 |
| 7 | 77 | 0.297 |
| 8 | 111 | 0.389 |

# Transform the Data to a Linear Form

The two-parameter Weibull can be re-expressed in a linear form:

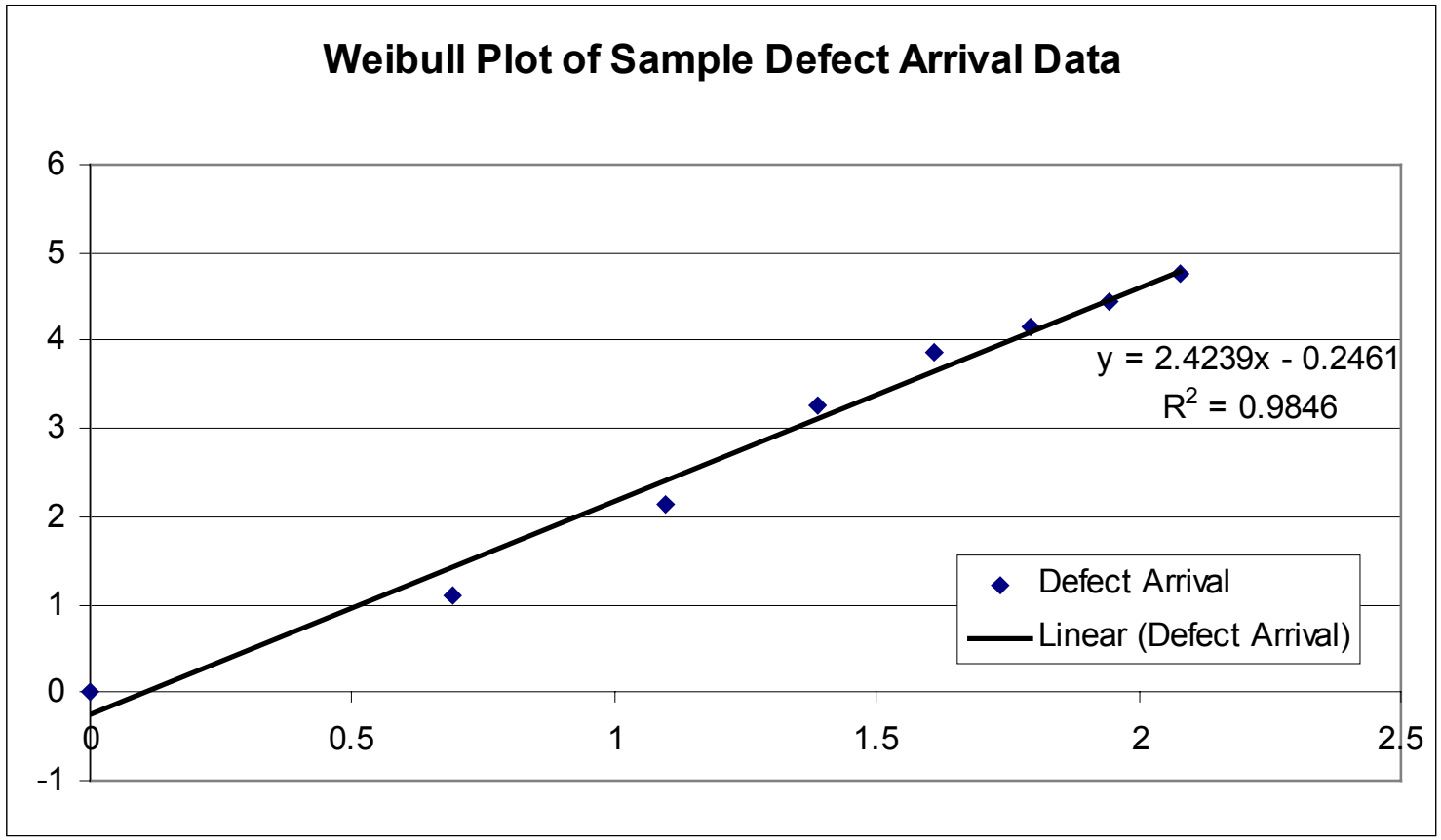
$$\ln\left(\ln\left(\frac{1}{1-F(t)}\right)\right) = \beta\ln(t) - \beta\ln(\eta)$$

Y = mX + B

# Transform the Data to a Linear Form

| ln(Week) | ln(ln(1/(1-F(t))) | Rescaled |
|---|---|---|
| 0 | -5.47855 | 0 |
| 0.693147 | -4.37574 | 1.102808 |
| 1.098612 | -3.33465 | 2.143905 |
| 1.386294 | -2.21576 | 3.262797 |
| .1609438 | -1.60701 | 3.871539 |
| 1.791759 | -1.32947 | 4.149070 |
| 1.94591 | -1.04434 | 4.434213 |
| 2.079442 | -0.70739 | 4.771166 |

# Fit a Least Squares Line to the Data



# If the Fit is Acceptable, Estimate $\beta$ and $\eta$

The $R^2$ for the line is .9846, so about 98% of the variation in the data is explained by the line

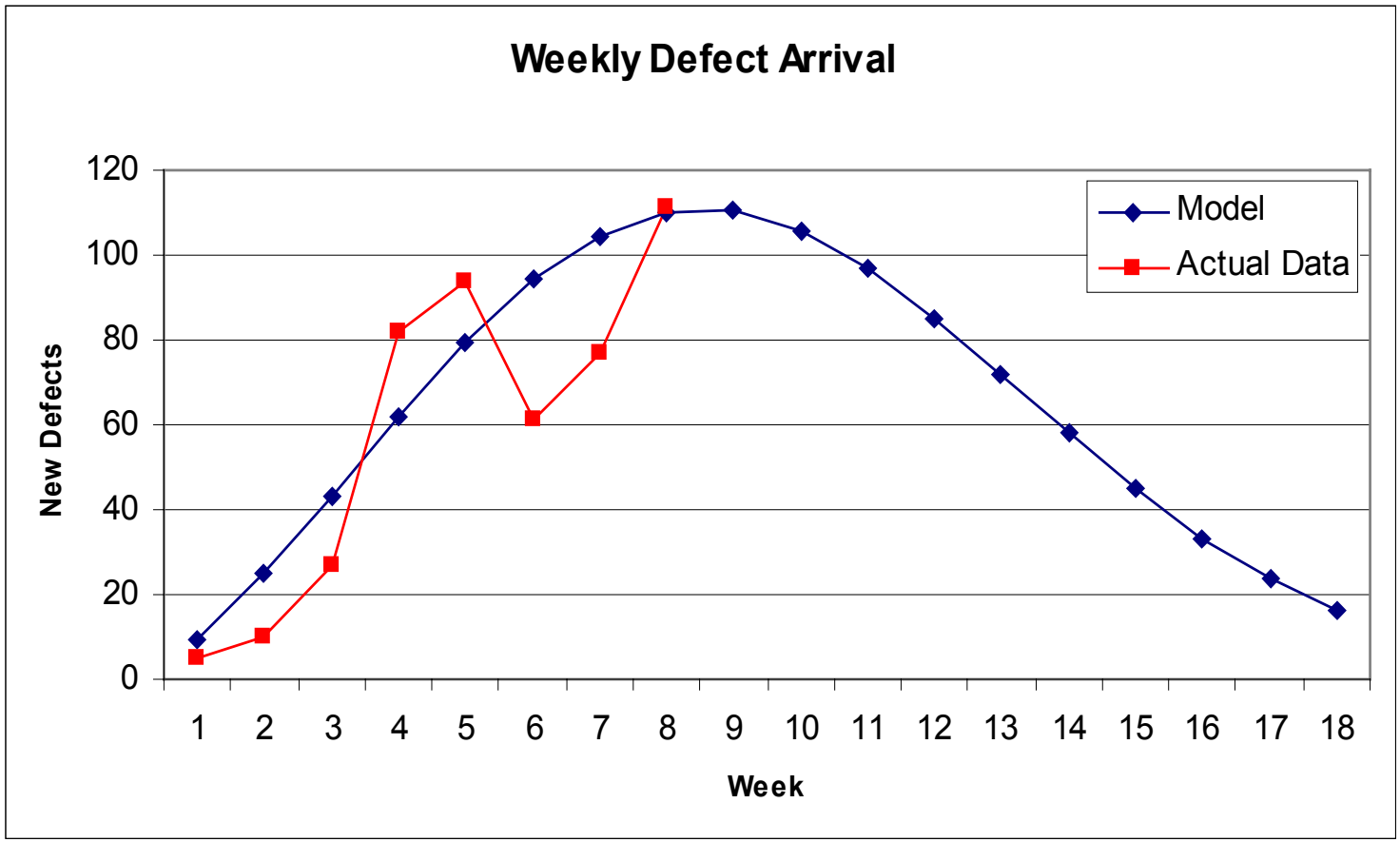
$\beta$ can be read from the regression equation as the slope: 2.4329

$\eta$ can be calculated by solving the following equation (remember that we rescaled the data):

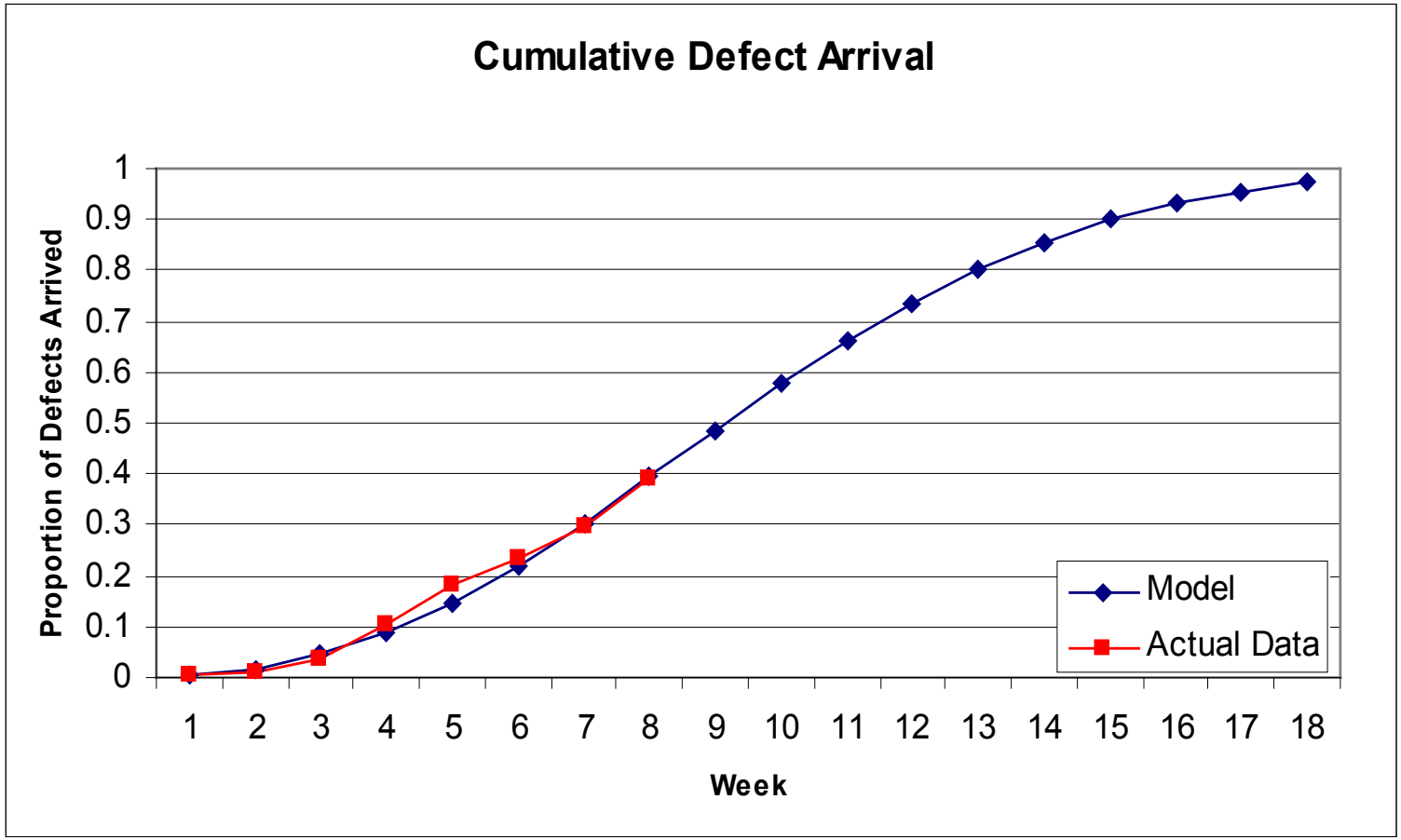
$$\ln\left(\ln\left(\frac{1}{1-.632}\right)\right)+5.47855=2.4239\ln(\eta)-.2461$$

Solving, we get $\eta$ = 10.6 weeks

# Plot the Model vs. the Actual Data



# Plot the Model vs. the Actual Data



# Case Studies

# Project A

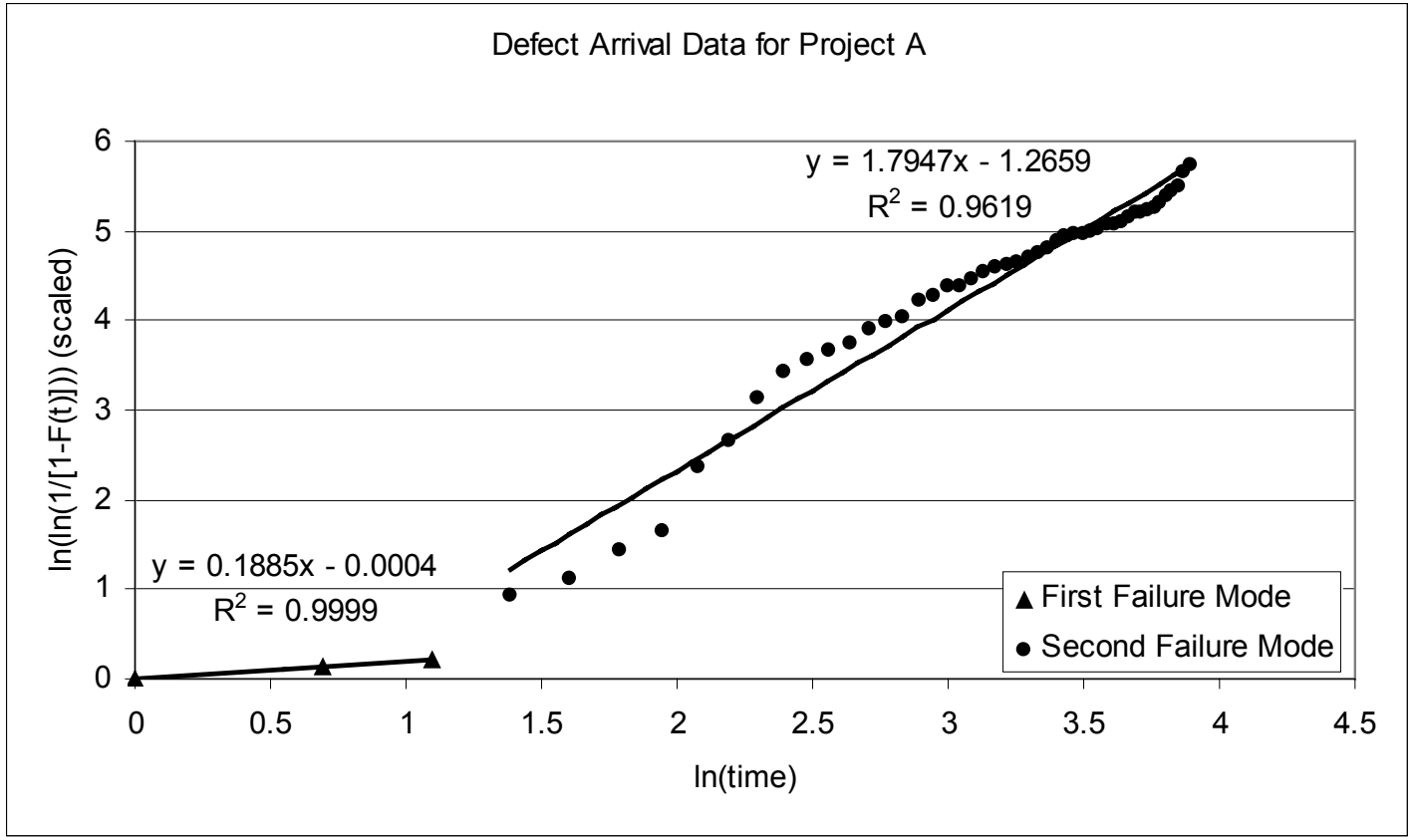
Two-tiered client/server application

4 developers, 1 tester

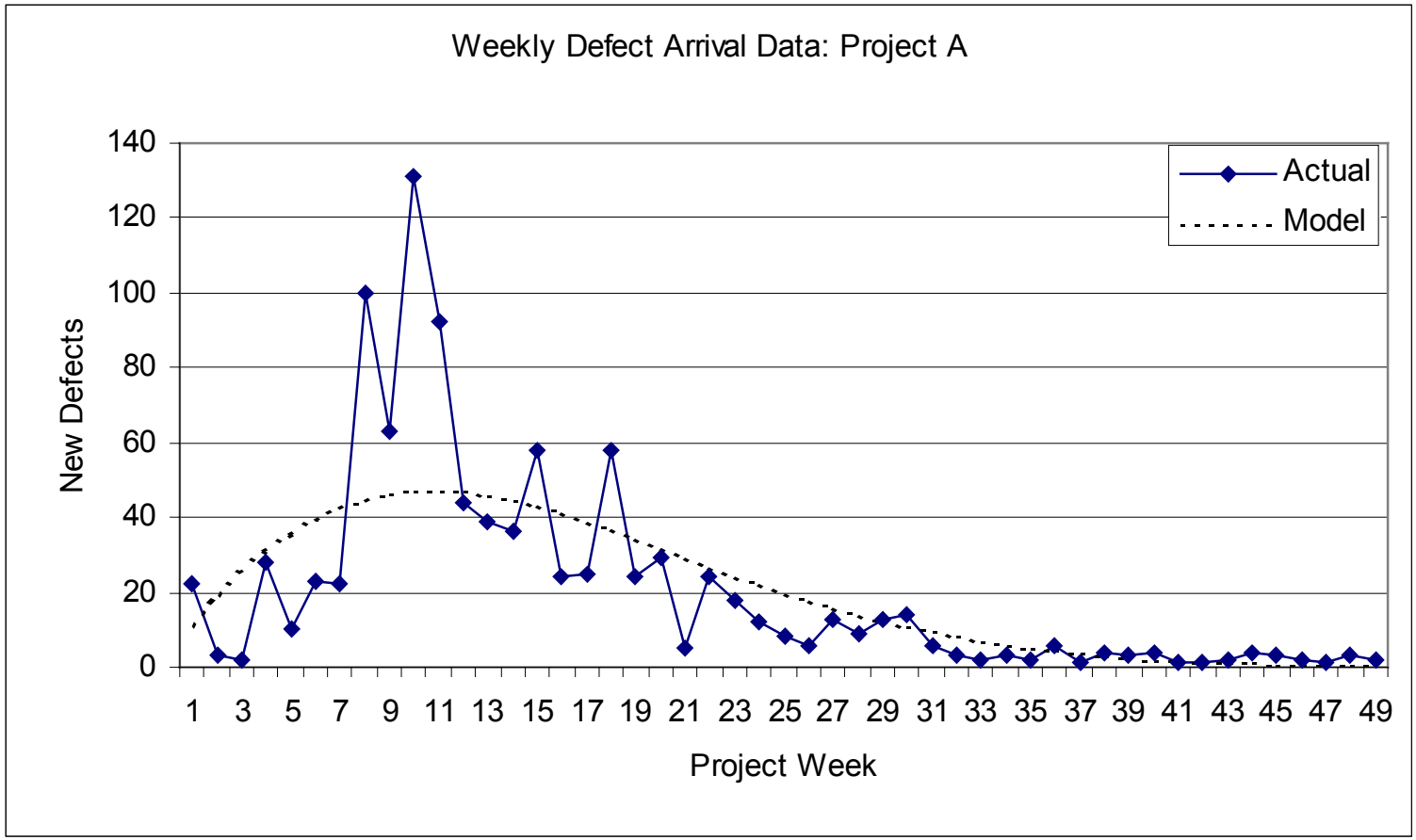
Manual testing according to a written plan

Weibull fit using 1008 located defects

# Project A: Weibull Plot



# Project A: Weekly Arrival



# Project B

Web-enabled three-tiered client/server application

8 developers, 1-2.5 testers

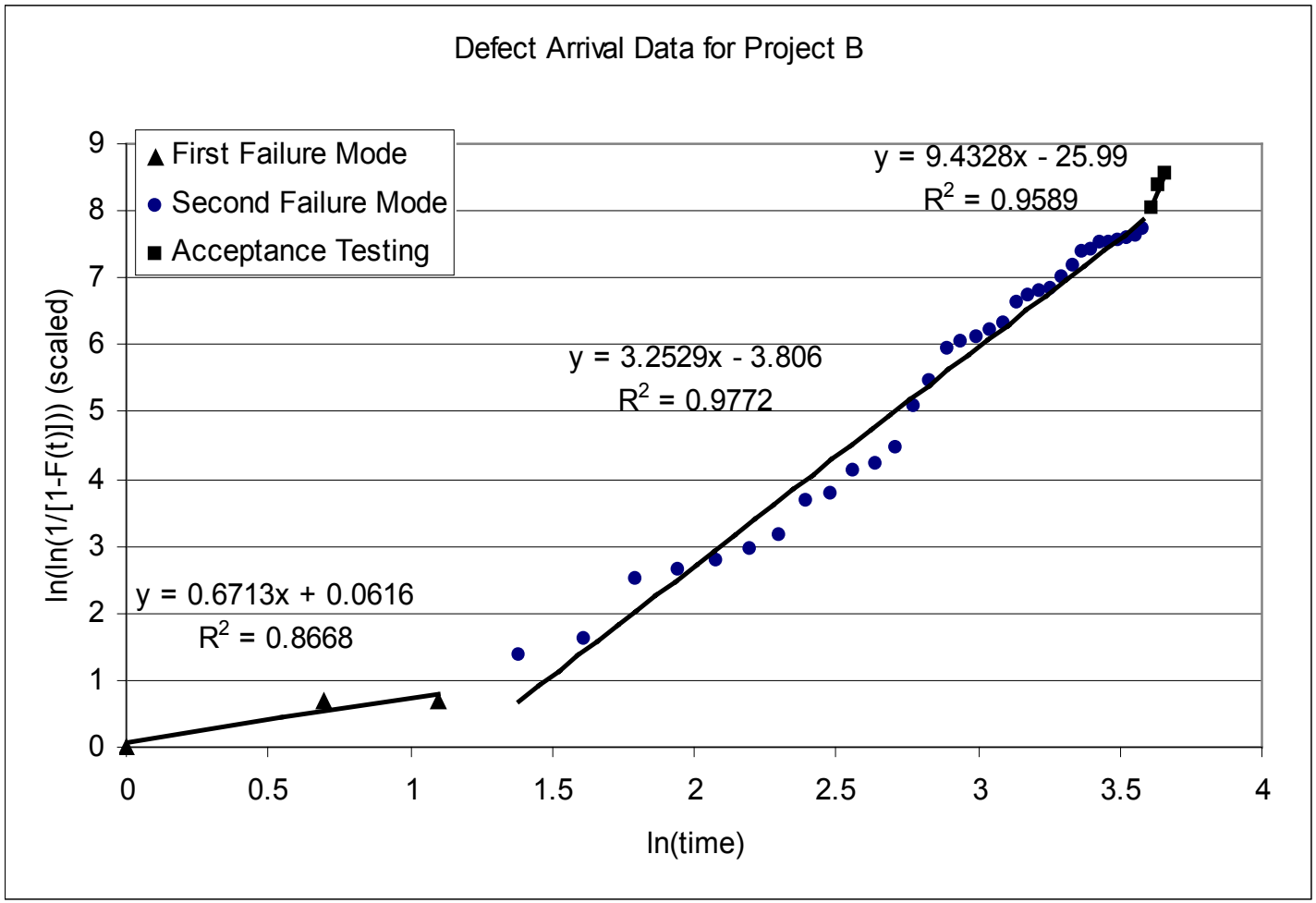
Manual testing according to a extensive, inspected plan

First Weibull fit to 1000 estimated defects

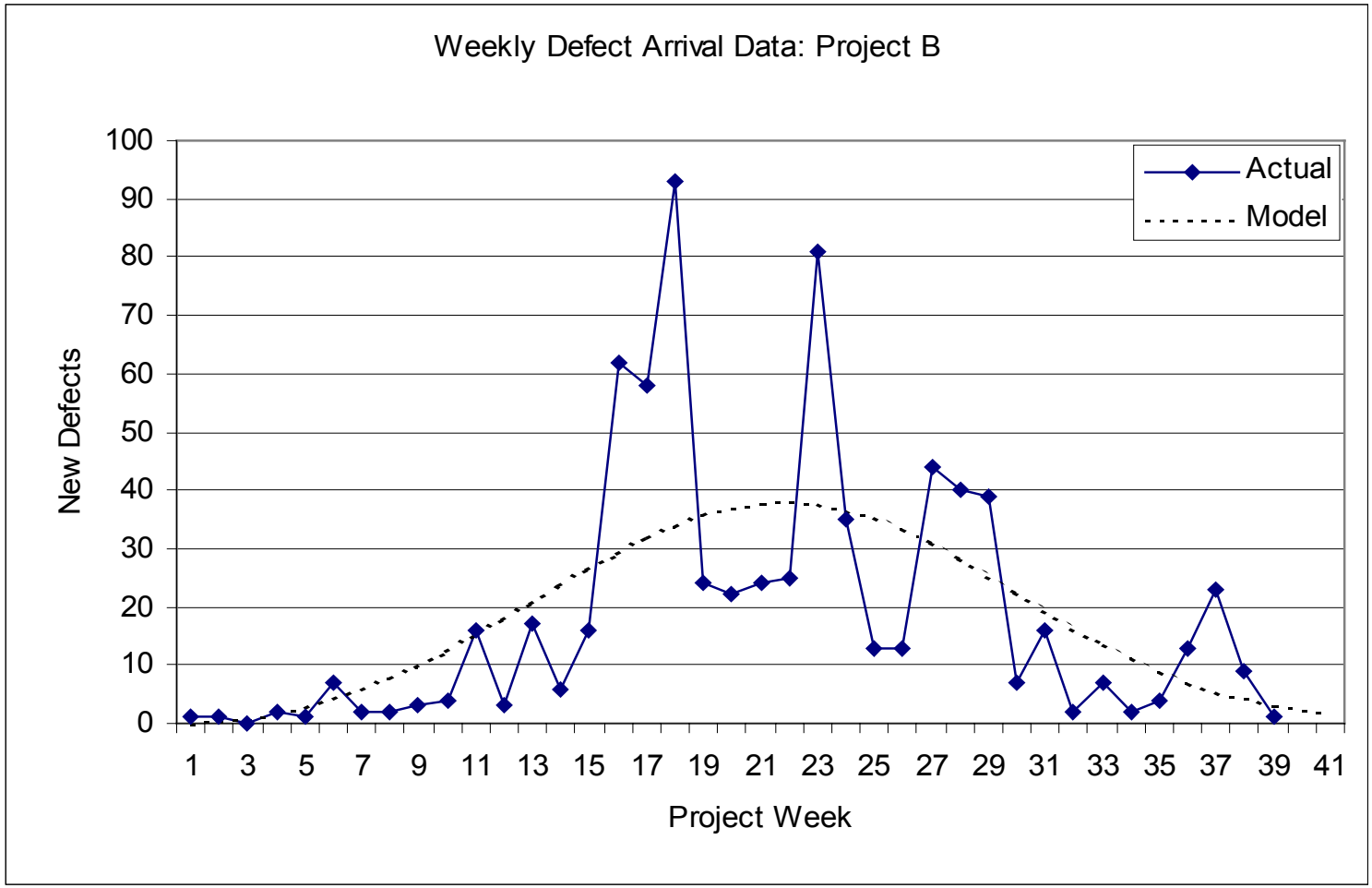
Total defect estimate trimmed to 800 near the end

Final fit to 736 defects

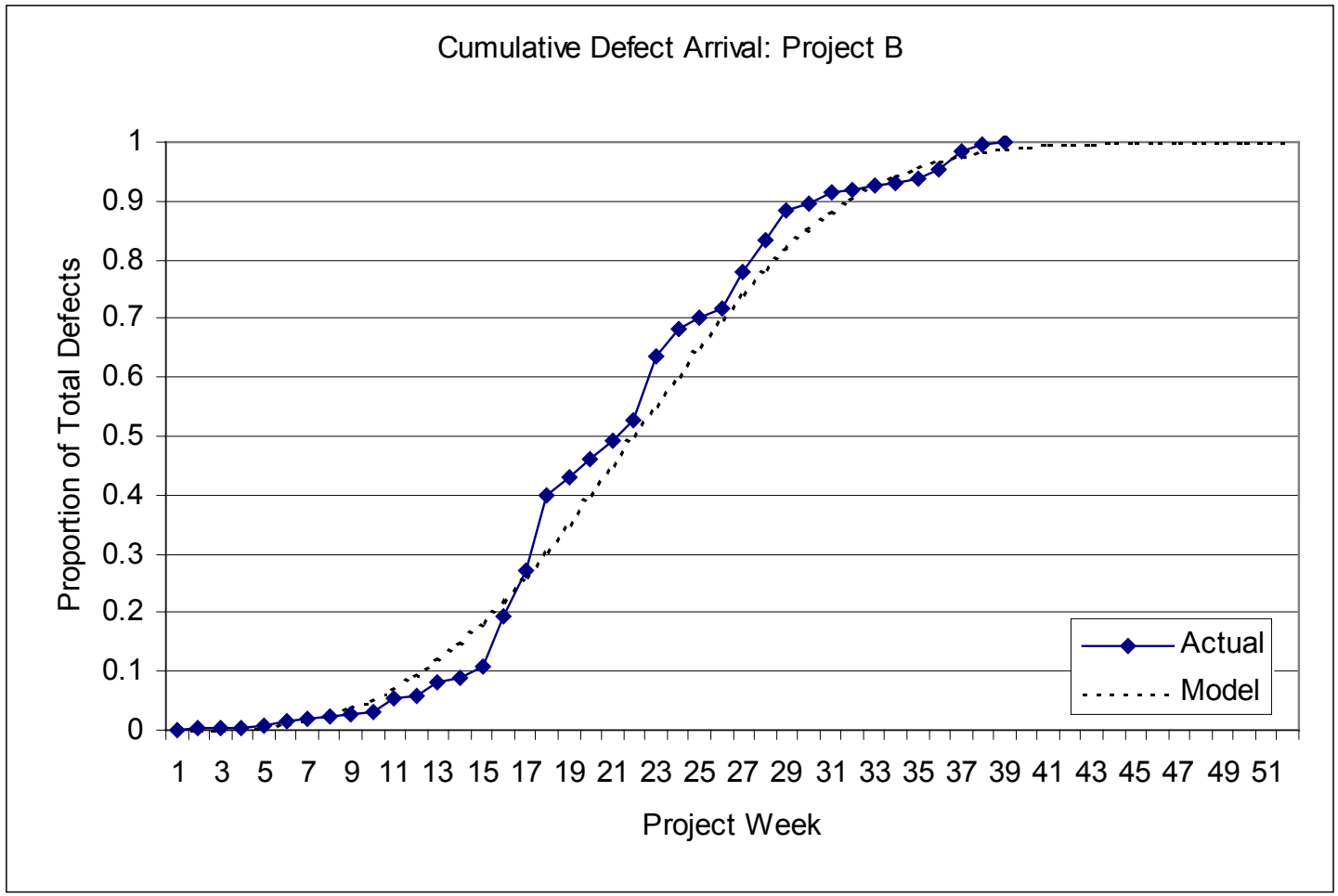
# Project B: Weibull Plot



# Project B: Weekly Arrival



# Project B: Cumulative Arrival



# Project C

Traditional two-tiered client/server application

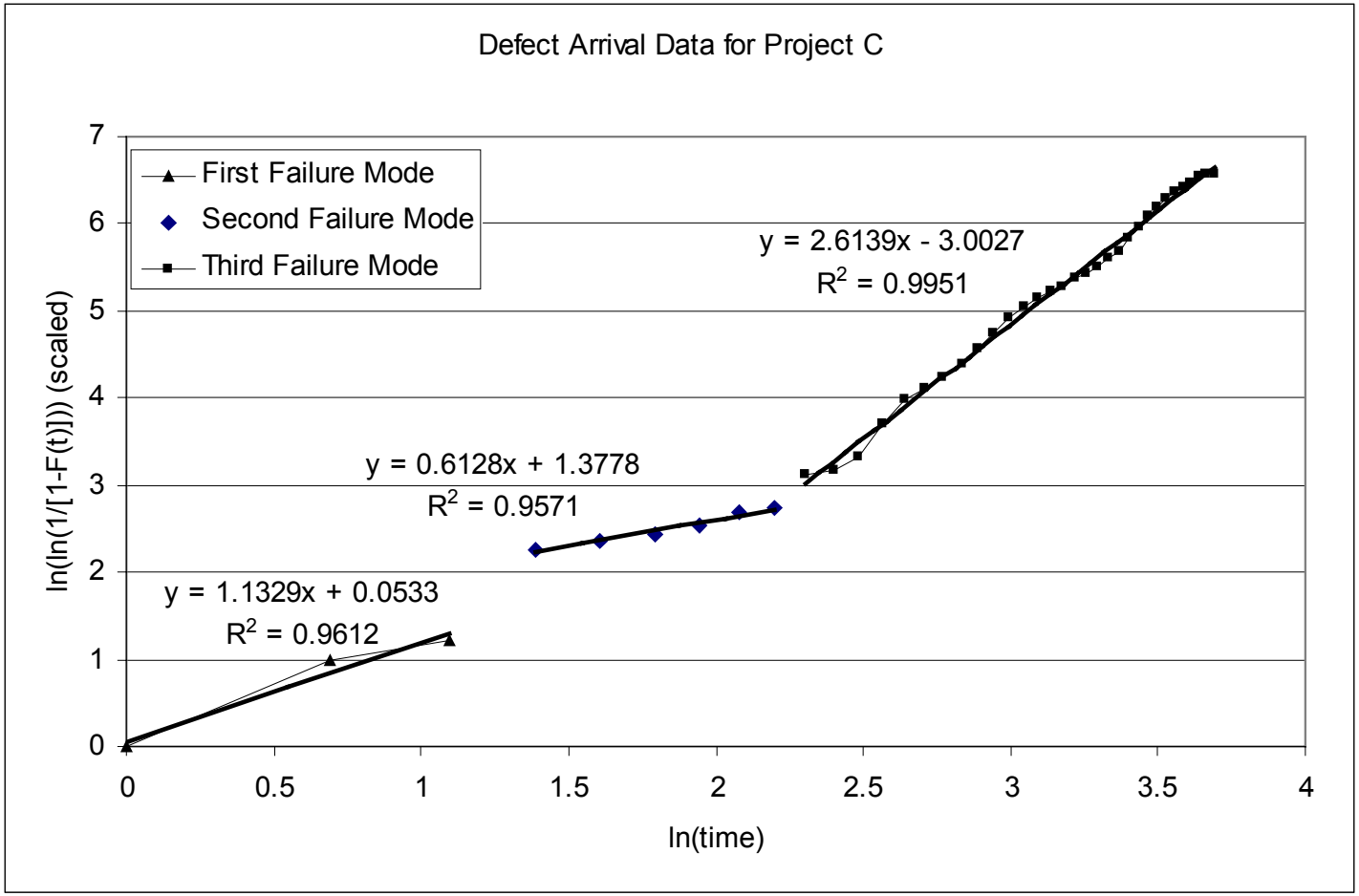
12 developers, 4-7 testers

Mix of mostly manual and some automated testing according to a extensive plans and specifications

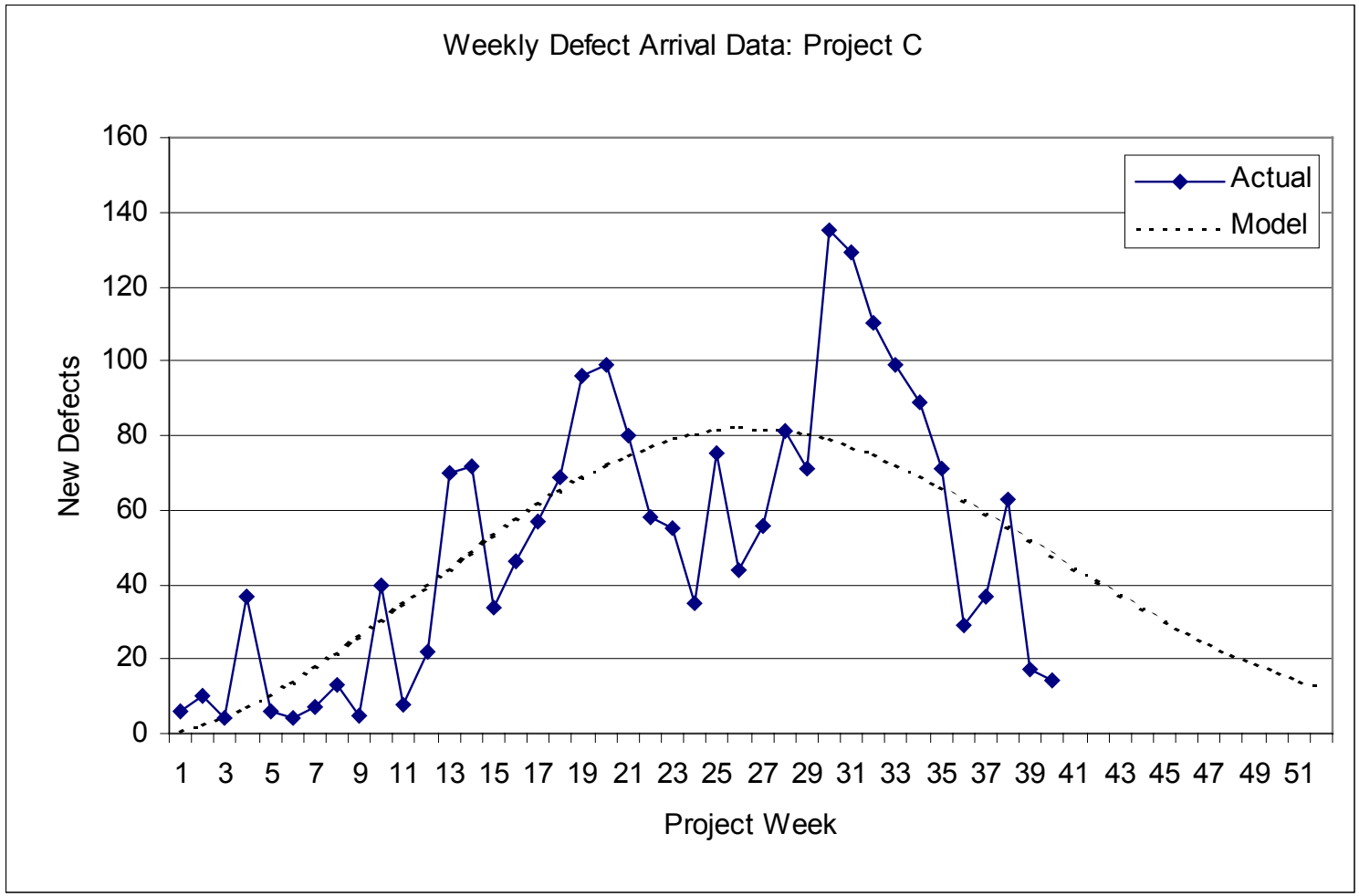
First Weibull fit to 2000 estimated defects

Total defect estimate raised to 2500 during testing

# Project C: Weibull Plot



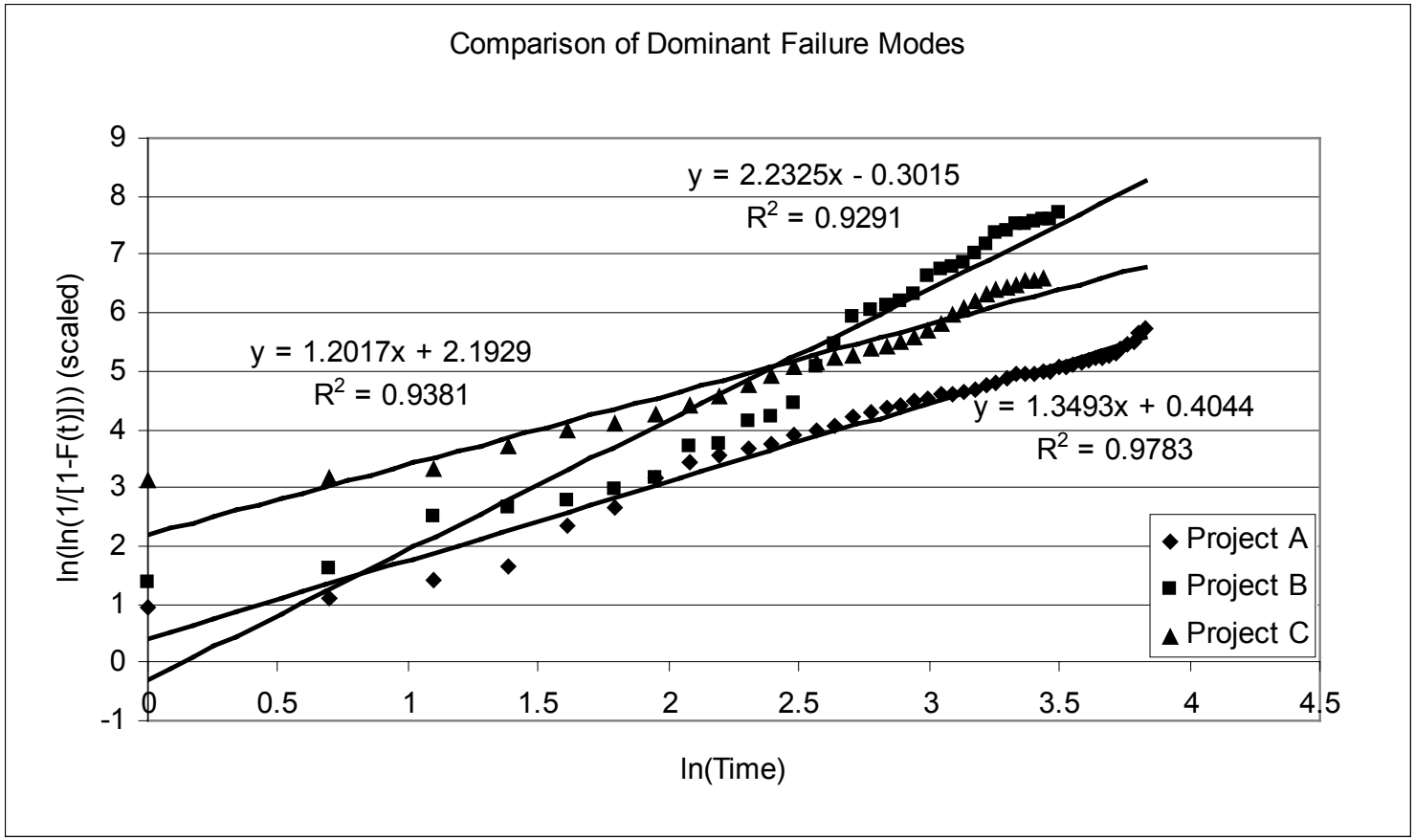
# Project C: Weekly Arrival



# Discussion

- Fits appear to be better when non-cosmetic defects are used
- 'Infant mortality' failure is common at the beginning of testing efforts
- Use case when comparing $\beta$ parameters between projects because of nonlinear axes
- Estimates appear to be stable enough to be useful only a few weeks after the main failure mode appears
- The project's lifecycle will influence defect arrival patterns

# Comparison of Main Failure Modes



# Possible Future Work

There are three main areas for more work:

- What factors influence $\beta$?
- Can the need for an estimate of total defects be removed?
- Are results significantly better with execution time domain and/or time between failures data?

# References


| | |
|---|---|
| Abernethy98 | Abernethy, Dr. Robert B., *The New Weibull Handbook, 3rd ed.*, Self-published 1998 |
| Jones97 | Jones, Capers, *Software Quality – Analysis and Guidelines for Success*, Thompson Computer Press 1997 |
| Kan95 | Kan, Stephen H., *Metrics and Models in Software Quality Engineering,* Addison Wesley 1995 |
| Lyu95 | Lyu, Michael (ed.), *Handbook of Software Reliability Engineering,* McGraw-Hill/IEEE Computer Press 1995 |
| Musa87 | Musa, John, *et al., Software Reliability – Measurement, Prediction, Application,* McGraw-Hill 1987 |
| Putnam92 | Putnam, Lawrence, and Myers, Ware, *Measures for Excellence – Reliable Software On Time, Within Budget*, Yourdon Press 1992 |

# Meeting Tight Schedules through Cycle-time Reduction

Dennis J. Frailey
Co-presenter: Mary Sakry

**Abstract:** *Meeting schedules is often the most critical factor in the success of a software development effort. Yet cycle-time improvement techniques that are well understood in other fields are often unknown to software developers and software project managers. Many times the techniques are counterintuitive, so they may not be the first things you would think about when there's a schedule problem. This paper provides an overview of the problem and some techniques that can improve cycle-time for any software project.*

**Authors:**

Dr. Dennis J. Frailey is a senior fellow at Raytheon Company, an adjunct professor at Southern Methodist University, a senior consultant with The Process Group, an ACM Fellow, and a Senior Member of IEEE. He's been a software developer for over 35 years at the Ford Engineering and Research Center, Purdue University, Texas Instruments and Raytheon. He currently specializes in software management techniques and software process improvement. Dennis teaches a number of courses on software engineering and software engineering management for Raytheon, SMU, UCLA, the University of Texas, and The Process Group. He holds a BS in mathematics from Notre Dame and an MS and PhD in computer science from Purdue. Dennis can be reached by email at frailey@acm.org.

Mary Sakry has 23 years of experience in software development, project management and software process improvement including 15 years at Texas Instruments. In 1988, Mary was a member of a corporate Software Engineering Process Group (SEPG) and responsible for SEI CMM self-assessments across TI worldwide. She has a BS in Computer Science from the University of Minnesota and a MBA in Business Management from St. Edwards University.

## Background - Why Cycle-time is Important

Your software is far from ready, but it's almost time to ship. You work overtime to rush through the most important functions -- you skimp on testing, gloss over the quality assurance, and somehow get it out the door. Then you resign yourself to customer complaints, frayed nerves and a never-ending update cycle. The next project will be the same. Your customers keep wanting faster service and your competitors are delivering sooner than you think possible. Why is there never enough time to do it right?

Shortening cycle-time would give you a competitive edge -- if you could figure out how to do it. Develop faster and you have more time to do the job right. Deliver before your competitors do and they'll have to catch up while you work on the next release. The life expectancy of software products is typically short, so meeting tight schedules can mean more profit for your company. Even if you have no competitor, faster software development means you can accept more business opportunities. And for those who have no market-driven need to do it faster, just having the ability to do so means you can start implementation later -- so requirements will be firmer, with less time to change.

## Where Cycle-time Problems Come From - Frequent Causes

### *Three fundamental problems*

Several years ago I was asked to join a newly-formed cycle-time improvement team. We assumed we would all be doing time and motion studies and figuring out ways to squeeze more work out of employees. But we soon learned that the true causes of cycle-time problems have little to do with individual performance and a lot to do with process and organization issues. We found that the causes of project delays are pretty much the same as the causes of traffic delays -- in a long freeway commute, you encounter frayed nerves, plans that go awry because of unexpected delays, and an infinite supply of "other guys" who seem to move into the lane just before you do. Experts on cycle-time, such as Eliyahu Goldratt, Christopher Meyer and Michael Hammer (see reference list), offer various tactics for solving these problems but the causes all boil down to three fundamental problems: variability, overly complex processes, and bottlenecks and constraints. Variability results in some parts of the process being starved for input while other parts are generating more output than the next steps can handle. Performance becomes inconsistent and unreliable. Variability is a major reason why traffic jams occur - each driver wants to go at a different speed. Complex processes mean more work to do and more opportunities to make mistakes. Bottlenecks and constraints slow everything down. In order to fix these problems you need to study what *causes* them. The specific causes will depend on the specific process being analyzed, but let's consider some of the causes most commonly encountered with software processes.

### *Typical Causes*

As my team implemented cycle-time improvements, we discovered that over half of the unnecessary delays in our software projects were caused by seemingly innocuous things that we had ignored in the past. For example, our top three causes were incompatible tools and data formats, inefficient approval procedures, and failure to plan effectively. Figure 1 shows how

these three causes relate to the three fundamental problems mentioned above. Incompatible data formats add complexity to the process because you have to do extra work to convert. They also tend to cause process bottlenecks. Approval procedures are natural bottlenecks, so inefficient ones just make a bad problem worse. And since approvals sometimes happen right away but can often take a long time, they introduce variability. On one maintenance project, for example, the average customer complaint took three months to resolve, and actual resolution times varied from 1 to 6 months. About 40 percent of the average delay was traced to an arcane approval procedure that nobody had thought to question because it had been done that way for ages and was set up to address a customer policy that all signatures had to be concurrent. The procedure called for each of five managers to sign a particular document on the same day. Finding all five of them in the building on the same day was difficult and unpredictable. We fixed the problem by creating an electronic approval procedure so they could do the job no matter where they were. The result was a reduction of more than a month in average complaint resolution time with a variance of only about 15 days. The worst case was now 2 1/2 months instead of 6!

| | Fundamental Problems | | |
|---|---|---|---|
| Causes Found in One Organization | Variability | Overly Complex Processes | Bottlenecks and Constraints |
| Incompatible Tools and Data Formats | | ✓ | ✓ |
| Inefficient Approval Procedures | ✓ | | ✓ |
| Failure to Plan Effectively | ✓ | ✓ | ✓ |

Figure 1 - Relationship between Causes and Fundamental Problems in one organization

### *Misplaced priorities*

Some delays are simply a matter of wrong priorities. During a benchmarking visit, our team spoke with a group of people whose job was to prepare the shrink-wrap packages for a popular line of software products. The packaging process added two to three months to the software delivery schedule. The team could have reduced this to under a month by starting much sooner, but the chief programming guru, who had to approve the artwork for the boxes, would not pay attention to such “trivial” graphic decisions until the code had passed final tests. As a result, the customers' CDs sat there for months, ready to ship, while the artists prepared the final packaging. This example illustrates both a fundamental mistake and a fundamental principle. The *mistake* is that the programming guru was not focused on the big picture. He was only concerned about his part of the process. The *principle* is that you need to optimize the *entire process*. This often means being sub-optimal for the individual steps of the process -- a counterintuitive concept many find hard to swallow. Reward systems frequently go counter to this - imagine rewarding someone for NOT producing their maximum output because it would be detrimental to the overall system. If you understand the entire process, you may need to do exactly that!

## Spotting the Opportunities - Common Symptoms of Cycle Time Problems

### *Excessive Backlogs or WIP (Work in Process)*

Cycle-time problems are often usually easy to see. For example, a tell-tale symptom is a backlog of work, such as software requirements waiting to be translated into a design or designs waiting to be reviewed or code waiting to be tested. Backlogs are clear evidence of a cycle-time improvement opportunity. Work in process or WIP is a necessary part of any process, but simple queuing theory shows that the more WIP you have, the longer your cycle time (see Gross and Harris in reference list). Here's a simple equation that shows the relationship:

**Average Cycle-time = WIP / Throughput**

To shorten cycle-time, you must increase throughput and/or decrease WIP. However, it's hard to increase throughput without increasing WIP, so the smart approach is to reduce the WIP - the excessive backlogs in your process. But excess WIP is just a symptom. What is the underlying problem? What causes excessive WIP? It can almost always be traced to one or more of the three factors mentioned before: variability, complexity, and barriers or bottlenecks. From there, you can pin down specific causes. For example, suppose you notice that products are waiting to be tested but there is not enough test capacity. The products waiting are escessive WIP. The testing process is a bottleneck. The cause of the bottleneck might be poor planning, insufficient test equipment, insufficient staffing of the test process, or perhaps inadequate maintenance on the test equipment.

### *Rework*

A more fundamental symptom of cycle-time problems is rework. The more you do things over, the more WIP you have, which means you add cost and introduce delays. Much rework comes from simple things: rushing the work (which causes more errors), miscommunication (which may result in doing the wrong thing) and inadequate training (which means you waste time learning and making mistakes on the job). Measuring rework - and taking action to reduce it - is an important cycle time improvement technique.

## Reducing Cycle Time - Fixing the Problems

### *Customer Value*

You improve cycle-time by attacking the three fundamental problems, but you should "pick your battles." A good way to start is by defining the "value stream," that is, the sequence of things you do that really matter to the customer -- the things you must do to deliver the product. *Value-added analysis* is a formal process to do this, but you can accomplish a lot by just thinking about what is really necessary and what is not. Examples of the necessary, value-stream tasks include designing the software, writing the code, integrating the components and preparing the help files. Many things you do are not in the value stream: debugging, rewriting bad modules, waiting for approvals, translating between incompatible tools and correcting misunderstandings. By focusing on the value stream, you open your mind to the things that really matter. Everything else is ripe for streamlining or removal. Once you make a list of the tasks that waste the most time and

resources, you should prepare a plan to reduce or eliminate them. Then make another list and repeat. Before long, you will be pleasantly surprised at how fast your projects go.

What about testing, reviewing and managing your projects? Activities like these are not in the value stream. This may be hard to accept, but it is true. Often, these activities are essential for getting the work done, so they are given a special name: "non-value-added essential." Such tasks are targets for streamlining rather than removal -- at least in the short run. But if we get good enough, we can think about getting rid of them! (Why are these not part of the value stream? Because your customer would be just as happy if you could write perfect code with no management direction and no need for testing. Indeed, some day you may know how to do just that. After all, some Japanese cars have outstanding quality records even though they are not tested until they arrive at the dealer's lot.)

You should design your process to maximize efficiency of the value stream. That is what cycle-time improvement is really all about. In most cases, there are plenty of processes that don't add value and are not essential, so there is plenty of opportunity.

Every method of cycle-time improvement is a way to make the value stream as optimal as possible. It's a lot like code optimization, so try thinking like a programmer. Imagine your program has one primary execution sequence that must proceed as fast as possible (the value stream). Think about how you might optimize the software to make that happen. You would design the input and output to minimize delays. You would make sure there are no empty input buffers or full output buffers. You would optimize the code in inner loops. You would match your hardware and software for maximum performance. You would evaluate your algorithms to see if there is a faster way. Every line of code you remove will speed things up, especially if you choose the ones in inner loops.

Now, stand back and imagine your software development process as a program. You are the processor and the value stream is the primary execution sequence. Where are the I/O bottlenecks? What are the repeated processes, the inner loops? Is there too much delay getting something approved or through configuration control? Is there a long wait for test resources? Do you wait and wait for requirements specifications or approvals? What steps of your process could benefit the most from better hardware? Are you taking too long to do a critical design step because the workstations are not up to the task? Is there too much paperwork to purchase needed development tools? Is your development process just too complicated? Could you simplify it? Every step you eliminate reduces complexity and means fewer opportunities to introduce defects.

Here's an example of value stream analysis. An organization studied their process and concluded that the procedure for writing design specifications was costing a lot of money and time, while the customers complained that frequent design changes rendered the specifications out of date by the time they were printed. They concluded that customer value was accurate documentation of the design, not the paper specification document. So they eliminated production of paper design specifications and gave the customer on-line access to their design model, as it was being created. This resulted in a more satisfied customer, a faster and less error-prone development process, and a significant reduction in cost. Later, the design model replaced paper design specifications for maintenance purposes, with similar benefits.

### *Justifying the Solution*

Once you understand the causes, you must convince the organization to accept your solutions. An important ally is measurement. Do you know how much time you spend doing various software development tasks? Can you measure your cycle-times and rework? If not, this may be the best place to start. My team found that by measuring and quantifying we could make things happen -- improvement recommendations were backed up by facts so they were more likely to be accepted. For example, we convinced the company to change the approval cycle for new capital equipment because we were able to show how much money the delays of the approval process were costing. The amounts saved were in the hundreds of thousands of dollars.

Measurement has other benefits. When we started to measure where we actually spent our time, we were sometimes surprised. For example, the programmers typically felt they should reach agreement on the best coding conventions. This typically took several long meetings at the start of the coding phase of a project. But when we measured the cost and benefit we found that the time spent reaching the best solution was not justified. A "good enough" convention that was resolved in under an hour would serve just as well. We saved about a half man-month of effort on each project.

### *Some Counterintuitive Techniques*

Sometimes the actions needed to reduce cycle-time go against intuition. Goldratt's "The Goal" (see reference list) has some excellent examples. For example, consider the ***Cycles of Learning*** technique, which says it is faster to do a job in several small increments rather than to do it all at once. Intuition might suggest that it would be faster to do it only once. But by attacking a job in small chunks, you make mistakes and learn from them on the early cycles but perform at top speed in later cycles, when the problems are usually more difficult. Most of us recognize this as true for software development: iterative or incremental development is often the fastest approach, provided you learn to do better each time through. But we may be working for managers from other disciplines who do not understand this and think it makes little sense until they are shown why it works.

The ***Small Batch*** technique is also counterintuitive. We are accustomed to the concept of economy of scale, which says that large batches are more efficient because they require less overhead. But economies of scale don't always work. Why? Because large-scale economies are only realized when the requirements do not change and the process is close to perfect. In software development, requirements typically change often, due to changing market conditions and better understanding of what is needed. And our processes often need adjustments as we learn how to do them better. For example, suppose we need to design 1000 modules. And suppose we decide to design them all at once before we start programming. Errors in our design process could be reflected in many of our 1000 modules. If we discover an error during the coding process (or if, in testing, we discover a requirements problem that must be resolved and that affects 100 of those modules) we may have to re-design, re-code and re-test a large portion of the software. Small batches reduce the amount thrown away or reworked when things change. If we designed and coded 10 modules, learned about our mistakes, and corrected them, we would only have to re-do 10 modules. The next batch of 10 would likely have fewer problems, and the next batch even fewer. We might even be able to go to larger batch sizes later on. With software, we are

usually better off building small things and using a modular approach than trying to do everything in one large “waterfall” cycle.

The ***Smooth Flow*** technique achieves the optimal process by having each step proceed at the same pace, like boxcars in a train, rather than having each step go as fast as it can, like automobiles on a highway. Although each individual driver thinks he or she is best off going as fast as they can, having everyone go as fast as possible often does more harm than good because people end up getting in each other’s way. What does this mean for software development? You should analyze the flow through your development process, find the slowest points—the bottlenecks—and spend more of your time helping there, even if it means taking time away from design or coding tasks and helping out the people at the bottlenecks, such as in test or configuration control. This may go against the grain of organizations where job functions are compartmentalized, illustrating a previously mentioned cycle-time principle -- you must look at the total picture, not just your local part of it.

Naïve cycle-time improvement efforts often start with the obvious approach of trying to shorten the longest step in the development process. However, rather than optimizing locally, you need to look at the big picture. Ironically, sometimes you get better results by lengthening a process step, even though this may not seem to make common sense at first. A classic example is taking longer to plan, especially things like product integration, or to be more careful when you define requirements. Both of these can result in fewer problems and much less rework later. One successful project defined the integration-and-test plan first and gave control over all interface changes to the integration-and-test team. This reduced integration time by more than half.

Another cycle-time mistake is to cut out “overhead” activities, such as quality assurance and configuration control, because they seem to slow things down. Often, the reason cited is that these are not part of the value stream. (These may, in fact, be "non-value-added essential" tasks, but that is not the main point here.) Effective management of the value stream requires you to examine tradeoffs. Some non-value-added tasks cost more than others and some are needed to eliminate others. If you look closely, you may find that some "overhead" activities are worthwhile investments. They may, by cutting out rework, reduce cost and cycle-time by more than enough to justify themselves. How can you tell? Measure and see. My team found that quality assurance and configuration management tasks, when suitably optimized and integrated with the software engineering process, often saved many times their cost in both time and labor.

## A Few Lessons Learned - Things We Do that Hurt Cycle Time

### *Whom do you Reward -- and Why?*

Do you reward programmers who work late every night and wonder about those who leave early? This is a natural tendency. However, instead of looking at how busy developers are, maybe you ought to measure how productive they are -- that is, how much value they produce. When you do, you may find that some of the busiest people are spinning their wheels a lot. Not always, of course, but we found many cases where the real work was being cranked out by people we had once considered “too slow.” Sometimes the tortoise really does beat the hare.

### *Precautions from the Past*

Overly complex processes and bottlenecks often result from precautionary measures that protect against problems that are no longer likely or expensive. Company processes and procedures tend to get more and more complex as time goes by. Each time a problem arises, the bureaucratic tendency is to create a new rule. One company saved a lot of time and money by eliminating the requirement to maintain pseudo-code descriptions of software after it had been coded. The rule had been imposed in the days of assembly language programming when the code was not very self-documenting. Over the years, as good coding conventions and high level languages became more prevalent, the maintenance staff relied less and less on pseudo-code and more and more on the code itself. But it took measurement to show that the requirement for maintaining pseudo-code was costing money. And it took time to convince die-hards that it was not really needed for maintenance.

## The Payoff

By attacking the root causes of cycle-time problems, you can improve your delivery schedules permanently. When you deal with causes instead of symptoms, you save money and improve product quality as well. The techniques are not hard. You just have to apply basic principles in a methodical fashion and be open to new ways of doing your work. The biggest problem is often selling your "counterintuitive" ideas, which is where measurement really helps. If you are not sure whether to try cycle-time improvement, remember that your competitors are constantly striving to be faster than you.

## References


Frailey, Dennis J., "Reducing Cycle-time," Management Forum, *Software Development* (August, 2000).

Goldratt, Eliyahu M. & Jeff Cox, *The Goal*, (North River Press, 1984). Also, *Theory of Constraints* and *It's Not Luck*.

Gross and Harris, *Fundamentals of Queueing Theory*, Wiley, pp 10-11, 101-102

Hammer, Michael & James Champy, *Reengineering the Corporation, - A Manifesto for Business Revolution* (Harper Collins, 1993).

Meyer, Christopher, *Fast Cycle-time* (Free Press, 1993).

# Meeting Tight Schedules Through Cycle Time Reduction

**Dennis J. Frailey** - Senior Fellow, Raytheon;
Adjunct Professor, SMU
frailey@seas.smu.edu

**Mary Sakry** - The Process Group
help@processgroup.com

P.O. Box 700012
Dallas, TX 75370-0012

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

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

# Outline

- **Background - Why Cycle Time is Important**
  - Cycle Time Reduction Issues and Examples
  - Defining Cycle Time
- **Three Fundamental Problems**
- **Typical Causes**
- **Spotting the Symptoms and Opportunities**
- **Fixing the Problems**
- **Justifying the Solution**
- **Counterintuitive Techniques**
- **Lessons Learned**

# Why Cycle Time is Important

# Why Cycle Time is Important

- Short Cycle Time Gives You a Competitive Edge
  - You sell your product while the competitor is still completing theirs
  - You start the next product while the competitor is still completing the current one
  - Lower your costs
  - Or start development later in the program cycle
  - And allow less time to change requirements

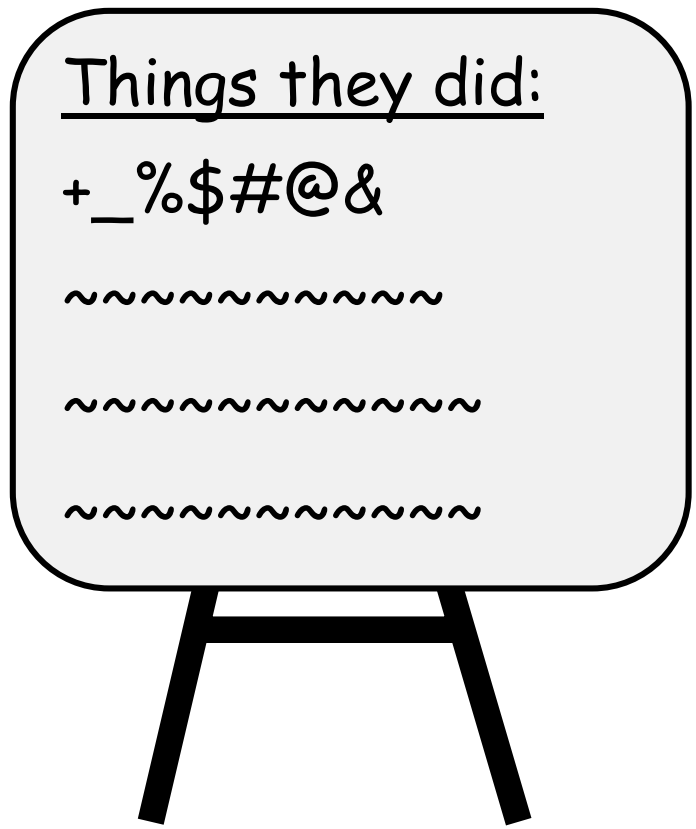
# How can Cycle Time be Improved?

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

**What did they do to reduce cycle time?**

# Some Lessons from the Cycle Time Video

- What did they do?
- Is there a software counterpart?



# Definition of Cycle Time

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

**Consider a “10 minute” oil change**

# How do You Measure Cycle Time?

## STATIC CYCLE TIME

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

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

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

  **… so we need a dynamic measure**

# Dynamic Cycle Time

## DYNAMIC CYCLE TIME

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

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

WIP = Work in Process

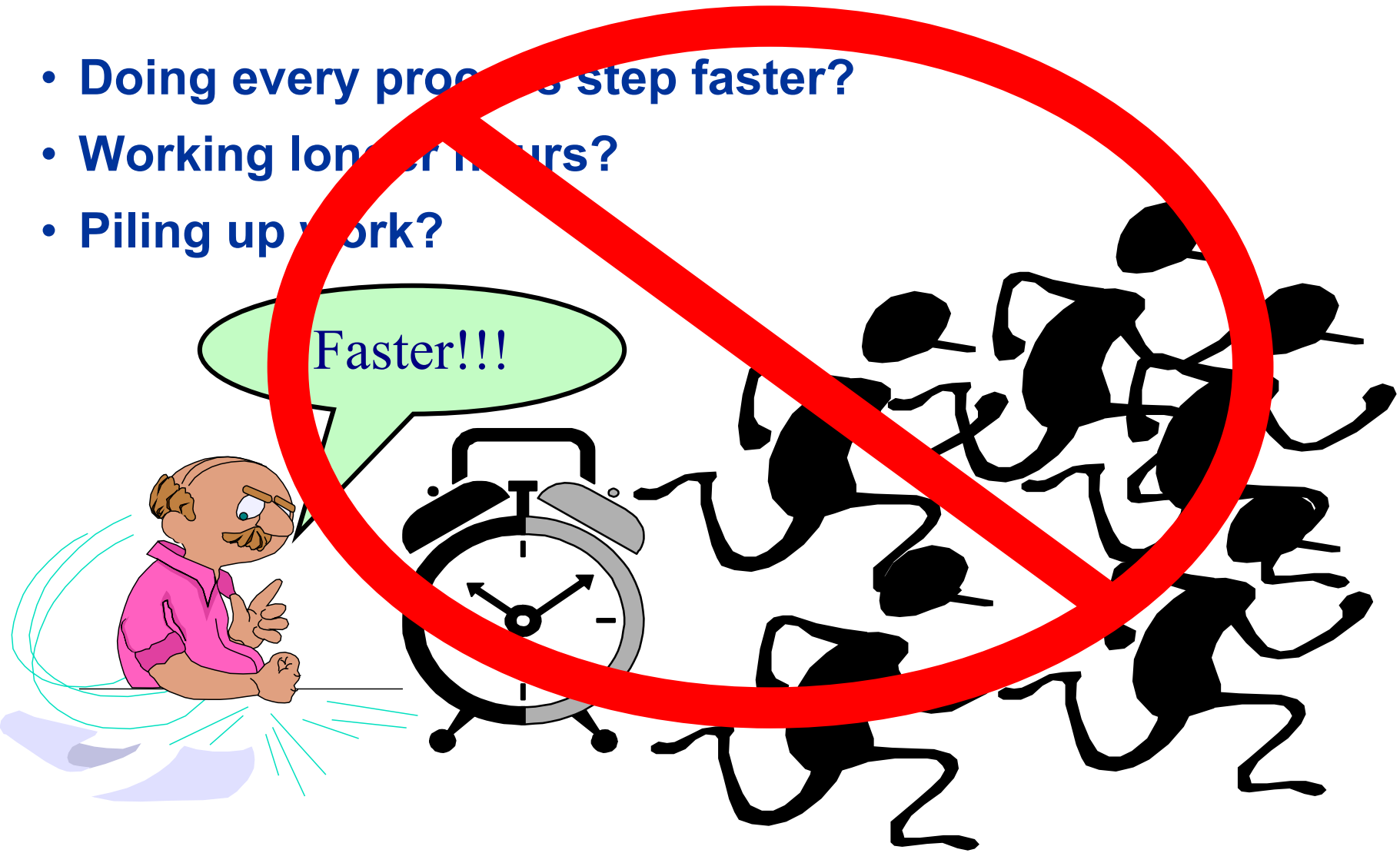
For related background, see Gross and Harris, in reference list of paper, p83.

# How is Cycle Time Improved?

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

# How is Cycle Time Improved?

- Doing every proc s step faster?
- Working lon r h urs?
- Piling up ork?



# Three Fundamental Problems

- **Variability**
  - Some parts of the process are starved while other parts are producing excessive output
  - Performance becomes inconsistent and unreliable
  - This is what causes traffic jams!
- **Complex Processes**
  - More work to do than is necessary
  - More opportunities to make mistakes
- **Bottlenecks and Constraints**
  - Things that slow everything down

# Typical Causes of Cycle Time Problems

| | Variability | Complexity | Bottlenecks |
|---|---|---|---|
| Misplaced Priorities | ✓ | | ✓ |
| Incompatible Tools | | ✓ | ✓ |
| Inefficient Procedures | ✓ | ✓ | ✓ |
| Poor Planning | ✓ | ✓ | ✓ |
| etc. | ✓ | ✓ | ✓ |

# Principle #1

# Look at the Entire Process

- Don’t optimize only your local part of the process
- Speeding up every step of the process will cost a lot and will not help as much as speeding up the bottlenecks
- Beware of inappropriate reward systems!

# Spotting the Symptoms and the Opportunities

- **Symptom: excess WIP or "work in process"**
  - work waiting to be done that is not being done -- waiting in queues instead
  - something is holding up the process
- **Causes: limited capacity, poor processes, poor execution, or various barriers imposed by the organization**

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

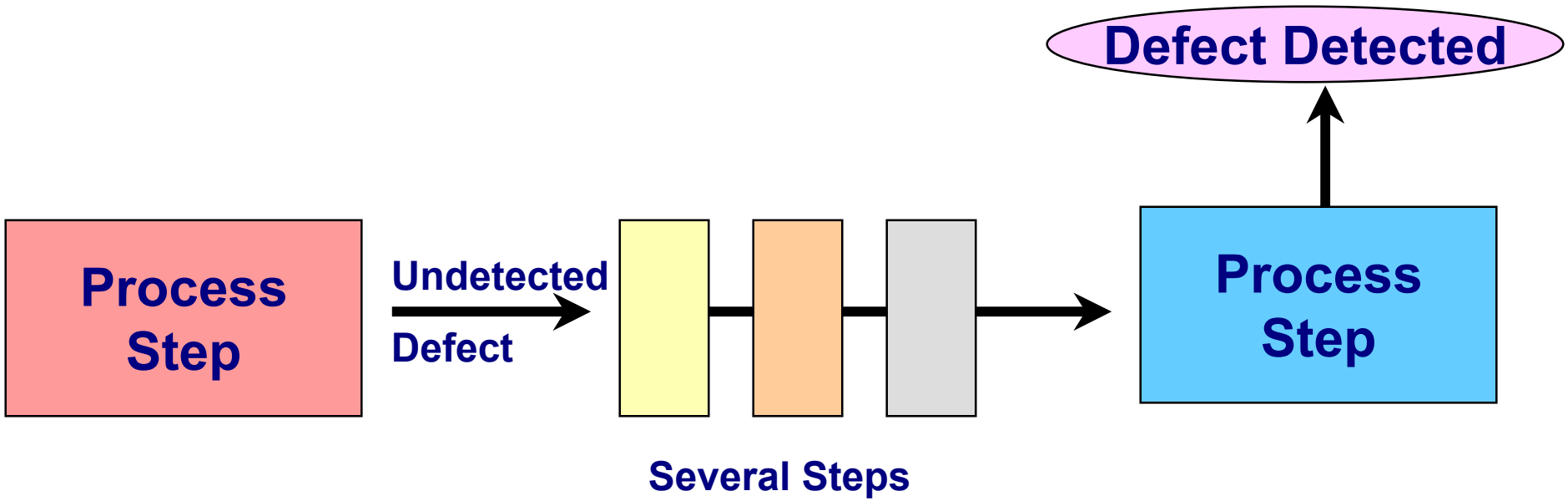
# Examples of Excessive WIP for Software

- Code waiting to be tested
- Designs waiting to be coded
- Specifications waiting to be inspected
- Change requests waiting for approval
- Hundreds more...

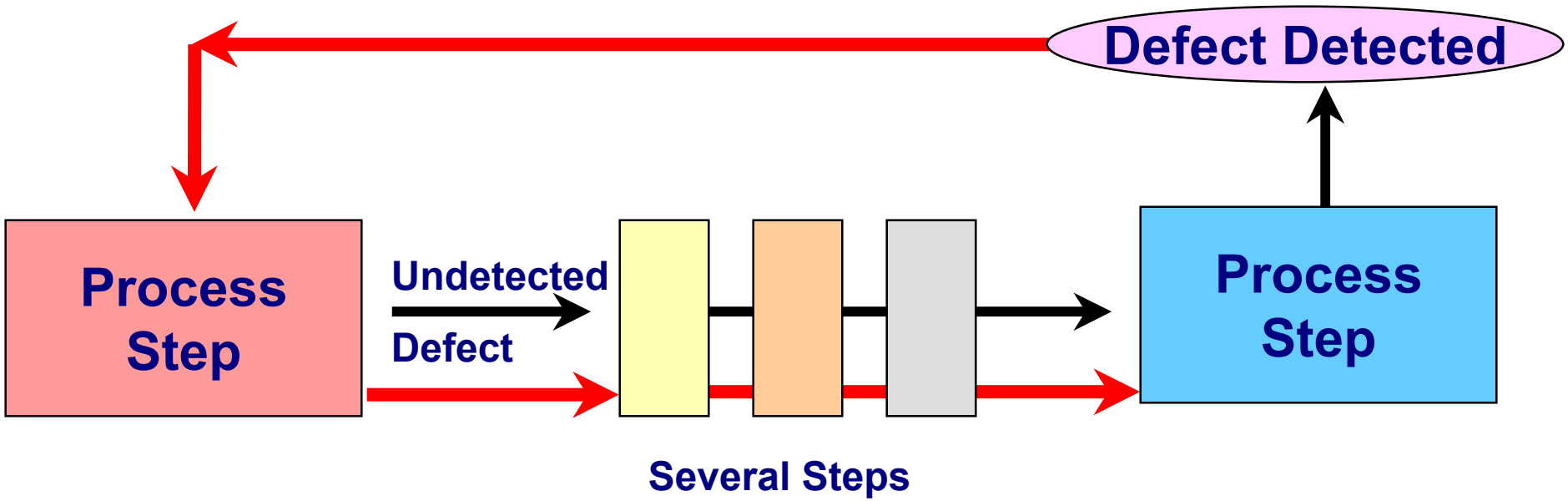
# Other Symptoms of Cycle Time Problems

- Long waits and queues
- High inventory levels
- Excessive overtime
- Rework / scrap

# Reducing Rework - The Impact of Defects on Cycle Time



# Doing it Over Again




**Rework costs money and time**

# Look for Rework

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

# Where are the Opportunities?

- **~30% of the improvement comes from technical changes**
  - Process changes
  - Tool changes
  - Changing rules and operations

- **~70% of the improvement comes from organizational, cultural and environmental changes, such as**
  - Education
  - Communication
  - Management
  - Teamwork

# Fixing the Problems

# Customer Value

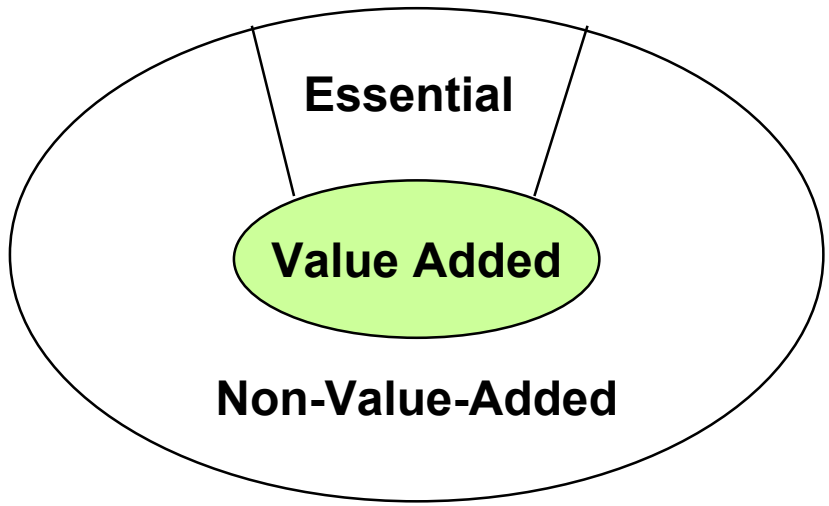
- What matters to the customer?
- Which things do you do that the customer does not care about?

# Value Added Analysis

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

# Non-value-added Essential

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




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

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

For example:

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

# Then You Work on the Non-Value-Added Essential

- Statistical process control techniques to reduce testing, reviews, etc.
- Reengineering the process to make it fundamentally more efficient

# Justifying the Solution

## The Key is to MEASURE

# Things to Measure

- What it costs to follow your process
- The value produced (benefits) of each step
- What it costs for the rework and other non-value-added activities
- How long it takes to do each step
- How much time and cost can be saved by changing the process or the organization

# Other Benefits of Measuring

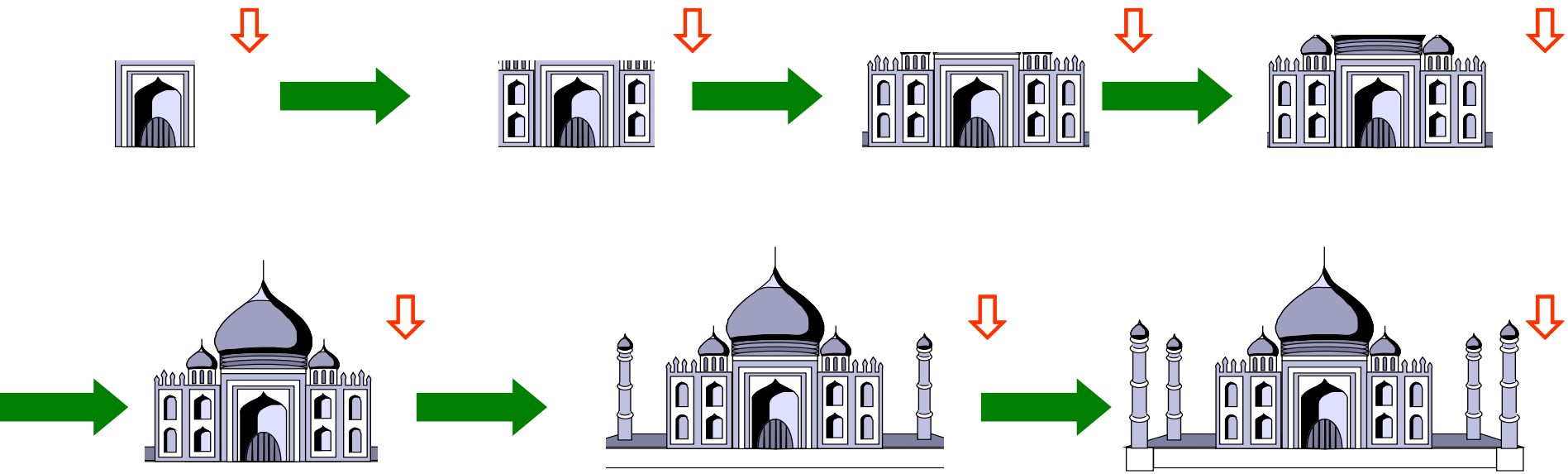
- **You replace intuition with facts**
  - Management by fact is the central concept of SEI CMM level 4
- **You learn things that surprise you**
  - Your intuition is not always right
- **You can justify your proposed solutions**
- **And you can discard good ideas that don't pan out**

# Counterintuitive Techniques

# Principle #2 - Use Cycles of Learning

- **Sometimes it is better to do the job many times, in small chunks, than to do it all at once**

Changes are received and processed here (⇩)

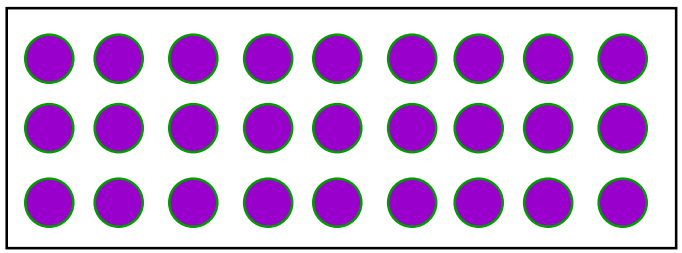


# Ways to Implement the Cycles of Learning Principle

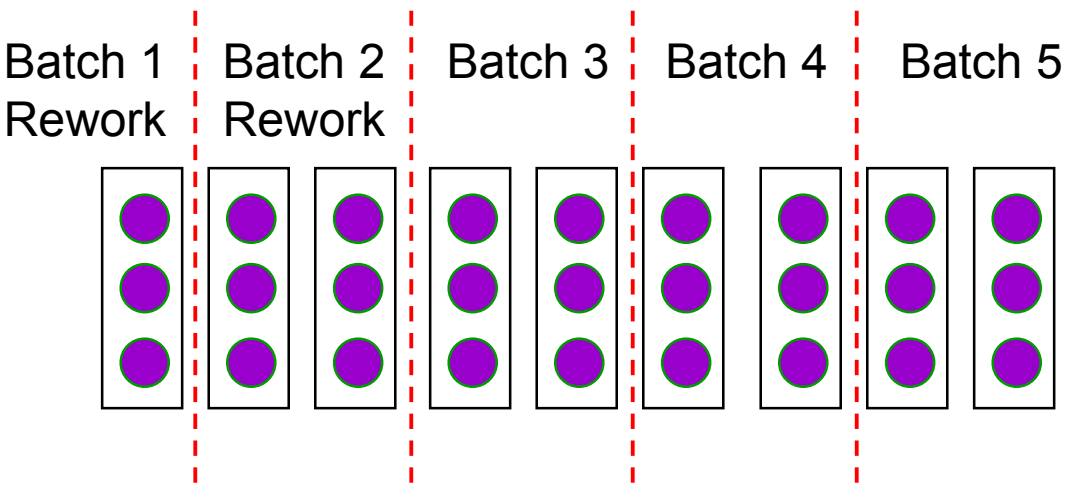
- **Try new compiler or configuration management tool tool on one module first to see how well it works**
  - Then decide whether to use it on more modules
- **Write one module using the new language**
  - Then decide whether to use the new language on a larger scale
- **Take one subset of the features through the whole process flow first to work out the quirks**
- **Test server performance using dummy data**

# Principle #3 - Use Small Batches

- **Important when requirements change a lot or the process is new**



The work to be done

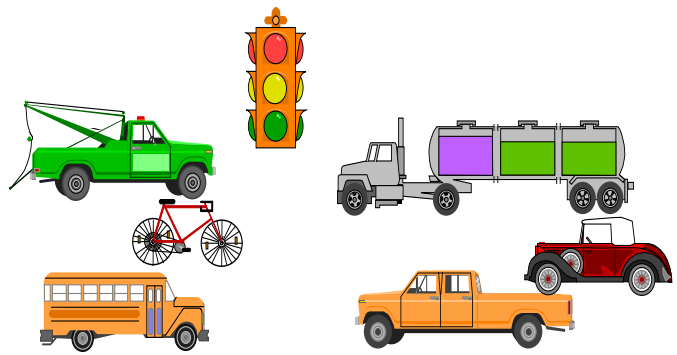



Small batch principle

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

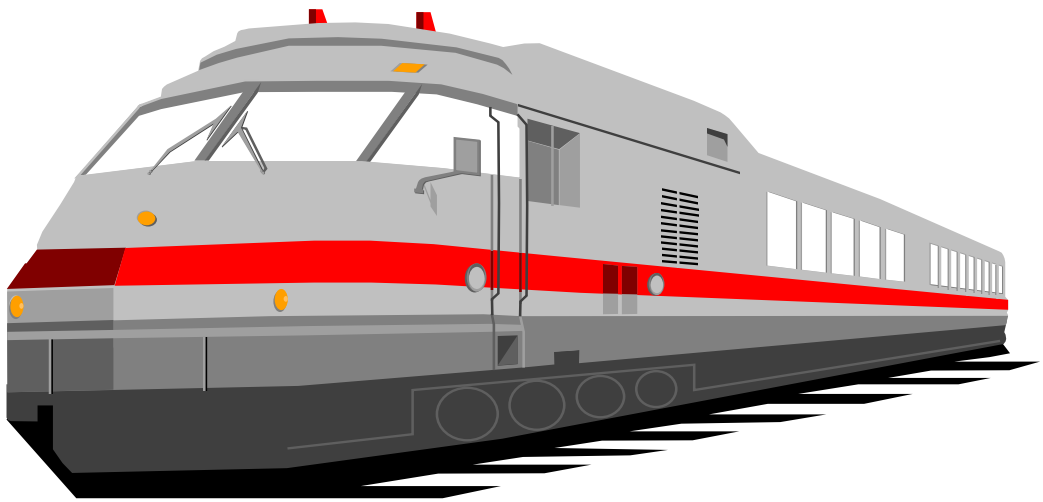
# Principle #4 - Achieve Smooth Flow

- **The typical process runs unevenly, like vehicles on a city street**
  - Lots of entrances and exits
  - Vehicles of different sizes and speeds
  - Some drivers uncertain of what they want to do
  - Lots of stoplights to “control” the flow (mainly to prevent collisions)
  - Note: streets are usually crowded



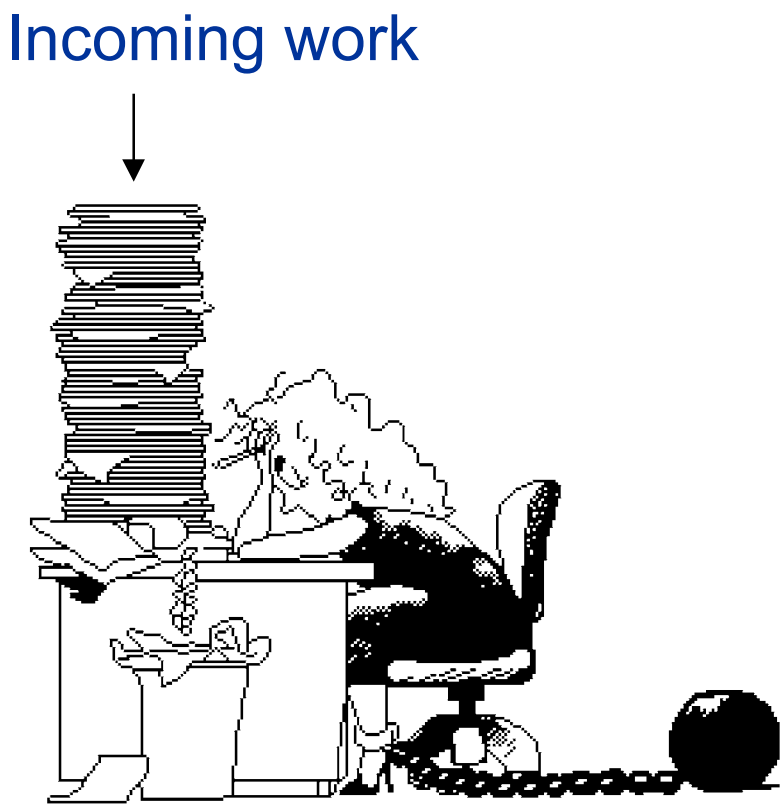
# The Ideal is Smooth Flow

- **The ideal process flows smoothly, like a train running on tracks**
  - Note: tracks are empty most of the time



# What Prevents Smooth Flow?

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



# Principle #5 - Avoid the Naïve, Obvious and Wrong Solutions to Cycle Time Problems

- Shortening the Longest Step of the Process
- Shortening Every Step of the Process
- Cutting the Overhead without Assessing the Impact

# Lessons Learned

# Be Careful Whom you Reward

- Know the Difference between ***Busy*** and ***Productive***
- Examine the ***Value*** ***Produced***, not the effort spent
- The Tortoise sometimes Beats the Hare

# Question Excessive Bureaucracy

- **Every problem, big or small, tends to generate an excessive, bureaucratic response**
- **Eventually, the bureaucracy strangles progress**
- **You must "clean house" or "re-engineer" now and then**
  - Question why
  - Measure
  - Challenge sacred cows

# Don't Measure Too Much or in Too Much Detail

- **The tendency is to measure too much and in too much detail**
  - Rough estimates usually identify the biggest problems and opportunities

# Independent Observers May See Problems and Opportunities the Best

- **Practitioners generally focus on their work and on what they THINK is happening rather than on what IS happening**
  - They tend not to see all of the waits, queues, etc. that they cause themselves
  - Their perception of how they spend their time is generally incorrect
  - They are too busy getting the job done to see how they might improve it

# The Athletic Coach Analogy

- Just as athletes rely on coaches, software engineers need to learn to trust in others to observe and help them do better

# Software Developers Are Accustomed to Improving Cycle Time

- Think of your software development process as a large computer program that runs too slow. How would you make it run faster?
- Imagine how you would speed up a computer program …….
- Then draw analogies to the software development process …
- And improve the process the way you would improve a program

# Cycle Time Bonus

**It Generally Turns Out That Improved Cycle Time Produces Lower Costs and Higher Quality As Well!**

- You don't pay for the steps you don't do
- Fewer steps means fewer opportunities to introduce defects
- Shorter cycles means less time to change requirements
- Shorter cycles means more time to iterate designs or benefit from cycles of learning

# Summary

- **Background - Why Cycle Time is Important**
  - Cycle Time Reduction Issues and Examples
  - Defining Cycle Time
- **Three Fundamental Problems**
- **Typical Causes**
- **Spotting the Symptoms and Opportunities**
- **Fixing the Problems**
- **Justifying the Solution**
- **Counterintuitive Techniques**
- **Lessons Learned**

# Conflict Management in Software Development Environments

Presented by Lisa Burk and Jean Richardson, authors; Lisa Latin, contributor

## Why This? Why Now?

Conflict management is one of the next frontiers in software development productivity improvement. Given the rate of technological change over the last three to five years and the projections for the continuance of this trend, perhaps you find it strange to highlight *conflict* as the next frontier. Or, perhaps you have recently experienced a project or corporate failure that was due to conflict between *people*, as opposed to software components, and you are not at all surprised.

Usually it takes something like a catastrophic failure or a continuing underperformance in the market to get us to pay attention to the obvious. Poor conflict skills at all levels of the technology development environment are the basis for tremendous loss of revenue every year. We executed this survey of the literature on conflict as it relates to technology development environments, particularly software, as part of the preparation for the workshop entitled "The Link Between Creative Product Development and Conflict Management: Problem Solving for Project Teams" to be offered at the Pacific Northwest Software Quality Conference 2000. This paper provides a context for that workshop. Based on our observations of the industry, focusing on conflict management skill building is an idea whose time has come.

Poorly managed conflict is related to impaired productivity. Employees spend more time in meetings trying, usually unsuccessfully, to make decisions – any decision, let alone a good one. The meetings end, the employees go back to their offices or cubes and shortly reform in virtual or actual knots of combusting humanity, rehashing the meeting, reviewing the problem, placing blame – and wasting precious time and bandwidth.

They rehash and blame because they either don't know how, or don't feel that they can, do anything else. And while they are paralyzed with indecision, frustration, rage, and fear, they're not focusing on getting the product to market with any degree of efficiency. They're wasting time and money and, quite possibly, polishing their resumes. Maybe they'll make a quick call to the shop down the road with five positions open for folks just like them, and tell their buddies on the team about the prospective financial packages available if they bail out on this lousy project that isn't going anywhere anyway. Or, just as likely, they're sitting and churning, developing chronic colds and flu, not to mention ulcers, heart disease, hypertension, and migraine headaches – all of which translate into recurring paid time off, again impeding the project.

We all know this. If you've been around the block for even a couple of years, you've probably spotted the pattern. But what's the cause? Skillful conflict management is not part

of the dominant high tech culture. Technology development is still largely a U.S. phenomenon, and the dominant culture is known to be short on good conflict skills.

From good old Daniel Boone, who felt compelled to move on whenever he saw the smoke from the neighboring cabins, to weapons packing, gang leaders at the local middle school the hard, cold, isolate individualists that Thoreau wrote about have made this country great through their highly competitive brand of capitalism. They have carved out and legitimized a way of working that has little to do with collaborative problem solving and very much to do with conquering the guy next door, even if he's ostensibly on your team. Because, in today's technology development environment, today he's on *your* team, next month he's on your *competitor's* team.

Improving conflict management skills across the workforce holds great potential for increased productivity. When people know how to operate effectively in the face of the conflicts that will inevitably come their way, they can more easily make decisions quickly. Those decisions will more often be good decisions, stripped of the baggage that accompanies dysfunctional conflict. *Engaging in conflict effectively, or functionally is the key*. Eradicating conflict from the environment is not only impossible, it is undesirable. It is no better than leaving it unaddressed to run rampant through the team. Technology development is all about creative problem solving; a certain level of conflict is essential to any creative endeavor.

## Overview of the Paper

This paper reviews the current literature on:

- Conflict response styles and their subsequent relationship to successfully managing conflict.
- The financial cost of conflict in technology development organizations, particularly as that relates to retention and project failure.
- The current thinking on ADR (Alternative Dispute Resolution) system design in organizations.
- How systemic and targeted ADR solutions can most effectively address conflict management issues in the current market and drastically cut the costs of conflict in your organization.

This paper includes an annotated bibliography of sources as well as an interpretation of the research. In addition, it presents the data upon which the workshop, "The Link Between Creative Product Development and Conflict Management: Problem Solving for Project Teams" is based.

We hope it in the remainder of this paper to raise your awareness of the connection between conflict management strategies and high tech project success and stimulate discussion around conflict management in technology development environments.

## Responding to Conflict

They say it takes two to tangle. Conflict can exist in any given situation without it being necessarily dangerous to relationships, communication, or productivity. However, we often equate conflict with damage and negativity. In a classic work on mediation, Folberg and Taylor give us two reasons why conflict can be damaging.

"Although conflict is not necessarily bad, wrong, or intolerable, our society often views conflict negatively because it is equated with win/lose situations….and conflict is commonly viewed by the participants as a crisis. A crisis mentality lends itself to destructive processes because people will often rush to use anything (usually not the best process) they believe will relieve the conflict."[1]

But to prevent damage, it's essential that at least some of the participants in the conflict respond in a productive manner. The more team members there are who are skilled in responding to conflict, the more likely the conflict will be productive and short-lived.

Conflict is also very complex even though it may appear very simple on the surface. Underneath every conflict is a host of possible competing and compatible reasons for the conflict. Fisher and Ury, outline some possible underlying reasons why people may be in conflict.

- Form vs. Substance
- Economic Considerations vs. Political Considerrations
- Internal Considerations vs. External Considerations
- Immediate Future vs. More Distant Future
- Ad hoc Results vs. The Relationship
- Hardware vs. Ideology
- Progress vs. Respect for Tradition
- Prestige and Reputation vs. Results[2]

The list above illustrates why we often begin believing we are in conflict about the same thing, but we end up fighting over apples and oranges.

We've researched the literature on conflict response styles and their subsequent relationship to successfully managing conflict, paying particular attention to software development environments. For a complete listing of sources consulted, see the attached annotated bibliography.

As we did this research, we considered the following:

- A certain level or volume of conflict is needed in an organization to motivate change and encourage creative thinking. Conflict is inevitable, and it can also be beneficial.
- There are particular styles of responding to conflict that are successful in any given situation. The skill in managing conflict is to assess the effective response given the situation.
- The degree to which technology development personnel as a population can be characterized as both highly competitive and conflict avoidant.
- How people can gain conflict management skills.
- That team members perceive a relationship between skillfully dealing with conflict, job satisfaction, and high performance.

## Conflict and Creativity

A certain amount of conflict is needed in an organization to motivate change and encourage creative thinking. Several sources speak to the issue of the importance of allowing conflict to flourish to stimulate creativity.

In *Managing Technical People* Watts Humphrey notes that "Nothing can destroy the effectiveness of a team more quickly or more completely than unresolved conflicts between the members. Inevitable differences and disagreements will crop up in any fast-paced organization, but the members themselves can generally work them out. Occasionally, however, the problems are too complex or pervasive, and a highly destructive process ensues. Rather than face the continuing unpleasantness, the disagreeing parties start to avoid each other. The reduced contact that results causes a total break in communication. This temporarily reduces the unpleasantness, but it also makes it almost impossible for the parties themselves to resolve the problem."[3]

Though communication is often fostered by standard organizational practices such as team meetings, status reports, and the like, human communication is a very subtle thing, and robust communication cannot be coerced. The most important subtleties of communicating and relating to others typically attended to by proactive team members drop away when unacknowledged conflict is present. For example, the very thought of approaching Bob regarding a design change in the new calculator interface becomes abhorrent to Jane as she remembers how she felt the last time she talked to him. She may not put the word "conflict" to her reluctance, but unacknowledged conflict is present. And, Bob doesn't find out about the design change until the application goes to field test, far too late for him to respond productively. He now has more fuel for the fire he intends to build under Jane.

The first rule of resolving conflict at any level, interpersonal to intergovernmental, is to keep the lines of communication open, even if shuttle diplomacy using a third party is required. Humphrey goes on to say, **"**Under these circumstances outside intervention is generally required."[4] This indicates the need for trained mediators to be engaged as a means of reestablishing communication between disputing team members. Humphrey further states that third party conflict resolution can be helpful because the mediator can help move parties off their polarized positions, can help the participants see what they both have to gain from reaching an agreement and can help reduce the stakes so that nether party feels pressed to give up too much. Mediators can also help people in conflict surface the underlying reasons for their conflict (see the Fisher and Ury list above) which can broaden the number of settlement options.

Additionally, Humphrey contends "no conflict between team members can be resolved by dealing with them separately. They both must be equally involved and must openly accept the conclusions. Since silence is not a reliable indicator of consent, both parties should air their opinions and restate the final agreement."[5] A skilled facilitator is required when either party is in any way conflict avoidant or hesitant to engage in direct communication.

Regarding conflict and performance, Humphrey states "No group of active and intelligent professionals can function for very long without generating friction of some kind. Disagreement is natural, and…it often stimulates performance. When a group of professionals are personally compatible but intellectually competitive, friendly rivalry generates the highest overall group performance. When the disputes become personal, however, performance invariably suffers."[6]

Humphrey's theory of "contention management" indicates that managing conflict is an ongoing process that "…leads to the most effective group performance."[7] Contention management makes sure all decisions to be made are "out on the table." People who disagree with a decision are responsible for escalating their opinion in the team, and if there is no opposition to a decision, the issue is deferred until sufficient opposition and contention is voiced.

According to Humphrey "The reason contention is so effective is that it both exposes the organization's latent conflicts and helps keep the discussions on a rational plane. When decisions are made in secret, the debates become political, and a feeling of distrust invariably develops among the top managers (and team members)."[8] This approach works well in an environment where most or all team members respond well to conflict. However, without good conflict skills, dialog can quickly devolve to argument and debate.

This seems to be the ideal form of conflict resolution to Humphrey, but we question how management by contention works for team members and managers who are conflict avoidant or who see expressing disagreement as disrespectful or unpleasant, as this author articulates in an earlier chapter. As we will show in a later section (see Effective and Ineffective Responses to Conflict on page 12) Gobeli and Koenig indicate that confrontation

(which we map to collaboration) and give and take (which we map to compromise) will likely increase team member satisfaction, thereby proving a more effective response to conflict.

In "Corporate Culture in Internet Time" Art Kleiner discusses the case of an e-commerce consulting firm that he feels typifies the difficulty of managing conflict in small technology development firms. In the case he cites, conflict has permeated the organization and has even leaked outside to the customer. He recounts the woes of one of his former students as she tries to manage the disputes in this small start-up.[9]

Kleiner postulates that there are two cultures present in every e-commerce organization; he identifies these as the cultures of "hype" and "craft". In this analysis, he draws a division similar to that found in William E. Souder's "Managing Relations between R & D and Marketing in New Product Development Projects."[10] Both authors point to what common sense indicates, that the personality and culture of the engineering department is markedly different from the marketing department.

The participants in the culture of hype are those who raise the capital, articulate new ventures, and sell the product. Kleiner finds this group in direct and necessary conflict with the participants in the culture of "craft". The culture of craft includes

> Programmers, designers and practitioners of the new profession of information architect . . . all artisans at their core. Like members of a craft guild, they like to delve deeply into a project, come to an understanding, and deliver an elegant solution. Even when individual artisans are sympathetic to the hype ethic (or when they stand to make millions from it), the culture of craft is innately persnickety, recalcitrant and suspicious. (It has to be, because craft work requires moving into a hype-free, reflective mental space where there is nothing but the hum of the work, where effort takes place on a semiconscious level, where the writing and design flow through the mind in a way that will not suddenly shift gears just because a new, more insistent client has arrived.)[11]

Kleiner then points out that software development is not exceptional in that these cultures have existed within new organizations for hundreds of years. It is exceptional in that these cultures are currently dominant.

> These cultures of hype and craft, of course, aren't limited to Internet and e-commerce businesses . . . Only in a few industries have hype and craft remained prominent as corporate cultures. Whether by coincidence or from some fundamental reason, these have included the most critical industries of the Internet Age: personal computers, Internet technology (but not mainstream telecommunications), software and media content.[12]

Some would argue that this is because the business of software development, though it has had a huge cultural impact so far, is still in its infancy.

The author holds out hope for reconciling these two conflicting cultures and points out why this reconciliation is relatively uncommon.

> The hype people, if they're smart, learn to protect the craft people, instead of draining them. They know that sophisticated craft people need clear boundaries within which to make choices time for reflection, and a chance to express their ideas. Similarly, smart craft people gradually learn, sometime during their career, to value the strategic primacy, and the groundbreaking audacity, of the culture of hype. They even learn to speak effectively to the strategists. But HTML and Java wizards in their early 20's typically haven't learned how to do that yet, and they won't learn it while under the gun of a Christmas Web site deadline.
>
> The result is debilitating and thoroughly unnecessary culture clashes. The management style that brings hype people success in their world makes it difficult for them to manage craft people effectively. [13]

Elsewhere in this same article Kleiner describes why and how a vicious cycle of intensifying conflict can build between these two cultures over the life of the organization.

As Gruber points out, we "often view conflict as the problem child…something to eliminate" yet "students of creativity often view conflict as it's necessary companion." We know that "conflict resolution requires collaboration, if not as the goal than at least as the means. Creative work has been treated, by and large, as an individual effort, sometimes painfully isolated." [14] This link is important to software development teams, as software team members must work with other team members from different professional backgrounds, and they all must creatively and collaboratively solve project problems on a daily basis.

Johnson, Johnson, and Tjsvold advocate constructive controversy as a necessary source of creativity. "Constructive controversy occurs when one person's ideas, information, conclusions, theories, and opinions are incompatible with those of another, and the two seek to reach an agreement."[15] "Constructive controversy tends to promote creative insight influencing individuals to (1) view problems from new perspectives and (2) reformulate problems in ways that allow new orientations to a solution to emerge." They have hit upon exactly the kind of problem solving required in software development. Further, the authors say "…constructive controversy increases the number of ideas; quality of ideas; creation of original ideas; use of a wider range of ideas; originality; use of more varied strategies; and the number of creative, imaginative, and novel solutions." [16]

Coleman and Deutsch also speak to the necessity of conflict in creative endeavors. "One of the creative functions of conflict resides in its ability to arouse motivation to solve a problem

that might otherwise go unattended." [17] Supporting the findings of Gobeli and Koenig, the authors state that "…a competitive, as opposed to cooperative approach to conflict leads to restricted judgement, reduced complexity, inability to consider alternative perspectives, and less creative problem solving."[18]

During the course of a project, team members tend to experience what might be characterized as "creative highs", those endorphin-laden periods that keep us working productively into the wee hours of the morning oblivious of the time, and "conflict lows", those points at which we "hate computers" and wonder what motivated us to join this house of horrors. Coleman and Deutsch help make sense of this experience. "Tension is the primary link between conflict and creativity. Conflict signals dissatisfaction with something or someone. This dissatisfaction brings tension into the system. If standard approaches to reducing tension are ineffective, it increases. This increase can eventually motivate people to seek new (creative) means of reducing the tension."[19]

Coleman and Deutsch point to using someone skilled in managing conflict to nudge participants stuck in an unproductive conflict into a more creative space. "A third party, such as a mediator, can bring new thinking (creativity) into a stuck conflict. Also, by making the parties aware of their potentially "creative" differences in what they value, their expectations, their attitude toward risk, their time preferences, and the like, we help them see that their differences can facilitate mutually satisfactory agreement."[20]

William Bridges also addresses the connection between creativity and conflict in *Managing Transitions: Making the Most of Change*. He distinguishes between change and transition (transition being the human process, or response to change), and calls attention to "the neutral zone", the place in between the ending of the old and the beginning of the new. "Transition is like a low pressure area on the organizational weather map. It attracts all the storms and conflicts in the area, past and present." [21]

He argues that change happens, and the related transitions must be well-managed in order for the change to "take". He draws our attention to "the neutral zone" as a critical aspect of change, one that is often ignored. He contends that individuals, organizations, and even whole nations deal with the discomfort of the neutral zone during times of change. In an organization, the neutral zone can be described by rising anxiety, falling motivation, and the re-emergence of old weaknesses and disputes. However, Bridges also points out the more positive opportunities inherent in the neutral zone, most significantly, that it is a time that is "ripe with creative opportunity." Properly managed, it can be mined for creative solutions and new innovations. He goes on to offer a laundry list of management suggestions, as well as commentary on managing in a world of non-stop change.

So, given that conflict is necessary to the creative process and software development is heavily laden with demands for creative problem solving, why do we need to manage conflict? Can't we just bulldoze our way through a conflict and figure that those who can't take the heat should get out of the proverbial kitchen?

Well, suppose that there is a way to minimize the potential damage of that approach. Some ways of dealing with conflict are very effective, increasing productivity, while others are actually destructive and can send a project into a tailspin.

## Responses to Conflict

There are particular styles of responding to conflict that are successful in any given situation. The skill in managing conflict is to assess the appropriate response to conflict given the situation.

Some of the many styles of responding to conflict include avoiding, accommodating, competing, compromise, and win/win, as shown in the table below.[22]

| ***Style*** | ***Purpose*** | ***Effect*** |
|---|---|---|
| Avoidance | I don't want to deal with the conflict. | "You win." |
| Accommodation | I'll let you have your way and it's O.K. | "You win." |
| Competition | I'm going to get what I want and you're not. | "I win, you lose." |
| Compromise | I'll give a little and you give a little. | "We both win a little and lose a little." |
| Win/Win, or Collaboration | We both get what we need when we listen to each other and work together. | "We both win the most." |

Each of these styles is described in more detail below. The details help you identify when a team member might use this response and what that might look like behaviorally.

### Avoiding

When a team member avoids a problem, he knows there is a problem but he acts like there is no problem at all. He would rather ignore the problem or walk away from it than confront it. The result is that the problem is still there, and he may feel guilty for not dealing with it.

However, avoidance has it uses. It can be useful when:

- The issue is trivial. Some things aren't worth fighting about.

- The avoider has little power or feels that he does in this dispute. He may work the overtime rather than speaking to upper management about unrealistic deadlines.
- The perceived potential damage of confronting outweighs the benefits. If he confronts upper management unskillfully, he could lose his job.
- People need to "cool down". There has been a yelling match, and he perceives that the conflict has become personal rather than about the issues.
- Time is needed to gather more information. He can make a better case later if he can do more research before addressing the conflict.
- Another person can resolve conflict more effectively. He does not feel that his skills, position in the organization, or perceived role is seen as appropriate credentials for addressing the conflict.

## Accommodating

When a team member uses accommodation to respond to conflict, she lets the other person "win" just to end the confrontation. She lets the other person have or do what they want in order to preserve and enhance her relationship with a co-worker. The other person may not have heard or understood what she was trying to say or what her interests were.

Accommodation can be useful if she wants to:

- Learn from others. She may need to change her lunch date in order to go to a noontime brown bag session on a new technology.
- Show she is reasonable, as in making a goodwill gesture. For instance, to invite participation in a negotiation, the team member may accommodate another team member's request for information or other resources.
- Acknowledge that an issue is more important to the other person. If she is in conflict over the deadline that a user documentation review will be returned to the writer, she may reprioritize her task list and arrange to do the review sooner than she had planned.
- Build up "social credits". This can be like trading favors. I accommodate you on this issue this time; implicitly, you will accommodate me on another issue in the future.
- Allow others to learn from mistakes. She may let her team member "win" even though she knows the solution will fail in order to give her team member an opportunity to learn a needed lesson.

## Competing

The team member will likely compete when she is out to "win" and make the other person "lose." Sometimes a team member feels she must get her point, her interest, across to the other person **no matter what**! The result is that she may get what she wants but she may also damage the relationship with the other person in the process.

Competition works as a response to conflict when:

- A quick response is necessary. The product is two months from shipping; she believes there is no time for negotiation.
- An unpopular course of action is needed. There are staff cutbacks coming; those who perform stay employed.
- Protection against aggressive behavior is desired. Someone has his eye on her role as a team lead.
- The rules demand competition and the rules of the competition are clear. Individual bonuses are paid out based on ranking system that gives the highest performers the biggest bonuses.

## Compromising

A team member is compromising when he gives up something and the other person gives up something. They don't necessarily discuss what the reasons are or what the impact is of what they are each giving up. Each person does not know the full perspective of the other's point of view. Each person gives a little and gets a little. The result is that the situation may be resolved quickly but it may not be the best resolution, and it may not last. It also can be a lose/lose situation where people "give in" and neither achieve their goals.

Compromising can be useful when:

- Goals are moderately important but not worth the time and effort of a more assertive style. If preserving the relationship with his supervisor is more important than the goal of not working overtime, he will compromise and work at least some overtime.
- The participants in the conflict have equal power and are committed to a mutually exclusive goal. His goal is to debug the component he is working on and rebuild by 5:00p. Jane's goal is to get bug 3285 fixed before the beta test of the software. They agree that he will update the problem report today, but he will not do the work until tomorrow.
- Temporary agreements and quick solutions are needed. Team members don't have time to go into elaborate detail about a solution so they agree to do a "quick fix" now and develop a more detailed solution later.
- Collaboration/competition fails and a back up plan is needed. Compromising can be a "Plan B" or part of an interim plan, as described in the example in the bullet above.

## Collaborating

When a team member collaborates, she listens to each disputant's point of view. All team members state their interests, give each other explanations and reasons why they feel the way they do (interests), and advocate for their version of the solution to the problem. Then all disputants research and communicate to figure out what the best solution is for all concerned. The result is that all disputants understand the reasons behind what each wants. The resulting solution is based on those reasons (interests), and will probably be the best for

both the team and the organization as a whole. Collaboration gets the most for all involved in the problem-solving process.

Collaboration works well when:

- The issues are too important to be compromised. She is participating in a feature set definition process. The features they include in the release being planned directly impact the organization's bottom line.
- People want to preserve/improve relationship. She found the poor relationship with marketing in the past resulted in marketing not sharing information about the customers that directly impacted the success of the last release.
- There is a desire to get insights from others who have a different perspective on a problem. She needs the customer information that marketing routinely gathers and stores only in the heads of it's specialists.
- It is important to gain commitment from all involved. She doesn't want the feature set changed after implementation has begun.
- An increase in follow-through is desired. When team members invest their time in a collaborative solution they are more likely to invest time in the implementation of the solution.

All of these modes of response to conflict are valid in some situations. However, two modes, avoidance and competition, are more harmful in work teams. One mode, collaboration, facilitates better decisions faster. Collaboration also improves team member relationships, performance, and follow through.[23]

## Effective and Ineffective Responses to Conflict

As show above, competition and avoidance tend to be effective under some circumstances, but they are antithetical to collaboration, which is what is needed on cross-functional teams, the norm in technology development environments. According to Gobeli and Koenig[24] in their 1998 study of conflict in software companies in the Pacific Northwest,

- Unresolved conflict has a strong, negative impact on overall software product success and customer satisfaction.
- Managing conflict well will increase the chances of success; not managing conflict well will worsen the conflict, decreasing the chances of success.
- Confrontation (collaboration) and give and take (compromise) will likely increase team member satisfaction.
- Withdrawal (avoidance) and smoothing (accommodation) will have marginal effects, and forcing (competition) will decrease member satisfaction. Forcing appears to have an even greater negative impact at the project team level as opposed to the organizational level.
- To the extent that management style affects success via conflict intensity, project level management should emphasize confrontation over give and take when conflict surfaces.

- In general, a sign of trouble for management to monitor is frequent use of the dysfunctional conflict management styles including withdrawal, smoothing, and forcing rather than the more functional approaches of give and take and confrontation.

The link between conflict and project success and team member retention is of particular interest to us. After reviewing Gobeli and Koenig's research, we felt it was important to attempt to uncover another layer of the team member experience strata where conflict is concerned. To do this, we surveyed 261 students who have attended the Project Management series at the Portland State University Professional Development Center series on project management, where Lisa Burk is an adjunct faculty member. In informal surveys of classes in this program it appeared that one third to one half of the students work in high tech and one third of the reasons cited for project failures is the unsuccessful management of conflict and poor or lack of communication.

Of the 261 students who received the survey, 180 were estimated to be working in technology development. We received 56 responses from qualifying recipients. The data gathered (discussed in Appendix A) was interesting in that it seems to validate two of the expectations that caused us to initiate this review of the literature:

- Conflict is not recognized as a cause for project failure, but rather "part of the job to be endured" that it "comes with the territory".

- Conflict skills training is "not on the radar" in this industry as a possible solution to communication problems, though human communication experts have long acknowledged conflict skills as trainable and a key component in effective communication.

As will be shown later (see Conflict is Related to Increased Costs on page 32), job satisfaction criteria far outrank financial rewards with regard to retention, and we believe that low staff retention is an indicator of a high rate of unmanaged conflict in an organization. The details of our research and the questionnaire we used appear in Appendix A of this paper.

Our conclusions include the following:

- As Gobeli and Koenig found, effective conflict management is related to team member satisfaction.
- There is a high correlation between ineffective conflict management and project failure.

As will be shown later, job satisfaction relates more closely to an employee's choice to stay with the organization than does financial reward. When conflict is managed poorly, the risk that the team member will seek work elsewhere dramatically increases. But does this mean that there needs to be a conflict manager on every team?

No, we don't think that's the answer. Rather, increasing the conflict response skill level of the entire team is more effective. Conflicts are ubiquitous. They arise quickly and usually between two individuals. Take for example, the "flare up" between two individuals at the fictitious Megatroid project. (The underlying reasons, or interests, are provided in italics.)

***Usability Engineer***

"Here is my proposal for a series of customer site visits to help us gather data to feed into the requirements definition phase in the development of the new Megatroid project."

***Product Manager***

"Why do you need to do this?"

***Usability Engineer***

*[Thinking] Isn't it obvious? Haven't we talked about this before? This is my job!*

"This process will gather the data we need to understand our customer's environments and work styles as well as how they use our current products, our competitors products, and any homegrown solutions they've created. Knowing this kind of information will help us create a set of requirements that are more reflective of the marketplace and therefore, result in a more saleable feature set."

***Product Manager***

*What is she talking about? If we didn't know that the customer needed this new product, we would never have funded a development team to build it. It's my job to know this kind of stuff!*

"Like I said, why do **you** need to do this?"

***Usability Engineer***

*This guy is nuts? What's wrong with him? Doesn't he think I know what I'm doing?*

"In the past we have only done usability testing. That's reactive, and at best, can only find problems in the design after they are already present. That's far more expensive than developing a sound set of requirements based on user and task analysis which results in a feature set and implementation criteria that prevents the problems we might identify in usability testing on the back end."

***Product Manager***

*That's it, she's operating in my area. I'm sick of this.*

"Listen, we're about out of time for this topic in this meeting. What I want to see is a list of the top five questions you are going to ask the users – by next week."

***Usability Engineer***

*What the hell does he think he's doing?! He's not my boss.*

"Why do you need this?"

***Product Manager***

*Now I've got her.*

"Because my budget is funding this, and I need to understand what the benefit is here. I need to have more information before I can decide whether it makes sense to do this."

***Usability Engineer***

*Damn. It is his budget.*

"Okay, fine. I'll send you something by next Wednesday."

[Departs in a subdued huff.]

Often those individuals are interacting without an audience or immediate supervision, as exemplified in our characters above. The point of the flare-up is where the flame must be tempered. All team members need proper conflict response skills. In our example above, this conflict could have been easily managed if both people had used effective conflict mangement skills, in this case, communicating the real reasons beneath their conflict.

## Conflict Skills Development

How do people who do not currently have good conflict response skills gain those skills? As with most adult learning, they internalize theory through practice. Raider, Coleman, and Gerson address this issue most concisely. They describe six insights gained from teaching negotiation and mediation to adult learners:

- The first pedagogical insight is that each learner has a unique and implicit "theory of practice" for resolving conflicts.
- Second, learners need both support and challenge to examine their own theory of practice. Intellectual and experiential comparison of competitive and collaborative processes can create challenging internal conflict for learners.
- The third insight is that experiential exercises shift the responsibility for learning from the trainer to the participant.
- Fourth, self-reflection based on video or audio feedback gives many learners motivation to modify problematic behavior.
- Fifth, user-friendly models and a common vocabulary enable a group of learners to talk about their shared in-program experience.

- The final insight is that learners need follow-up and support after workshop training to internalize new concepts and skills."[25]

The authors also give seven workshop modules for conflict resolution training, which we support. These modules are:

- Overview of Conflict
- The Elements of Negotiation
- Communication Behaviors
- Stages of Negotiation
- Culture and Conflict
- Dealing with Anger and Other Emotions in Conflict Situations
- Introduction to Mediation[26]

Slaikeu and Hasson see conflict resolution skills training for all employees as a method to reduce stress and save time, and as critical component to a successful conflict management system.[27] They suggest two types of training.

- An orientation to the approach the organization or team is taking regarding managing conflict.
- Skills training for all employees, managers and specialists in communication and conflict resolution similar to the outline listed above.

They also suggest that managers with supervisory authority need additional training in "coaching" and "informal mediation."[28]

Slaikeu and Hasson provide several suggestions for best practices in conflict skills training:

- First, distinguish between orientation and skills training, neglecting neither.
- Second, review existing skills courses.
- Third, target skills-based training to individual job functions.
- Fourth, consider several training formats geared towards the needs of participants.
- Fifth, Coordinate with other training programs in the organization.

## Conflict Skills and Professional Mastery

Given how difficult dealing with conflict is for almost everyone, why would a team member be interested in gaining conflict resolution skills? And, why would an organization be interested in encouraging the team member to take time away from designing and implementing a new product to spend it on learning how to respond to conflict?

The answer from the organization's point of view is fairly straightforward: Conflict-skill-competent team members are more likely to be contented team members. Contented team members are less likely to leave the organization. Hiring and training cost **a great deal of money** (see Conflict is Related to Increased Costs on page 32).

> "…If your organization is more like the norm, there will be line managers who are clueless about what people really want and what makes them vulnerable to talent theft. According to the *Harvard Management Update* (June 1988) 9 out of 10 managers think people stay or go because of money. We know that's not the case. Money and perks matter, but employees tell us again and again that what they want most are challenging, meaningful work, good bosses, and opportunities for learning and development.
>
> A 1999 Hay Group study of more than 500,000 employees in 300 companies found that of 50 retention factors, pay was the least important. Our research of more than 2,000 respondents from diverse industries and functions shows similar results….The top three reasons people stay are: 1) career growth, learning and development; 2) exciting work and challenge; and 3) meaningful work, making a difference and a contribution….
>
> Once people attain a certain level of material comfort, they care most about what they do every day and who they do it with. They care about the content of their work and whether there are opportunities to stretch and grow in the job and in the organization. They want feedback, recognition, and respect from their bosses.[29]

The above studies were not specific to software development environments. And, to be sure there are individuals who do migrate after dollars. However, according to the multiple studies cited in "Retention: Tag, You're It!"[30] money is not the primary motivator for the vast majority of people.

The answer from the team member's point of view relates to employability and professional mastery. People in pain, as people in conflict frequently are, do not typically care to perpetuate that pain. Given the choice, they'll take an aspirin, submit to surgery, even make far reaching lifestyle changes – including, perhaps, building a new set of conflict resolution skills to help them be safe, happy, and more successful in their careers.

## Employability

Technical skills are the stock in trade of technical professionals. However, skills such as conflict management make those technical skills accessible to the organizations employing these team members, much as the transmission in your car requires hydraulic fluid in order to transmit the power from the engine to the wheels.

According to a recent survey, more than one-third of the skills identified by managers as important are non-technical skills such as good communication, problem-solving, and analytical skills, along with flexibility, and the ability to learn quickly. Training after the employee is hired is rated as significantly more effective than pre-hire methods of training; 84 percent of the managers rated on-the-job training as effective or very effective compared to 41 percent rating pre-hire training as high. Managers strongly prefer on-the-job training when it has a structured format and a defined curriculum..[31] This will be true of conflict skills training as well as technical training inasmuch as organizations have identifiable personalities and typical conflict styles just as individual people do.[32] It is best to train team members in the environment in which the conflicts will occur.

About 35 percent of the companies taking part in another survey said they are actively looking for people with the right mix of technical abilities and business savvy. Included in this chief information officers' wish-list of skills requirements is an understanding of business modeling, project management, **leadership**, the **ability to work as part of a team**, and **communication abilities**. [33]

The most valuable technical professionals are those who get results, the people who work well in groups to quickly execute complex product designs. These people are either good conflict managers or they are team-eaters who burn out their people, cycling through a new team every project or so, bad risks in terms of cost to the organization (see Conflict is Related to Increased Costs on page 32).

## Health and Safety

Perhaps you've witnessed extreme examples of unhealthy responses to conflict on the job: heart attacks, fist fights, shouting matches, and chronic stress-related illnesses resulting from a depressed immune system. Modern medicine has established beyond contention that emotional stress directly relates to physical stress and disease. When people are in conflict and they see no way to resolve the conflict and maintain their sense of personal and professional safety, they are under tremendous stress. Over time, this stress can, and often does, culminate in physical illness, which impairs the productivity of the team member, in extreme cases, even ending his or her career.

Two physicians, Drs. Brinkman and Kirschner, have become so convinced of the link between job stress and poor physical health that they have developed a training program and written a book to help people develop skills to minimize job stress. They say, "Time and again, we found that when people clarify their values, update their concepts, learn effective communication and relaxation skills, set and then work to fulfill their goals, they feel better. And as their mental and emotional health improves, many of their specific physical symptoms disappear."[34]

### Professional Mastery

Skill in responding to conflict situations is one that crosses all occupations. However, in high pressure positions such as most technical professionals in software development hold, this skill is vital. It can easily mean the difference between personal and professional success or failure. People in positions of leadership are under even more intense pressure to quickly master and apply conflict skills.

One of the most common responses of project managers to team conflict is panic. … Consequently, any evidence they interpret as damaging to the prospects of project success, such as team conflict, represents a very real source of anxiety. In reality, however, these interpersonal tensions are a natural result of putting individuals from diverse backgrounds together and requiring them to coordinate their activities. Conflict, as evidenced by the stages of group development, is more often a sign of healthy maturation in the group.[35]

Clearly, developing a strong set of conflict skills is of benefit to both the team member and the organization she works within. The team member stands to safeguard her health, increase her job satisfaction, and attain greater perceived value in her field. The organization is more likely to retain the team member, to have a much more productive team member, and to more efficiently execute projects – all of which translate in to tremendous cost savings.

## An ADR System Design Perspective

In this section, we will introduce you to the concept of integrating conflict management into a system within an organization. This concept is a fairly recent one, being introduced into organizations within the last seven to ten years. We will discuss three books written on this subject, and will highlight a local example of a conflict resolution system that is currently being utilized by federal and local agencies in Oregon and SW Washington. As stated earlier, managing conflict effectively can increase your bottom line, therefore it might be useful for software development organizations to examine and adapt some of these strategies to fit your particular environment.

Dispute system design (DSD) grew out of the ADR movement in the late 1980's. At about the same time, the first book on DSD was published: *Getting Disputes Resolved* (Ury, Brett and Goldberg, 1988). This book provided the first framework and core principles for designing conflict management systems in organizations. Two subsequent books followed. First, Costantino and Merchant in their book *Designing Conflict Management Systems* (1996) advance the field of system design by integrating organizational development principles into management of conflict in organizations. Next, Slaikeu and Hasson write the most recent book on system design: *Controlling the Costs of Conflict* (1998). This book further advances the field by providing readers with a cost effective blueprint on how to design systems that control and manage conflict in your organization. Costantino and Merchant describe why DSD is such a recent phenomenon.

> Typically, organizational leaders do not view the management of conflict as systematically as they do information, human resource and financial management systems. Rather, conflict in organizations is viewed and managed in a piecemeal, ad hoc fashion, as isolated events, which are sometimes grouped by category if the risk exposure is great enough but that are rarely examined in the aggregate to reveal patterns and systemic issues. In a sense, most organizations regard disputes as "local" events. Viewing the management of conflict systematically provides unparalled opportunities for an organization to learn critical information about its operations, its population, and its environment – that is, to achieve a more global perspective.[36]

All three books are based on the premise that there is something wrong with the way most businesses manage conflict. The failure lies in a systemic reliance on higher authority, power play, conflict avoidance, and weak or only partial use of collaborative options. They also examine the relationship between interests, rights, and power. They advocate interest-based, collaborative negotiation, and detail the practical realities andcosts when power or rights dominate a conflict management process.

Ury, Brett and Golberg[37] describe interests, rights and power as:

- **Reconciling interests** - negotiation and mediation; also called interest-based problem solving.
- **Determining who is right** –adjudication; courts, administrative agencies, arbitrators.
- **Determining who is more powerful** – coercive; imposing costs or threat thereof; acts of aggression (sabotage/attack) or withholding the benefit of a relationship (strike, divorce).

They the go on to say that a "distressed" system allows power to dominate rights and rights to dominate interests. An "effective" system is the reverse; most disputes are resolved by interests, some by rights, and the fewest by power.

Slaikeu and Hasson state "most conflicts start small and present many opportunities for resolution before growing into full-blown disputes." They continue: "Our experience suggests that many organizations focus more on full-blown disputes than on creating procedures geared to reveal conflicts early and resolve them in the most efficient and productive manner." [38]

Do these symptoms "play out" in software development environments? We found two recent examples.

One author sites how conflict impacted their system at LookandFeel New Media, a dot com start-up in Kansas City.

> Territorial issues sprang up between departments. These issues began to have an impact on the quality of our work. Deadlines slipped, the quality of our work dropped, clients called to voice their disappointment, and employees began to leave. …we had always operated on the belief that if we hired good people and gave them a good environment that good work would result. Individually, everyone on the team was a solid professional. Collectively, however, we were dysfunctional. We wondered how we had fallen short of our ideals. [39]

Kleiner, in his article "Corporate Culture in Internet Time," cites a recent trend in e-commerce businesses to turn to more traditional organizational development systems and techniques.

> In the last few months in particular, there's been a growing sense that something is wrong. Dot-com company executives are beginning to try the same kinds of "employee empowering" measures that their mainstream, Fortune 500 counterparts have tried: Talking about values, building shared vision, even hiring facilitators to run meetings. Six months ago most of these ideas would have been dismissed out of hand as hopelessly old-fashioned. [40]

We certainly could add DSD to Kleiner's list of suggestions above.

Costantino and Merchant differentiate between *dispute* and *conflict* (which many people use interchangeably).

> Conflict is a process; a dispute may be one of several products of conflict. Conflict is the process of expressing dissatisfaction, disagreement, or unmet expectations with any organizational interchange; a dispute is one of the products of conflict. Collections or clusters of disputes are simply one of the many ways that conflict manifests itself in an organization.[41]

They believe that an organization's response to disputes and its overall strategy for managing conflict is directly related to the overall "culture" of the organization, or the informal and formal "rules" of the organization.

They also provide us with examples of how conflict shows up in many ways in an organization:

- In *individual disputes* such as grievances, disciplinary action, complaints and disagreements with internal or external people.
- As *unhealthy competition* within teams or between internal departments.
- *Sabotage efforts* (an extreme form of unhealthy competition), where individuals engage in covert actions to undermine or "get back" at an individual or an organization.
- *Poor or slow performance and low morale* can also be a symptom of conflict in an organization.
- Individuals or groups can withhold knowledge, particularly in an organization where knowledge is power.[42]

Constantino and Merchant then conclude that "…though conflict is a given (in organizations), weak systems are not. You can strengthen the system and thereby reduce costs associated with unresolved conflict." [43]

> Some organizations have moved towards more systemic approaches to conflict management. Others continue to use fight (competition) or flight (avoidance) methods to deal with conflict or avoid it. In the middle are those organizations that have designed an alternative dispute resolution (ADR) approach or program to deal with particular types of disputes. [44]

The authors site the following interests behind the growing movement in ADR:

- Backlash against attorneys, lawsuits, and legal costs,
- Societal movement to more natural and humane methods of dispute resolution,
- Increasing interest in flexible dispute resolution, and
- Interest in confidentiality and avoidance of publicity.[45]

How might this growing interest in ADR and DSD transfer to software development environments? Costantino and Merchant give a pertinent hypothetical example of Montro, a leading manufacturer of word processing software whose consumers are both home users and commercial businesses.

> Last year, Montro introduced "Hole in One," a new "umbrella" software package that includes word-processing, graphics, spreadsheets, and presentation applications. It is Montro's first entry into the spreadsheet and presentation software market. Recently, Montro has begun to receive complaints, particularly from home users, that the presentation package does not interface properly with the word processing package, with the result that certain categories of presentation data (particularly bar graphs and pie charts) are deleted in certain modes of operation. Yesterday, an irate home user who runs her own business called the Montro CEO, complaining about a large presentation file (and, she claims, a potential client) she lost because of this problem. The CEO, sounding annoyed, immediately called Ms. Jones, the director of the Consumer Service Department. He instructed her to meet with Mr. Tate, director of research and design, to deal with the customer, to explore the problem, and to come up with some solutions "once and for all." Ms. Jones is not looking forward to today's meeting since she and Mr. Tate have a strained working relationship based on several problems on which they have worked together in the past.[46]

The authors use this example to illustrate that conflict can take the form of either an internal or an external distress signal that comes from either inside or outside the organization. Organizations can choose to ignore the distress signal and hope it "goes away" or develop systems that assess the need to pay attention to the signal and appropriate solutions.

## Organizational Differences

Although all the system design sources we evaluated offer a template or blueprint for developing conflict management systems, they acknowledge that every organization is unique and needs it's own "game plan."

Kleiner also notes that each company has a unique culture and that "…anyone who has tried to create a culture, however, knows it can't be done on Internet time. Cultures aren't designed. They simmer; they fester; they brew continually, evolving their particular temperament as people learn what kind of behavior works or doesn't work in the particular company." [47]

In order to understand organizational differences, including how they respond to conflict, we have found the work of William Bridges to be very helpful. Organizations, like individuals, have inherent characteristics that must be understood and respected if real growth or change is to take place. Based on the work of Carl Jung and the Myers-Briggs Type Indicator (MBTI), Bridges explains why organizations act the way they do and how an understanding

of organizational character benefits an organization as well as the individuals in an organization. Bridges looks at growth, change, response to conflict, and organizational development through the myriad perspectives held by different "types." Costantino and Merchant shed additional light on how the MBTI can characterize an organization's response to conflict.

The application of MBTI information about individuals in an organizational setting provides immensely valuable and otherwise unavailable insights into why individuals and groups either do or do not work together effectively. "One possibility is that where the organizational response to conflict differs markedly from a particular individual's preferred response to the same conflict, dissonance results."[48]

The opportunity to design a dispute resolution system most often arises in one of three situations: there is a crisis, an insider has "a better idea", or new relationship or organization is being formed. In crisis, designers are usually called in to fix the situation (do a training, make recommendations, or resolve a particular dispute), not to redesign the system. The opportunity to redesign the system comes only after the designer has gained credibility and familiarity with the parties. Insiders can be the sparks for system overhauls. They may even be the designer and lead person implementing the change. For example, grievance mediation has been effectively introduced by both companies and unions, not because of a crisis, but simply because it's a better way to handle disputes. Both the real time company, LookandFeel New Media, and Costantino and Merchant's hypothetical company, Montro, were in crisis and could have used their crises to springboard the development of a conflict management system.

Costantino and Merchant link, for the first time, conflict management principles with the principles of organizational development. In other words, conflict resolution tools (mediation, arbitration, facilitation, etc.) have been used to resolve isolated disputes for over twenty years in workplace environments, it is only in the last five to seven years that some organizations are looking at managing organizational conflict as a system issue – not just housing it with HR or a legal department.

When you look at managing conflict through the lens of organizational development, conflict management is viewed systematically, it is planned and intentional, it is tailored to "fit" the organizational culture, and it is as durable as possible over time. Unfortunately, this takes time –in two ways. First, in all three books reviewed for conflict management system design, they suggest a process that could take up to two years to design. Second, an organizational development "rule of thumb" regarding changing an organizational system is three to seven years. This means that once the system is designed, successful implementation takes another chunk of time for the individuals in the system to successfully use it.

A classic organizational development text states, "The key elements (in organizational development)…are long range, planned and sustained, and strategy. There is a long-range

time perspective on the part of both the client system and the consultant in OD programs. Both parties envision an ongoing relationship of one, two, or more years together if things go well in the program. The reasons for OD practitioners and theorists conceptualizing OD programs in long-range are several. First, changing a system's culture and processes is a difficult, complicated, and long-term matter if lasting change is to be effected. OD programs envision that the system members become better able to manage their culture and processes in problem-solving and self-renewing ways. Such complex new learning takes time. Second, the assumption is made that organizational problems are multifaceted and complex. One-shot interventions probably cannot solve such problems, and they most assuredly cannot teach the client system to solve them in a short time period."[49]

## System Design Methodology

All three of our primary sources on conflict resolution system design outline blueprints that would involve intensive staff involvement in developing a system over a year or more period of time. The organization should be an active participant in all phases of the system design process, involving team members from the outset in both the diagnosis and the design. Establish a design committee: they serve as liaison and representative for the organization, creating more ownership, easier acceptance and higher commitment to joint problem solving. In a good design process, end-use acceptance is built in by involving people in the design process.

Slaikeu and Hasson outline four basic principles for designing conflict management systems in organizations:

- Acknowledge that there are different options for dealing with conflict, and choose the option you want to reward more than others.
- Create options that prevent conflict from escalating.
- Develop clear, internal systems that support collaboration – policies, roles and responsibilities, documentation, selection, training, support and evaluation.
- Use mediation techniques (consensus and collaboration) to build the internal conflict resolution system.[50]

Ury, Brett and Goldberg offer the following suggestions for designing systems:

- Gain the support of key stakeholders. This may involve canvassing potential users of the system to understand their motivations.
- Expect to make adjustments as the system is implemented. Adjust to keep parties motivated in using new procedures, and help them develop the skills to do so.
- Make the new procedures attractive to disputants. Demonstrate procedures, use leaders as examples, use peers as proponents, set goals, provide incentives, and publicize early successes.

- Training the parties together is valuable. It gives the parties a common vocabulary, instills common expectations, and offers a safe environment to try out new procedures. Provide continuity by establishing an ongoing program for familiarizing and training new people.
- The designer is a temporary support in the construction of a new system. Balance benefits of managing implementations vs. risks of over-reliance on the designer. At some point, the system has to stand on its own.
- Evaluate the system. Are the costs reduced? Are benefits realized? Are there any unintended consequences? Fine tune changes. Learn from initial experiments. Does the new system work? What are the limits of the effectiveness of the changes? Why do the changes work?

## Essentials of System Design

When designing a conflict management system, the designer embarks on a system diagnosis. A system designer looks at:

- What kinds of disputes are likely to arise, how often, and between whom?
- How are disputes being handled currently, and where might lower cost procedures be used? (Establishing current costs also sets a base line against which to measure costs of new system.)
- Why are current procedures being used? New system must match or beat motivations and benefits met by current system.
- Identify who needs to be trained, coached and/or replaced.
- What resources are available to assist disputants (people, information, institutions)? Are more resources needed?
- What type of organization are you looking at? An organization can avoid its weaknesses, compensate for its weaknesses, develop new strengths, and/or capitalize on its existing strengths.

Throughout the design process, the system designer plays many roles:

- Coach
- Evaluator
- Expert
- Mediator
- Negotiator
- Evangelist[51]

## System Design Options

Slaikeu and Hasson advocate for the use of system options that range from site-based resolutions between employees with a higher authority back-up, to internal support people (HR, Peer Review Panels, internal mediators), to external ADR resources, to external higher

authorities (Courts, unions, administrative hearings). Good systems should start with encouraging the least intrusive option and the option that keeps the most control of the outcome in the parties' hands, and feature "loopbacks" to lower interventions at any time.

Another author stresses that organizations need to pay attention "to characteristics of the conflicts (content, relational and situational dimensions), desired outcome of the participants, and awareness of available conflict management strategies when choosing conflict management strategies.[52] This author also identifies three internal options for resolving conflict: face-to-face discussions with no third party intervening, an intervening manager or intervening other in the organization (HR, internal mediator), as well as turning towards external third parties for intervention.

Ury, Brett and Goldberg list the six basic principles dispute system design…

- Put the focus on interests. Use negotiation and mediation.
- Provide rights and power procedures that loop back to negotiation.
- Provide low-cost rights and power backups (low cost means for resolution if interest-based procedures fail).
- Build in consultation before disputes arise, and feedback after they are resolved.
- Arrange procedures in low to high cost sequence.
- Provide motivation, skills and resources needed to make all these procedures work.

There are many types of ADR techniques that could be part of an organizational conflict management system. As shown in the table below, they range from preventative (partnering, conflict resolution training) to negotiation (parties involved in conflict directly negotiating a settlement) to facilitattion (mediation, ombudsperson) to fact-finding (settlement conferences evaluation, non-binding arbitration) to imposed ADR (binding arbitration). We have defined these ADR techniques in more detail for you in a chart below. As one can see, the techniques are listed in order from the least invasive (parties have more control of the process and the outcome) to the most invasive (parties have less control of the process and the outcome).

*Alternative Dispute Resolution Ladder*[53]

| | |
|---|---|
| Communication | Verbal and non-verbal behaviors, or events that are perceived by one or all parties to have meaning assigned to it. The parties may be experiencing conflict or no conflict. |
| Partnering | A preventative approach to deal with potential conflict that features a negotiated "partnering agreement" that states how people will resolve future problems. It often includes joint training in conflict resolution and communication. |
| Negotiation | A process whereby, disputants communicate with each other, either directly or indirectly, about issues in disagreement. |
| Facilitation | The use of a third party (insider/partial or outsider/impartial) to help multi-party groups accomplish their work by providing process leadership and expertise. The group may be experiencing conflict or no conflict. |
| Mediation | The use of a third party (insider/partial or outsider/impartial) to facilitate communication between negotiating parties which may enable the parties to reach settlement. |
| Settlement Conference | A neutral and impartial third party conducts an informal assessment and negotiation session, with the goal of settling the dispute. The third part may advise the parties and suggest a settlement of the dispute. |
| Med-Arb | A neutral and impartial third party facilitates communication between negotiating parties, and failing settlement, receives evidence and issues a binding decision. |
| Arbitration | One or more impartial third parties hear and consider the evidence and testimony of the disputants and issue a binding or non-binding decision. |
| Litigation | An impartial judge or jury consider the evidence and testimony presented by the disputants and issues a binding, enforceable court order. |

We would now like to highlight two of the ADR techniques that have been working in other workplace environments:

## Partnering

The construction industry has been using a technique known as "partnering" for over ten years to prevent problems from escalating before, during, and after construction projects. "Partnering is a structured management approach to facilitate teamworking across contractual boundaries. Its fundamental components are formalized mutual objectives, agreed problem resolution methods, and an active search for continuous measurable improvements. It should not be confused with other good project management practice, or with long-standing relationships, negotiated contracts, or preferred supplier arrangements, all of which lack the structure and objective measures that must support a partnering relationship. The critical success factor for partnering is the commitment of all partners at all levels to make the project a success. The result is that the partnering agreement drives the relationship between parties rather than the contract document."[54]

## Developing Internal Mediators

Shared Neutrals is a remarkable local example of how an ADR program can work. Shared Neutrals is a cooperative arrangement between several federal and local agencies in Oregon and Southwest Washington. Member agencies share "neutrals" – employees who have been trained and mentored in communication, conflict facilitation, and mediation. Neutrals mediate employee and agency/public conflicts outside of their own department or agency. These neutral individuals are called in to handle interpersonal conflicts (disagreements, harassment, communication problems), discrimination allegations, sexual harassment issues, contract disputes, policy disputes, and agency/client disputes. They have no personal stake in the outcome and remain impartial in resolving the conflict. Their purpose is to help the disputing parties arrive at an outcome acceptable to all parties involved.

In the Shared Neutrals program, participating agencies each "contribute" a staff member or members. By interagency agreement, these staff members are also available for service in another agency or department as a Shared Neutrals mediator. (Time limitations apply – cases are not to exceed 35 hours; a normal case take 4 – 6 hours to resolve.) Staff members undergo thorough training and mentorship to prepare them for their role as mediators.

Costs for implementing an ADR program such as Shared Neutrals are minimal, especially when compared to the savings such programs deliver. "We've saved a ton of money," says Julie Wells, US Forest Service Regional Manager for Employee Relations in Oregon and Washington. Ms. Wells signed her agency onto the Shared Neutral program. She enthusiastically reports, "I have no regrets. The average discrimination complaint costs us $5,000 just to investigate. We've been resolving many of those cases much more cheaply and successfully using mediation." [55]

"In-house" mediation networks, such as the Shared Neutrals program, are a recent development. Assessment on these programs is favorable, so much so that the Federal Equal Employment Opportunity Commission has mandated implementation of ADR programs in all federal agencies by the year 2000.[56]

## Developing Your Own System

How might a software development company develop an ADR program? Let's re-visit our friends at LookandFeel New Media, our real time e-commerce company.

First, this company found itself in a crisis state – "Many of the problems we faced seemed to be steeped in emotion and had defied our attempts at rational analysis and solution."[57] Next, they hired a former psychologist, turned communication consultant (a quasi system designer and mediator/facilitator) who interviewed employees privately to assess the communication and problem-solving issues the company faced (a system diagnosis). They then had a retreat and had a discussion where a sense of confusion prevailed, but were able to identify they needed to do something in the organization to better deal with conflict - "And almost universally, people felt that when problems were identified, they took too long to be solved."[58]

At the one day retreat mentioned in Logan's article, **employees, managers and partners** (it is important to have all involved) developed some basic guidelines (a minimal, but effective conflict management system) for resolving problems more effectively:

- Understand that problems arise every day: Nothing is more natural. The way we respond to problems is what differentiates us.
- If you see a problem, address it. Nobody has permission to stay silent. Everybody is obligated to address issues as soon as possible.
- If you have a problem with someone else, address it with that person directly, privately and as constructively as you can.
- If that doesn't work, bring in a third party, a manager, or the partners to arbitrate.
- And most importantly, except in the framework of this process, talking about problems with anyone other than the person involved is strictly forbidden.[59]

They then established a "leadership council," comprised of all levels of employees and representative of all departments to monitor the above system evaluation.

The author then goes on to state his positive assessment of the process:

> Because building this kind of culture (the above problem-solving process) requires an ongoing commitment, Cone (the consultant) continues to serve as a company counselor (facilitator/mediator), helping us identify problems and brainstorm solutions. Having an objective counselor has been one of the biggest benefits of the entire process. When this began, we were all really excited about creating a culture of open communication, but none of us had really seen it in any workplace. Being able to talk honestly to an objective, outside source made it easier to do the same thing with my peers. The result of this investment (in an objective, outside consultant) in effort and time has been a remarkable turnaround in almost all facets of our work. The quality of our culture is reflected in not only our employees' morale, but in our client satisfaction as well. And these improvements have had a definite impact on our bottom line – revenues have more than doubled from last year.[60]

## The Costs of Conflict

This section discusses the how conflict impacts your bottom line. We have come to take high turnover rate for granted in technology development. We may suspect other organizations of "pilfering" our human resources. Often we see companies throw money at the problem of retention: stock options, signing bonuses, performance bonuses, and salary increases. However, the literature indicates that tangible rewards do far less to retain employees – including the star performer – than overall job satisfaction. Skillful conflict management drastically increases job satisfaction.

### The Big Picture

There are costs associated with conflict. People pay, teams and projects pay, and companies pay.

Costs can be computed quantitatively and qualitatively. In the examination of conflict resolution skills and conflict management in the software development arena, qualitative criteria – how people feel about and experience workplace conflict – translate into quantitative factors, i.e. dollars. Specifically, job satisfaction rates, project success, and team member retention rates all impact the bottom line.

Organizations pay the price for poorly managed conflict: when projects and products "fail", and when good people walk. Both scenarios are commonplace, and expensive. If you consider that turnover costs are generally estimated at 150 – 200% of salary[61], you can see the problem when workers are lured away. Turnover rates continue to be high, in the 15 – 20% range (see below for details). Combine these turnover costs with the big hits companies take when projects and products don't meet expectations or requirements, and the costs really add up.

Individuals also fare much better when conflict is skillfully addressed. People (managers, staff, and individual contributors alike) bear the costs of conflict in a much more personal manner – they pay with loss of quality of life, and, too often, their health. The relation between stress and health is well documented: as stress increases, quality of life decreases, and potential impacts on health increase. When you add it up, it pays to deal with conflict constructively, creatively, and proactively. Ideally, you want to move beyond managing flare-ups; you want to set up conflict resolution systems in advance. Setting up systems to resolve conflict creates a safety net. You can catch and constructively deal with conflict. You can minimize costs, and even derive some benefit from conflict if you set up effective systems for dealing with it. (See Conflict and Creativity on page 4 and An ADR System Design Perspective on page 19.)

## Conflict is Related to Increased Costs

The benefits of collaborative conflict resolution stand in stark contrast to the costs of poorly managed conflict. Consider the following:

The economic impact of the software industry is significant; employment numbers as a percentage of the workforce keep growing:

- Software development is a component of "high technology" – which accounted for 14% of all employment in the U.S. as of June, 1999.[62]
- In 1999, "high technology manufacturing" accounted for 44,800 jobs in the Portland-Vancouver area (including Clackamas, Columbia, Multnomah, Washington, and Yamhill counties in Oregon and Clark County, Washington), up 48% from 1990.[63]
- In Oregon, there were 1,675 companies associated with the software industry employing 16,987 people in 1998. The average wage was $51,875.[64]
- Software development is a piece of the larger IT pie. (IT occupations include database administrators, computer support specialists, and all other computer scientists; computer engineers; and system analysts.[65]) The IT worker shortage, well documented and lamented, is expected to escalate in the U.S. and worldwide. Projections vary, but all reports predict increasing shortages of IT workers, for the next few years at least.[66] This is bad news for companies. It's also bad news for the managers and workers who get caught in the crunch and have to take up the slack.

Salaries and bonuses keep increasing:

- Compensation programs within IT companies increased from about 30 percent of the entire IT budget in 1998 to 40 percent or more in 1999. In addition, 15 percent "skills bonuses" are not uncommon as companies struggle to retain employees.[67]
- In April, 1999, Information Week reported, "The median annual [IT] salary increase across the board was a hefty 8.9% compared to all industry average of 5.9% reported by the Labor Department. Internet managers report the second-highest median annual percentage increase--12.9% over last year--among all managers surveyed. They also enjoy a healthy median annual salary of $70,000. This study was based on 21,398 responses."[68]

- IT professionals with the hottest skills are often receiving base pay increases in the 10-20% range per year, compared to national average annual increases of 4-5% for non-IT salaried exempt U.S. employees.[69]
- IT salaries are a factor in staffing problems, with an average 20 percent annual increase. That contrasts with 4 percent annual increases for non-IT salaries (as of a February, 1999 report).[70]

Attrition costs more than retention.

Retention is a big problem for U.S. corporations in all sectors, software development included. People walk, frequently. A sampling of turnover statistics:

- A survey of 1,400 Chief Information Officers, released in December, 1998, reported turnover in IT departments averaging 19%, with one-fifth of respondents citing attrition rates more than 25%.[71]
- A February, 1999 survey reported IT turnover averaging 11 – 20%, and hitting as high as 50% in some corporations.[72]
- Turnover among technical workers is high: 15% as of 6/99, according to a Meta Group study.[73]
- Turnover of IT employees averages just below 16%, but some companies in the study reported turnover rates as high as 35%. The average time reported for filling an IT position was three months. Twenty-eight percent of companies reported at least 10% of their high tech positions were vacant, and while the vacancies are down from 1998 reports, 75% reported using contract IT employees in some capacity.[74]

While it is not the only factor related to low job satisfaction, we do know that poorly managed conflict is, indeed, strongly associated with low job satisfaction levels. Common sense indicates that low job satisfaction levels are associated with retention problems, and the literature supports this:

> "Money and perks matter, but employees tell us again and again that what they want most are challenging, meaningful work, good bosses, and opportunities for learning and development."

People who are not satisfied with their jobs are easy targets for recruitment, and turnover is expensive. Estimates vary. One source put turnover costs at 70 – 200 percent of that employee's annual salary.[75] Another source cited turnover costs as much as one and one-half to two times salary.[76] Consider the impact of a 150 - 200% turnover cost, multiplied by some average salaries in Oregon:

| Position | Avg Oregon salary '98[77] | Multiplier | Cost of Turnover |
|---|---|---|---|
| Computer programmer | $51,600 | 150 – 200% | $77,400 – $103,200 |
| Computer systems analyst | $54,000 | 150 – 200% | $81,000 – $108,000 |
| Computer engineer | $65,000 | 150 – 200% | $97,500 – $130,000 |
| Technical writer | $48,000 | 150 – 200% | $72,000 – $96,000 |
| Computer support specialist | $35,490 | 150 - 200% | $53,235 –$70,980 |
| Database Administrator | $49,810 | 150 - 200% | $74,715 – $99,620 |

In an IT labor market report released September, 1999 the International Data Corp stated, "IT firms that lower their attrition rates can cut their recruiting needs by as much as half. The cost of attrition to a company is often "far greater" than recruiting costs."[78]

"The Costs of Loss" include: newspaper ads; search firm; interview expenses (air fare, hotel, meals, etc.); manager's and team members' time spent interviewing; work put on hold until replacement is on board; overload on team, including overtime to get work done during selection and training of replacement; orientation and training time for replacement; lost customers; lost contracts or business; lowered morale and productivity, time spent talking about it around the water cooler [or online]; sign-on bonus and other perks; moving allowance; and loss of other employees.[79]

These figures would seem to lead anyone concerned with bottom-line to ask a few questions:

- Considering the current job market, who is responsible for attrition and turnover expenses?
- When you know that people are routinely solicited by competitors, who is responsible for keeping good people on board? And how is that demonstrated?
- Is keeping people on-board a priority?
- If retention is valued, who is responsible for creating a "retention culture"[80]?

One company instituted an eye-catching bottom-line approach; they assigned accountability squarely with the manager who let the valued team member get away. "We know of a CEO who charged $30,000 to a manager's operating budget because he needlessly allowed a talented person to leave." [81]

Conflict impacts more than job satisfaction and retention. Conflict impacts product and project performance. Sometimes the problems boil down to two people who can't communicate. Sometimes whole teams or departments don't work well together. Many companies deal with the duality of "hype and craft". [82]

Lack of communication, poor teamwork, faulty team coordination, rivalries, turf battles and the like – all are symptoms of poorly managed conflict. All impact product or project success. All cost money.

## Computing Costs

There are several ways you can compute the costs of conflict.

You can measure the cost in dollars. Add staff turnover expenses to the costs of project or product failures. You may also want to add in the opportunity costs of what could have been, had things been handled a bit differently.

Another method, advocated by conflict resolution system designers, takes a broader approach. Ury et al suggest determining the "costs of disputing" based on four factors: [2]

- Transaction costs (time, money, emotional energy expended in disputing, resources consumed and destroyed, opportunities lost)
- Satisfaction with outcomes (Is the outcome mutually satisfactory? Does it fulfill the interests that led to dispute? Is the outcome fair? Were the procedures fair?)

  Note: fairness has several components, including:

  1. disputants had an opportunity to express themselves
  2. disputants had control over accepting or rejecting the outcome
  3. disputants participating in shaping the settlement
  4. the third party, if there is one, acted fairly
- Effect on relationship (What is the long-term effect of disputing? What approach will best support relationship maintenance/improvement?)
- Recurrence of disputes (How durable is the solution? Does the conflict end or merely transfer to other members in the system?)

In *Designing Conflict Management Systems*, Costantino and Merchant note that most organizational representatives consider the costs of conflict "…not only in terms of dollars and time spent in resolution efforts and litigation, but also in terms of the negative impact on important, ongoing relationships – both within and outside the organization - with employees and customers."[83] In this statement, they point to yet another factor to consider in calculating the cost of conflict: What is the impact on your customers? How will this impact your business relationships, present and future?

When viewed from a conflict resolution system designer's perspective, there is an obvious relationship between effective conflict management and increased retention rates, increased project success rates, and a better bottom-line. The system designer's goal is to create solutions that consider all of the costs of conflict, and to educate and train people so they can manage conflict with the lowest total cost.

## Conclusions

This review of the literature has taken us into exciting territory. When we started it, we suspected certain things may be true, and our research has validated suspicion about the nature of conflict in software development environments. This review is necessarily abbreviated. But it will be ongoing; the annotated bibliography will continue to grow. Certain topics we were not able to take the time to broach. For instance, we suspect that conflict has a regional face, that the reasons and ways in which software developers enter into conflict is different in Portland, Oregon than it is in Portland, Maine, or Austin, Texas, or Atlanta, Georgia. This will likely be one of our next areas of inquiry around the topic of conflict resolution in software development.

What is indisputable regardless of the locale is that conflict is inevitable, in software development environments, as well as in any business or organization. It is a human process, common at some level to all human beings. Successful response and resolution to conflict is highly correlated to employee satisfaction, retention, and productivity. Both the literature review and the results of our local survey point us in this direction. It is clear that the costs of not resolving conflict are high for any organization – costs in employee retention, performance and project failure. And finally, we know that it is a good, and cost effective idea for organizations to spend time and money developing a basic conflict management system, and that this may mean reliance on external resources depending on the conflict and the environment it occurs in.

What then are our conclusions for improving conflict resolution and problem-solving in software development? We have the following four recommendations:

- **Implement skills training for all team members who have to work together.** Skills training in communication and conflict resolution are well worth the cost. The benefits are improved job performance and employee satisfaction, increased retention rates, health and safety, and professional mastery.

- **Hire external mediators or develop internal mediators to resolve conflict in a timely manner**. Mediation helps resolve conflicts efficiently and swiftly, preserves and enhances working relationships, gives people the opportunity to practice conflict resolution skills and improves follow through with the implementation of an agreement.
- **Use partnering techniques at project start-up.** Although partnering would take some time at project start-up, it may well save time during the life cycle of the project.
- **Implement conflict management systems where feasible**, specifically in mid-sized to large software companies which are planning for the long-term. It could be a minimal system like our LookandFeel New Media example developed, or it could be a more elaborate system as described by Slaikeu and Hasson, Ury, Brett and Goldberg or Costantino and Merchant. We suspect that your organizations are used to moving at a fast pace, and therefore believe you could speed the development time when you design a conflict management system.

As Kleiner concludes regarding the clash of cultures between the craft and hype people in dot-com environments,

> Instead of self-consciously trying to build a corporate culture, or aping the worst bureaucratic tendencies of the dinosaur companies they are replacing, there are things that Internet and e-commerce startups…can consider to sustain themselves past, say, this summer. These ideas are tough to implement at breakneck speeds, unless you are willing to think through your work design right at the beginning. And they're counterintuitive, to some extent, to both the culture of hype and the culture of craft, because they start with an explicit appreciation of the differences between the two cultures – differences that are generally unobserved.[84]

Conflict is not to be feared; rather, it can be welcomed, advisedly and with skill, as an opportunity for growth, increased creativity, and a spur to productivity. As Constantino and Merchant have so concisely stated:

> Conflict is like water: too much causes damage to people and property: too little creates a dry, barren landscape devoid of life and color. We need water to survive; we need an appropriate level of conflict to thrive and grow as well. How we manage our natural resources of water through dams, reservoirs, and sluices determines whether we achieve the balance necessary for life. So, too, with conflict management: a balance must be struck between opposing forces and competing interests.[85]

# Appendix A: Results of Survey

In attempt to gather data about conflict in technology development environments, we, in cooperation with the Project Management program at Portland State University's Professional Development Center (PDC) surveyed 261 students who have attended the Project Management series at PDC. Of the 261 students who received the survey, 180 were estimated to be working in technology development. We received 56 responses (31.1 %) from qualifying recipients.

Our major goal was to determine how frequently respondents attributed unresolved conflict and lack of communication to unsuccessful high technology development projects.

The 56 qualified respondents accomplished their work in the following structures:

- 65.45% accomplished their work within cross-functional teams .
- 5.45% within uni-functional teams .
- 27.27% through a group of individuals coordinated by a manager.
- 1,82% through other work systems.

## Questions Asked

We asked respondents to estimate the percentage of technology development projects that have a less than successful final outcome. The average rate of projects falling short of expectations among respondents was 33.1 %.

We then asked respondents to attribute reasons for unsuccessful projects by ranking reasons for overall project failure. To gather more data about why they ranked the causes as they did, we asked them to rate the most frequently occurring factors that:

- Impact project success.
- Most adversely impact team member satisfaction.
- Most adversely impact team member performance.

Choices offered included:

- Conflict that is not addressed effectively and/or resolved (Conflict)
- Incompetent team members (Incompetence)
- Impossible deadlines - not enough time (Time)
- Instability/changes in company structure and/or focus (Instability)
- Insufficient resources - not enough money, people, and/or tools (Resources)
- Lack of communication - among team members and/or between teams (Communication)
- Lack of planning before implementation (Planning)
- Unclear goals (Goals)

51 out of 56 respondents, or 98%, responded to these rating questions. The tables below illustrate our findings.

*Table 1. Ranking Reasons for Overall Project Failure*

The scale for this ranking set *1* as the most important/impactful reason projects are less than successful, and *8* as the least important/impactful reason projects are less than successful.

| **Reason** | **Average Ranking Score** | **Response Rate (n=56)** |
|---|---|---|
| Planning | 3.5 | 82% |
| Goals | 3.9 | 88% |
| Resources | 4.0 | 86% |
| Communication | 4.1 | 88% |
| Time | 4.2 | 84% |
| Instability | 4.7 | 84% |
| Conflict | 5.2 | 79% |
| Incompetence | 6.3 | 82% |

Note that, explicity, conflict is ranked quite low. This is consistent with our expectation that conflict is not recognized as a cause for project failure, but rather "part of the job to be endured" that it "comes with the territory".

Note also that Communication is ranked midway in this distribution. Now note that Planning and Goals are ranked highest. Planning is a communication process and commonality of goals across a project requires good communication skills. We suspect that poor planning and lack of common goals operate in the project to generate conflict. Unresolved conflict can masquerade as a communication problem, which ultimately it is. However, simply communicating without effectively applying conflict management skills will not resolve conflict, facilitate truly effective communication, or improve the planning or goals development processes.

*Table 2. Most Frequently Occurring Reasons That Projects Are Less Than Successful*

| **Reason** | **Rating** | **Percentage (n=56)** |
|---|---|---|
| Resources | 1 | 18.07% |
| Planning | 2 | 16.87% |
| Communication | 3 | 15.06% |
| Time | 4 | 13.86% |
| Goals | 5 | 12.65% |
| Instability | 6 | 11.45% |
| Conflict | 7 | 8.43% |
| Incompetence | 8 | 3.61% |

The placement of Planning, Communication, and Conflict remains fairly consistent for this question. Since this question is about frequency rather than intensity, this indicates to us that, overall, these three contributors remain constant across the projects these respondents have experienced.

*Table 3. Most Frequently Occurring Reasons that Most Adversely Impact Team Member Satisfaction*

| Reason | Rating | Percentage |
|---|---|---|
| Communication | 1 | 20.13% |
| Conflict | 2 | 15.72% |
| Goals | 3 | 15.09% |
| Time | 4 | 13.21% |
| Resources | 5 | 12.58% |
| Instability | 6 | 10.06% |
| Planning | 7 | 7.55% |
| Incompetence | 8 | 5.66% |

This response is interesting in light of the discussion of how team member satisfaction impacts retention (see

*Conflict Skills and Professional Mastery* on page 17). Communication and Conflict rank highest on this rating. While the respondents focused on Planning and Goals as contributors to overall project failure, they focused on Communication and Conflict when asked the question most directly related to retention.

*Table 4. Most Frequently Occurring Reasons that Most Adversely Impact Team Member Performance*

| Reason | Rating | Percentage |
|---|---|---|
| Communication | 1 | 20.73% |
| Goals | 2 | 15.85% |
| Planning | 3 | 12.20% |
| Conflict | 4 | 10.98% |
| Incompetence | 5 | 10.98% |
| Instability | 6 | 10.37% |
| Resources | 7 | 9.76% |
| Time | 8 | 9.15% |

It is again interesting to see the disconnect between Communication and Conflict occur again in this rating. This again supports our expectation that conflict skills training is "not on the radar" in this industry as a possible solution to communication problems, though human communication experts have long acknowledged conflict skills as trainable and a key component in effective communication.

### Summary and Conclusions

Though poor communication is more frequently identified as being related to unsuccessful projects, poor communication is one of the reasons conflict occurs.

Although conflict is ranked and rated low in the first two tables (overall reasons for less than successful project outcome), it is much more highly correlated as a reason for team member's dissatisfaction (a #2 rating) and adversely affecting team member performance (a # 4 rating). This indicates that improving communication and conflict resolution methods in technology development environments would increase team member satisfaction and performance, as well as retention, according to research cited elsewhere in this paper.

It appears that if technology development companies are concerned with building and retaining high-performing project teams, successfully resolving conflict should be high on their priority list. Further, these results indicate that further ethnographic research or contextual inquiry should be applied to the topics covered by this survey to gather more specific data about the disconnect in the minds of the respondents between communication and conflict and between poorly managed conflict and project success.

## Acknowledgement

We would like to thank our assistant, Lisa Latin, for her invaluable support in the research and development of this paper. Without her support and urging, our schedules and competing priorities would have precluded this paper being finished in the foreseeable future. While we preferred to stir the pot of the present, she reminded us that today is yesterday's tomorrow.

## Sources Cited

---

[1] Folberg, Jay and Taylor, Alison, 19.

[2] Fisher, Roger and Ury, William, 74.

[3] Humphrey, Watts S. Managing Technical People. Reading (MA):Addison Wesley Longman, 1997, 177-179.

[4] Humphrey, 178.

[5] Humphrey, 178.

[6] Humphrey, 179.

[7] Humphrey, 197-198.

[8] Humphrey, 198.

[9] Kleiner, Art. "Corporate Culture in Internet Time." Culture & Change :18 – 24.

[10] Souder, William E. "Managing Relations between R & D and Marketing in New Product Development Projects." Journal of Product Innovation Management 5 (1988): 6 – 19. Reprinted in Ralph Katz, ed., The Human Side of Managing Technological Innovations. Oxford: Oxford University Press, 1997. 523-534.

[11] Kleiner, 20.

[12] Kleiner, 19.

[13] Kleiner, 21.

[14] Gruber, Howard E. "Creativity and Conflict Resolution." The Handbook of Conflict Resolution. San Francisco: Jossey-Bass, 2000, 345.

[15] Johnson, David W., Johnson, Roger T. and Tjosvold, Dean. "Constructive Controversy." The Hanbook of Conflict Resolution. San Francisco: Jossey-Bass Publishers, 2000, 66.

16 Johnson, 73.

17 Coleman, Peter T. and Deutsch, Morton. "Some Guidelines for Developing a Creative Approach to Conflict." The Hanbook of Conflict Resolution. San Francisco: Jossey-Bass Publishers, 2000., 355.

18 Coleman, 356.

19 Coleman, 361.

20 Coleman, 364-365.

21 Bridges, William. Managing Transitions: Making the Most of Change. Reading (MA): Addison-Wesley Publishing Company, 1991.

[22] Burk, Lisa. Training materials. June 2000.

[23] Burk.

24 Gobeli, David H., and Koenig, Harold F., and Bechinger, Iris. "Managing Conflict in Software Development Teams: A Multilevel Analysis." Journal of Product Innovation Management 15 (1998): 423-435.

25 Raider, Ellen, Coleman, Susan and Gerson, Janet. "Teaching Conflict Resolution Skills in a Workshop." The Handbook of Conflict Resolution. San Francisco: Jossey-Bass Publishers, 2000, 499–500.

26 Raider, 504–518.

27 Slaikeu, Karl A. and Hasson, Ralph H. Controlling the Costs of Conflict. San Francisco: Jossey-Bass, 1998, 122.

28 Slaikeu, 124–125.

29 Kaye, Beverly and Jordan-Evans, Sharon. "Retention: Tag, You're It! How to Build a Retention Culture." Training & Development, April 2000: 29–34, 29–30.

[30] Kaye.

31 Executive Summary - Bridging the Gap: Information Technology Skills for a New Millennium. 11 May 2000. Information Technology Association of America. <<www.itaa.org/workforce/studies/hw00execsumm.htm>>

32 Bridges, William. The Character of Organizations: Using Jungian Type in Organizational Development. Palo Alto (CA): Davies-Black Publishing, 1992.

33 Scannell, Tim. IT Worker Gap Will Hit 1.2 Million. 29 June 1999. TechWeb: Technology News. 18 May 2000 <<www.techweb.com/wire/story/TWB19990629S0010>>

34 Brinkman, Rick and Kirschner, Rick. Dealing With People You Can't Stand., xvii.

35 Pinto, Jeffrey K. and O.M. Kharbanda. "Lessons for an Accidental Profession." Business Horizons (March – April 1995): 41 – 50. Reprinted in Ralph Katz, ed., The Human Side of Managing Technological Innovations. Oxford: Oxford University Press, 1997. 215 - 226.

36 Costantino, xiii–xiv.

[37] Ury, William L., and Jeanne M. Brett and Stephen B. Goldberg. Getting Disputes Resolved: Designing Systems to Cut the Costso of Conflict. Cambridge (MA): The Program on Negotiation at Harvard Law School, 1993.

38 Slaikeu,. 13.

39 Logan, Mark. Design Shop Counseling. 5 June 2000. Designshops.com: Design Shop Counseling. <www.Designshops.com>>" page 1.

40 Kleiner

41 Costantino, 5.

42 Costantino, 6.

43 Costantino, 16.

44 Costantino, 33.

45 Costantino, 36-37.

46 Costantino, 3-4.

47 Kleiner

48 Costantino, 17. and Bridges, William. The Character of Organizations: Using Jungian Type in Organizational Development.

[49] French, Wendell L., Bell, Cecil H. Jr. and Zawacki, Robert A. Organizational Development and Transformation – Managing Effective Change. Boston: Irwin McGraw-Hill, 2000, 6.

50Slaikeu, 17-18.

[51] Ury

52 Jameson, Jessica Katz. "Toward A Comprehensive Model for the Assessment and Management of Intraorganizational Conflict: Developing the Framework." The International Journal of Conflict Management 10:3 (July, 1999): 268-294, 268.

[53] Burk, Lisa. Volunteer Manual Basic Mediation Training 2000. A joint work between the City of Portland Neighborhood Mediation Center and CommunicationWorks.

54 "Construction Industry Board - Partnering in the team: Introduction and executive summary." Thomas Telford Publishing. 7 November 1999. <<http://www.t-telford.co.uk/pu/cib_part1b.html>>

55 "Shared Neutrals." Oregon Mediation Association Newsletter Spring 1999.

56 Equal Employment Opprtunity Commission - Federal Management Directive 110, Chapter 3. November 1999.

57 Logan, 2.

58 Logan, 2.

59 Logan, 3.

60 Logan, 3-4.

61 "Meta study found demand for IT workers high after Y2K; found current turnover 11 to 20%." PC Week 22 February 1999. ITAA website: workforce resources/articles/IT workforce studies and statistics. 10 June 2000. <<http://www.itaa.org/workforce/resources/articles.htm>>

62 Hecker, Daniel. "High-technology employment: a broader view." Monthly Labor Review (June 1999): 18 – 28.

63 Oregon State. Oregon Employment Department Research and Analysis. Portland-Vancouver PMSA Economic Profile. Salem (OR): Oregon Employment Dept. 19 May 2000.

64 Oregon Software Industry 2000 Directory. Tigard (OR): Software Association of Oregon. 2000.

65 "Abundant career opportunities projected in information technology." Monthly Labor Review – The Editor's Desk. 28 October 1998. US Dept. of Labor Bureau of Labor Statistics. 28 June 2000 <<http://www.bls.gov/opub/ted/1998/oct/wk4/art03.htm>>

66 Scannell

67 Oregon Software Industry 2000 Directory.

68 Mateyaschuk, Jennifer. "1999 National IT Salary Survey: Pay Up." Information Week 4 April 1999. Information Week Online. 28 June 2000. <<http://informationweek.com/731/salsurvey.htm>>

69 "Hewitt study finds that demand for 'hot skills' continues." Hewitt Associates Press Release 29 July 1999. ITAA website: workforce resources/articles/IT workforce studies and statistics. 10 June 2000. <<http://www.itaa.org/workforce/resources/articles.htm>>

70 "Meta study found demand for IT workers high after Y2K; found current turnover 11 to 20%."

71 "Survey reveals turnover rates in information technology." RHI Press release 16 December 1998. ITAA website: workforce resources/articles/IT workforce studies and statistics. 10 June 2000. <<http://www.itaa.org/workforce/resources/articles.htm>>

72 "Meta study found demand for IT workers high after Y2K; found current turnover 11 to 20%."

73 Scannell

74 "Hewitt study finds that demand for 'hot skills' continues."

75 Kaye

76 "Meta study found demand for IT workers high after Y2K; found current turnover 11 to 20%."

77 Occupational Information Center. 28-6-2000. OLMIS (Oregon Labor Market Information System). <<http://olmis.emp.state.or.us/>>

78 “IDC study finds current worlwide shortage of 1 million workers; projects shortage of 850,000 IT workers over next 3 years.” Newsbytes 21 September 1999. ITAA website: workforce resources/articles/IT workforce studies and statistics. 10 June 2000. <<http://www.itaa.org/workforce/resources/articles.htm>>

79 Kaye

80 Kaye

81 Kaye

82 Kleiner

83 Costantino, Cathy A and Merchant, Christina Sickles. Designing Conflict Management Systems. San Francisco: Jossey-Bass, 1996.

84 Kleiner

85 Costantino, xii.

## Sources Consulted

"Abundant career opportunities projected in information technology." Monthly Labor Review – The Editor's Desk. 28 October 1998. US Dept. of Labor Bureau of Labor Statistics. 28 June 2000 <<http://www.bls.gov/opub/ted/1998/oct/wk4/art03.htm>>

Information technology jobs are projected to be among the fastest growing occupation between 1996 and 2006. Includes projected employment rates and listing of jobs classified as IT.

Avruch, Kevin, Peter W. Black, and Joseph A. Scimecca. Conflict Resolution. New York: Greenwood Press, 1991.

The first part of the book provides the reader with the history and roots of conflict resolution. The remainder of the book has many articles that examine conflict resolution in different cultural contexts.

Blessing/White, Skillman. "Retention Rodeo: How to Keep Technical Professionals From Straying." Training & Development April 2000: 20.

Improve the skill level of managers who supervise tech professionals. Provide management development training, including effective leadership skills. Clarify your understanding of employees' needs. Reinforce frequent communication.

Bowers, Brent. "On the Fast Track, Those in Charge Must Change." New York Times 24 Nov. 1999, late ed. (east coast): sec. C: 8.

Adjusting to growth is one of the toughest tasks managers face. Anecdotal story demonstrates the trials and tribulations of success: shifting management styles, streamlining communication, and counting minutes saved.

Bridges, William. Managing Transitions: Making the Most of Change. Reading (MA): Addison-Wesley Publishing Company, 1991.

Bridges distinguishes between change and transition (transition being the human process and response to change), and draws our attention to "the neutral zone" as a critical mid-point of transition: simultaneously very challenging, and ripe with creative opportunity. In an organization, the neutral zone can be described by rising anxiety, falling motivation, and the re-emergence of old weaknesses and disputes. Transitions can be thoughtfully and successfully managed; the author offers numerous suggestions, as well as commentary on managing in a world of non-stop change.

Bridges, William. The Character of Organizations: Using Jungian Type in Organizational Development. Palo Alto (CA): Davies-Black Publishing, 1992.

Organizations, like individuals, have inherent characteristics that must be understood and respected if real growth is to take place. Based on the work of Carl Jung and the Myers-Briggs Type Indicator, Bridges explains why organizations act the way they do, and how an understanding of organizational character benefits an organization as well as the individuals in an organization. Bridges looks at growth, change, conflict and organizational development through the myriad perspectives held by different "types". Filled with theory, examples, and

practical suggestions. Useful reading for navigating in a diverse world. Includes the Organizational Character Index for self-assessment.

Brinkman, Rick and Kirschner, Rick. Dealing With People You Can't Stand.

Two naturopathic physicians discuss practical strategies for responding to conflict based on differences in interpersonal style. The authors come from the perspective that unmanaged conflict is dangerous to the physical health of the disputants, based on their experience as physicians.

Buie, Elizabeth A. "Personality and System Development: What's the Connection?" System Development January 1988.

<http://www.aesthetic-images.com/ebuie/article_type_and_sd.html>

System developers (referred to as "SDers") represent a disproportionate share of certain personality types. Discusses the Myers-Briggs Type Inventory (MBTI) as it relates to personalities typical in the software development environment. Also discusses the benefits of applying MBTI understandings in your environment. "The Myers-Briggs approach to personality is non-judgmental: no preference or type is inherently better than any other. Each type contributes to teamwork. Different types do tend to choose different occupations and work environments, but this doesn't guarantee that an individual will be suitable (or nsuitable) for work that is generally chosen (or avoided) by his or her type. . . . "What's more, over half of all SDers are ISTJs, INTJs, or INTPs (highlighted areas), while only about 8.2% of Americans belong to one of those three groups."

Burk, Lisa. Volunteer Manual Basic Mediation Training 2000.

A joint work between the City of Portland Neighborhood Mediation Center and CommunicationWorks.

Christian & Timbers. "What Opens the Barn Door, or Why Employees Leave." Training & Development April 2000: 21.

Top ten reasons employees leave.

Coleman, Peter T. and Deutsch, Morton. "Some Guidelines for Developing a Creative Approach to Conflict." The Handbook of Conflict Resolution. San Francisco: Jossey-Bass Publishers, 2000.

The authors analyze how conflict facilitates creativity and specifies seven guidelines for dealing with creativity and conflict. They also address the issue of fostering "optimal" tension and suggest some ways of generating novel ideas.

"Construction Industry Board - Partnering in the team: Introduction and executive summary." Thomas Telford Publishing. 7 November 1999 <<http://www.t-telford.co.uk/pu/cib_part1b.html>>

Background, introduction to, and purpose of industry report for construction professionals, clients and advisors. Report includes five Partnering case studies.

Costantino, Cathy A and Merchant, Christina Sickles. Designing Conflict Management Systems. San Francisco: Jossey-Bass, 1996.

This book gives a comprehensive blueprint for designing conflict management systems for organizations. The provide a brief history of both ADR and Dispute System Design (DSD), as well as a step by step process for DSD.

Dobrzynski, Judith H. "Online Pioneers: The Buzz Never Stops." New York Times 21 Nov. 1999, late ed. (east coast), sec. 3: 1.

Interview with four dot.com chief executives. A wide ranging converstion touching on the need for speed and sleep, employee retention, managing growth, travel, building a company, values and the unique characteristics of managing dot.coms.

Edleman, Joel and Crain, Mary Beth. The Tao of Negotiation. HarperBusiness, 1993.

The nature of conflict is discussed with a focus on resolving conflict through negotiation. A strategy for preparing to negotiate or resolve conflicts in the face of anticipated resistance is discussed in depth. The uniqueness of conflict in the workplace is discussed.

Eisenhardt, Kathleen M. "Speed and Strategic Choice: How Managers Accelerate Decision Making." California Management Review 32, no. 3 (1990). Reprinted in Ralph Katz, ed., The Human Side of Managing Technological Innovations. Oxford: Oxford University Press, 1997. 424-343.

How do you make fast, high-quality strategic decisions? A study of twelve microcomputer firms reveals that fast decision making is essential when technical and competative changes are rapid. Successful leaders constantly gather and use real time information, meet with key people frequenlty, have seasoned counsel, and make decisions by "consensus with qualification". Being fast involves accelerated information processing, building confidence, and maintaining the cohesiveness of the group.

Equal Employment Opprtunity Commission - Federal Management Directive 110, Chapter 3. November 1999.

Overview of statutes enforced by EEOC and executive orders encouraging the use of ADR in resolving employment disputes in all federal agencies. This directive outlines ADR program design, information to be provided to disputants, ADR core principles, training qualifications for "neutrals," format of resolutions and ADR definitions.

Executive Summary - Bridging the Gap: Information Technology Skills for a New Millennium. 11 May 2000. Information Technology Association of America. <www.itaa.org/workforce/studies/hw00execsumm.htm>

Summary of "one of the largest and most comprehensive studies of the IT workforce ever conducted." Includes current figures on growth, demand & worker shortages; details on where the jobs are; and how to acquire skills.

Fisher, Roger and Ury, William. Getting to Yes. (second edition) New York: Penguin Books, 1991.

This was the premiere book on negotiation that "coined" the term "interest-based" negotiation. They take the reader through a 4 step win/win methodology that is easy to read and straightforward. It was first published in the United States in 1981 and is used as a basic text in most conflict management and negotiation training. The second edition doubled the

size of the book and answers four questions about principled negotiation, dealing with irrational people, tactics and power. This book is also full of great real life examples.

Fitzgerald, Beth. "No Hard Feelings: Strive to Resolve Conflicts in the Workplace Before They Damage Your Company." San Jose Mercury News 25 April 2000, morning final: 18C.

Corporate executives and managers are turning to conflict training and conflict management consultants to help guide them through internal strife and resulting casualties brought on by increasing demands and ever-faster timelines. Corporate America is recognizing the need to acknowledge conflict and deal with it productively and proactively.

Folberg, Jay and Taylor, Allison. Mediation: A Comprehensive Guide to Resolving Conflicts Without Litigation. San Francisco: Jossey-Bass Publishers, 1984.

This is a classic first text on mediation written by two Northwest authors. The book briefly explores conflict, but goes into depth outlining the mediation process and applying the process in many different contexts – labor, community, crisis, family, divorce, workplace, etc. Finally, the authors delve into the ethical, educational and practical issues of the profession.

Ford, John. Workplace Conflict: Facts and Figures. 16-4-00. Jossey Bass Publishers. <http://www.josseybass.com/index-con.html>

Abstracts and links to workplace conflict facts and figures.

French, Wendell L., Bell, Cecil H. Jr. and Zawacki, Robert A. Organizational Development and Transformation – Managing Effective Change. Boston: Irwin McGraw-Hill, 2000.

A classic, comprehensive text on organizational development. Various articles and essays take the reader through the history, applications, interventions, strategies and challenges in the field of organizational development.

Gibbs, Mark. "Managing By Messaging." Network World 14 Sept. 1998.

Overreliance on email as a communication channel is extremely common. Many managers dive behind their computers and issue edicts that manage the process rather than the people. Includes strategies for managing by email.

Gobeli, David H., and Koenig, Harold F., and Bechinger, Iris. "Managing Conflict in Software Development Teams: A Multilevel Analysis." Journal of Product Innovation Management 15 (1998): 423-435.

For New Product Development (NPD) organizations, a little conflict can be a good thing; the tension can engender innovation. The authors note that conflict must be managed not only for the satisfaction of team members, but also to achieve strategic project success. In a study of 117 respondents, unresolved conflict had a strong, negative effect on overall software product success and customer satisfaction. Impact of various conflict styles is considered, with a recommendation to resolve conflict with true problem solving (i.e., collaboration).

Gordon, Corbett, Attorney at Law. Personal interview. 28 June 2000.

Conversation regarding employment law and dispute resolution vis-à-vis technology development environments.

Gruber, Howard E. "Creativity and Conflict Resolution." The Handbook of Conflict Resolution. San Francisco: Jossey-Bass, 2000.

The relationship between creativity and conflict is explored from the perspective of the importance of point of view. Using two experiments with human subjects, Gruber illustrates his hypothesis that free and open communication facilitates problem solving. This article indicates that developing the skill of assuming multiple points of view is a creative exercise critical to conflict resolution.

Hecker, Daniel. "High-technology employment: a broader view." Monthly Labor Review (June 1999): 18 – 28.

High technology employment, 14% of total employment, is projected to grow much faster than in the past due to employment gains in high-tech services and among suppliers to computer and electronic components manufacturers.

"Hewitt study finds that demand for 'hot skills' continues." Hewitt Associates Press Release 29 July 1999. ITAA website: workforce resources/articles/IT workforce studies and statistics. 10 June 2000. <<http://www.itaa.org/workforce/resources/articles.htm>>

Competition has never been keener for IT employees with " hot" skills. IT professionals with the hottest skills are often receiving base pay increases in the 10-20% range per year, compared to national average annual increases of 4-5% for non-IT salaried exempt U.S. employees. Turnover of IT employees averages just below 16%, but some companies in the study reported turnover rates as high as 35%. The average time reported for filling an IT position was three months. Twenty-eight percent of companies reported at least 10% of their high tech positions were vacant, and while the vacancies are down from 1998 reports, 75% reported using contract IT employees in some capacity.

Humphrey, Watts S. Managing Technical People. Reading (MA):Addison Wesley Longman, 1997.

Includes practical highlights and tools for leading technical professionals. He focuses on the critical role of innovative people, and gives concrete advice on how to identify, motivate and organize people into highly productive teams.

"IDC study finds current worldwide shortage of 1 million workers; projects shortage of 850,000 IT workers over next 3 years." Newsbytes 21 September 1999. ITAA website: workforce resources/articles/IT workforce studies and statistics. 10 June 2000. <<http://www.itaa.org/workforce/resources/articles.htm>>

Worldwide, the shortage of IT workers currently stands at more than 1 million - a problem will only intensify during the next millennium. The US will experience intense recruiting for nearly 850,000 IT openings in 2002. IT firms that lower their attrition rates can cut their recruiting needs by as much as half. The cost of attrition to a company is often "far greater" than recruiting costs. Based on two separate reports: "The Resource Gap in the IT Industry: Too Much Work, Not Enough Skilled People" and "Employing Critical IT Talent in the 21st Century."

IT Workforce Studies and Statistics. 11 May 2000. Information Technology Association of America. <www.itaa.org/workforce/resources/articles.htm>

Compilation of abstracts and links addressing workforce studies, employment projections, IT industry compensation trends, retention and turnover, state level workforce studies and international workforce studies.

Jameson, Jessica Katz. "Toward A Comprehensive Model for the Assessment and Management of Intraorganizational Conflict: Developing the Framework." The International Journal of Conflict Management 10:3 (July, 1999): 268-294.

The author proposes framework for a model of intraorganizational conflict assessment and management. A contingency-based model, it would consider conflict characteristics, desired outcomes, and awareness of available conflict management strategies, with an aim to assist in identifying the most appropriate conflict management strategy for a given conflict.

Johnson, David W., Johnson, Roger T. and Tjosvold, Dean. "Constructive Controversy." The Hanbook of Conflict Resolution. San Francisco: Jossey-Bass Publishers, 2000.

A method of addressing conflict in a manner which generates the highest level of learning, the best and most creative decisions, and the greatest degree of personal growth and psychological health is discussed. Using case studies to describe their theory, the authors describe how to practice this method in a variety of contexts.

Katz, Ralph, ed. The Human Side of Managing Technical Innovation. New York: Oxford University Press, 1997.

A comprehensive collection of readings that give a variety of insights into successfully managing the people in technology environments. The readings cover motivating professional performance, managing creativity, managing project team dynamics, the role of project managers and leaders, cultural differences, decision-making processes and organizational practices and policies. There are also several readings that link successful project teams and managers with resolving conflict; however, there is little in the way of practical skills for managing conflict.

Kaye, Beverly and Jordan-Evans, Sharon. "Retention: Tag, You're It! How to Build a Retention Culture." Training & Development April 2000: 29 – 34.

Retention is corporate America's number 1 challenge. Of 50 identified retention factors, pay is the least important. HR has to lead retention efforts by making managers the owners – responsible and accountable for keeping good employees.

Kepner-Tregoe Business Issues Research Group. "Avoiding the Brain Drain: What Companies Are Doing to Lock in Their Talent." Training & Development April 2000: 20.

Researchers studying high employee retention identified practices common to organizations dubbed "Retention Leaders", including having a stairstepping process for conflict resolution; offering legitimate alternative avenues that allow employees to circumvent their immediate supervisor, if necessary, to get their problems resolved; and viewing people management as a strategic business issue.

Kleiner, Art. "Corporate Culture in Internet Time." Strategy and Business :18 – 24. First Quarter, 2000.

The author discusses the impact of the tension between two cultures essential to technology development, those of hype and craft. He describes how e-commerce varies from

traditional businesses in this cultural aspect. He goes on to discuss how to develop and support effective project teams in the context of e-business.

Kreitzber, Charles B., Ed. "Usability as Therapy." The UPA Voice 2:2. <http://www.upassoc.org/voice/vol2no2/index.htm>

Postulates a conflict management role for usability professionals as e-commerce evolves. Speaks particulary to the lack of alignment between "developers" and "the business". " As we enter the second act of the e-business transformation, the cost and riskof business' failure to assume full partnership with developers isunacceptable."

Kwahk, K-Y and Kim, Y-G. "A Cognitive model based approach for organizational conflict resolution." International Journal of Information Management 18:6 (1998): 443-456.

The authors analyze organizational conflict, seek to identify cause-effect relationships, and propose a model for capturing and alleviating conflict.

Lebedun, Jean. Managing Workplace Conflict. West Des Moines: America Media Publishing, 1998.

A self-help work book with a series of "tests" the reader can take to reflect on their individual response to conflict, analyze a conflict situation, resolve a real conflict in four easy steps, etc. There are some examples, but mostly, the authors ask the reader to reflect about conflicts in their own life.

Logan, Mark. "Design Shop Counseling." 5 June 2000. Designshops.com. <www.Designshops.com>

This article describes how a dot com company in Kansas City faced internal communication problems. In response, they hired a communication consultant, who walked them through a series of trainings and meetings that improved internal communication and resulted in a remarkable turnaround in all areas of their work.

Maruca, Regina Fazio. "How Do You Manage an Off-Site Team?" Harvard Business Review July-August 1998.

Case study demonstrating the challenges of managing an off-site team when team members are in conflict with one another. Four experts offer their advice on the complexities of managing off-site employees.

Mateyaschuk, Jennifer. "1999 National IT Salary Survey: Pay Up." Information Week 4 April 1999. Information Week Online. 28 June 2000. <<http://informationweek.com/731/salsurvey.htm>>

Reports on a survey of more than 21,000 respondents: IT salaries continue to climb. IT managers earn a median base salary of $71,000, up 9.2% from 1998. IT staff members earn a median annual base salary of $54,000, up 8% from 1998. Job-hopping is common, with staffers moving on after four years, while managers stay about five years. That's not surprising, considering 70% have been contacted by headhunters this year.

Maydan Nicotera, Anne, ed. Conflict and Organizations-Communicative Processes. Albany: State University of New York Press, 1995.

This edited collection examines conflict management (vs. conflict resolution) through the lens of communication theory. Most works on organizational conflict focus on negotiation processes, usually with a prescriptive bias. This collection focuses on conflict in three broad categories: ways of thinking about organizational conflict, individual processes within the organization, and interaction processes in organizational conflict.

McCarthy, Jim. Dynamics of Software Development. Redmond (WA): Microsoft Press, 1995.

The author discusses the practical aspects of managing software product development teams based on his personal experience. His comments regarding establishing shared vision, "the most difficult feat of all to pull off", bear on the issue of communication across the project team.

"Meta study found demand for IT workers high after Y2K; found current turnover 11 to 20%." PC Week 22 February 1999. ITAA website: workforce resources/articles/IT workforce studies and statistics. 10 June 2000. <<http://www.itaa.org/workforce/resources/articles.htm>>

Summary of IT labor report. Meta Group predicted no relief from the IT labor shortage. Retention is a major issue, with turnover averaging 11 percent to 20 percent and even, in some corporations, hitting the 50 percent mark. Such turnover is painful: The cost of replacing an IT worker is estimated at one and one-half to two times annual salary. IT salaries are a factor in staffing problems, with an average 20 percent annual increase. That contrasts with 4 percent annual increases for non-IT salaries.

Occupational Information Center. 28-6-2000. OLMIS (Oregon Labor Market Information System). <http://olmis.emp.state.or.us/>

OLMIS offers a variety of information, including wage and employment statistics for Oregon. Wages cited are as of 1998.

Oregon Software Industry 2000 Directory. Tigard (OR): Software Association of Oregon. 2000.

Offering an annual snapshot of the software industry in Oregon, the SAO report includes industry overview, funding and resource information, schools, and an industry directory.

Oregon State. Oregon Employment Department Research and Analysis. Portland-Vancouver PMSA Economic Profile. Salem (OR): Oregon Employment Dept. 19 May 2000.

Excel spreadsheet detailing Portland-Vancouver's Economic Profile, 1989 – 1999. Received file 19 May 2000.

Pinto, Jeffrey K. and O.M. Kharbanda. "Lessons for an Accidental Profession." Business Horizons (March – April 1995): 41 – 50. Reprinted in Ralph Katz, ed., The Human Side of Managing Technological Innovations. Oxford: Oxford University Press, 1997. 215 - 226.

Based on interviews with dozens of senior project managers, the authors offer The Vital Dozen: 12 Points for Project Managers to Remember. Among them, "Recognize project team conflict as progress."

Raider, Ellen, Coleman, Susan and Gerson, Janet. "Teaching Conflict Resolution Skills in a Workshop." The Handbook of Conflict Resolution. San Francisco: Jossey-Bass Publishers, 2000.

The authors describe insights gained from teaching negotiation and mediation to adult learners, and suggest workshop models for conflict resolution training.

Rahim, M. Afzalur and Albert Blum. Global Perspectives on Organizational Conflict. Westport (CT): Praeger, 1994.

The authors analyze organizational conflict orientation and management practices in France, Japan, The Netherlands, Norway, South Africa, Spain and Turkey. Each analysis looks at social, cultural and economic factors; managerial styles; styles of handling interpersonal conflict; alternative dispute management; conclusions and implications.

Ruch, Will. "How to Keep Gen X Employees From Becoming X-Employees." Training & Development April 2000: 43.

Studies show that, on average, US companies experience a 50 percent turnover of employees every four years.

Ryals, Frank, Oregon Employment Dept. Telephone interview. 8 June 2000.

Discussion regarding software industry employment in Oregon.

Salopek, Jennifer J. "Listen Up!" Training & Development April 2000: 15.

Managers and executives must increase their ability to listen. Researcher Rex Gatto found that 89% of managers showed a high level of assertiveness and an ability to succincty present their points of view. They also tend to be challenging or argumentative, don't listen to others, and cut off real communication. Managers tend to be task-oriented, they do not to shy away from conflict, they fight harder, and are less disturbed by disagreement. They favor brief, focused communication, and tend to respond in such a manner. It all adds up to a glaring need for managers to develop better listening skills.

Scannell, Tim. IT Worker Gap Will Hit 1.2 Million. 29 June 1999. TechWeb: Technology News. 18 May 2000 <<www.techweb.com/wire/story/TWB19990629S0010>>

The IT worker shortage is getting worse. As turnover climbs, companies struggle to find and keep people with technical and business skills. A wish-list of skills needed includes understanding of business modeling, project management, leadership, teamwork, and communication.

"Shared Neutrals." Oregon Mediation Association Newsletter Spring 1999.

Quarterly newsletter describes and details success of the Shared Neutrals mediation network program, as implemented by various governmental agencies in Oregon.

Singer, Linda R. Settling Disputes: Conflict Resoluton in Business, Families, and the Legal System. Boulder: Westview Press, 1990.

A primer for the beginning conflict resolution specialist. This book gives the reader a history of the dispute settlement movement, defines conflict resolution techniques and gives example of dispute resolution in various contexts – from families, communities, and businesses to the legal system.

Slaikeu, Karl A. and Ralph H. Hasson. Controlling the Costs of Conflict. San Francisco: Jossey-Bass, 1998.

The authors base their work on the premise that there is something wrong with the way most businesses manage conflict. Weak conflict resolution systems (higher authority, power plays and avoidance) equal high costs for businesses. The authors examine how collaboration saves money, and how mediation saves time and money.

Souder, William E. "Managing Relations between R & D and Marketing in New Product Development Projects." Journal of Product Innovation Management 5 (1988): 6 – 19. Reprinted in Ralph Katz, ed., The Human Side of Managing Technological Innovations. Oxford: Oxford University Press, 1997. 523-534.

Research and development teams (R & D) and marketing teams depend on each other for the creation of new products. Yet, in nearly two-thirds of the 289 projects studied, their work relationship was disharmonious, often very much so. Disharmony between R & D and marketing continues to be prevalent, chronic and disruptive to successful new product development. The author suggests a variety of remedies, all related to communication, an acknowledgement of interdependence, and a need to work together as one team.

Steckler, Niki, Oregon Graduate Institute. Telephone interview. 20 June 2000.

Conversation regarding the state of conflict management skills in the software development industry.

"Survey of Technical Development Professionals." Portland State University – Professional Development Center. Email survey. June 2000.

Survey sent to 180 qualified contacts, primarily project managers in technology development arenas. 56 responded (31.1%). Questions concerned development environment (hardware, software, telecommunications, biotechnology, other), customer (business to business, business to consumer, internal use, other), team configuration (cross-functional team, uni-functional team, individuals coordinated by a manager, other), percentage of projects that fall short of expectations, cause of project failures, impact of various causes on project failure, impact on job satisfaction and job performance of various causes of project failure.

"Survey reveals turnover rates in information technology." RHI Press release 16 December 1998. ITAA website: workforce resources/articles/IT workforce studies and statistics. 10 June 2000. <<http://www.itaa.org/workforce/resources/articles.htm>>

With today's record-low unemployment levels, the issue of staff retention is top of mind for technology executives. But how much turnover is "normal" in the industry? In a recent survey, chief information officers (CIOs) said the average annual turnover rate within a typical IT department is 19 percent, and about one-fifth of respondents cited attrition rates of more than 25 percent. The survey includes responses from 1,400 CIOs from a stratified random sample of U.S. companies with more than 100 employees.

Tamler, Howard, Usability Consultant. Telephone interview. 15 June 2000.

Conversation regarding the state of conflict management skills in the software development industry.

Ury, William L., and Jeanne M. Brett and Stephen B. Goldberg. Getting Disputes Resolved: Designing Systems to Cut the Costso of Conflict. Cambridge (MA): The Program on Negotiation at Harvard Law School, 1993.

Outlines a framework for understanding the dispute resolution process, with case studies. Examining the relationship between interests, rights and power, the authors advocate interest-based negotiation, detail the practical realities/costs of conflict, and offer guidelines for dispute resolution system diagnosis and design.

Van Slyke, Erik. "Resolving Team Conflict." PM Network June 2000: 85 – 88.

Misunderstandings, personality clashes, and differences of opinion are standard fare for team interaction. They can be an opportunity for growth and innovation. Listening to and understanding the other party's perspective and interests is critical to constructive conflict management. This article briefly outlines how to prepare for the interaction, initiate the exchange, facilitate relationship, understand interests, examine solution, reach consensus and mediate conflict.

Vander Vliet, Amy, Oregon Employment Dept. Telephone interview. 18 May 2000.

Conversation regarding software industry employment in Oregon.

Whitaker, Ken. Managing Software Maniacs: Finding, Managing, and Rewarding a Winning Development Team. New York: John Wiley & Sons, Inc., 1994.

While conflict management is not expressly the topic of this book, this software manager's guide book does unwittingly discuss some of the ways that conflict is handled in software development environments as well as some of the ways it is engendered in the culture.

# Author's Biographies

## Lisa Burk

Lisa Burk has over eighteen years of professional mediation experience and ten years of work as a business consultant and trainer. She provides expert assistance to businesses, public and private non-profit agencies in the areas of:

- conflict resolution and mediation
- diversity and communication
- organizational development and assessment
- group facilitation and strategic planning
- management/team leader coaching
- conflict management system design

She is the owner of Communication Works, an Oregon Limited Liability Company. Her contracts with various organizations range from one day to two year commitments. Lisa specializes in resolving complex organizational disputes. She has trained over five thousand individuals in conflict resolution, communication, diversity and mediation.

She is an adjunct faculty member at Portland State University's Professional Development Center where she teaches "Workplace Conflict Management" and "Communication and Conflict Resolution for Project Managers." Through these teaching efforts, she has become committed to bringing mediation and conflict resolution skills to project managers and team members in software development environments.

## Jean Richardson

Though she has been writing in a variety of areas since 1984, Jean Richardson has been working specifically in hardware and software development environments since 1990. During that time she has designed and implemented a number of user information sets, lead a number of process improvement initiatives, mentored individuals entering the field of technical and business communication, and lead professionalization and education efforts for techical communicators.

Her interest in Alternative Dispute Resolution is an outgrowth of her experience as an independent consultant in high tech, an industry firmly wed to processes fraught with conflict. Some years ago she began to weigh the damage done by unresolved conflict in the workplace and to read about its corollary effects on professionals involved in conflict.

As a businessperson, she is firmly aware of the value – as well as the cost – of excellent customer service. She cautions fellow consultants against too strictly applying the addage "it's just business," because business is done by human beings, and at the root of most conflicts and most customer/vendor, employer/employee, or client/consultant disputes are human issues. Jean believes that if we ignore this basic fact we dehumanize ourselves and imperil our society.

Her fondest focus professionally is software development process improvement for the purpose of delivering higher quality products in more humane working environments.

# Managing Software Product Quality

Manny Gatlin, Intel Corporation

Manny is an Engineering Manager with Intel's Corporate Quality Network. He has an extensive software development background, including hands-on product development, project management, and software process engineering. His current assignment is developing and deploying a software qualification methodology for Intel's software products and components.

Greg Hannon, Intel Corporation

Greg is a Sr. Software Engineer with Intel's Validation Technology. He is a well-seasoned software engineer who has spent the last 5 years devoted to software development process improvement and project management. He is a former member of the Corporate Quality Network at Intel Corporation where he worked on company-wide software process improvement projects.

**Abstract**

*Many software development teams don't realize they have significant product quality problems until just a few days or weeks prior to the release of their product. This is commonly due to intra-team communication problems, lack of defined quality expectations, and compressed development schedules with testing beginning very late in the development process. A means of reducing these late-breaking quality issues is the concept of "No Quality Surprises": plan what level of quality you want your product to have at release, and continuously monitor the quality level throughout the development cycle. This provides visibility into quality problems when they first appear and are likely to be small and relatively easy to resolve. Implementing No Quality Surprises entails planning for desired product quality, monitoring these attributes of quality, assessing risk to product health using these attributes, and using this risk assessment to manage the development and release of the product. We present a project management method implementing this concept that has been developed at Intel along with results from our pilot deployment effort.*

## Introduction

It is usually expected that teams who develop a software product know what their product features are, and how the software functions under normal use conditions. They can usually describe in great detail the functionality of the software, and provide abundant technical information about how product features are implemented. But if you ask the same people to describe the quality of their software product, they generally will respond with an ambiguous description, such as "good quality". Some might suggest that the product quality is high since the defect count is low. Another common measure is that the quality of the software is high if it meets the needs of the customer. The difficulty with this approach is measuring the quality of the software before it is delivered to the customer.

One common reason that the quality of a system is not described using measures similar to those used in determining conformance to functional requirements is that the quality objectives are typically not stated explicitly. Without clear quality objectives developers tend to pay attention to that which is clearly required (i.e. the functional requirements) and little attention is given to characteristics of program quality that at best would be a rough approximation of what is desired, since they were never stated in the first place.

The typical outcome of this type of development effort is that the quality of the software is a surprise. If you don't plan for quality, why should you expect quality from the final product? Is this a reasonable expectation? The answer seems obvious, but the reality of software development is that measurable, demonstrable quality of the product is often an afterthought, not something planned for during product requirements specification and analysis. For example, if a product has few defects, does that suggest that it's a high quality product? The absence of defects doesn't really say anything substantial about software quality, even though the presence of many defects may. We suggest that by planning for quality from the beginning of product development, and establishing quality requirements and objectives early, there will be no quality surprises at the end.

In order to measure quality, you must first define what you mean by quality. It's not enough to state that a product must have "good quality". The product's desired quality should be defined in unambiguous, measurable terms. So how do you define software quality? The Institute of Electrical and Electronics Engineers (IEEE) defines software quality as "1) the degree to which a system, component, or process meets specified requirements, or 2) the degree to which a system, component, or process meets customer or user needs or expectations" [1]. The first definition would apply to products that met the functional requirements specified for a product, but this definition alone ignores the very important aspect of quality which is customer satisfaction. In order to assess the quality of a software product using the second definition, however, one must understand the attributes of quality that are important to a customer or user. In addition, there are some measures of quality that may not directly impact the customer, but are valuable nonetheless in determining whether or not a software product meets quality objectives. An example is how well a product conforms to company policies for trademark and copyright usage. The customer or user may not care how well a product conforms to these guidelines but a company has good reason to protect its trademarks, brands and image.

A good definition of software quality should then contain the following:

- Measures of quality that demonstrate some value to the customer or user
- Measures of quality that demonstrate some value to the development team and/or company
- Quality objectives that are measurable, unambiguously defined, and agreed to by all stakeholders

Many organizations think of quality activities as those activities associated with quality control; lots of testing in an attempt to find defects before the customer does. Other organizations add more tools to the quality toolbox, using inspections and formal validation and verification methods. The view of quality assurance we want to present here is one that is part of the product development lifecycle, involved in product definition and integrated with every development activity through product release. The method we discuss in this paper is concerned with the management of product quality, not simply the tracking and oversight of quality. The approach taken is to integrate this method with other project management practices. It is a method we have recently deployed to software product development teams at Intel in a major software quality initiative.

The objective of our method was to provide a solution to software product development teams that would enable them to establish measurable quality objectives, and to demonstrate quantitatively how well the product met those quality objectives. The quality solution we defined consists of quality attributes, which are the general parameters of

quality (e.g. Functionality). Defined within each quality attribute are a number of quality assessments, which consist of the specific compliance criteria and measurable targets or goals used to assess the quality of that given attribute.

The process described here is a process for managing software product quality. It describes a method for setting product quality objectives, defining clearly measurable quality attributes, monitoring progress towards quality objectives, and determining product quality at release.

## Quality Planning

An essential step towards achieving desired product quality is planning for the necessary activities to ensure attention will be given to meeting quality objectives. An important part of this is specifying quality objectives for the software product in the first place, a powerful technique that is often not practiced or is overlooked. Without explicit quality objectives that are quantifiable and measurable, developers often will give emphasis to unstated or assumed quality objectives that may vary radically from the unstated but desired quality objectives.

Planning for quality should start during product planning. Specifying quality requirements in a product requirements document does not address all the quality goals of a product, however. For example, the estimated number of defects present in a software product is not something typically stipulated in a requirements document. In some cases, measures of quality are only indirectly specified in a requirements document. Test coverage of the functionality specified in the requirements is an indirect measure of the quality of the software, as measured by the earlier definition of quality as conformance to customer requirements. When considering quality goals for the software, it is important to think about quality objectives that are meaningful, and measures that directly demonstrate whether or not the product achieves those objectives. One of the first steps we took when defining our method was establishing a framework for quality that articulated software quality attributes, and provided guidance for establishing quality goals and measuring product conformance to those goals.

## Establishing Software Quality Objectives

Planning for product quality requires some work in establishing quality objectives. Determining what those quality objectives are and how to measure them is often the most difficult task a development team is faced with during the initial planning stages. In order to help development teams with this critical step, we developed a software quality model for this quality initiative, which defines the attributes of quality and the measurement of those attributes. The format we defined describes the quality attributes, specific measurable criteria for each attribute, and quality targets for each criterion. The relationships of the attributes, assessments, criteria, and quality targets are shown in Figure 1.

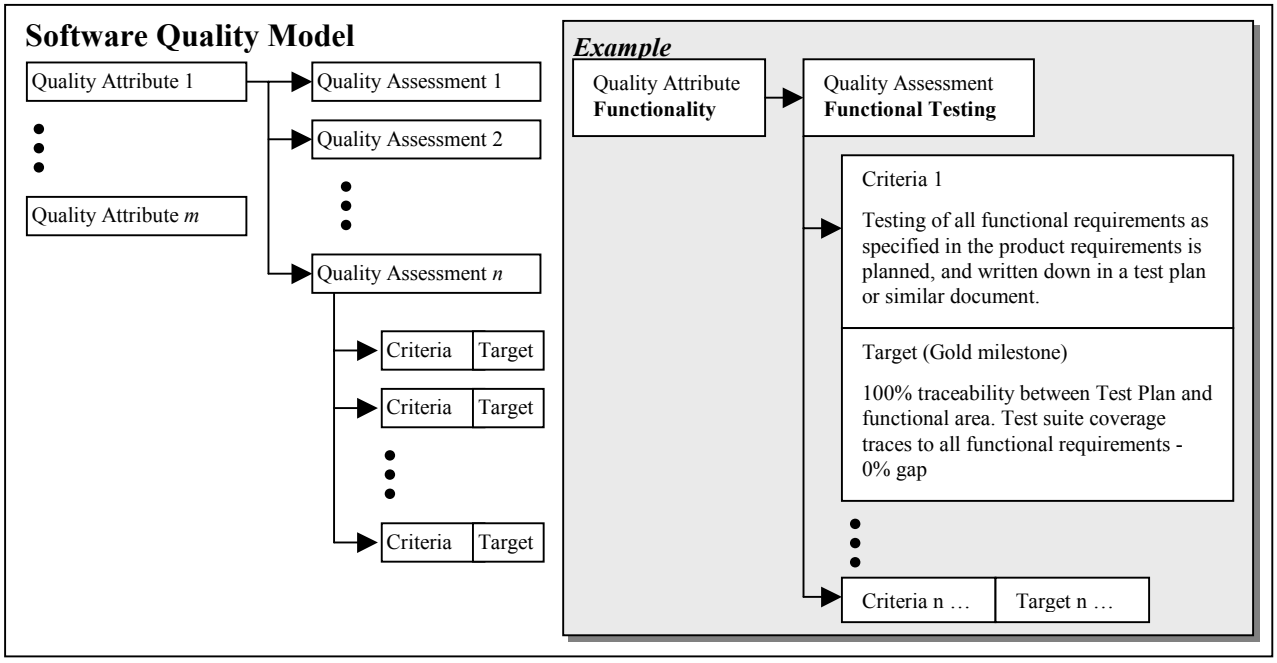



*Figure 1. Software Quality Model*

We looked at several industry models of quality, including Hewlett-Packard's FURPS+[1] model [2] and IBM's CUPRIMDSO[2] quality model [3]. We decided to incorporate many of the key aspects of industry models as a starting basis for our software quality model. Internal software development experts were polled for their inputs on what attributes they thought would constitute quality of software products.

The quality attributes we selected narrowed down to the following list:

- Functionality
- Usability
- Reliability
- Performance
- Service
- Compliance
- Stability
- Installability
- Maintainability
- Extensibility
- Reproducibility
- Compatibility
- Documentation

Not all the attributes in the model were addressed in this quality initiative. Since our goal was to develop a method that could be applied across the company to a wide variety of software products, it was felt that the focus should be on those attributes of quality which were of primary importance from our customer's perspective (based on data from customer reported issues) and from our own internal quality and reliability perspective.

Each quality attribute was broken down into one or more quality assessments. The purpose of each assessment was to assess one measure of a quality attribute. For example, an assessment of software Reliability would be Continuous Operation, or how long the software operates failure-free based on an operational profile that comprehends typical use of the software. Benefits of this approach are that you end up with an organized set of standardized quality assessments, each focused on a particular measurement of software quality, and adding new assessments or identifying duplication of effort is much easier.

Each assessment is composed of one or more criteria, which specify the actual measures of software quality. Associated with each criterion is a quality target, which is the specific quality goal defined for each software product for that particular measure. An example of this is a criterion from the continuous operation assessment which states,

> *"The product is able to achieve the specified continuous failure free operation target defined in the Product Requirements Document or Software Quality Plan".*

The targets are set for each major milestone, and are defined by the product development team when setting the quality objectives. The method does not define what those targets should be in most cases, but provides guidance in terms of what reasonable limits are, expectations for reliability, etc. The actual targets, however, are established by the team accountable for the quality of the final product.

A set of standard assessments was created that established the recommended set of quality assessments to ensure that software products met minimum quality expectations. This set of assessments was not intended to be a comprehensive set of quality measures for any given software product, but rather a basic set that would provide substantial value to a software product development team by addressing some of the most common and potentially damaging software quality issues. An initial set of extended assessments was also created to help some software product development teams perform quality checks against quality expectations that were not necessarily applicable to all software products, but may have specific applicability depending on the particular market or customer at which the software was being targeted. The standard set and initial set of extended assessments are described in Table 1.

[1] The quality attributes of the FURPS+ model are Functionality, Usability, Reliability, Performance, and Supportability, plus other quality attributes.

[2] Capability, Usability, Performance, Reliability, Installability, Maintainability, Documentation, Service, and Overall quality attributes comprise the IBM quality model.

| Standard Quality Assessments | Description |
|---|---|
| Functional Testing | Evaluation of the testing of features and functions to determine whether the testing is adequate for ensuring quality. |
| Product Reproducibility | Evaluation of the ability to archive, accurately and consistently reproduce/rebuild the software product, and ensure the consistency of the product package with the content expectations for the released product package. |
| Legal Compliance | Evaluation of product compliance with legal requirements. |
| Continuous Operation | Evaluation of the detection of time-dependent product defects such as memory leaks and race conditions. |
| Defect Data | Evaluation of product defect information to verify that it is adhering to the requirements of the release criteria established for the product. |
| Compatibility Testing | Evaluation of the product under test to perform its required functions while sharing the same environment with one or more systems or components. |
| Install Testing | Evaluation of install and uninstall testing to verify correct install and uninstall of product. |
| Customer Documentation | Evaluation of product user documentation for accuracy and consistency with product operation. |
| **Extended Quality Assessment** | **Description** |
| Customer Acceptance Verification | Evaluation of the testing performed to determine whether the product meets its customer acceptance criteria. |
| Performance Validation | Evaluation of the adequacy of the performance validation of the software under test. |
| Standards Compliance | Evaluation of product compliance with company standards or applicable external engineering standards. |
| Regression Testing | Evaluation of the testing of the product to determine that changes have not caused unintended effects and the product still complies with requirements. |

*Table 1. Software Quality Assessments*

## Quality Monitoring

Once quality objectives are established, and the development team has clear, measurable quality targets to aim for, the process of actively monitoring progress to those quality objectives begins. The method we defined involves evaluating the risk of not meeting those quality objectives.

The basic approach of the method is to define quality objectives, with clearly defined measurable quality targets for each intermediate milestone, up to and including the final release milestone. In an ideal world, product development would not proceed past a milestone unless all the requirements for meeting that milestone had been passed. This is not always a common practice, however. In the case of quality objectives, we determined that a particularly useful type of information to a manager who was trying to determine how well the product was achieving its quality targets was the relative level of risk there was to not meeting those objectives. In other words, if the product was not likely to meet its quality targets at an early integration milestone, for example, what was the risk to the product? A high-risk indicator early in the monitoring cycle would highlight areas of concern, giving the manager opportunity to find out what issues existed and apply corrective action.

### Monitoring quality data

Two things are required for quality information to be meaningful: 1) The information has to be timely and 2) the information has to be accurate. To achieve these two objectives, a project must put in place a few processes that will help them collect meaningful data. First, data collection should be conducted on a regular, frequent basis. We suggest weekly monitoring and data collection, but this may vary accordingly with the duration and reporting needs of different projects. For most projects, collecting data on a weekly basis provides a snapshot of the current state of quality of the software, and can provide development teams with trending information and adequate time to respond to negative trends if necessary. More frequent data collection may be called for the closer a project gets to delivery,

but the data collection should be synchronized with any project status meeting at which decisions based upon the data are made.

The accuracy of the data is an important consideration as well. For any quality indicator, a project should define a standard measurement method and ensure that everyone collecting the quality data is doing so using the defined method consistently. This includes agreement on the definition of such things as units of measurement and defect severity classifications prior to data collection. In any case, measurements should be made using the same tools or methods. Training is a big help in this area.

## Presenting quality data

The method we defined uses a standardized set of checklists and a scorecard to help organize and present the quality data for any given software product. Standard checklists are available for Alpha, Beta, and Gold quality milestones, but these can be adapted to alternative milestones where the intent of each software release is mapped to the quality milestones defined by the method. The checklists are used to check the product's quality measurements against the expected levels for each quality milestone. The data generated from the checklists is then rolled up into the Product Health Scorecard. Figure 2 shows an example scorecard.

**Scorecard**
Product: Widgit 2.0
Owner: M. Smith
Last Update: August 1, 2000
Current Software Milestone Objective: Beta

Risk Legend:
1: Showstopper
2: High Risk
3: Medium Risk
4: Low Risk
5: Negligible Risk

| ID | QUALITY ASSESSMENT | DESCRIPTION | RISK | | OWNER | NOTES/COMMENTS |
|---|---|---|---|---|---|---|
| | | | CURRENT | PREVIOUS | | |
| | **Summary** | Overall product health. | | | | |
| SWFT | **Functional Testing** | Evaluation of the testing of features and functions to determine whether the testing is adequate for ensuring quality. | Medium | Medium | Smith | 75% completed test plans, 50% completion of planned TDSs. |
| SWDD | **Defect Data** | Evaluation of product defect information/verification of correct product operation. | Medium | Medium | Smith | Defect data trending toward bug fixing vs implementation. |
| SWPR | **Product Reproducibility** | Evaluation of the ability to archive, accurately and consistently reproduce/rebuild the software product, and ensure the consistency of the product package with the content expectations for the released product package. | Low | Low | Taylor | Reproducibility obtained 3 times (4th time planned). |
| SWIT | **Install Testing** | Evaluation of install/uninstall testing to verify correct install and uninstall of product. | Low | Low | Jones | No installation problems, plan in place to address component update. |
| SWCD | **Customer Documentation** | Evaluation of product user documentation for accuracy and consistency with product operation. | | | | |
| SWCT | **Compatibility** | Evaluation of the product under test to perform its required functions while sharing the same environment with one or more systems or components. This | | | | |

*Figure 2. Example Scorecard.*

The Product Health Scorecard is the summary of the results of the quality assessments performed against the software product. It is a record of the current state of the product health, as measured against the quality goals established in the software quality plan. It provides a visual indicator of the current health of the product, with at-risk areas highlighted in red and low-risk areas in green.

## Release Management

Simply planning for product quality and monitoring the quality attributes and criteria is not a complete solution. The rest of the overall picture is to use this data in the management of product development. This is accomplished by the frequent review of this quality data by the product management team. This has been implemented either as a weekly review of the quality scorecard by the project manager, or by reviewing the quality scorecard at weekly development team meetings.

The intent of the scorecard is to provide sufficient data about the current quality of the software to aid in the decision-making process that determines whether or not to advance the product to the next level of release candidacy, or ship the product to the end user. The decision about whether or not to release the product is not part of

the method, however. Project management has the responsibility of making that decision. By design, the method is intended to provide the software quality data that is used in making an informed release decision.

When reviewing the scorecard data, there are two types of items to look for. The first are assessments whose product health risk is rated at the highest risk level (e.g. showstoppers). The second type are assessments whose risk level has increased since the last review. This identifies the most significant risks to the product's health. You can very effectively manage the quality of your product by performing a full risk assessment and developing and executing risk mitigation plans for each of these identified risks. By using this method, release decisions can be made effectively using current data, with potential quality risks identified and managed.

## Pilot Results

As a means of validating this project management methodology and gaining feedback on the attendant collateral we developed, we deployed this to a small number of pilot teams within Intel. These were existing project teams that were in the midst of product development and whom we were able to convince that this management methodology would be useful and well worth their time to adopt.

These eight pilot teams ranged in size from less than ten people (all co-located at one site) to more than 80 people (covering 3 geographic locations). The software these pilot teams were producing included:

- Interface drivers
- Embedded software
- Software development kit
- Shrink-wrapped application software
- Network management applications
- Server & server management products

### Modifications

Most of our pilot teams modified the criteria and assessments in only small ways in order to make this work within their environment. The most significant customization (made by our largest pilot team) was to apply the quality planning and tracking in a hierarchical fashion in order to make the effort more manageable. They used a separate checklist and scorecard for each of the six major components of their product, and created a top-level scorecard to summarize the product health risks for the entire product.

### Key Lessons

Of the many lessons that we learned during our pilot deployment, two stood out as particularly important:

- The need for clear communication and well-defined expectations is critical. This was demonstrated many times and with most of our pilot teams. Two of our pilot teams (our largest team and one of the medium sized teams) both discovered through repeated experienced with miscommunication that clear, well-defined criteria, definitions, and expectations were critical. However, this quality management method appeared to be the vehicle by which these miscommunications were exposed.
- One of the most critical aspects of our success with every one of the pilot teams was our focus on enabling them to be successful. This focus was on helping them to customize the method collateral to work for their situation. This customization included tailoring the assessments and criteria, redefining the risk level definitions, and modifications to the layout and format of the collateral.

## Significant Events

Several significant events occurred during the pilot deployment that proved the value of the methodology and tools:

- 3 different projects took schedule slips based on data from scorecards
- 1 project reallocated resources based on defect data presented in a scorecard
- QA personnel from a project successfully elevated a showstopper issue to their program manager. This was an issue that was well known to several individual contributors and first-line managers, but had not previously been successfully raised as an outstanding issue despite several weeks of effort.

## Quotes from Pilot Team Members

Several quotes from various participants in our pilot deployment support the effectiveness of this quality management methodology.

A comment from one manager after our solicitation for pilot participation: "Someone will have to convince me not to use it." This project joined our pilot deployment shortly after this comment was made.

"I think this has been critical to successes we have made bringing commonality to multiple QA groups together across sites. As well, it has been of tremendous benefit to me personally as a QA Manager in ensuring I am consistently doing the right thing. … I am really pleased with the [quality method] material and support it's existence and continued success. My biggest gripe is always writing documents and never looking at the documents again. The material is very interactive - you use it everyday/week - I like that."

"Overall, the [quality method] has helped the SW team focus on meeting requirements versus delivering SW to a schedule only."

The quality method is "the greatest thing since sliced bread" – mentioned after the data in a method scorecard had provided the justification for taking a 2 week schedule slip.

## Pilot Team Performance to Schedule

The intent of our pilot deployment was **NOT** to:

- Directly improve pilot team performance to schedule
- Prevent products from shipping with low quality

Rather, we focused on enabling product development teams to know their product's quality prior to release and having them adapt as they judged appropriate. We accomplished this by giving them tools with which to plan and track their product quality. One appropriate adaptation to knowing their product quality may mean delaying project release milestones and/or releasing the product with a quality level that is less than what was planned. Because of this, we did not attempt to measure the performance to schedule of our pilot teams.

In spite of this, three of our pilot teams reported project schedule slips whose justification was based on data in their quality scorecards. We view these schedule slips as positive indications that the method's tools are useful in managing product quality.

## Pilot Program Success Criteria

We conducted a post-pilot survey in order to gather information about the successes and shortcomings of the method and accompanying collateral, as well as the pilot deployment program. The pilot deployment program's success or failure (as defined in our pilot deployment plan) was to be determined from feedback received from the post-pilot surveys. Table 2 summarizes the survey findings.

| Survey question | Pilot Program Performance | Comments |
|---|---|---|
| "At least 50% of the pilot teams indicate they will adopt the [quality method] (or some derivative thereof) as part of their standard development process for future releases and projects." | **100% of pilots** | In some cases, the Divisional organization which sponsored the pilot project indicated that it would use the method for all projects in the Division. |
| "Project teams were able to reliably and consistently determine the quality of their product using the [quality method]." | **100% of pilots** | All survey respondents indicated that the method was useful in assessing the quality of their product. |
| "[Quality method] data was used in project decision making." | **6 of 9 pilots** | Those indicating that the scorecard data had not been used in decision-making were from pilot projects that were either in the adoption phase or early usage phase where it is reasonable to expect no impact as yet. This performance is most likely a result of terminating our pilot deployment program prematurely. |

*Table 2. Pilot Program Survey Results*

Currently, the pilots have been using the method for over 6 months, and follow-up interviews with the pilot teams indicate that they plan to continue using the method.

## General Deployment

Our organization is chartered with developing and deploying software methods that are of direct benefit to Intel projects developing software products and components. In order for us to be successful and our work to be useful to the various software development groups throughout the company, we used the pilot program concept to not only verify the utility and benefit of the method, but also to learn from the users what worked, what didn't, and what types of adaptations and changes to the method actually occurred during use. By carefully observing and understanding why changes were made and how they helped the pilot projects use the method more effectively, we were able to make changes and clarifications to the method that would benefit future users. The method has currently been updated with inputs from the pilot program, and a revised baseline version has been published. This version is intended for general use within the company, and will be deployed to divisions developing software products over the next year.

## Conclusion

Software quality doesn't just happen. Planning for quality requires the same attention as any other aspect of product development. Establishing quality goals early in the development process and tracking how well the product conforms to those goals throughout development gives timely information that can help identify and resolve quality issues before they become a problem. The method described here isn't anything new conceptually; the real value of the method is that it is designed to be an integrated part of the development cycle. Roles and responsibilities are clearly defined, tracking and reporting frequency expectations are set, and performance to quality goals is measured continuously so that there are no surprises at release.

## References


1. IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610.12, p. 60, 1990
2. R. Grady, Practical Software Metrics For Project Management and Process Improvement, p. 32, Prentice-Hall, Inc., 1992.
3. S. Kan, Metrics and Models in Software Quality Engineering, p. 116, Addison-Wesley Publishing Company, 1995.

# A Goal-problem Approach for Scoping a Software Process Improvement Program

**By Neil Potter and Mary Sakry**
**The Process Group**

Email: help@processgroup.com
Tel:972-418-9541
Fax:972-618-6283
Web:www.processgroup.com

## Abstract

The most common approach for process improvement we have seen, over the past ten years, is for an organization to document all processes. This approach is amplified when an organization rushes to adopt a sweeping solution such as ISO9001 or the SEI CMM[1]. In light of a goal stating, "Be SEI CMM Level 3 by December," the approach of documenting all processes is reinforced and might even appear natural. This process-centric approach can work, but it has a high risk of failure.

In this paper, we will explain an approach to scoping an improvement program based on problems and goals of the organization. By adopting this approach, organizations are able to make significant progress on real issues, and make progress on the process improvement model or standard they are trying to achieve. The problems and goals of an organization can scope and sequence your improvement program.

## Key Words

Capability Maturity Model (CMM), Key Process Area (KPA), sequencing, problems, goals.

## Introduction

The most common approach for process improvement we have seen during the past 10 years is to document all processes. This approach is amplified when an organization rushes to adopt a sweeping solution such as ISO9001 or the SEI CMM[1]. In light of a goal stating, "Be SEI CMM Level 3 by December," the approach of documenting all processes is reinforced and might even appear natural. The white line in Figure 1 describes this approach. It starts, wanders around, and ends without reaching any specific goal.

[1] Software Engineering Institute, Capability Maturity Model 1.1

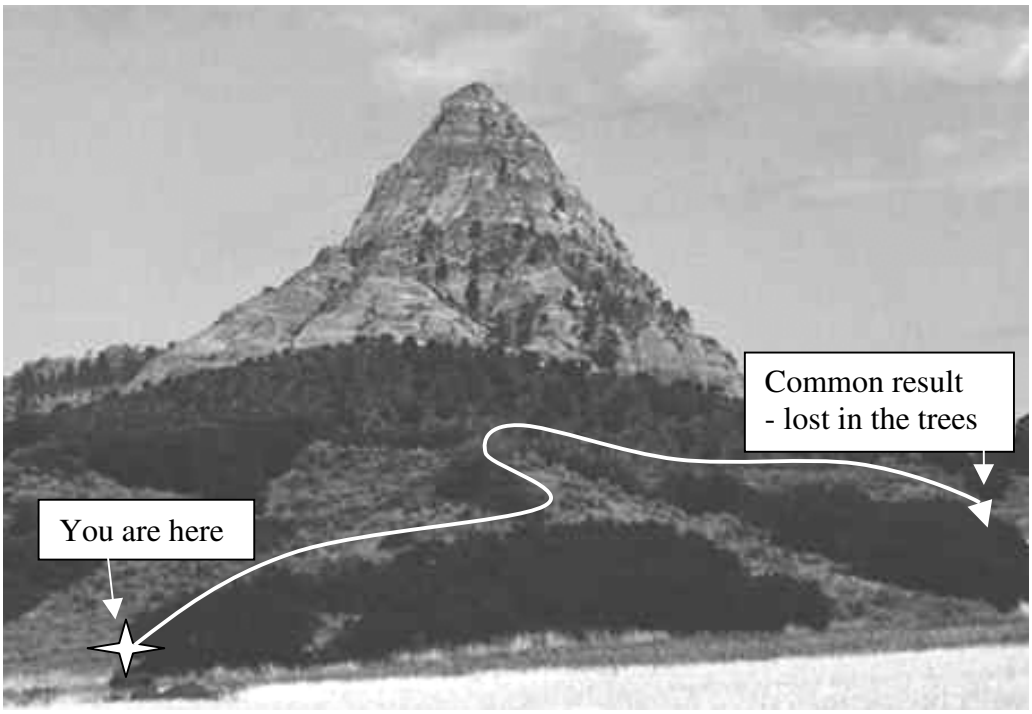



**Figure 1. A process-centric approach to improvement.**

This process-centric approach can work, but it has a high risk of failure. It usually results in a large stack of unused process documents.

**Goal-problem approach for scoping an improvement program**
In figure 2 we have highlighted one of the business goals the organization is trying to achieve. Examples might include the delivery of a product, the completion of a software installation, or the upgrade of a database. The goal could also be the desired outcome when a critical problem has been solved. For example, a critical problem might be the inability to hit delivery deadlines, or the fact that 75% of the organization's resources is spent on rework. Related goals might be to meet deadlines 100% of the time or reduce rework to 25%.

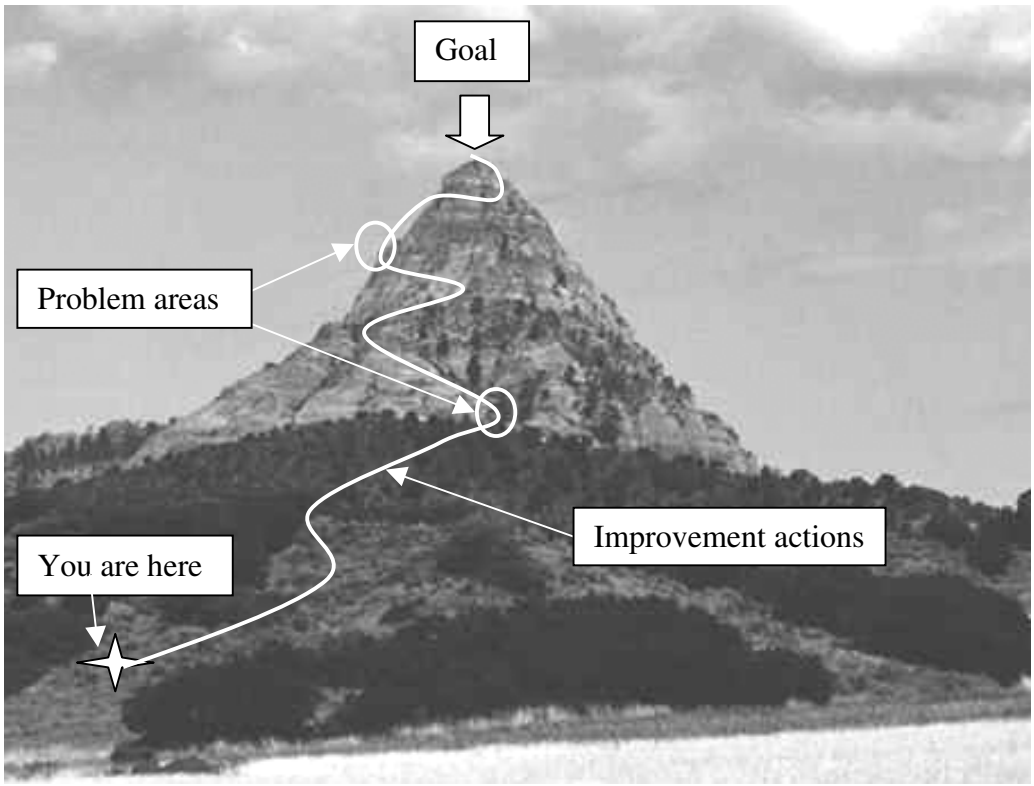



**Figure 2. A goal-problem approach to improvement.**

The goal-problem approach starts by determining business goals and problem areas. These are compared with the elements of the improvement model or standard being used. The appropriate elements are then selected to achieve the goals and address the problems.

During one session to help a company plan a process improvement program, we learned that the group was about to establish six teams to work on the six Key Process Areas of the CMM Level 2. We suggested that the developers and managers temporarily forget about Level 2 and state all the major problems they had. Then they were asked to state the business goals they were trying to achieve over the next six to 18 months. After one hour of discussion, they created the following list (figure 3).

A comparison was then done with the SEI CMM, Level 2 and 3 (figure 3). The related KPA names and activities are in parentheses after each item[2]. If the company had been using ISO9001 or The Malcolm Baldrige Award, we would have mapped the problems and goals to those documents.

[2] SEI Level 2: RM = Requirements Management; SPP = Software Project Planning; SPTO = Software Project Tracking and Oversight; SCM = Software Configuration Management; SQA = Software Quality Assurance; SEI Level 3: TP = Training Program; SPE = Software Product Engineering; PR = Peer Reviews; IC = Intergroup Coordination; ISM = Integrated Software Management.

| Problems | Goals |
|---|---|
| **1. Need better requirements. Requirements tracking not in place - changes to requirements are not tracked; code does not match specification at test time.** *[Level 2: RM - activities 1, 2, 3]* | **1. Orderly plans for development.** *[Level 2: SPP - activities 2, 5, 6, 7, 8, 13, 14]* |
| **2. Management direction unclear for product version 2.3. Goals change often.** *[Level 2: RM - activities 1, 3, verification 1]* | **2. Understand what our capacity is - develop one list of all the work we have to do.** [*Level 2:* SPP - activity 7, ability 1] |
| **3. Hard to revise project plan - items drop off, new things get added, plan is out of date.** *[Level 2: SPTO - activity 2, 8, 9]* | **3. Improve schedule tracking and communication of changes to impacted groups.** *[Level 2: SPTO - activities 3, 4]* |
| **4. Wrong files (e.g., DLLs) get put on CD - don't know what the right ones should be.** *[Level 2: SCM - activities 4, 7, 8, 9, 10]* | **4. Successfully deliver product X.** *[Level 2: RM - activities 1, 2, 3, SPP - activities 6, 10, 13]* |
| **5. Defect repairs break essential product features.** *[Level 2: SCM - activities 5, 6, 7, 9, 10, abilities 1, 2, 4, 5, verification 3, 4]* | **5. Improve performance of core software product.** *[Level 2: SPP - activity 11, SPTO - activity 7]* |
| **6. Customers are unhappy. There are approximately 300 outstanding defects that have not been addressed.** *[Level 2: SCM - verification 1, RM - activity 3; Level 3: IC - activity 1]* | **6. Identify needed skills for new designers and hire/promote and train accordingly.** *[Level 3: SPE activity 3, ability 2]* |
| **7. Difficult to find time to do critical activities (product development) versus crisis activities.** *[Level 2: SPP - activities 4, 10, 12]* | **7. Identify tools to support software developers.** *[Level 2: SPP - activity 14; Level 3: SPE - activity 1]* |
| **8. Lack of resources and skills allocated to software design.** *[Level 2: SPP - activity 10]* | **8. Keep making a profit. Keep customers happy.** *[Level 2: RM - activities 1, 2, SPP - activities 10, 12, 13, SPTO - activities 4, 6, 8, 10, SQA - activity 5, Level 3: SPE - activities 2, 7, IC - activity 1, PR - goal 2]* |
| **9. Quality department - need team training (product and test skills).** *[Level 2: SQA - abilities 2, 3, 4]* | **9. Identify tools to support software testers.** *[Level 2: SPP - activity 14; Level 3: SPE - activity 1]* |
| **10. Changes to specifications and documentation are not communicated effectively to documentation and test groups.** *[Level 2: RM - activities 1, 2, 3, SCM activities 5, 6, 7, 9, ability 1]* | **10. Empower Quality Department to have final say on product shipment.** *[Level 2: SQA - activities 6, 7]* |
| **11. Unreliable project schedule estimates.** *[Level 2: SPP - activities 5, 9, 10, 12, 13, 14, ability 4]* | |
| **12. Unclear status of software changes.** *[Level 2: SCM activities 8, 9]* | |
| **13. Testing does not necessarily comprehend things that matter to the customer.** *[Level 3: SPE activities 5, 6, 7]* | |

**Figure 3. Problems and goals list.**

What was the scope of the improvement program?

The scope of the improvement program was to address the problems and the goals of the organization. As you can see, 21 out of the 23 items (91%) map to Level 2. When all the problems and goals have been addressed, 46% of the Level 2 activities will have been addressed.

The essential difference between this approach and addressing the six KPAs in parallel is that the problems and goals indicate which pieces of each KPA to address first. Regardless of the model or standard being used, the problem-goal approach helps scope and sequence an improvement program.

**Dealing with items that don't match the improvement model or standard**
In figure 3, not all of the problems in the list closely match the areas of CMM Level 2. For example, there is not much in the CMM to address goal #5 specifically. In this situation, one has to determine which areas are the most important for the organization to fix now. Serious problems should be worked on first.

**What can be learned using this approach?**
There are five lessons to be learned from adopting the goal-problem approach:

1. All process improvement can be meaningful.

2. The problems and goals can help identify which pieces of a process improvement model or standard to work on first. A model or standard should no longer be seen as an all-or-nothing approach, which often leads people to do everything at once, regardless of whether it is appropriate. A model or standard can be treated as a large toolbox of little actions, ideas and solutions, each of which is useful at different times.

3. Any process document developed to solve a problem will be meaningful and useful. A process improvement team is less tempted to gold-plate the process since its scope is defined by a problem.

4. The motivation of the group to work on improvement is increased when its problems and goals are systematically addressed.

5. When an organization is focused on goals and solutions to problems, it does not create academic process documents.

**Using the approach at a project level**
Below is an example from a project of a different organization. We asked the project manager for a significant project goal. From this goal we derived areas that needed improvement by asking two further questions. The first, "*What is preventing you from achieving the goal?"* was asked to uncover problem areas related to the goal. The second, "*What other problems do you have related to this goal?* " helped elicit further problem areas. The resulting problem list formed the scope of the improvement program for this project.

*What is your goal?*
- Reduce release cycle to 6-9 months.

*What is preventing you from achieving the goal?*
- Changing requirements.
- Loss of resources - difficult to replace people that leave the project due to specialized skills.
- Too many features to put into a 6-9 month development cycle.
- Poor quality of incoming code from other groups.
- Inadequate availability of test equipment.

*What other problems do you have related to this goal?*
- Lack of visibility within any life cycle phase -- it is difficult to know whether we are ahead or behind schedule.
- Don't always have the resources available to complete the planned work.
- Difficult to find defects early.

We then stepped through each of her answers and made a note of the KPA activity that could significantly help address the problem area (figure 4). We recommended some of the more advanced Level 3 KPA components since her group was almost Level 2.

| **Goal: Reduce release cycle to 6-9 months.** | |
|---|---|
| **Problems** | **KPA component that would help this problem** |
| Changing requirements. | Level 2: RM - activity 3, SCM - activity 5. Level 3: SPE - activity 2. |
| Loss of resources - difficult to replace people that leave the project due to specialized skills. | Level 2: SPTO - activities 2, 8.<br>Level 3: TP - activities 1, 2, SPE - ability 2. |
| Too many features to put into 6-9 month development cycle. | Level 2: SPP - activities 4,12,13.<br>Level 3: SPE - activity 2. |
| Poor quality of incoming code from other groups. | Level 3: IC - activities 2, 5, 6, PR - activity 2. |
| Access to equipment to test code. | Level 2: SPP - activities 13, 14.<br>Level 3: SPE - activities 6, 7. |
| Lack of visibility within any life cycle phase -- it is difficult to know how much we are ahead or behind schedule. | Level 3: ISM - activities 4, 7, 11, verification 2. |
| Don't always have the resources available to complete the planned work. | Level 2: SPP - activities 4,12, 13.<br>Level 3: ISM - activities 3, 5, 10, 11. |
| Difficult to find defects early. | Level 3: PR - activities 1, 2, ability 1. |

**Figure 4.**

In this example, five out of the eight problems (63%) mapped to SEI Level 2, and 100% mapped to SEI Level 3. The problems and goal became the scope of the improvement program. By addressing the problems and the goal, she will make significant progress toward completing Level 2 and starting Level 3.

What are the questions that help you scope your improvement effort?
To scope the improvement effort for one goal, we ask the following questions:

1. State one goal that you will be held accountable for over the next six to 18 months?
2. What is preventing you from achieving this goal?
3. What other problems do you have related to this goal?
4. If you are using a process improvement model or standard, which items help each of the problems listed?

**Addressing all of the items in the model or standard being used**
One of the primary concerns with this approach is that an organization will not address all of the items in the model or standard being used, since there might not be goals or problems related to all of the items.

When the first set of problems and goals have been worked, the next step is to repeat the cycle and determine the next set of problems and goals. This new set can then be compared to the remaining items in the improvement model or standard. Over a one-to-three year period, each section of the model or standard is matched with a problem or goal.

Process auditing is an example of a practice that is sometimes premature. In the beginning of an improvement program, there is usually little benefit to performing process audits, an element that is called out in many improvement models and process standards. However, the need for auditing a process becomes apparent once the process has been defined, used and proven effective.

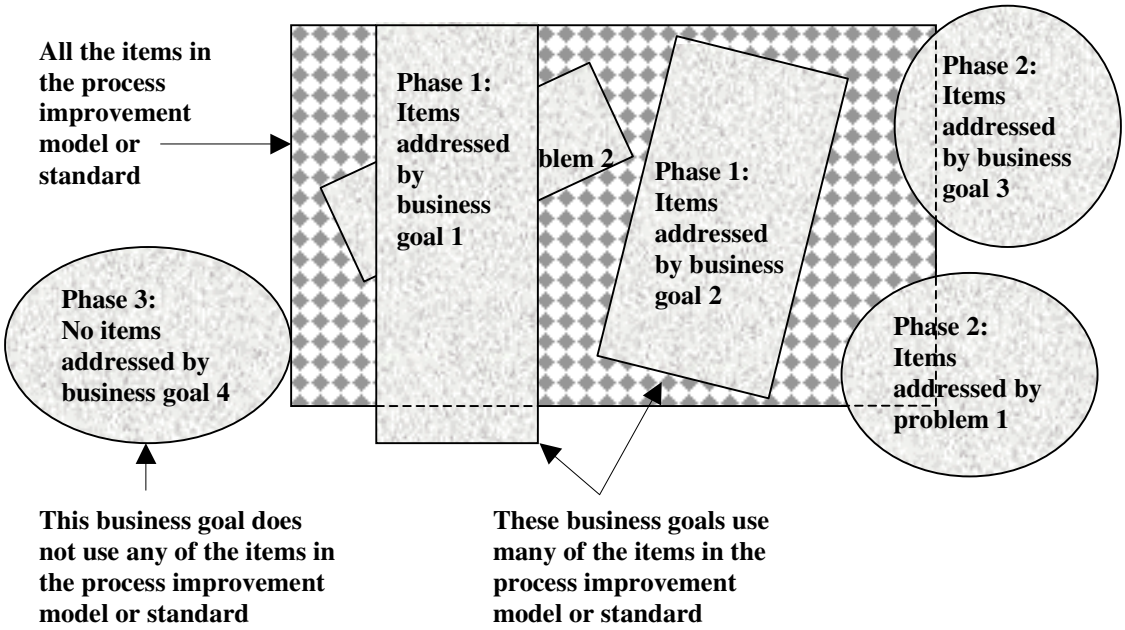
One company highlighted this with its software release management process. Before release management had been improved, performing an audit on the related software configuration management (SCM) activities would have been futile. When SCM and release management were in place, one employee by-passed the process and incorrectly released a software patch by E-mail to a customer. The software did not work and the

customer was furious. The need for SCM auditing became apparent. After the audits took place, the developers and managers realized that they now had a mechanism to verify the execution of the defined release management activities.

In the diagram below, we show the typical phases of an improvement program using the goal-problem approach. Each time a goal or a problem is addressed, it has the potential to use some of the items in the improvement model or standard. After several goals and problems have been addressed, typically 90% of the items have been used.

For one company, five significant problems, during phase one, used 66% of the elements of SEI Level 2. For another company, 100% of Level 2, 3 and 4 activities were used over a six-year period by taking a goal-problem approach.




There will, of course, be situations where some of the items in the model or standard are not used when solving a problem or achieving a goal. These items should be left until the end of the improvement cycle. At that time, one of three scenarios occurs. First, the outstanding items will be put to good use. Second, the items will be declared "Not applicable." Third, the items will be performed academically to meet the letter of the law. The focus should, of course, be on the first scenario.

**Summary and Conclusions**

Scoping an improvement program can be difficult and frustrating. The task becomes daunting when a process model or standard is adopted wholesale. However, a simple solution exists, one that is immediately available. The goals and problems of an organization can provide a timeless and effective scope for any improvement program. A model or standard can then be used as a source of ideas, solutions and actions to achieve this scope. The resulting goal-problem improvement program is compelling and practical.

**References:**

Paulk M., Weber C., Curtis B. and Chrissis M. B. (1994). *The Capability Maturity Model - Guidelines for Improving the Software Process.* Addison Wesley.

Robbins A., *The Time of Your Life*, audiocassette program. (1998). Robbins Research International.

---

**Biographies of Mary Sakry and Neil Potter**

Mary Sakry is co-founder of The Process Group, a company that consults in software engineering process improvement. She has 23 years of experience in software development, project management and software process

improvement. For 15 years, she was a Project Manager and Software Engineer in Texas Instruments (TI) in Austin, TX. In 1989, she worked on the Corporate Software Engineering Process Group within TI to lead software process assessments across TI worldwide. The last two years of TI were spent consulting and educating software developers and managers on software project planning, risk management, estimation, SEI CMM, inspection and subcontract management. Mary was the first SEI authorized lead assessor for CBA-IPI process assessments. She has an MBA from St. Edwards University, and a B.S. in Computer Science from the University of Minnesota.

Neil Potter is also a co-founder of The Process Group. He has 14 years of experience in software design, engineering and process management. For six years, Neil was a Software Design Engineer in Texas Instruments, Dallas, developing Electronic Design Automation software. The last two years at TI he was a manager of a Software Engineering Process Group performing consulting within TI in America, England and India. Consulting included software project planning, risk management, estimation, SEI CMM and inspection. Neil is an SEI authorized lead assessor for CBA-IPI process assessments. He has a B.Sc. Computer Science from the University of Essex in England.

# A Goal-problem Approach for Scoping a Software Process Improvement Program

The Process Group
P.O. Box 700012 • Dallas, TX 75370-0012
Tel. 972-418-9541 • Fax. 972-618-6283
E-mail: help@processgroup.com
http://www.processgroup.com

# Learning Objectives

- Scoping an improvement program from business or project goals
- Addressing project-level problems along the way
- Developing action plans that tie to problems and goals
- Using improvement models effectively, such as the SEI CMM*

*Software Engineering Institute, Capability Maturity Model 1.1

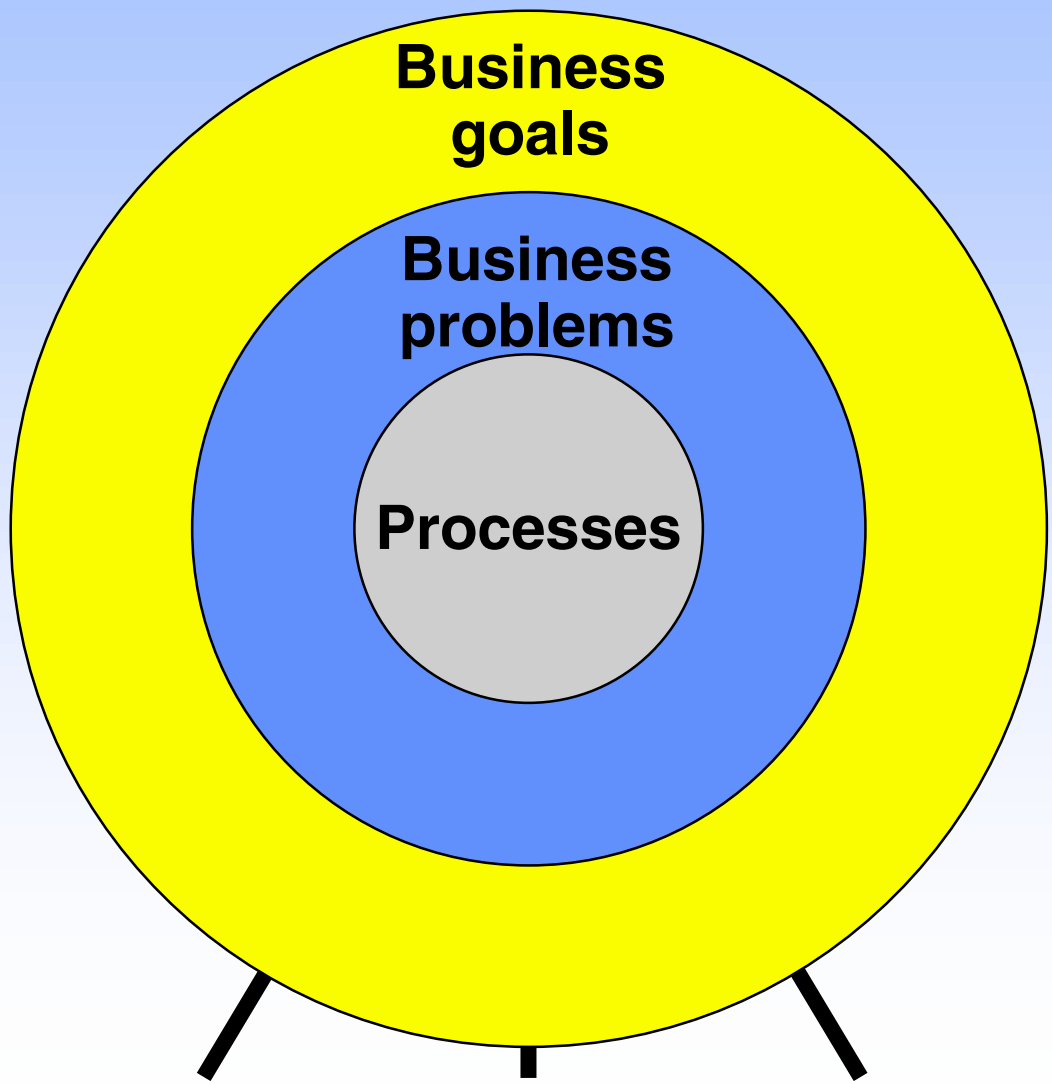
# The Problem

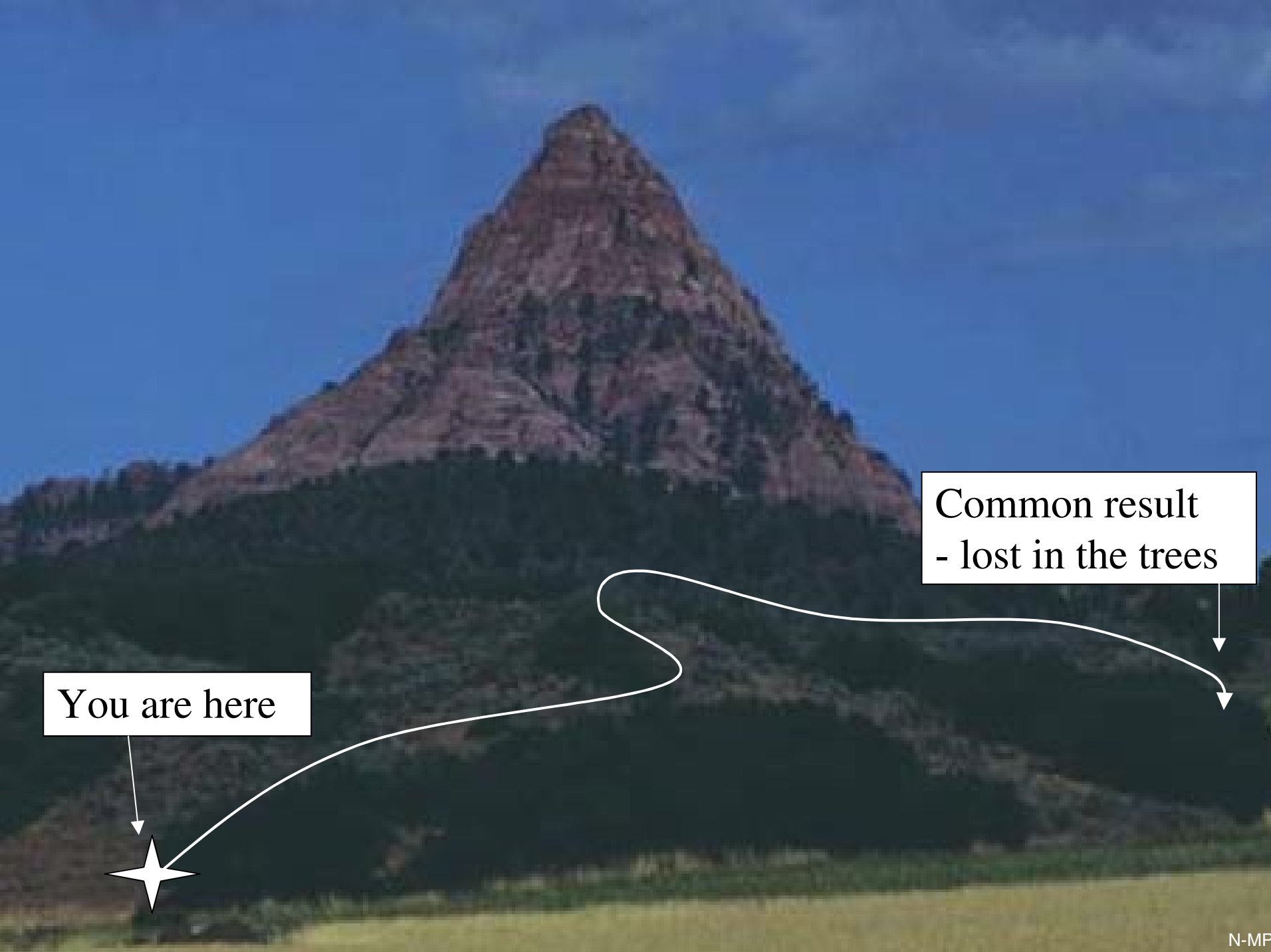
**Process-centric improvement**

- SEI CMM
- ISO9001
- Bellcore

**It can work!**

- High risk of failure

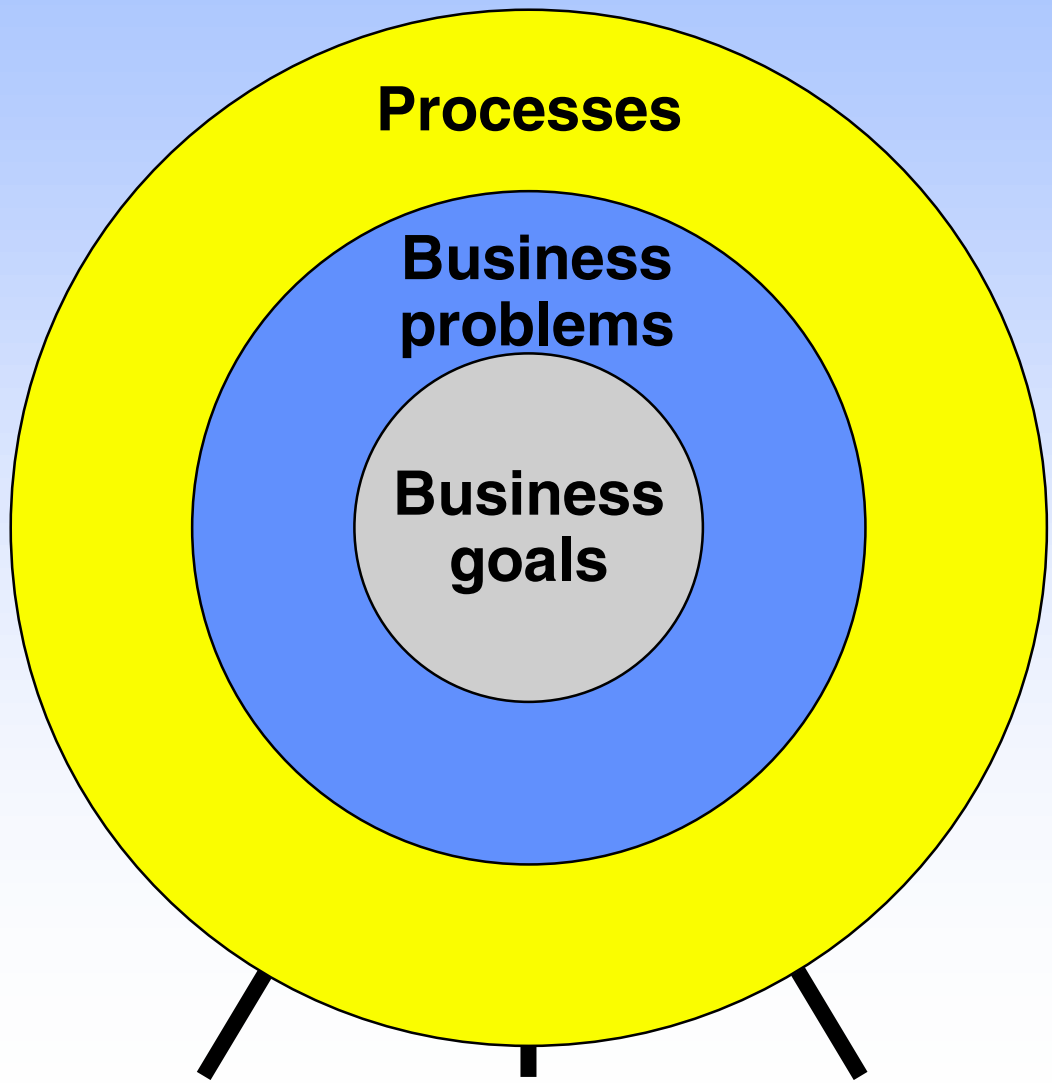


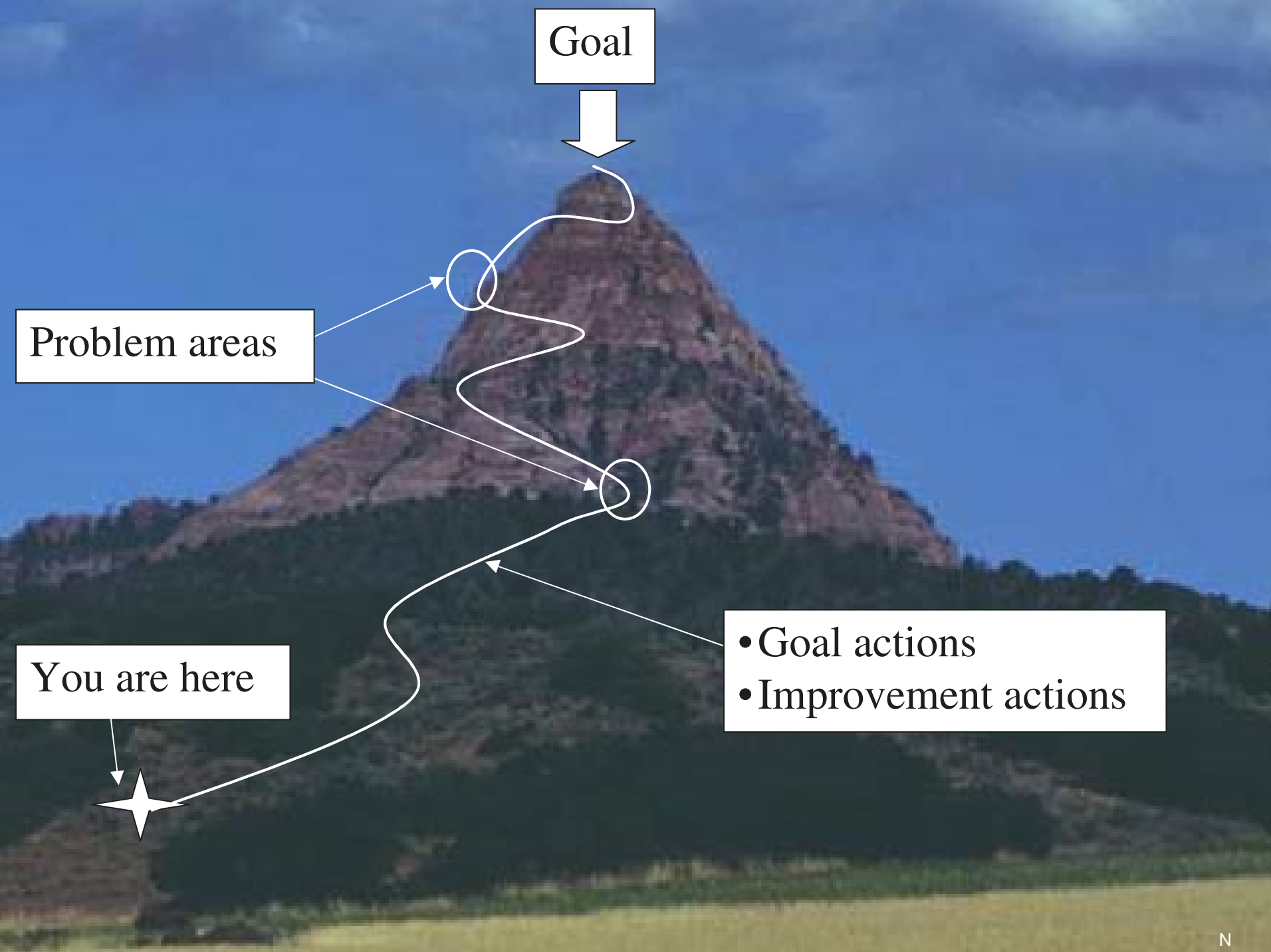

Common result
- lost in the trees
You are here

# The Solution

**Goal-problem-centric improvement**

**Goals and problems can be used to scope and sequence the improvement effort**




Goal
Problem areas
You are here
• Goal actions
• Improvement actions

# Goal-problem Approach

- **STEP 1**
  - State your **goals** for the next 6-18 months
- **STEP 2**
  - State your current **problems**
- **STEP 3**
  - If you are using a process improvement model or standard, determine which **elements help** each of the problems/goals listed

---

- **STEP 4**
  - Set priorities

# Step 1: State Goals for Next 6-18 Months

1. Orderly plans for development
2. Understand what our capacity is - develop one list of all the work we have to do
3. Improve schedule tracking and communication of changes to impacted groups
4. Successfully deliver product X
5. Improve performance of core software product

# Step 2: State Your Current Problems

1. Need better requirements. Requirements tracking not in place - changes to requirements are not tracked; code does not match specification at test time.

2. Management direction unclear for product version 2.3. Goals change often.

3. Hard to revise project plan - items drop off, new things get added, plan is out of date.

4. Wrong files (e.g., DLLs) get put on CD - don't know what the right ones should be.

5. Defect repairs break essential product features.

# Step 3: If you are using a process improvement model or standard, determine which elements help each of the problems/goals listed

## Goals and Mapping to SEI CMM 1.1

1. **Orderly plans for development**

   *[Level 2: SPP - activities 2, 5, 6, 7, 8, 13, 14]*

2. **Understand what our capacity is - develop one list of all the work we have to do**

   *[Level 2: SPP - activity 7, ability 1]*

3. **Improve schedule tracking and communication of changes to impacted groups**

   *[Level 2: SPTO - activities 3, 4]*

4. **Successfully deliver product X**

   *[Level 2: RM - activities 1, 2, 3, SPP - activities 6, 10, 13]*

5. **Improve performance of core software product**

   *[Level 2: SPP - activity 11, SPTO - activity 7]*

## Problems and Mapping to SEI CMM 1.1

1. **Need better requirements. Requirements tracking not in place - changes to requirements are not tracked; code does not match specification at test time.**

   *[Level 2: RM - activities 1, 2, 3]*

2. **Management direction unclear for product version 2.3. Goals change often.**

   *[Level 2: RM - activities 1, 3, verification 1]*

3. **Hard to revise project plan - items drop off, new things get added, plan is out of date.**

   *[Level 2: SPTO - activity 2, 8, 9]*

4. **Wrong files (e.g., DLLs) get put on CD - don't know what the right ones should be.**

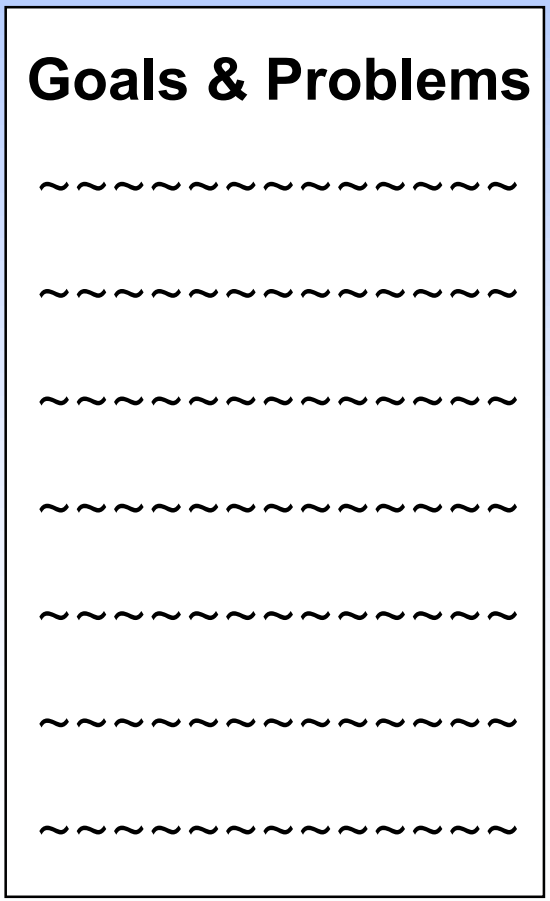
   *[Level 2: SCM - activities 4, 7, 8, 9, 10]*

5. **Defect repairs break essential product features.**

   *[Level 2: SCM - activities 5, 6, 7, 9, 10, abilities 1, 2, 4, 5, verification 3, 4]*

# Mapping Against CMM 1.1

**91% map to Level 2**




**Initial goals and problems address 46% of Level 2**

| Key Process Areas |
| --- |
| Process change management<br>Technology change management<br>Defect prevention |
| Software quality management<br>Quantitative process management |
| Peer reviews<br>Intergroup coordination<br>Software product engineering<br>Integrated software management<br>Training program<br>Organization process definition<br>Organization process focus |
| Software configuration management<br>Software quality assurance<br>Software subcontract management<br>Software project tracking & oversight<br>Software project planning<br>Requirements management |

# Goals / Problems That are not a Good Match

Example

**5. Improve performance of core software product**

*[Only 2 activities]*

**If it is important, work on it!**

# Using the Approach at a Project Level

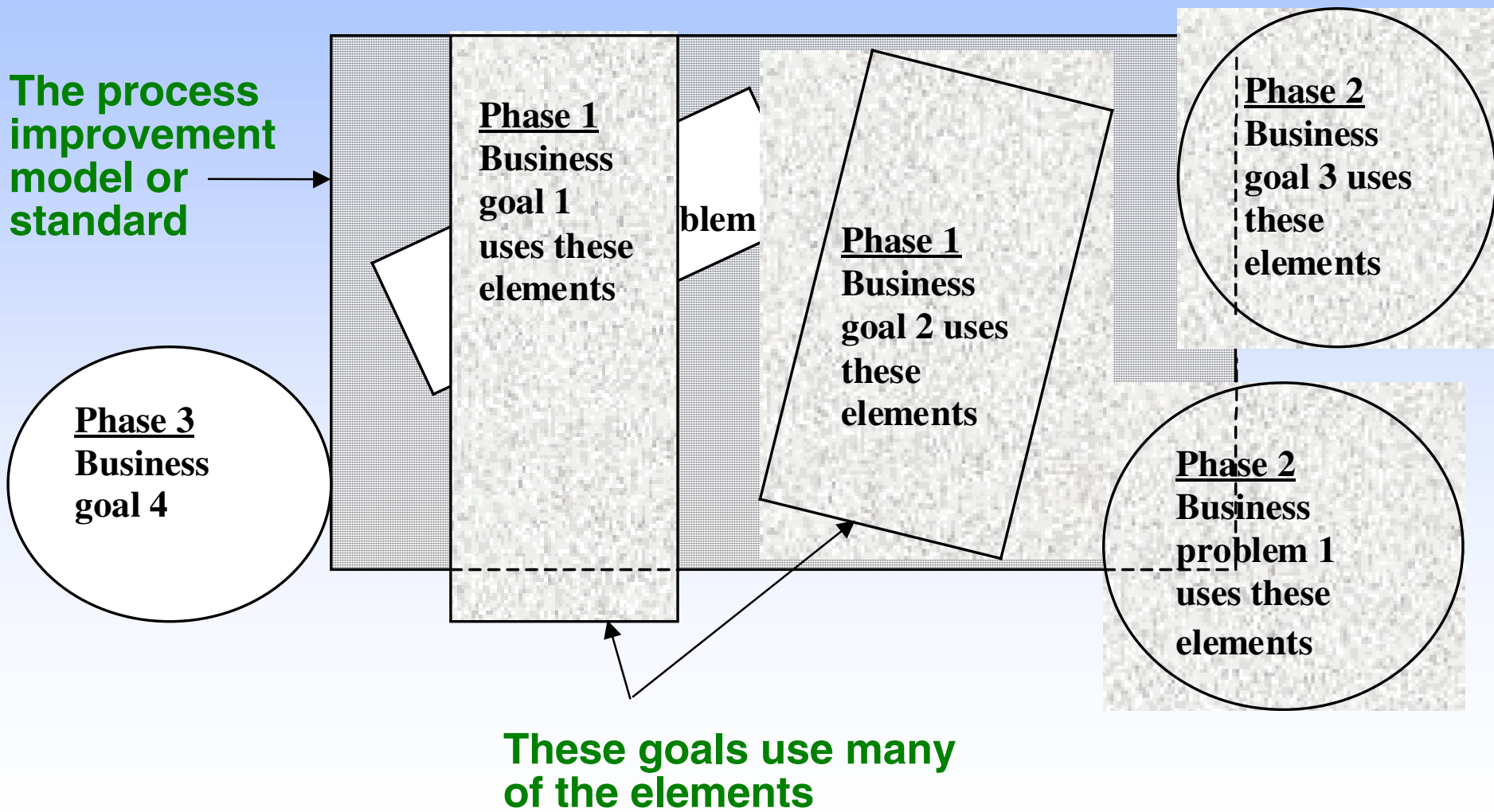
- **What is your goal?**
  - Reduce release cycle to 6-9 months
- **What is preventing you from achieving the goal?**
  - Changing requirements
  - Poor quality of incoming code from other groups
  - Inadequate availability of test equipment
- **What other problems do you have related to this goal?**
  - Don't always have the resources available to complete the planned work
  - Difficult to find defects early

| Problems | Mapping to CMM 1.1 |
|---|---|
| Changing requirements | Level 2: RM, SCM<br>Level 3: SPE |
| Poor quality of incoming code from other groups | Level 3: IC, PR |
| Inadequate availability of test equipment | Level 2: SPP<br>Level 3: SPE |
| Don't always have the resources available to complete the planned work | Level 2: SPP<br>Level 3: ISM |
| Difficult to find defects early | Level 3: PR |

| **Goal and Intermediate Goals** (The results you want) | **Purpose of Goal** (Why do you want to achieve the goal?) | **Actions / Solutions** | **Priority** (*= essential) |
|---|---|---|---|
| | | | |
| **REDUCE PRODUCT DEVELOPMENT CYCLE TO 6-9 MONTHS** | **INCREASE MARKET SHARE BY BEING IN THE MARKET PLACE 1ST WITH PRODUCT XX1** | | |
| | | | |
| | | | |
| Managed changing requirements (Problem#1). | | Only allow changes to the application interface, not the kernel routines. | 1* |
| | | Establish a group with the authority for managing the project's software baselines. | 2* |
| | | Improve the library control system to minimize version control errors.<br>Buy requirements management tool. | 3 |
| | | Record and track change requests and problem reports for all configuration items/units. | 4 |
| | | Review the initial requirements and changes before they are incorporated into the project plan. | 5 |
| | | Solidify the requirements earlier. | 6 |
| | | | |
| | | | |

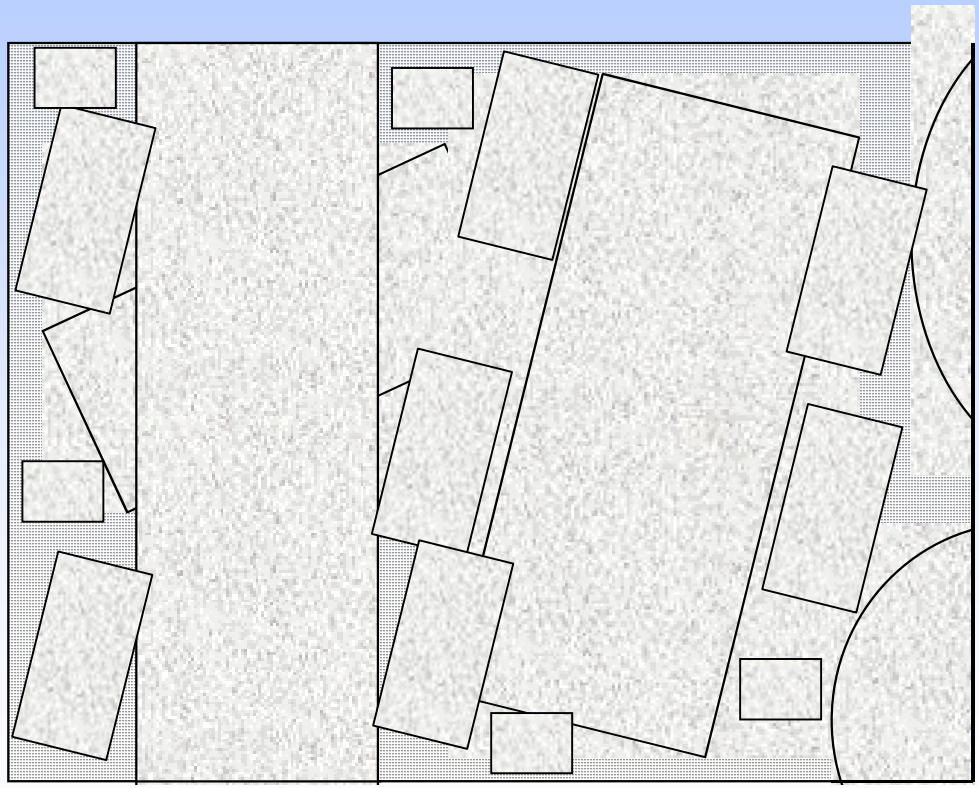
| | | | |
|---|---|---|---|
| Set feature priorities for a 6-9 month development cycle (Problem#3). | | Establish review process with clients to negotiate features for a 6-9 month development cycle. | 1* |
| | | Rate each feature based on value to the customer (1..10) and cost to develop (1..10). | 2* |
| | | Review project commitments with senior management, engineering and the customer to obtain agreement. | 3 |
| | | Perform risk management related to the schedule, resource and technical aspects of the project. | 4 |
| | | Establish incremental delivery plan to phase in lower priority features. | 5 |

# Using Models and Standards



# What to do with the Remaining Elements?

- **Put each to good use**
  - –What problem could it solve?
- **Declare them not applicable**
- **Meet the letter of the law**



# Summary

- Scoping an improvement program can be difficult and frustrating
- The task becomes daunting when a process model or standard is adopted wholesale
- The goals and problems of an organization can provide a timeless and effective scope for any improvement program
- A model or standard can then be used as a source of ideas, solutions and actions to achieve this scope
- The resulting goal-problem improvement program is compelling and practical

# References


- Paulk M., Weber C., Curtis B. and Chrissis M. B. (1994). *The Capability Maturity Model - Guidelines for Improving the Software Process.* Addison Wesley.

- Robbins A., *The Time of Your Life*, audiocassette program. (1998). Robbins Research International.

# Acronyms

- **SEI Level 2:**
  - RM = Requirements Management
  - SPP = Software Project Planning
  - SPTO = Software Project Tracking and Oversight
  - SCM = Software Configuration Management
  - SQA = Software Quality Assurance

- **SEI Level 3:**
  - SPE = Software Product Engineering
  - PR = Peer Reviews
  - IC = Intergroup Coordination
  - ISM = Integrated Software Management

# "A"is for "Assurance"–A Broad View of SQA

Alan S. Koch

## Abstract

Software Quality Assurance (SQA) means different things to different people.

- To some, "QA" means "test" ("We're ready to QA the software.")
- Others embrace a wider meaning of assuring quality by reviewing specifications, designs, user interfaces, and even code to assure that what is developed is high-quality in the first place.
- Some see QA's role as collecting appropriate metrics (e.g. peer review defect detection data) and analyzing them to assure that the software development work is effective.
- In the Capability Maturity Model for Software (CMM)®[SEI], SQA is mainly focused on assuring that organizational processes (e.g. Requirements Management) have been faithfully followed, and that the processes are effective.

But how can we truly *assure* quality? The only way we can assure quality is to embrace *all* of these things.

In this presentation we will explore the various dimensions of software quality (e.g. Usability, Maintainability, Lack of Defects) and identify the variety of actions that are necessary to truly assuring each dimension of quality.

## The Author

Alan S. Koch's 23 years in software development includes:

- 9+ years designing, developing and maintaining software,
- 5+ years in Quality Assurance (including establishing & managing a QA department), and
- 2+ years in Software Process Improvement.

Mr. Koch was with the Software Engineering Institute (SEI) at Carnegie Mellon University (CMU) for 13 years where he became familiar with the Capability Maturity Model (CMM)® earned the authorization to teach the Personal Software Process (PSP)[SM] and worked with Watts Humphrey in pilot testing the Team Software Process (TSP)[SM].

After leaving the SEI, Mr. Koch was the Manager of Software Process and Quality Assurance for a software company. He is now an independent consultant helping companies to improve the return on their software investment by focusing on the quality of both their software products and the processes they use to develop them.

ask@pgh.net http://www.pgh.net/~ask (724) 226-3044

## What does it mean to "assure" quality?

In our headlong rush to do our jobs, we can often find it difficult to keep our objectives in sight. We can get so caught up in the things that we do; the planning and reviewing, the writing and testing, the reporting and tracking; that we can lose track of *why* we are doing all of these things. From time-to-time, we need to take a few steps back from the busy-ness of our work to ask ourselves if we are indeed fulfilling our charter.

We who are responsible for Quality Assurance may find that our objective is not easy to identify. We don't build a product (though we are members of a team that does). We don't provide a service (though we are a key part of our employers' ability to do so). Our job is not as concrete as that of many others in our company, because our job is to "assure".

To "assure". What does that mean? And how does this "assuring" translate into the activities in which we involve ourselves every day? Let us begin to answer these questions by looking briefly at what it means to "assure" something.

### Webster's view

as.sure v.t.

1. to make (a person) sure of something; convince.
2. to give confidence to; reassure: as, the news assured us.
3. to declare to or promise confidently: as, I assure you I'll be there.
4. to make (a doubtful thing) certain; guarantee.

The first thing we see is that "assure" is a transitive verb –an "action" word. To assure is to actively do something. It requires forethought, direction and effort. Each of the four definitions listed above adds a distinctive flavor to our understanding of this action we call "to assure".

### The company's view

"To make a person sure of something" and "to give confidence to; reassure" –These definitions speak to our relationship with our employer. Our employer pays us for assurance. What they need from us is a level of confidence in the integrity of the product that will allow them to confidently release and market it to your customers. Whether our software is the product that is sold, a component of a larger "thing", or it supports a service that is provided to the customer, our employer needs to know that the software does what it is supposed to do, and has a minimum level of defects.

Our management is *not* looking to us for quantification of how *bad* the product is. They don't want to hear about problems from us (though raising issues is an important part of our job). Of course, they want us to ferret out any defects that are hiding in the products; but they want much more than that. They want us to make sure that the defects are corrected. But beyond that, they want to hear (truthfully) that the product is good.

In short, what they need from us is confidence. Confidence that selling the product will enhance the company's image and position in the market. Confidence that the product will make more money than it cost to produce. Confidence that the customer's experience with the product will be good. Confidence that the product will not become a support nightmare. Confidence.

### The customer's view

"To declare to or promise confidently" and "To make a doubtful thing certain" –These definitions relate more to the customer. Every time someone commits to purchase a product or service, they are taking on a certain amount of risk. Will it do what I expect? Will it work properly? Will I be able to use it easily? Can I depend upon it in my business? The customer needs reassurance that the money they are paying to buy the product, and the time they are spending to install and/or learn about it will not be wasted.

Although the customer may not view Quality Assurance as a separable part of the company, we none-the-less have a very real responsibility to them. They don't care about our Quality Plans, our test cases, our reviews, or the fact that hundreds of defects were corrected before the software was released. The customer cares about the value they get for their money, and the problems they encounter in using the product.

In short, the customer needs a useful and trouble-free product. Something they can rely on. A purchase they can be proud of.

### Our Job: To Assure

With these definitions in mind, our job of assuring quality becomes much larger than finding problems and tracking them to resolution. Although making sure that the defects and other problems that we find do not get shipped is important, we can see that neither our senior management nor our customer cares directly about those things.

What they care about is the quality that the customer experiences. You won't hear them boasting about the number of defects that were removed from the product before release. But you will *definitely* hear about it if the number of delivered defects is too high! And even if defect levels are low, the other dimensions of quality may be lacking, resulting in an unsatisfactory experience for the customer.

## The dimensions of quality –A quick survey

Before we look closely at the implications these definitions of "assurance" have on our day-to-day tasks, let's spend a few minutes expanding our definition of "quality". Quality has many dimensions of which lack of defects is but one. This is not to say that assuring that the product lacks defects is not important. To the contrary, a defective product is of little use to anyone.

But it is important that we expand out attention beyond defects and assure that the other dimensions of quality also contribute to our company's success by pleasing the customer. The following few sections will look at each of a representative set of dimensions of quality. Readers are invited to consider other dimensions of quality that may be important in their domains, and subject them to similar analyses.

### Functionality

The first order of business is to provide the functionality that the product's users need. If the product or service is not useful to the customer, then everything else is wasted effort.

In QA, we tend to view the specified functionality as a "given", and the baseline against which we evaluate the software. But even if the software requirements are imposed upon us by the customer or by higher-level system specifications, the quality of that specification is critical to

the success of the project. If incorrect, inconsistent or ambiguous functionality has been specified, then a successful project will be impossible.

**Usability**

Usability is a qualitative measure of the ease with which the intended user of the product can use it. While this sounds straightforward, the realities of our customer-base can complicate it significantly. For example, our product may be used by:

- Technically savvy people who understand and want to exploit the capabilities of the computer,
- Technical novices who are intimidated by the computer and afraid they will break something,
- New users who need to learn how the product can meet their needs,
- Experienced users who want to use the product to quickly and efficiently do a job,
- Advanced users who want to exploit the full functionality of the product.

In reality, it will most likely be used by a whole constellation of people who represent a mix of these attributes. Assuring the usability of the product for each of them becomes a real challenge!

**Reliability**

Reliability is a difficult attribute of the product to quantify, but it may be the most important to the product's users. If they can not count on the product being available and usable, or if they can't be sure it will function properly and handle data accurately, then they will avoid using it. No product at all is preferable to one the can not be depended upon.

The reliability issues that must be addressed with software include:

- Mean Time to Failure (how long the software is likely to work without fault),
- How gracefully it handles hardware and network failures that it will encounter,
- How gracefully it handles user errors,
- How gracefully it handles its own internal faults, and
- How it works under the variety of different situations, configurations, modes of use, and unusual conditions to which it will be subjected by its users.

Another reliability attribute is Mean Time to Repair. There are two components to this:

- When the product aborts or crashes, how much time does it take to restart it? and
- How much time does it take to repair a defect in the product and get the new version into use at the customer's site?

**Install-ability**

Like usability, the operative issue with install-ability is who will do the installing. For products that will be installed by someone in the customer's organization, the technical expertise of the installer may vary widely. A technically novice new user will require a whole different installation procedure from an expert who has worked with and installed similar products many times.

Even if your company's own professional installation team handles the job, their level of knowledge and expertise must be taken into account in determining an appropriate installation procedure.

### Upgrade-ability

Will the same people who originally installed the product install product upgrades? Or do you expect a more casual user to perform this task?

### Maintainability

For some types of products, we must pay attention to the effort that the customer must expend to maintain the software in proper working order (e.g. backups, performance tuning, and load balancing). What activities must the customer engage in? What level of technical expertise do those activities require? How much time do they require? What impact would they have on a 24x7 operation?

For most of us, maintainability is an internal company issue (though as the open-source movement picks up steam, the issues that were discussed under usability, above may complicate it). Either way, this type of maintainability focuses not on how the product works, but on how it was built.

- Will the architecture allow for the types of enhancements that we can foresee?
- Does the design specification provide the information that an engineer would need to understand how the product works?
- Is the code readable?
- Have appropriate levels of abstraction and information hiding been employed?
- Are comments liberal and informative?
- Are the compilers and other required tools available?

### Performance

In a perfect world, everything would be instantaneous. But reality is that we must wait for things to happen. The issue then becomes, "How long is *too* long?" A recent radio commercial pointed out that we are willing to wait different amounts of time for different things: a couple of minutes for a light to turn green; an hour for a table at a trendy new restaurant; as long as possible for a dental appointment.

In the same way, our customers are willing to wait different amounts of time, depending on what they perceive the product to be doing. Our challenge goes beyond building a product with acceptable performance; we must figure out what level of performance our customers will accept.

### Security

Security refers to a broad spectrum of issues:

- Protecting the user's data from unauthorized viewing,
- Protecting the user's data from accidental or malicious alteration,
- Protecting the system from unauthorized use,
- Protecting the software from piracy.

Which of these issues is most important, and to what degree are questions that must be answered before the security features of the product can be designed or evaluated.

### Lack of Defects

When most people think of "quality", they think in terms of defects. (Of course, problems with any of the other dimensions of quality could be called defects, but that is not usually how people think about defects.)

Defects hold an interesting position in people's perception of the product's quality. If there are many defects, the customer sees the product as "defective" and of poor quality. But if there are few defects, they do not necessarily view the product as being good. The quality of a product that is not "defective" is generally judged based on the other dimensions of quality (outlined above).

So we see that while it is important to reduce the number of defects that the customer will experience, that alone will not assure that they judge the product to be of good quality.

## SQA activities – How each contributes to assuring the dimensions of quality

There is a wide variety of activities in which we may engage to assure the quality of our software. Table 1 on the next page illustrates the relationships among the dimensions of quality discussed above, and a variety of QA activities, including work product reviews, process audits and different types of testing. Although not exhaustive, this set of QA activities represents the broad spectrum that is the heart of this discussion. As we can see, no single activity will assure all dimensions of quality. Each QA activity has a unique focus that makes it more or less likely to address the various quality dimensions.

In the following sections we will look at each of these activities to determine which dimensions of quality it is best suited to assuring. Readers are encouraged to subject any additional activities in which they engage to a similar analysis.

### Requirements Review

The responsibility for requirements specification rests with different people in different organizations. But regardless of whether the requirements are written by Marketing, Product Management, Development, or someone else, requirements reviews can be the organization's first and most cost-effective means of insuring product quality and project success.

Besides assuring that an appropriate set of features and functions has been specified, the requirements specification should be reviewed for:

- Reasonableness – Can the system be implemented in a reasonable time using available technology and people?
- Verifiability – Can the specified attributes be verified in a reasonable time using available technology and people?
- Compatibility – Will the product as specified work acceptably with prior versions, or with existing systems?

But above all, a requirements review provides the opportunity to ensure completeness. Have all the critical attributes of the product been specified – especially those that represent the various dimensions of quality? For example:

- Usability – What types of people are expected to use the product? If there are divergent types of users, which features and functions are each type of user expected to use? Are there specific User Interface (UI) attributes that certain users should expect? Are there specific ways that novice or expert users should be accommodated?

| Quality Activities | Functionality | Usability | Reliability | Install-ability | Upgrade-ability | Maintainability | Performance | Security | Lack of Defects |
|---|---|---|---|---|---|---|---|---|---|
| **Requirements Review** | X | ? | ? | ? | ? | ? | ? | ? | |
| **Architecture Review** | | | ? | ? | ? | ? | ? | ? | |
| **User Interface Review** | X | X | | | | | | ? | |
| **Design Review** | | | ? | ? | ? | ? | ? | ? | X |
| **Code Review** | | | | | | ? | ? | ? | X |
| **Documentation Review** | | X | | ? | ? | | | | X |
| **Process Audit** | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| **White-Box Test** | | | ? | ? | ? | | | ? | X |
| **Black-Box Test** | X | ? | ? | ? | ? | | | ? | X |
| **Regression Test** | | | | | | | | | X |
| **Performance Test** | | | | | | | X | | |

| Key | |
|---|---|
| X | Commonly Related |
| ? | Can be Related |
| | Not Generally Related |

**Table 1: Quality Dimensions vs. Activities**

The Requirements Specification should *not* contain a UI design. It should, however, identify the requirements for the UI, and provide the UI designers with enough information to do their job effectively.

- Reliability – What variety of different configurations, and other use conditions must the product support? How tolerant must it be of user errors or hardware or network failures? How much recovery time is tolerable when it crashes? How important is it that bug fixes can be made in the field?

- Install-ability and Upgrade-ability – Who is expected to install and/or upgrade the product? What types of technical expertise do we expect them to possess? What assumptions can we make about their knowledge and background? What kinds of prompting and help will they need to successfully complete the installation or upgrade? Is an installation document necessary?
- Performance – For each of the features and functions, has an appropriate response time been specified? Are the specified response times reasonable? Are they achievable?

  What level of resource usage is acceptable? (E.g. disk usage, memory requirements, network bandwidth) In a multi-tiered architecture, what resource constraints are there at each tier?

  What is the recommended configuration? What is the minimum configuration? In a minimum configuration, what amount of response time degradation is allowable over that of the recommended configuration?

  All of these are likely to be negotiating points between those who specify the requirements and the developers. To allow the developers to deliver whatever performance-level they can is asking for customer dissatisfaction. To give the requirements writers freedom to require whatever they like is asking for impossible requirements. Getting the performance requirements right is a matter of the customer experts and the technical experts reaching compromises on all points of conflict.
- Security – What security issues are important in this product? Must the system be protected from only accidental data loss? Or must it protect against malicious use as well? Is piracy of the software an issue that must be addressed? If so, what level of inconvenience will valid users tolerate?

As can be seen from the discussion above, requirements reviews must include a variety of people to assure that all critical attributes are scrutinized. For most organizations this includes Marketing or Product Management, Development, QA, Technical Writers, and often management.

In most instances, QA's role in these reviews focuses on completeness and verifiability: Have all important attributes been specified? And will it be reasonably possible to verify that the final product contains the specified attributes. In addition, QA should audit the review process (see "Process Audits", below).

Reviewing the requirements for all of these attributes goes a long way toward mitigating the risk of embarking on a project that is doomed to failure. Good, complete, well thought-out requirements do not guarantee a successful product. But poorly conceived, incomplete requirements virtually guarantee failure.

**Architecture Review**

The system architecture specification is not focused on any particular attribute of quality. The point behind this level of analysis is to ensure that the product is structured in the best way to support the specified features and functions, as well as future enhancements that can be foreseen. None-the-less, the architecture review can provide the opportunity to assure that some of the dimensions of quality have been given appropriate consideration.

- Reliability – Is the product structured in such a way to allow support of multiple existing or added configurations? Will the structure adequately isolate failures so that catastrophic system collapse can generally be avoided? Does the structure promote easy repair of problems and distribution of fixes?

- Install-ability and Upgrade-ability – Is the product structured in such a way that the Install-ability and/or Upgrade-ability requirements can be satisfied? Would changes to the Architecture result in a simplified install/upgrade process?
- Maintainability – The Architecture is your first opportunity to address the issue of maintainability. While the first order of business is to support the required features and functions, we must not lose sight of the fact that the structure must allow them to be implemented (and later, maintained) with reasonable effort.

  The most common maintainability problems arise from the fact that your most capable engineers are likely the ones who develop the Architecture; but your less expert developers will have to implement it. Often the Architecture can seem quite straightforward to those who wrote it, but be a total mystery to others who must actually use it! The Architecture review is the organization's opportunity to discover if the architecture is over the heads of some of the development staff.
- Performance – Is the product structured in such a way that the Performance requirements can be satisfied? Indeed, your ability to satisfy performance requirements is often constrained by the Architectural choices that were made early in the project. Therefore, it is imperative that the Architecture be reviewed for performance impacts.
- Security – The product's ability to provide the required levels of protection is likewise constrained by the architectural choices that are made. Will the structure of the product support the Security requirements?

The primary participants in Architecture reviews are drawn from the engineering staff. They are generally your most competent engineers; but as discussed above, involving less experienced personnel can have great value. QA's role in Architecture reviews may be constrained by the level of expertise of your Quality Analysts. In the best case, they would be qualified and capable of fully participating in this review, providing the necessary technical critique from their unique perspective. At the very least, they should audit the review process (see "Process Audits", below).

Specifying the Architecture of a software product is challenging mainly because of the competing priorities. The features and functions must work, the staff must be able to build and maintain it, the product must be installed, it must perform appropriately, and it must be secure. The final architecture will be a series of tradeoffs, with possibly none of these competing priorities receiving optimal treatment. The function of the Architecture review to assure that the tradeoffs have resulted in acceptable treatment of all of the dimensions of quality.

**User Interface Review**

The User Interface (UI) review is fundamentally focused on two dimensions of quality: Functionality and Usability. For example:

- Have all of the features and functions been provided for in the UI?
- Have the needs of the different types of users been accommodated?
- Has the UI been structured to ease the most likely modes of use? That is, are the items that are used together displayed together? Is the sequence of steps the users follow mirrored by the sequence of screens or other interactions?
- Can the user easily recover from errors they will commit?
- Will the novice user be comfortable with the UI?
- Will the expert user be able to efficiently navigate and use the system?

Besides these two obvious issues, the UI review should also assure that the security requirements of the product are factored into the UI. Can the security requirements be met without unduly complicating the UI? Will the added steps that are needed to enforce security be acceptable to the users? (Especially expert users?)

A similar group as reviewed the Requirements specification should review the UI specification. Marketing or Product Management and possibly management will be concerned with the over-all look and feel of the product and engineers will need to be sure that the system can be implemented as specified. The Tech Writers must assure that the specification is complete (so they can use it as the basis for their work) and that the specified UI will behave reasonably. The addition of an HCI (Human-Computer Interaction) expert to the review team would be well worth any costs this would involve.

QA's role in the UI reviews will overlap many of the others, but is most closely related to that of the Tech Writers. In addition, as with the other reviews, they should audit the review process (see "Process Audits", below).

A well-conceived UI Design will help the project team to assure that the user's experience (the ultimate basis for judging the quality of the product) will be a good one.

**Design Review**

In addition to determining that the best design alternatives have been chosen, the main focus of the Design Review (whether of the high-level or detailed level design) is to look for defects. For example:

- Have all of the features and functions specified in the requirements been included in the design specification?
- Do all parties agree that the features and functions as designed meet the intentions of the requirements specification?
- Is the design faithful to the product architecture?
- Have all of the elements of the UI been specified in the design?
- Do all parties agree that the design represents an implementation of the UI that is consistent with the intention of the UI design?
- Can the details of the product be successfully implemented as designed?
- Will the software as designed work correctly?

Besides these issues, the Design Review provides the opportunity to specifically address five other dimensions of quality:

- Reliability – Have all conceivable failure modes been trapped? Does the handling of failures meet the Reliability requirements? Does the design partition functionality to avoid cascading failure? Will the product protect against data loss or corruption when failures occur? Does the design promote easy repair of problems and distribution of fixes?
- Install-ability and/or Upgrade-ability – Have the installation and upgrade steps been designed? And do those steps satisfy the Install-ability and Upgrade-ability requirements?
- Maintainability – In what ways does the design complicate or simplify the maintainability of the product? Are the features and functions isolated so that changes to any one of them would involve only localized code changes? Can features and functions be added without compromising the structure or complicating existing ones? Is the structure straightforward enough to be understood by the less experienced members of the maintenance team?

- Performance – Is the product as designed capable of achieving the performance requirements? What could be done to improve the response times of the various features and functions? What can be done to reduce the resource usage?
- Security – Have the security requirements for the product been designed appropriately? Do the security features as designed provide the levels of protection that are needed? Do they comply with the UI design, and will they be reasonably usable?

As with the Architecture review, the main participants in design reviews are normally selected from the engineering staff. Again, it would be best if your Quality Analysts were technically competent to fully participate, but their participation may be constrained by their level of expertise. At the very least, they should audit the review process (see "Process Audits", below).

**Code Review**

As with design reviews, code reviews are generally focused on detecting and removing defects. For example:

- Have the Design and UI specifications been accurately reduced to code?
- Is the syntax of the code correct? (This is an issue even though nearly all code reviews are performed after the code successfully compiles, because some syntax errors result in syntactically correct code that does the wrong thing.)
- Are the algorithms that were used correct and appropriate?
- Will the code work as expected?

But beyond the issue of defects, the Code review provides an important last opportunity to verify three dimensions for quality that are impossible to test (or not easily testable):

- Maintainability – Was the code written in a straightforward way? Can the less experienced members of the development and maintenance teams easily understand it? Have appropriate comments been included? Is it formatted to enhance readability? Were appropriate names used for variables, functions, classes, etc.? Have all organizational coding standards been adhered to?
- Performance – Will the code, as written be capable of meeting the performance requirements? Can anything be done to improve its speed, response times or resource usage? Are memory leakage or other resource-wasters avoided?
- Security – Have the security features been implemented correctly? Have the common security holes been accounted for (e.g. buffer overflows, or null data)? Have all currently known precautions been included?

As with the Architecture and Design reviews, the main participants in code reviews are normally selected from the engineering staff. Again, it would be best if your Quality Analysts were technically competent to fully participate, but their participation may be constrained by their level of expertise. At the very least, they should audit the review process (see "Process Audits", below).

**Documentation Review**

Software product documentation generally undergoes two different types of reviews:

- Editorial reviews focus on the usability of the documents. They are usually based on organizational standards for the way information is organized and presented, as well as common grammar and structure guidelines.

- Content reviews focus on removing defects. They look at the information that is provided with the intent of identifying information that is incorrect, ambiguous or missing.

In addition, the documentation would benefit from additional attention:

- Usability – In addition to the basic stylistic and language structure issues, the documentation must also be assessed against the expected users' knowledge level and expected activities.
  - For each section of the documentation, are things described in a level of technical detail that is appropriate to the expected user? Will novice users understand the explanations? Will technical experts get all of the information they need?
  - Is the documentation structured to support the way the users will want to use it? e.g. Reference manual style: A complete explanation of each feature and function, complete with all options and permutations. How-to style: Step-by-step procedures for doing particular jobs, especially when a number of discrete features are involved.
- Install-ability & Upgrade-ability – Has all of the necessary documentation been written for those who will install or upgrade the product? Is the level of detail of these documents appropriate to the types of people who will be doing the installation/upgrade? Have the last-minute documents (e.g. release notes) been subjected to all appropriate reviews?

The writing staff most often does the editorial reviews, though QA can provide an important extra set of eyes. The content reviews should include the engineers (since they are the most familiar with the details of the way the product is actually being implemented) and QA (who generally address completeness, compliance with specifications and usability). And, as with the other reviews, QA should audit the review process (see "Process Audits", below).

**Process Audit**

Process audits do not directly address *any* product quality dimensions. Their point is to assure that the organization's processes have been faithfully executed, and that they have been effective in guiding the development work. But in doing so, process audits *indirectly* affect all dimensions of product quality. The following examples of processes that are relevant to this discussion are drawn from the Capability Maturity Model for Software (CMM)[SEI]:

- The Requirements Management process assures that an appropriate set of functionality, as well as all of the other critical attributes of the product are specified, understood and agreed to by all of the appropriate parties. This will involve many activities, including the Requirements Specification review.

  An audit of the Requirements Management process would ensure that all of the appropriate requirements elicitation, analysis, review and approval steps have been taken by the appropriate people in a timely manner. It would also assure that the issues raised in the requirements review are recorded, tracked and resolved in an appropriate and timely way.
- The Peer Reviews process assures that the work-product reviews (Architecture, Design, Code, etc.) are effective in assuring all of the various dimensions of quality.

  An audit of the Peer Reviews process would ensure that all of necessary work products have been subjected to review by the appropriate people in a timely manner. It would also assure that the issues raised in the reviews are recorded, tracked and resolved in an appropriate and timely way.

Every organizational process has a goal of helping the organization to produce a better product efficiently. The process audits check to be sure that this goal is being met.

In some organizations, responsibility for process assurance (e.g. audits) is vested in a different group from product assurance (e.g. testing). But regardless of the organizational structure, both types of activities are important to assuring product quality.

**White-Box Test**

(The term “White-box” is meant to be a counterpoint to the following term, “Black-box”. “White-box” refers to testing, such as unit and integration that is based on a detailed understanding of the implementation details of the software. The point behind White-box testing is to check for problems that can be expected based on the way the software was structured, designed and coded.)

As with most testing, White-box tests mainly focus on detecting defects. Do the algorithms work correctly? Do loops terminate properly? Have variables been initialized? Is memory freed when appropriate? Do the components integrate properly? Do functions return appropriate values? Is data stored in the correct form? Can external files be read and processed? Are exceptions detected and handled appropriately?

In addition to these things, White-box tests provide the opportunity to verify things that would be difficult or impossible to verify using Black-box testing.

- Reliability – A wide variety of configurations can be simulated (especially during Unit testing) so that the software’s ability to work in environments and under conditions that don’t exist in your organization can be verified. You can also simulate a wide variety of error conditions to assure that the software handles them as it should.
- Install-ability & Upgrade-ability – The discrete steps of the installation and upgrade processes can be checked to be sure that all files and data elements are handled correctly, and that all of the necessary elements are put into place.
- Security – The security mechanisms can be tried out to be sure that the controls and protections are working properly, and that data corruption has in fact been avoided.

The engineers usually perform White-box testing. This raises an interesting problem, since many programmers have never been instructed in techniques for designing and performing effective tests, and have often not acquired those skills on their own.

The natural remedy for this problem would be to involve the Quality Analysts in the developers’ test planning (and possibly testing) activities. This would assure that the white-box tests are designed and executed well, and would have the added benefit of helping your developers to refine their skills in these areas. One way of establishing this interaction is for QA to review the developers’ test plans. This could be structured as a reciprocal relationship (see the note about test plan reviews under “Black-box testing, below).

**Black-Box Test**

(“Black-box” testing is done with no concern for – and often no knowledge of – the implementation details of the product. It is done on the basis of the Requirements Specification, and from the standpoint of the expected users. The customer acceptance test is a classic Black-box test; though most organizations perform Black-box tests prior to subjecting the product to formal acceptance.)

Like White-box tests, Black-box tests mainly focus on finding defects. But in this case, the focus is at a much higher level: Have all of the required features and functions been included? Do they work as expected and produce the required outputs? Does the system do unexpected

things, or does it crash under various conditions? Are the specified configurations and environments supported as they should be?

In addition; many other forms of verification can be included during Black-box testing:

- Usability – If at all possible, a few real-life users of the product (representing an appropriate mix of abilities) should test it. They will be your best barometers of your success at building the product to suit them.
- Reliability – As many different configurations and environments as possible should be tested. Beta test programs provide this exposure by allowing the trial use of the product in environments that differ from those in the development organization.

  Also, any types of hardware or network failures that can be reasonably tested should be, to assure that the product handles those situations properly.
- Install-ability & Upgrade-ability – The product should be both installed and upgraded using the supplied documentation by a person of the appropriate skill level.
- Security – Attempts should be made to defeat all of the security measures specified in the requirements; including unauthorized use, inappropriate attempts to view or update data, and attempts to corrupt data.

Black-box testing is usually the job of SQA, and it is generally what is meant when "QA" is used as a verb. ("We're ready to QA the software.") It can, however, be beneficial to enlist the developers in reviewing QA's test plans prior to the start of testing. The developers can often suggest tests that have been overlooked, and they appreciate it when the Quality Analysts subject their own work to the same level of scrutiny that is required of them.

This also introduces the opportunity for reciprocal test plan reviews. SQA reviews the developers' white-box test plans, and the developers review SQA's black-box test plans. This reciprocity goes a long way toward encouraging respect and cooperation among these groups.

### Regression Test

The purpose of regression testing is to assure that a modified product has not been damaged by whatever changes have been made. It consists of re-running previously successful tests (usually automatically) to be sure they are still successful. Because of its very specific focus, regression testing should not be complicated by attempts to add other types of verification.

Regression tests are usually written (and automated if the facilities are available) by SQA. It can be quite beneficial for a simple regression test suite to be incorporated into the build process. This would allow the developers to assure that each new build works at some basic level before declaring it to be available for testing or other use.

### Performance Test

The only effective way to quantify the product's performance is with an explicit performance test. These tests generally require specialized test environments and monitors, as well as special skills and knowledge. These tests are usually automated, and are very special-purpose. Although they will often uncover various kinds of defects, that is not (and should not be) their focus.

Depending on the performance requirements for the software your company writes, it may make sense to hire the necessary experts and purchase the tools and equipment to do this sort of testing

within SQA. Many companies find that contracting this work to a specialized testing contractor is a better choice.

## Who is responsible for quality?

How could a QA department (of *any* size) possibly do all of the activities that have been discussed so far? The answer is, “They can’t.” And they shouldn’t. Any organization that hopes to achieve a good level of product quality must involve every member in a variety of product quality activities.

The idea that the QA department is responsible for quality is a fatal flaw that will sap the strength of any software development organization. This idea promotes the creation of marginal software and pits the QA folks against everyone else. It results in missed schedules, quality deficiencies and lots of finger pointing and blame placing when things go wrong.

Conversely, when the entire team and each member treats the quality of the product as his or her own responsibility, the organizational environment improves, development problems are avoided, and the customer is pleased by a superior-quality product. In this environment, the QA department changes from being the problem or the scapegoat to being expert consultants on software quality and the organizational conscience that keeps the appropriate focus on all of the dimensions of quality throughout the entire development cycle.

Let’s look at the parts of the organization to see how each is responsible for quality.

### Software Specifiers (e.g. Systems Analysts, End users, Marketers)

The people who hold the responsibility for specifying the requirements for the product must see themselves as holding the primary responsibility for the quality of the requirements. They must assure that all of the important features, functions and attributes of the product are specified, and at the appropriate level of detail. They must assure that all of the issues discussed in the “Requirements” section above have received the attention they deserve.

They must also assure that everyone who has a stake in the requirements, or who must use them has the opportunity to review and comment on them. They must assure that the requirements specification represents a true shared understanding among all of the parties, and is not a document that different people will interpret in different ways.

And as the requirements change (which they certainly will do during the course of the project), these people must assume responsibility for controlling those changes. They must collect impact analysis from all of the affected people so that the decision to make the change can be done on the basis of facts, and all parties can commit to working from the changed specification. And they must assure that all changes are clearly communicated so that there is no confusion about what must be developed and verified.

Finally, when requirements defects or requirements process problems crop up, these people must take responsibility for understanding why they happened, and improving the Requirements Management process so that the problems will not happen in future projects.

### Software Developers (e.g. Architects, UI Designers, Software Designers, Coders, Writers)

The people who build the product are the ones who are in the best position to assure the product’s quality. They must understand that quality can not be tested in, or otherwise added at the end. Quality must be built in or it will not be present. Therefore, the only people who can

assure that the product is high quality are the folks who build it. This represents a significant shift in mind-set for most developers; their job is not just to build something, but to build something that is of high quality.

Software developers have a wide variety of quality enhancement tools available to them, including personal reviews, peer reviews, compilers, spelling checkers and other tools, unit testing and integration testing. Many developers are not proficient with most of these are tools, but this proficiency can be learned. When developers become proficient at assuring the quality of their own work, they can begin to produce very high quality products, usually in less time and at lower cost that they otherwise require.

Besides the responsibility for the quality of their own work, the developers must be responsible for assuring that the requirements specification contains all of the information they need and at an appropriate level of detail to provide sufficient guidance for the development effort. In addition, it would be good for the developers to assist QA by reviewing their test plans.

**Software Testers**

Most people view the software testers as being responsible for the quality of the product. In reality, they can do *very* little about quality. Software testers' jobs consist of finding specific defects, tracking them to resolution, and re-testing to be sure they are gone.

That would be like the State Highway Department exercising its responsibility for the quality of the roads by driving around randomly. When one of them hits a pothole, he adds it to his pothole list. Then he meets with the maintenance department manager every week and nags about each hole on his list until it is fixed. Finally he drives back to that spot and drives over it again to be sure the patch worked.

The only real contribution that software testing makes to product quality is in quantifying how effective the other quality activities have been. If a good set of test cases generates very few defect reports, then you can be fairly confident that the product quality is good.

Contrary to most managers' impressions, finding and removing lots of defects during testing is *not* a good sign. This is because good testing efforts generally find about half of the defects that exist in the product at the time of the test. That means that for every defect you find, there is probably another one that you will ship with the product! Put another way:

- If you found 100 defects in system test, then you are likely shipping 100 defects with the product.
- If that number is 1,000, then you are shipping another 1,000 to the customer.
- If you found only one defect, then you are shipping a nearly defect-free product.

So, the paradox is that of all of the members of the development team, the QA folks contribute only indirectly to the quality of the product. But that contribution is an important one. By doing the testing, checking and verifying, they continually challenge the organization, pushing it to produce better software with each project.

**Process Developers and Auditors (e.g. Software Engineering Process Group – SEPG)**

Although most professionals really want to (and try to) do good work; doing consistently good work in practice day-in and day-out requires the support of well-thought-out and defined processes. The function of the SEPG is to assist all of the members of the software development team to define, establish, and assure the consistent use of appropriate processes.

The process auditors (the Capability Maturity Model (CMM) calls them "SQA") are responsible for assuring that the defined processes are followed, and that they are effective in helping people to do their jobs well. Although this could degenerate into a policing function, the process auditors provide a very important conscience for the organization. They help everyone to keep an appropriate focus on the process discipline that will help them to produce the level of quality to which most aspire.

Like the testers, the process developers and auditors contribute to product quality only indirectly. They do it not by looking at the product itself, but by providing the infrastructure to support everyone else's best work.

### Organizational Prioritizers (e.g. Managers)

The most important quality assurers in the entire company are the managers. It may seem odd to think that the people who may not touch the product at all during its development hold this distinction. But consider that the managers:

- Set the priorities. If they decide that quality is not important, then very few people in the organization will give it any serious attention.
- Allocate resources. If the time and money to do reviews or run special tests is withheld, then those things will not happen.
- Establish the reward system. If fire fighting is rewarded then people will set fires.
- Model behavior. They can say whatever they like; but their behavior will be mimicked. If management chooses reactionary responses over disciplined action, that is the way the entire organization will operate.

If the organization is to adopt the resolve to produce high quality products, then that resolve as well as the concrete actions that it provokes must start at the top and work its way down through the entire management structure.

## Making it work in your company

The concepts presented in this paper represent a significant departure from business-as-usual in most organizations. Adopting these ideas into the organization cannot be done quickly. Like any organizational change, these changes must be taken one step at a time. It will take a frustratingly long time to make these sorts of changes; but there is simply no way to change people's behavior quickly.

With this in mind, you must begin by identifying changes that are compatible with your current organizational practices and culture. Identify some relatively quick and easy changes that will demonstrate real progress toward improved quality. Then plan to build on those initial successes over an extended period of time.

### Organizational Implications

Although these changes are all about people's behavior, the main challenge to them is organizational structure and people's understanding of their responsibilities under the division of labor. The prior section, "Who is Responsible for Quality?" identifies significant challenges to the way many people view their job responsibilities. The idea that QA is responsible for Quality is deeply rooted and affects most people's work. Many who are outside of the QA group honestly don't see quality improvement activities as being within their sphere of responsibility.

This is not to say that they believe quality is not important; only that they don't see it as being a part of their jobs.

So the organizational implications of all of this revolve not around changes in corporate structure, but in a broadening of each person's understanding of his or her job. For this change to take place, it must be nurtured in several ways:

1. The company management (starting at the top) must begin to espouse and live by the notion that quality is *everyone's* job. Each person must begin to view the assurance of the quality of their own work as being *their* job. The idea that Quality is QA's job must be actively fought, and must be replaced with the idea that quality must be built in at every step of the development process.
2. At the same time, the QA group's charter must change to support this new view. They must be given a set of responsibilities and activities that actively support everyone else's responsibility for quality. Software testing will likely continue to be a large part of this responsibility. But a new role for QA in reviews is likely. Also, making the QA folks into consultants who can help engineers to devise appropriate Unit and Integration test plans would go a long way. And, of course process auditing in some form must be a part of the job.
3. Many people's formal job descriptions must expand to include the quality enhancing activities that apply to their work.

   If job descriptions do not exist, they should be written. There is no substitute for a clear understanding of the extent and boundaries of one's job.
4. The processes that are employed in doing the various phases of the development work must be similarly expanded so that every task ends with some sort of validation. This is the heart of the ETVX process definition model – that every task (T) has its own validation step (V) as well as entry and exit criteria (E & X).

   Again, if your processes have never been written down, then that effort should begin immediately. The initial work need not be a long, drawn-out, mind-numbing affair. Initial process descriptions can be a simple as a single page of numbered steps with responsible groups or individuals identified.
5. Finally, the organization's reward system must be changed so that the necessary new behaviors and job responsibilities are rewarded. If the current system rewards on-time delivery of code without any regard for its quality, it will undermine all of these other efforts. That kind of system must be augmented so that the best rewards go to those who deliver *high quality* code on time.

**Ideas to take home**

If we take our job of assuring quality seriously, then we must recognize that it involves a wide variety of things. Our company looks to us for a measure of confidence in the goodness of the products it produces. And our customers depend on us to make sure their purchase experience is a happy one.

In order to assure quality, we must be sure to give appropriate attention to *all* of its dimensions, including Usability, Reliability, Install-ability, Upgrade-ability, Maintainability, Performance and Security in addition to Lack of Defects. We tend to focus almost exclusively on defects, and while a defective product is useless to anyone, a product with few defects is not necessarily a *good* product. In order to satisfy our customers' expectations, all of the dimensions of our product's quality must be good.

In order to provide appropriate attention to all of the dimensions of quality, the organization must engage in a wide variety of quality enhancing activities, including reviews of Requirements, Architecture, User Interface, Software Design, Code and Documentation; Process Audits, and a variety of testing including White-box, Black-box, Regression and Performance. No one of these activities will provide good coverage of all of the dimensions of quality; doing a good job requires all of them.

Clearly, this variety of activities is beyond what QA (or any other single department) can do. It requires that every participant in the development effort take responsibility for the quality of his or her own work, and ultimately for the product's quality. Of all of those involved in building a software product, QA has the least opportunity to build quality in. Those who specify and build the product, along with the management team have a much more direct impact on the final quality of the product.

That means that to produce truly high quality products, most companies must re-think their ideas about who is responsible for quality: augment job descriptions, add process steps and alter their reward system. To become a quality-conscious organization, each person must become quality-conscious; and that consciousness must be demonstrated and fostered starting at the top.

This paper provides snapshots of a variety of activities that can be added to your company's quality arsenal. It also provides the compelling arguments for enlisting *all* members of the development effort in the battle for quality. It is up to you to carry these ideas into your organization and start a quality revolution that can eventually put your products at the top of the heap.

Your move.

---

[SEI] Software Engineering Institute, *The Capability Maturity Model: Guidelines for improving the software process* (Addison Wesley, SEI Series in Software Engineering, 1994)

# Practical Lessons Learned in the Coordination of a Multi-site SQA Team

**John Suzuki, CQSE, CSTE**
Software Consultant
JKS and Associates
168 Chandon
Laguna Niguel, CA 92677 USA
+1 (949) 363-1229
john_suzuki@msn.com
and
Metro Information Services
Irvine Office
20 Corporate Park Suite 285
Irvine, CA 92605

and

**George McKewen**
SQA Manager
PSG/PSSU/PCDT
Xerox Corporation
101 Continental Blvd.
ESC1-14D
El Segundo, CA 90245 USA
+1 (310) 333-5521
george_mckewen@usa.xerox.com

**Author's Biographies**

John Suzuki is the principal of JKS & Associates. He currently provides consulting in software requirements analysis and design, software verification and validation, testing, safety and hazard analysis, ISO 9001 and ISO 9000-3/TickIT software auditing, software process improvement, software metrics, Software Engineering Institute's (SEI) Capability Maturity Model (CMM) certification/auditing and provides assistance in documenting various software lifecycle activities. He is a member of the International Council On Systems Engineering (ICOSE), ACM, IEEE, SOLE, IPSE, AAMI and the SRE. He is a certified Software Quality Engineer (CSQE) through the American Society of Quality, a certified Software Test Engineer (CSTE) through the Quality Assurance Institute and trained to perform Software Capability Evaluations (SCE's).

George McKewen is the manager of Software Quality Assurance for the Production Controller Development Team at the Xerox Corporation in El Segundo, California. His team provides SQA support to a large group of engineers developing software for multiple electronic printer products. The development staff, as well as the SQA team is collocated on both the East and West Coasts. This organization was recently successfully assessed at SEI CMM Level 2. Prior to this assignment, Mr. McKewen has held various positions in software development, both as a programmer and as a project software manager responsible for developing future electronic printer products and providing support for current products.

# Practical Lessons Learned in the Coordination of a Multi-site SQA Team


## Abstract

Performing technical activities across geographic and time zone boundaries presents unique challenges to both management and practitioners in today's business environment. The problems of separated groups are becoming a common challenge due to business needs and the multiple (and even multi-national) locations of modern companies. These issues apply to any discipline in which close coordination is required between members of a group. Although this paper discusses the challenges faced in forming and managing a Software Quality Assurance (SQA) group, the principles described here apply to many other multi-site activities.

While it may be intuitive that geographical separation leads to problems in maintaining consistency between groups, the actual problems are not easy to solve. The major problem faced by multi-site operations is difficulty in communication between team members. The barriers to communication that cause the problem equally limit the solution, and this challenge is equally applicable for SQA. Since the audit activity is paced by the progress of the development group, the SQA team is constantly in a reactionary mode, and inter-group coordination and communication fall ever lower on the priority list. This paper addresses some of the problems associated with startup and management of a multi-site group. Although our experience was with a SQA group, we feel the principles involved apply to any multi-site organization. We also describe the barriers to communication, and a number of remedies we found to be useful.


## 1.0 Introduction

Many companies now distribute software development, as well as other technical work, among separate groups and locations. Decreasing budgets and a smaller pool of qualified job candidates are forcing many software development organizations to staff projects with less than the optimum number of skilled personnel. Mergers, acquisitions and consolidations, along with the increase in the physical source size and complexity of many software applications, force companies to distribute work between widely separated physical locations. For many organizations, this leads to difficulties in collaboration and coordination. Historically software development has been an extremely difficult and time-consuming process and requires more than just common access to source code. Proper development in a distributed environment requires tightly coupled work and extra effort in managing the various locations. Splitting the development work geographically and across time zones makes the development effort more difficult to manage and execute.

By analogy, splitting software quality assurance (SQA) activities across two different locations and time zones would make SQA coordination difficult to manage. We have noted that coordination of SQA activities across multiple sites is made difficult primarily by the breakdown in normal communication channels that occurs when groups are not co-located. SQA

coordination is also challenging because of the dynamic nature of a software project as it moves to completion. SQA activities, such as audits and reviews, are paced by the development work, and are therefore constantly struggling to stay current.

## 2.0 Coordination in Development and SQA Teams

It is generally recognized in the software community that proper coordination is a major contributor to project success in large-scale development. Over the past decade, researchers and some progressive software organizations have been actively studying an area called coordination theory [1,2,3,4]. This interest is motivated by the continued growth in the size and complexity of application development projects and the use of the Internet and World Wide Web as virtual development tools. Another driver is the realization that the context of a software work environment and the formal and informal mechanisms of communication are important to project success, especially in large development projects. The desire to improve coordination has also spawned a number of popular computer-supported cooperative work (CSCW) tools and teamware applications which have reinforced the importance of coordination theory.

Coordination has been defined as the managing of dependencies between activities [1]. Kraut and Streeter [2] have combined the various definitions from other coordination researchers and formally defined coordination as "the direction of individuals' efforts toward achieving common and explicitly recognized goals and the integration or linking together of different parts of an organization to accomplish a collective set of tasks". In a software development environment this typically means that a common definition of the project exists, all developers are striving to build and organize the various parts so they fit properly, and that everyone is sharing information and coordinating design activities as the project proceeds. In essence, all the information and activities are being integrated so the application can be handed off to the customer in an expeditious fashion.

In a typical SQA organization, quality assurance information and activities must also be shared and coordinated. Like the development team, the SQA team must also have a common goal or vision. The goal might be a certain level of project audit coverage, a measure of the overall compliance to process and work product standards, or a standard measure of the quality of the tested application. Much like the software development team that is often required to coordinate with a hardware development team, the SQA organization must work in parallel with the software development organization, scheduling work product or process reviews and audits as the development team executes their planned activities. Audits and reviews cannot be performed unless the development team is about to or has recently completed that software activity. As a result, SQA schedules and audit activities must remain flexible to accommodate schedule slips, feature additions, high bug rates and design changes that occur regularly in software projects. This requires a high level of coordination within the SQA organization in order to free up SQA resources to address these unplanned tasks. Additional coordination among SQA engineers will be required to insure that common audit results are obtained from the unplanned development activities throughout the entire organization.

Within the SQA team, a team member who resides at one location must also be careful not to

duplicate work product audits or process audits done by another team member on the same project. Audits must be performed when they minimize the impact on the development schedule if time-to-market is a key business success factor for the organization. This requires careful attention to project and audit schedules between team members. If a SQA engineer is supporting multiple projects, care must be taken to ensure that the audit process is applied in a uniform fashion across the multiple projects. If it is not, some development groups may feel that they are being overly scrutinized or "policed." Conversely, some groups may feel that they are not receiving full value for their contribution to the SQA organization. This uniform process suggests that some SQA standards have to be implemented that will require additional coordination.

For large development projects, portions of the application or features can be assigned or moved to different groups. These may be co-located or geographically distinct. There is some inherent flexibility in the assignment since the work can be distributed among a large group of developers. In a typical SQA organization, resources are limited or fixed (such as 1 SQA engineer per project or 1 SQA engineer per 25 developers) so one cannot easily assign the new assurance tasks to another SQA individual without affecting current schedules or disrupting planned audits. Special care and coordination must take place if the new assurance tasks are assigned to an existing SQA member at a different location who may already be over burdened. When the SQA team is split geographically, additional effort and communication must be expended to ensure that the application of SQA is uniform across the projects and various locations.

## 3.0 Site Environment

Our information for this paper is drawn from several projects within the Xerox Corporation. This particular organization employs over 200 managers, analysts, software developers, testers, and other non-technical support staff. Collectively the group works on a wide range of applications and solutions in the printer industry. Specifically the information was drawn from eight different development projects including several that were split geographically across two locations and time zones. The SQA organization, which was also split between two locations, was formed with three SQA engineers and a manager located on the West Coast. The SQA team was formed as part of the division wide Software Process Improvement initiative to move the organization to CMM Level 2. The SQA group was chartered to comply with the CMM requirements for SQA, and is primarily focused on insuring compliance to defined process. This team is not involved in software testing or evaluation. The team was quickly ramped up to provide eight SQA engineers located on both coasts to support these eight projects. The eight projects represented application sizes from hundreds of thousand of lines of code to millions of lines of code. The eight projects surveyed were at various stages of the software life cycle. The hardware environment included both personal computers and Unix-based workstations with Windows and Unix development tools. The various development members and managers on the eight projects represented a wide range of experience and capabilities. Developer and manager backgrounds included in this survey ranged from a few years to over 25 years of software development experience. Similarly, the SQA organization also represented a wide range of experience and capabilities. A key issue that faced the SQA team was the fact that most of the development projects were already in full development mode. This meant that the SQA team did not have the luxury of developing a full set of processes before starting work. In effect, the team was "changing a tire on a moving car".

The organization recently achieved a SEI CMM Level 2 rating from the Software Productivity Consortium using the CBA-IPI method description. The SQA group was found to have "no observed weaknesses".

## 4.0 Barriers to Coordination

We have identified a number of barriers in Table 1 to coordination and collaboration with multi-site SQA teams. The list of barriers is very similar to the findings of Herbsleb and Grinter [6] for R&D organizations in the telecommunications industry. The similarity is not surprising since most coordination problems are people-related.

**Table 1: Barriers To Coordination**

| | |
|---|---|
| **4.1** | **Physical separation** |
| **4.2** | **Communication differences or preferences** |
| **4.3** | **Personal work style differences** |
| **4.4** | **Lack of trust** |
| **4.5** | **Different backgrounds of team members** |
| **4.6** | **Work load differences** |
| **4.7** | **New SQA team formation** |
| **4.8** | **Not realizing there is a need to communicate** |

### 4.1 Physical Separation

The root cause for most of our inter-group coordination problems was the physical separation of the team members across different locations. While we have identified other contributors, we feel that this is the major driver for all of the barriers. These are the key problems we observed from physical separation of our teams.

- **Loss of Ad Hoc Communication.** We have found that the separation of team members reduces both the opportunity and frequency of formal and informal communication. When you do not see someone on a regular basis, there is a tendency not to call them or to inquire about the status of their activities. We also have noticed that we lose other informal exchanges such as meetings in the coffee room, lunchroom or at the “infamous” water cooler due to location separation. The impact of this loss of informal communication is usually highly underestimated. Kraut and Streeter [2] found in their study that informal communications is the primary way information flows into and through research and development organizations. Herbsleb and Grinter [6] found several consequences from the relative lack of casual contact among software developers. They reported that unplanned contact seems to be one of the primary mechanisms that bring conflicts and issues to light within a development organization. They found that the lack of unplanned contact made it much less likely that conflicts and issues would surface. As a result many more conflicts went unrecognized until much later in the development lifecycle. Addressing the problems later in development was likely to cost the organization more in terms of time and money. Another consequence that they listed from reduced contact was the general lack of information across the various locations. This included how they do their work, what were the most important

issues to them, location specific terms and language, and various responsibilities and expertise of team members. Harder to measure is the loss of gossip and group bonding. From a SQA perspective, we found similar consequences. For instance, we did not realize early on in the team formation process that we were performing audits and reviews in a different fashion. There was no awareness of any conflicts in the way we performed our responsibilities. We also realized that we interpreted various SQA process descriptions differently. If a conflict had been surfaced earlier, the teams at both locations could have resolved the differences at that time.

- **Unique Sub-team Perspectives.** Physical separation seems to foster a unique sub-team perspective (caused by geographical and culture isolation). This is manifested by location cohesiveness (adhering to a particular location's set of internal practices) and unwillingness to share information and knowledge (caused by a lack of trust and a lack of familiarity with other team members). These are not intentional actions but seem to be common human traits that arise naturally from separation, and are probably best described by the old cliché "out of sight, out of mind".

- **Duplication of processes.** We have also found that physical separation led to the duplication of key SQA processes. This was partially a result of the separate evolution of the basic audit process at each location and the fact that different development processes were used in each location. Being new, the team had few defined processes or standards in place. An audit of a development process may have been conducted in a slightly different fashion by the two locations. The geographic distance between the two sites increased the tendency of the team not to check with each other to see how a particular process was implemented, or to see if their process could be used unmodified at the other location. Often, these differences were driven by unique development practices at each site. Considerable time and energy would have to be expended to overcome this tendency. When universal processes were adopted by the SQA organization, each location had to modify their current practices to incorporate and accommodate the new standard SQA practices.

- **Time zone changes.** Although we experienced only a three hour time separation, it was long enough to experience some coordination difficulties. Because of the time zone difference, the effective communication time for the SQA team was reduced from 8 hours to less than 5 hours a day. Team meetings, videoconference calls, and telephone exchanges had to be scheduled to insure the availability of both conference rooms and video equipment, as well as the various team members. Different time constraints due to commuting, different lunch hours, and quitting times, all contributed to a reduction in coordination of group activities and scheduling.

### 4.2 Communication differences or preferences

Another barrier to SQA coordination is the communication differences and preferences of the various team members. There were several distinct methods preferred by team members.

- *Telephone.* The telephone is a great tool for urgent conversation but is limited by the availability of the other person to pick up the call when you need a quick reply to a question. Because of the immediacy of the technology, the telephone is generally considered intrusive

and disruptive.

- *Email.* While generally less intrusive, email suffers from not being read or answered in a timely fashion. Another problem with email is that it is hard to gauge the urgency and importance of the message. In addition, with an ever increasing load of email being received, responses to those you do not know well may be delayed.
- *Face-to-face.* This is the preferred method when important messages or information (i.e., layoffs, site closures, reorganizations) are being communicated or transferred. If face-to-face communication must occur, it also suffers from the time delay until both parties can physically get together.

If differences exist in communication style, there may be a reluctance to communicate with other team members. Ultimately this leads to some loss and a higher cost in coordination.

### 4.3 Personal work style differences

Personal work style differences definitely affect coordination. Some individuals prefer to work collaboratively, while others prefer to work alone. For some individuals, the analysis of documentation is preferred to making appointments to question and talk to individuals. The nature of SQA generally requires the auditors to work independently on their assignment and come together infrequently to report their experiences and progress. This approach generally reduces the interaction with the various team members. The independent work style of the SQA engineers tends to promote the lack of coordination since they tend to focus only on their primary audit and review responsibilities. The SQA manager also has less visibility into what each engineer is doing. This is detrimental since the manager cannot exercise managerial powers to counteract the forces that degrade coordination. Inexperienced SQA engineers may have a tendency not to seek out the more experienced SQA engineers for help since they feel they are interrupting. We have noticed that some SQA engineers prefer to make a detailed plan and follow it closely. Other SQA engineers prefer a more flexible schedule and plan and are thus more tolerant to disruptions and development changes. A more flexible plan however, requires extra attention so the SQA engineer can finish his or her assigned and scheduled audit activities.

### 4.4 Lack of trust

During the development of the multi-site SQA team, trust was not originally perceived as an issue or barrier to coordination. The team communicated by email and with videoconferences on a regular basis. We noticed however, that after the team met face-to-face for the first time for a team building session, the relationships between the team members and the project activities changed. The consensus from team members was that the SQA individuals from different locations seemed to work better together. There was an improved sensitivity and understanding of the problems that each location had to deal with. Project problems were resolved faster after the team building session. Part of this improvement is attributed to increased familiarity with each other and an increase in trust among the various team members. The remainder can be attributed to the feeling that both groups now realized that they were dealing with the same problems and situations on all projects. We believe that both locations finally felt that they were actually part of the same team despite being physically separated.

### 4.5 Different backgrounds of team members

When the SQA organization was first formed, representatives of the multi-site team represented a

wide range of SQA experience and capabilities, from novice to expert. This led to some difficulties in coordination. The difficulties usually manifested itself in two ways. First, less experienced SQA engineers had difficulties in defining the processes and activities that needed to be audited and reviewed over the lifecycle of the project. They were missing the SQA experience to know what needed to be audited and reviewed. This knowledge comes from previous SQA experience on other projects. Despite the lack of formal SQA knowledge, some of these engineers had considerable development talent and valuable application domain experience. The second group consisted of experienced SQA engineers who had little or no domain knowledge of the current application. While they had extensive experience in general SQA, they had trouble initially understanding the processes and techniques that the various teams had been using to build the software. Both situations led to coordination inefficiencies while both sets of individuals tried to correct their respective deficiencies. The tendency was to try to solve the problems first by themselves instead of seeking assistance from the various experts within the SQA team. And finally, the ratio of skills and experience was not evenly distributed between sites.

### 4.6 Work load differences

Another coordination barrier that we saw was a result of workload differences among the various team members. Because the development efforts were not equally balanced between locations, and the distribution of SQA Engineers was also not balanced, the workload on individual team members was not equal. Those team members that supported more than one project often found it difficult to spend time on coordination activities with their team members since they were so busy supporting their primary project SQA responsibilities. Their focus was on the next audit or review they were supposed to perform, not on how to communicate or coordinate with the other location. Schedule pressure and productivity goals (such as the number of audits completed) shift the focus from communication and coordination to activity results. In the instances where two SQA engineers supported a common project from two different locations, one engineer was typically assigned primary responsibility while the other had secondary responsibility. The secondary SQA engineer typically focused on his or her primary activities instead of support of the second project. This often led to coordination and prioritization problems, especially in a resource limited environment.

### 4.7 New SQA team formation

Another barrier to coordination that the SQA team faced was the challenge of starting a new team while also trying to work out the problems of multiple site coordination. Not only were many of the SQA engineers new, but the management team was inexperienced with multi-site management issues. Interestingly, Herbsleb and Grinter [6] stated in their research that the biggest unidentified challenge of assembling new development teams was the new relationships that the development organizations had to forge with the other locations. We found a similar situation in our analysis.

### 4.8 Not realizing there is a need to communicate

Noted software engineering author Gerald Weinberg made the recent statement [7], “the fish is always the last to see the water. Culture is like that, and the only way to find out that you’re in it is the same way the fish finds out that it’s been in water all its life - by suddenly finding itself out

of the water." Technical folks are notoriously unaware that effective communication is missing, that communication needs to occur more often, or that the current forms of communication are not as effective as they believe. Sometimes it takes a study like this for software individuals to be the fish out of water and to realize that email, collaboration tools, teamware, and formal communications sometimes are not enough to facilitate effective teams, processes and organizations. Other times we need to step away from the logical and technical side of development and acknowledge the difficulties with communication from a human perspective. This difficulty is probably a result of so many of us being trained technically and not realizing the importance of the human side of software. Software development continues to be a human performance and social activity [8]. Communication plays a large role in the relationships of individuals and teams and ultimately in the success of software projects. After forty years of studying software development organizations, Weinberg has noted and stated that all technical problems are human problems in disguise [9].

## 5.0 Impact of Coordination on Processes

During the initial stages of organization of the multi-site SQA organization, it was recognized that geographically separate SQA teams would pose some challenges in managing and coordinating the SQA activities for the projects. Our initial efforts on SQA coordination focused on creating a basic set of standard SQA processes and templates.

- SQA Process Description - The high level description of SQA tasks that all team members could reference and use as a general roadmap for each development project.
- Document Management Process - A guide for maintaining the version control of SQA documents.
- SQA Master Plan - A generic SQA plan for use by all projects using SQA services.
- SQA Project Plan template - Template for creating project specific SQA plans.
- Audit process - A generic guide on how to conduct and report on audits.

These were the initial work products or processes created to facilitate better coordination among SQA team members. As the work progressed, additional guidelines and procedures were developed. These supplemental documents were developed as a team to achieve consistency and buy-in between locations.

**Table 2 : Coordination Activities versus Time and Location**

| | | LOCATION | |
|---|---|---|---|
| | | **Same** | **Different** |
| **TIME** | **Same** | Lunch Room Conversation<br>Water Cooler Conversation<br>Office/cubicle exchange<br>Hallway exchange<br>Co-Located Meeting<br>Team Building Session | Regular Phone Calls<br>Telephone Conferencing<br>Video Conferencing<br>Web Conference - Net Meeting<br>Weekly SQA Staff Meeting<br>with Video Conference |
| | **Different** | Posted Notices<br>White Boards<br>Bulletin Boards<br>2nd or 3rd Shifts<br>SQA Peer Training | Email<br>Voice Messages<br>Snail Mail/Interoffice Mail<br>Audit Reports<br>Monthly Status Reports<br>Central Deviation Database<br>DocuShare Repository<br>Telecommuting<br>Workload Balancing<br>Project Preferences/Assignments<br>General Software Training<br>Standard Checklists<br>Standard Templates<br>Standard SQA Plan Templates<br>Self - Audits<br>Newsletters<br>Standard Audit Process<br>Standard SQA Process Description |

## 6.0 Practical Tools and Techniques to Reduce Barriers to Coordination

Over the course of setting up the multi-site SQA organization we have identified and incorporated a number of techniques, processes and methods to improve coordination. These coordination activities, along with a few additional activities not described in this paper are summarized in Table 2. This table shows the various coordination activities arranged by time and location requirements. These tools, techniques, and processes also are effective in improving coordination of Software Development and SQA teams that are co-located. We have organized these factors into three categories: People Techniques, Process Techniques, and Technology Techniques.

### 6.1 People Techniques

The consensus among SQA engineers from both locations was that the team building session was the most important and effective technique that we used to reduce the effects of location on coordination. This meeting brought the two teams together to a common, off-site location for a week of team building and working sessions. Although there are many formats for structuring an effective team building session, we found the following activities to be effective for us.

- **Bring remote groups together in a common location.** Although techniques such as phone and video conferencing can be effective, we found the face to face meeting of team members to be the most effective way to break down the communication barrier.
- **Hold the team building meeting offsite, away from the work environment.** The most effective team building occurs when team members are not being constantly interrupted by

phone calls and "drop-ins". Being in the work environment, there is always the tendency to check voice and email, leading to interruptions in the team flow. Although it is sometimes difficult to reserve a large block of time for the entire group, the payoff was well worth it for us.

- **Conduct actual working sessions to resolve key issues and develop common processes.** As part of the team session agenda, specific SQA team tasks were also scheduled and discussed before returning to our respective locations. This insured a common understanding of the goals for each task, and helped gain buy in from all team members.
- **Have a professional facilitator assist with team building exercises.** During one team building session, a moderator from our Human Resources department assisted us in conducting a valuable team exercise on personality and temperament. This exercise helped to explain the work style and communication differences and preferences of the individual team members. This proved to be a very enlightening and valuable exercise. It really led to a noticeable improvement in the coordination of the team. There are several profile tools in common use (i.e. Life Orientation (LIFO), Birkman, Personal Profile System, Meyers Briggs Type Indicator (MBTI), etc.) and most Human Resource groups have personnel skilled in conducting moderated sessions.
- **Extended meeting length (all day meetings over several days).** We found that it took several days before the barriers to open and direct communication began to fall.
- **Lunch and dinner together as working and social events.** We found that having lunch together really accelerated the bonding process among team members. The more informal and relaxed atmosphere of meals fostered good communication among the team members.
- **Upper management participation.** The working sessions included separate meetings with several levels of management and other organizational representatives with whom SQA interfaced. These sessions were very valuable for gaining a common understanding of the expectations of the management chain.

The team building session accomplished several key enablers for better communication.

1. *Lowered cultural barriers.* The meeting helped reduce the cultural barriers between the two locations. The two sites had been entrenched in their own culture of doing audits and reports based on the local development environment. This had not been conducive to fostering or promoting coordination.
2. *Improved trust and communication.* The team building exercises and various team meetings, including attending lunch and dinner together, seemed to improve trust and communication between the locations. Part of the improvement came from an understanding of the communication and work style differences of the team members as well as an improved sensitivity to the challenges that both locations faced on each project. Since that first team building session, we have recommended to senior management the need for an annual team building meeting.

Building on the success gained from the offsite team building session, we also employed a number of other people techniques to promote better coordination.

- *Weekly SQA team communication and status meetings.* These were most effective when video conferencing was used.
- *A separate videoconference between the SQA manager and the team at the other location.*

This was a valuable addition to identify and resolve issues and alleviate feelings of isolation that were unique to the remote team.

- *Workload balancing and task alignment.* SQA team members were assigned to development projects based on work preferences and work strengths as another means to enhance coordination.
- *Standard training process for new SQA engineers.* This training has been recently developed, and utilizes a mentor or buddy approach for all new SQA engineers.
- *Participation in outside software activities such as conferences and group classes.* These activities are added as budget and time restrictions will allow.

### 6.2 Process Techniques

Although they are not as powerful as people techniques, process techniques can also be used to improve coordination. Simple process techniques can usually be implemented easily since they do not require extra cost or expensive tools. Some of the techniques we employed included:

- **Standard work products.** Each standard document or template used helps insure that audits and reviews will be carried out consistently across locations and across projects. Some of the standards we use are audit report forms, standard checklists, standard templates, standard method to report deviations, standard processes and common definitions.
- **Use of self-audits by development teams.** Recently, in an attempt to reduce the audit and project workload among SQA engineers, we have been experimenting with self-audits. These audits tend to emphasize process compliance with the development team members and managers and effectively allow the SQA team to perform more audits or investigate other areas of the project. The self-audits are all standardized and require a developer or development manager to fill out a simple audit questionnaire. The initial results from these types of audits look promising and will allow more time for the SQA team to work on improving group coordination.

### 6.3 Technology Techniques

Aside from the regular use of email and direct person-to-person telephone conversations, we have utilized a number of technology solutions to improve coordination.

- **Video conferencing.** Despite some limitations with video conferencing, (speech and video delays, slow speed, unable to secure video conferencing rooms) it has provided the closest form of face-to-face communication without actually being physically present, and had proven to be the most important technology tool we have used. The cost and time savings from reduced travel are substantial. A weekly SQA staff meeting is conducted with the entire team, as well as a separate meeting with the remote site using this form of technology. The video conferencing systems used have dedicated telephone lines, and are all housed in conference rooms.

- **Telephone conference calls.** In addition to video conferencing, several SQA engineers have scheduled the use of telephone conference calls to increase coordination on shared projects. The SQA engineers are using the phone to hold regularly scheduled project status meetings. These phone conferences are used to update the schedules, allocate extra work, and talk about

challenges and suggestions for improvement. The phone is still an important tool to facilitate coordination. However, it is still amazing to realize how reluctant we can be to pick up the phone and use it, especially when we are busy.

- **SQA deviation database.** Another technology tool that the two different locations utilized is a common SQA deviation database utilizing client server architecture. This database, built from Microsoft Access®, allows a centralized and standard method of entering, tracking and reporting of project deviations. Valuable metrics and project statistics are gathered from the database, which help the SQA manager better coordinate the various development projects and SQA organizational activities. The centralized database prevents duplication of the deviation entering and tracking process, and reduces reporting efforts. This leads to a more productive group and enables better communication when sharing project responsibilities. Currently we have one SQA engineer responsible for database administration of the deviation database.

- **SQA Process Asset Library.** As the SQA organization executed its plans and processes, we created a collection of SQA knowledge, general information, and a record of its audit activities. We realized early in the formation of the team that we needed a central repository to deposit, store and retrieve key project and SQA information. While not all team members need access to all the information, the ability to share such information with the SQA manager and with project management was considered key. With multiple SQA engineers supporting more than one project, the need to coordinate project information, knowledge, and results was considered a top priority. The SQA group organized their reports, plans, standards, methods and results in a process asset library (PAL) using DocuShare®, a document repository application available from Xerox. DocuShare® allows the team to use standard Internet browsers to create, edit and access a team portal where project schedules, forms, templates, references, audit checklists, audit reports, and SQA tools can be centrally stored. One of the reasons for using this document database was that various groups within the organization, including developers, were already using the tool for storage and retrieval of project information. Another benefit of DocuShare® is that it allows the team to keep any document in its native format (i.e., Word, Excel, PowerPoint®, Acrobat®, and Microsoft Project). The use of this tool promotes document consistency, and allows for a wide range of search options.

- **Microsoft NetMeeting®.** Recently, technology has allowed us to enhance our telephone communication. Microsoft NetMeeting®, is currently being tested within the SQA group to facilitate internal document review and document collaboration. This tool allows multiple users to view and edit the same document in real time. It has worked best for us by ignoring the sound capability of the application and using regular telephone conference calls for the audio. We have realized that the ability to view and share a common window is an invaluable feature and really promotes document and process coordination within the team.

## 7.0 Conclusions

We have presented a brief list of barriers to coordination and some methods to improve coordination in multi-site SQA organizations. Although tools and technology do help to improve

productivity within a team, the largest gains for improving coordination are all related to the human aspects and relationships between people, teams and organizations. Many of the listed techniques or processes that we utilized or described here can be implemented in any group or organization with minimal cost. We also discovered that the most important method in improving coordination between groups separated by distance is to physically bring the parties together as soon as possible. Face-to-face meetings prove to be valuable in promoting group norms, increasing sensitivity among team members, and improving communication. We feel that the benefits of a face-to-face meeting far outweigh any expense incurred by the SQA organization in bringing the team members together from separate or distant locations.

We have seen a number of short-term benefits to the organization and the various projects from the increased emphasis on coordination. First, there appears to be better communication between the various team members. In addition, there is an improved sensitivity to the communication challenges between the two locations. The various standards, templates, forms, checklists, and processes have all improved SQA productivity and have insured more consistent audit results between the locations and the different projects.

The longer-term benefits of improved coordination include reduced costs, and a more simplified and uniform approach to performing SQA across the organization. From a managerial perspective, the improved coordination within the team makes it easier to manage and lead. The standard methods and processes will also make it more efficient to bring new SQA engineers on board and to enable them to be more productive more quickly. The improvement in productivity and the reduction in SQA costs are important factors today in a competitive business environment and contribute to project and organizational success.



**References**

[1] T.W. Malone and K. Crowston, The Interdisciplinary Study of Coordination, *ACM Computing Surveys*, Vol. 26, No. 1, 1994, pp. 87 - 119.

[2] R.E. Kraut and L.A. Streeter, Coordination in Software Development, *Communications of the ACM*, Vol. 38, No. 3, 1995, pp. 69 - 81.

[3] R.E. Grinter, From Workplace to Development: What Have We Learned So Far and Where Do We Go?, *International ACM SIGGROUP Conference on Supporting Group Work, GROUP '97, 1997. Phoenix, AZ.* ACM Press, pp. 231 - 240.

[4] N.B. Harrison and J. O. Coplien, Patterns of Productive Software Organizations, *Bell Labs Technical Journal*, Summer 1996, pp. 138 - 145.

[5] R.E. Grinter, J.D. Herbsleb, and D.E. Perry, The Geography of Coordination: Dealing with Distance in R&D Work, *International ACM SIGGROUP Conference on Supporting Group Work, GROUP '99, 1999. Phoenix, AZ.* ACM Press, pp. 306 - 315.

[6] J.D. Herbsleb and R.E. Grinter, Splitting the Organization and Integrating the Code: Conway's Law Revisited, *21st International Conference on Software Engineering, ICSE '99*, *Los Angeles, CA.* ACM Press, pp. 85 -95.

[7] Gerald Weinberg, Personal communication, Shape Forum, May 31, 2000.

[8] Gerald Weinberg, *The Psychology of Computer Programming,* Silver Anniversary Edition, Dorset House, New York, New York, 1998.

[9] Gerald Weinberg, Problem Solving Leadership (PSL) Workshop Notes, December 1999.

**Acknowledgements**

The authors would like to thank Vanessa Otto for her helpful comments. John Suzuki, who works as a consultant for Metro Information Services, Irvine, CA office, would like to acknowledge Metro Information Services for their continuing support and training.

# Using Reading Techniques to Support Inspection Quality

## Lessons learned from a large-scale software development experiment on the individual defect detection effectiveness of reading techniques in two inspection cycles

Stefan Biffl  Thomas Grechenig  Michael Halling
*Vienna University of Technology, Institute of Software Technology*
*Karlsplatz 13, A-1040 Vienna, Austria*
*Stefan.Biffl@tuwien.ac.at*

**ABSTRACT**
Defects in the requirements of a software product, like wrong, incomplete, and inconsistent information, cause costly development cycles, delay time to market and lower product quality. Inspection of a requirements document can detect defects in an early stage of development, improve software quality, and prevents effort for unnecessary rework. Individual reading is an important step in the inspection process, where inspectors go through the document independently and record defects found. The effectiveness of individual reading depends on inspector ability and the reading process applied.

This paper presents an experiment to investigate the effectiveness of reading approaches to help individual inspectors find given sets of defects (regarding the severity level and the location in the document) in a requirements document. After correction of the defects found in the initial inspection, the inspectors used the same reading technique to find more defects in a second inspection cycle.

Main findings of the experiment are that the reading techniques focused inspector attention on specific aspects of the inspection object, i.e. they made a difference for the probability of inspectors to find minor and major defects and to find groups of defects in different document parts. Consequently building inspection teams of 6 or more persons from inspectors using a mix of all reading techniques was more effective than taking any mix that excluded one or more reading techniques. The introduction of a second inspection cycle on average improved product quality but not inspection quality compared to the initial inspection. These results suggest to improve defect detection in the initial inspection rather than conducting a reinspection.

*Keywords:* Software Inspection, Defect Detection, Reading Techniques, Defect Classification.

The authors can be reached via email at {biffl, grechenig, halling}@swt.tuwien.ac.at.

Stefan Biffl is an assistant professor of software engineering at the Vienna University of Technology. His research interests include project management and quality management in software engineering. Biffl received an MS and PhD in computer science from the Vienna University of Technology and an MS in social and economic sciences from the University of Vienna. He is a member of the ACM and the IEEE.

Thomas Grechenig is an associate professor at the Vienna University of Technology. His research interests include human-computer interaction and project management in software engineering. Grechenig received an MS and PhD in computer science from the Vienna University of Technology.

Michael Halling is research assistant at the Vienna University of Technology. His research interests include project management and quality management in software engineering as well as economic models that account for various sources of risk and international financial markets.

## 1 INTRODUCTION

Defects in the requirements of a software product, like wrong, incomplete, and inconsistent information, cause costly development cycles, delay time to market and lower product quality. Product quality, a low number of deviations from specified quality levels, is a key to market success.

Inspection of a requirements document can detect defects in an early stage of development, improve software quality, and prevent unnecessary rework effort (see [Gilb93]). Inspection leaders in industrial development need defect detection approaches for their environment that are effective. Such approaches must find a high proportion of the defects present in a given work product, and be resource-efficient by exhibiting a low cost per defect.

Individual reading and the systematic coordination of individual reading activities among a group of inspectors are particularly effective defect detection approaches [Port94].

Effective individual defect detection is a necessary basis for effective inspection team performance and high inspection quality. Reading techniques (RTs) are the procedures that guide an inspector during defect detection. They are expected to influence the probability of finding a given set of defects in a work product.

A RT can be built to examine the work product from one viewpoint, out of a set of different viewpoints. The most effective combination of a set of RTs is a major concern when planning the best mix of individual activities for a cost-effective inspection team.

If the initial inspection of a work product suggests that a substantial number of defects still remain undetected, a second inspection cycle (reinspection) can be scheduled to raise product quality to an acceptable level. There are very few reports on the effects of an actual reinspection.

In this work we investigate the influence of 2 RTs and 3 RT roles on the effectiveness and efficiency of individual inspectors to detect defects in a software requirements document in a controlled experiment. We use data on the detection probability for given defects by inspectors who use different RTs to estimate the number of defects a group of inspectors will find with a particular mix of RTs. We examine the effectiveness of a second inspection cycle to improve the product quality.

Section 2 presents related work on reading techniques that support inspectors in their reading tasks: non-document-specific checklists and focused scenario-based reading techniques tailored to document types and defect classes. Section 3 describes the controlled experiment with software engineering students to investigate effects of defect detection techniques on the number and type of defects found. Section 4 presents the experiment results. Section 5 discusses lessons learned from the experiment, their significance for inspection, and concludes with guidelines for inspection planning.

## 2 APPROACHES TO DEFECT DETECTION

The goal of inspection is to identify and measure product and inspection process quality. Product quality is determined by the number of defects remaining in the product after development and after inspection.

A key objective of defect detection is to find as many major defects as possible within a certain timeframe and given set of inspectors. The effectiveness of an inspection process is defined as the ratio of defects found to the total number of defects in the product, or in a given defect class like regarding defect severity or location.

The inspection process consists of these 3 steps: defect detection, collection and repair. Variations of the number and sequence of these steps can occur in the number and sequence of the steps or the execution of a step. In this work we focus on the individual inspector's contribution in the defect detection step using a specific reading technique and on the effects of reinspection.

### 2.1 Reading techniques

Reading, a key activity in defect detection, is to understand a given software artifact representation and compare it to a set of expectations regarding structure, content, and desired qualities. The recognition of differences between expectations and understanding of the artifact helps to spot defects. The proceedings of ad hoc inspections depend completely on the capability of the inspector and not on a repeatable process.

Defect detection in an inspection can be supported with techniques that guide the inspectors through the inspected document and instruct them what to look for and how to perform the quality checking. These techniques are commonly referred to as reading techniques.

Major characteristics of a reading technique are its applicability, the repeatability of the inspection process, and the responsibility that it assigns to inspectors. Applicability denotes the scope of documents, which the technique may aid to inspect, e.g. the product type and notation. Repeatability is essential for a well-defined inspection process. Inspection responsibility may be general, to identify as many defects as possible, or specific, to focus the inspector's attention on a limited coverage of the document or of a set of defects.

An example for a general reading technique is a checklist that lists all of the defect types or symptoms to look for in a particular document type, usually independent from a specific notation. The inspector has to map checklist questions to tasks and plan how to traverse the document. Checklists can cover a broad range of issues but may require the inspector to read through the document several times sequentially, which limits the appli-

cability to documents of a limited suitable size, or makes the inspector decide which parts of the document to skip.

Basili *et al.* [Basi96] have developed a set of reading techniques[1], called scenario-based reading (SBR), that uses procedures to detect specific classes of defects. These scenarios consist of a viewpoint or emphasis description, a set of procedures to abstract information from the document, and a set of questions to identify defects. The scenarios guide readers through a document with a particular emphasis or viewpoint, and, thus, must be combined to provide complete coverage of the document

Perspective-based reading (PBR) is a SBR technique that focuses on the point of view of the customers of a software artifact, e.g. users, designers or testers of the requirements document. For each customer perspective there can be a scenario to produce a model, e.g. a user manual for the user's view, high-level design sequence diagrams for the designer's view, or test cases for the tester's view. These models can be analyzed to answer questions based on the particular perspective's qualities. The assumption is that the union of perspectives provides extensive coverage of the range of defects present in the document, while each reader is responsible for a narrowly focused view of the document, which is supposed to lead to a more in-depth analysis of potential defects in the document.

Basili *et al.* [Basi96] and Ciolkowski *et al.* [Ciol97] report on experiments on the inspection of requirement documents in natural language with professional subjects and students, where PBR performed better than checklist-based reading (CBR).

Software requirements artifacts typically consist of descriptions for functional and non-functional requirements of the future system in natural language and model notations, e.g. UML class, state or sequence diagrams. Object-oriented models represent multiple views on the system that are created at different times. Developers of large systems have to concentrate on a part of the system and have to rely on the consistency of all parts. They need checks for horizontal consistency to avoid contradictions between models in the same development stage, and vertical consistency to trace correct refinement through the different development stages (see [Lait99]).

A goal of inspection is to find incorrect, inconsistent, ambiguous, missing or extraneous information in these descriptions that may lead to problems in later development stages.

[1] See *http://www.cs.umd.edu/projects/SoftEng/ESEG/* for material on reading techniques (SBR, PBR, TBR) and experiments.

Travassos *et al.* [Trav99] have suggested a SBR technique for defect detection in object-oriented design documents called traceability-based reading (TBR), which describes how to perform correctness and consistency checks among various UML models. Due to this specific focus on a limited set of important defects TBR is to be used in combination with other reading techniques for complete coverage of the artifact. The researchers have performed a case study to show the feasibility of the approach, but not with experimental control.

The literature on RTs proposes different effectiveness and efficiency levels of inspector groups who use a range of reading techniques. We will investigate the effects of CBR and a combination of PBR and TBR techniques on the individual level.

### 2.2 Individual defect detection task expertise

Sauer *et al.* [Saue00] propose that the effectiveness of both individual preparation and the group meeting depend on the level of inspector task expertise for defect detection and defect discrimination, i.e. the ability of an inspector to discern a defect, to distinguish among defect types, and to detect certain defect types. They state that a bad inspection process can hinder the application of good task expertise, but even an excellent process can not make up for poor inspector task expertise.

The tasks of a RT, that are to help an inspector discover and discriminate defects, vary with the parameters of the inspection object (e.g. product type and notation) and the target set of defects (e.g. defects with syntax or semantics). Further they follow implicitly or explicitly a plan to cover all or some parts of the product, striving for an overview of the product parts and then setting a more specific focus to check details. This plan has to be calibrated to allow inspectors in the target ability range to execute the reading tasks within the allotted inspection time.

The individual performance of the inspectors in a team is the basis for the team's performance. The definition of tasks can be used for selection, training, and the activation of inspectors' defect detection capabilities. Gathering empirical data on practical task expertise in inspection contexts is a key to improve RTs and tailor them to the goals of a project or an organization.

### 2.3 Combining individual defect detection activities

The effectiveness of the meeting step in inspection has been questioned in literature [Vott93, John98]. The meetings of experiment teams did not contribute more new defects than were lost during the meeting. Thus we deal in this work with nominal teams of inspectors without team interaction.

The individual defect detection activities in RTs are designed to focus the inspector's attention on different sets of defects. Individual inspectors with specific RTs

cover only part of the document. Thus the combination of RTs is expected to yield better defect coverage for a given team size than a group using a single RT.

### 2.4 Reinspection to improve product quality

A second inspection cycle can be considered by the project manager, if the initial inspection yields evidence for low product quality or for low inspection quality (see [ElEm99]). From a reinspection we expect an improvement of product quality (see also [Biff00a]) and want to investigate the effects on individual defect detection effectiveness and efficiency.

Between the inspection cycles the inspectors and their detection aids stay the same, while their knowledge of the document, the defects in the document, and their experience with the RT all increase. On the other hand, the number of defects in the document decreases. The effects of team interaction between the cycles on individual performance in reinspection merit investigation.

## 3 Experiment Description

This section introduces the experiment environment, the inspection object and process, the RTs applied, and a list of expectations for experiment results.

### 3.1 Experiment environment

The experiment was part of a university software development workshop that teaches 200+ undergraduate students every year to develop a real medium-size software product. All students knew how to develop small programs; some 10 percent had professional experience.

The inspection object was a 35-page requirements document containing 9000 words and 6 diagrams. The document described a distributed administrative information system for managing ticket sales, administration, and location management. The system description consisted of 4 parts: context information in natural language text and illustrating diagrams, business functions and non-functional requirements in structured natural language and tables, a relational database model in entity-relationship notation, and an object-oriented class model and class description in the notation of the Unified Modeling Language.

In the inspection object 86 defects, 48 minor and 38 major defects, were re-seeded that had been found during development of the requirements document. All defects in the requirements models and business functions, such as wrong, missing, unclear, or inconsistent information, could be found without need for reference to external documentation. Defects were classified according to their location in the document and the severity of their impact on development and product quality, if not detected. Gilb and Graham [Gilb93] distinguish minor and major defects according to the difference of their cost (an order of magnitude or more) to find and remove in later development or operational stages.

To ensure that the inspection process in the experimental design was representative of software development practice, we applied inspection activities that had been used in a professional development environment ([Port94]).

### 3.2 Experiment process

The experiment consisted of 3 phases: A training phase and two inspection cycles.

The main experimental steps of an inspection cycle were individual defect detection, a team meeting, and the correction of true defects found by the team.

During individual reading each subject applied the assigned reading technique independently to the inspection object; results for each subject were a defect list and an inspection protocol providing feedback on their effort and use of the assigned reading technique. After this step all members of a team met to consolidate their individual defect lists to a common team defect list and to write a protocol on meeting effort.

Inspection supervisors performed an initial data analysis: They checked the completeness and validity of the collected defect and effort data and tried to match each defect on the team defect list with a defect in the reference defect list, which had been provided by the experiment team. Every matched defect was corrected in the requirements document creating a unique version of the specification for further work.

All teams received feedback from their workshop supervisor that there was still substantial numbers of major defects in their specification and proceeded to reinspect their version of the specification. After the reinspection the corrected requirements document became the official specification for the development effort of the team.

### 3.3 Reading techniques used in the experiment

The experiment investigated effects of CBR and SBR viewpoints, tailored to a requirements document as described in section 3.1.

The general checklist consisted of 7 quality sections, e.g. completeness and testability, and questions regarding defects in the respective quality section, e.g. ‘Are there business functions that lack proper description of input or necessary information on processing?’ or ‘Are there defects in the notation of data, function, or object models?’

We tailored scenarios from PBR ([Basi96]), which originally dealt with natural language requirements, and scenarios from TBR ([Trav99]), which deal with UML models, to our experiment environment. The experiment scenarios were made for the viewpoints of: user (SBR-A), designer (SBR-D), and tester (SBR-T). They each consisted of a PBR-related part, and two TBR cross-checks between models. The user role was to derive use

cases from context information and business functions, and to check business functions and their constraints with data model entities and their integrity rules. The designer role constructed sequence diagrams from the business functions and the object model, and checked the consistency of these models. The tester role built test cases from business function input and non-functional requirements, and checked the consistency of entities and attributes of the data model with the object model.

### 3.4 Focus on individual defect detection

Initial data analysis revealed that the meeting step in the experimental environment was not effective, by not uncovering a substantial number of new defects. Thus we concentrated on the individual defect detection step and on combining inspectors into effective nominal teams. These were teams without interaction, where collating the defect lists of individual inspectors is done independently by a single person with tool support.

We analyze the number of defects an inspector found to describe his contribution to product quality. We compare inspectors' detection effectiveness as this measure normalizes the number of defects found by the number of defects remaining.

### 3.5 Expectations for experiment results from theory

From experiment reports and intuition during the experiment preparation we listed expectations for experiment results.

*Expectations for the effect of RTs (process in-the-small):*

1. RT focus: CBR and SBR roles influence the probability of a given defect to be found. Thus the readers of different RTs find different sets of defects regarding defect severity and location.
2. RT effort and efficiency: RTs encourage different approaches to read the document. We expect these approaches to differ in their associated effort and efficiency.
3. RT combination effects: We expect a larger gain of effectiveness from an additional inspector for a nominal group with different RT viewpoints than for a group of homogeneous RT readers, due to the combination of different defect sets they are supposed to find.

*Expectations for the effect of a second inspection cycle (process in-the-large)* where the inspectors and RT assignments stay the same, while the set of defects in the document as well as the experience with the document, the kind of defects, and the RT all change.

4. $A_1 > A_2$: The number of defects remaining in the document is smaller during reinspection ($T-B_1$), thus, using the same process the number of defects found during reinspection, $A_2$, is expected to be smaller than during inspection, $A_1$.
5. $M_1 <= M_2$: The number of major defects found can be higher, if the inspectors are more familiar with the product and the easy defects have run out. On the other hand ineffective inspectors may not find major defects even in the second cycle.
6. $F_1 >= F_2$: Reinspections are not more effective than inspections, since the process stays the same and the number of defects in the document decreased, especially the ones that are rather easy to find.
7. $I_1 > I_2$: Reinspections are less efficient, than inspections, i.e. the average effort to find a defect is higher, since defects, that are easy to find, are expected to be found during the first inspection cycle.

*Table 1: Definition of experiment variables.*

| | |
|---|---|
| T | The number of defects in the inspection object (known in the experiment only). |
| $A_i$ | The number of defects found by an inspector in inspection cycle i. |
| $B_i$ | The number of defects found by a team in inspection cycle i. |
| $m_i$, $M_i$ | The number of minor/major defects found by an inspector in inspection cycle i. |
| $F_i$ | Individual effectiveness, that is the individual defect detection rate in an inspection cycle, $F_1 = A_1/T$, $F_2 = A_2/(T-B_1)$. |
| $H_i$ | The cost of an inspection cycle in staff hours, the effort for individual reading. |
| $I_i$ | Individual efficiency, $A_i/H_i$, i.e. the average number of defects found per staff hour. |

## 4 Results

Valid data was gathered from 169 inspectors in 31 teams of 4 to 6 persons: 86 checklist-based readers (CBR), 83 scenario-based readers (SBR) with the viewpoints user/ designer/tester with 29/29/25 inspectors respectively.

In this section we present data from both inspection cycles. First we investigate the overall contribution and effectiveness of inspectors, the RT focus on minor and major defects as well as by document part. Then we examine the effectiveness of nominal teams from a mix of RT roles. Third we explore the effect of team interaction, the individual inspector contribution for reinspection as well as effort and efficiency changes between the 2 cycles.

### 4.1 Reading technique focus

The contribution of inspectors to product quality is the absolute number of defects they find in each inspection cycle ($A_1$ and $A_2$).

*Table 2: Absolute number of defects found by RT role.*

| | Inspection $A_1$ | | Reinspection $A_2$ | |
|---|---|---|---|---|
| | mean | std.dev | mean | std.dev |
| CBR | 17.3 | 8.3 | 7.7 | 4.7 |
| SBR | 13.0 | 6.9 | 7.4 | 4.6 |
| SBR-A | 13.8 | 6.4 | 7.5 | 4.3 |
| SBR-D | 12.6 | 6.7 | 7.5 | 5.0 |
| SBR-T | 12.6 | 7.8 | 7.3 | 4.8 |

In the initial inspection CBR readers find significantly ($p<<0.01$) more defects than SBR readers (see table 2). Differences between contributions of the SBR roles are not significant on an alpha level of 0.1. The difference between the contribution from the first and second inspection cycle is significant ($p=0.01$) for all RT roles. In the second inspection cycle the contributions of all RT roles do not differ significantly.

*Table 3: Individual detection rate by RT role.*

| | Inspection $F_1$ | | Reinspection $F_2$ | |
|---|---|---|---|---|
| | mean | std.dev | mean | std.dev |
| CBR | 20.1% | 9.7% | 18.8% | 13.4% |
| SBR-A | 16.0% | 7.4% | 16.2% | 9.2% |
| SBR-D | 14.7% | 7.8% | 16.9% | 11.3% |
| SBR-T | 14.7% | 9.1% | 15.4% | 9.6% |

Table 3 shows the individual defect detection rate. For the first inspection cycle the statements from the absolute number of defects hold. The effectiveness of CBR readers is on average lower in the second cycle, while SBR readers improve somewhat. The effectiveness differences between the two inspection cycles are not significant.

Overall the defect detection effectiveness is similar in both inspection cycles. The RT focus on minor and major defects calculates the effectiveness of a RT for the respective set of defects. Minor defects are found significantly ($p=0.02$) more effectively in the first cycle, while major defects are found significantly ($p=0.03$) more effectively in the second cycle (see table 4).

*Table 4: Detection rate by inspection cycle and severity level.*

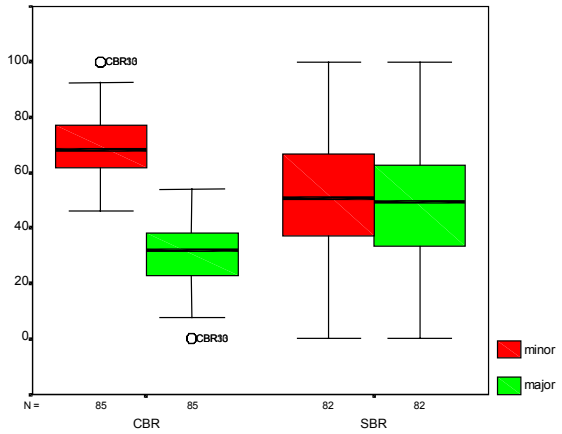
| | Inspection $F_1$ | | Reinspection $F_2$ | |
|---|---|---|---|---|
| *Severity* | mean | std.dev. | mean | std.dev. |
| Minor | 19.5% | 11.3% | 16.8% | 13.6% |
| Major | 15.5% | 9.9% | 18.4% | 13.5% |
| *All defects* | 17.7% | 9.2% | 17.5% | 11.9% |

*Table 5: Detection rate for minor and major defects by RT and inspection cycle.*

| | CBR | | SBR | |
|---|---|---|---|---|
| *1. cycle* | mean | std.dev. | mean | std.dev. |
| Minor | 24.4% | 10.7% | 14.3% | 9.4% |
| Major | 14.7% | 9.6% | 16.3% | 10.2% |
| *2. cycle* | mean | std.dev. | mean | std.dev. |
| Minor | 17.8% | 14.8% | 15.7% | 12.2% |
| Major | 19.7% | 14.8% | 17.0% | 12.0% |

In the initial inspection CBR readers are significantly more effective ($p=0.01$) finding minor defects, while SBR readers find a significantly ($p=0.09$) higher proportion of major defects (see figure 1 and table 5) regarding the number of defects an inspector found. CBR readers find major defects more effectively in the second inspection cycle, and minor defects significantly better ($p<0.01$) in the first cycle. SBR readers get more effective in finding both kinds of defects.

*Figure 1: Boxplot[2] of the focus of RTs on minor and major defects in the first inspection cycle (% of defects found).*




The focus of a RT on document parts denotes the proportion of defects an inspector found for the 4 document parts. Table 6a and 6b exhibit the mean focus by document part for both inspection cycles and all RT roles. The distribution of the proportion of defects found is an indicator whether two RTs focus inspector attention on the same or different document parts. In the first inspection cycle SBR-A and SBR-T follow a similar distribution as well as CBR and SBR-D. A more detailed

[2] We use boxplots to show data distributions: The box spans the central 50% of the data. The upper and lower ends of the box mark the upper and lower quartiles. The data median is denoted by a bold line within the box. Vertical lines attached to the box indicate the tails of the distribution and extend to the standard range of the data. All other detached points are outliers.

analysis of the probability of individual defects to be found confirmed that the focus of the SBR roles was different.

*Table 6a: Means of focus distributions for 1st cycle by RT role.*

| | Cont | Busi | Data | Obj |
|---|---|---|---|---|
| CBR | 19.9% | 26.9% | 42.0% | 11.2% |
| SBR-A | 21.1% | 43.1% | 34.5% | 1.4% |
| SBR-D | 15.3% | 25.3% | 38.1% | 21.2% |
| SBR-T | 25.0% | 36.2% | 33.1% | 5.7% |

In the second inspection cycle the distributions changed; particularly the document part with the object model received more attention at the cost of the business function model. Here the RTs CBR and SBR-A show a more similar distribution.

*Table 6b: Means of focus distributions for 2nd cycle on document parts by RT role.*

| | Cont | Busi | Data | Obj |
|---|---|---|---|---|
| CBR | 18.5% | 22.9% | 38.6% | 20.0% |
| SBR-A | 18.5% | 26.6% | 32.5% | 22.4% |
| SBR-D | 9.7% | 15.5% | 31.9% | 42.9% |
| SBR-T | 7.7% | 23.4% | 54.4% | 14.5% |

### 4.2 Combination of reading techniques

Inspection planners want to optimize defect coverage for a given team size with the right mix of reading techniques assigned to the inspectors.

We can calculate the expected defect detection effectiveness for a nominal team with a given mix of reading techniques. As input we take the probability of each individual reference defect in the experiment to be found by readers with a given RT. Then we determine the probability that at least one reader in the nominal group will find this defect and sum up the expected probabilities to the expected team effectiveness (the algorithm considers the covariance of defect pairs, see also [Biff00b]).

The first column in table 7 shows the mean defect effectiveness of a single inspector for initial inspection and the 4 RT roles: C denotes a CBR reader, A/D/T denote an SBR-A/D/T reader. CBR readers are significantly more efficient than the other 3 RT roles. When looking at the top performing inspection team with 3 inspectors, all teams with one or more CBR readers (column 2) are on average more effective than teams without a CBR reader (column 3).

For the teams without a CBR reader the team is most effective that combines all 3 SBR viewpoints. Teams that contain 2 or 3 RT roles are more effective than the best non-mixed SBR team.

*Table 7: Detection effectiveness of RT combinations (%).*

| Single inspectors | | Top 4 teams with 3 insp. | | Viewpoint combinations | | Effectiveness gains | |
|---|---|---|---|---|---|---|---|
| C | 20.1 | CCC | 42.9 | ADT | 37.8 | C | 20.1 |
| A | 16.0 | CAD | 41.9 | AAA | 34.9 | CC | 13.4 |
| D | 14.7 | CAT | 41.5 | DDD | 34.2 | CCC | 9.4 |
| T | 14.7 | CDT | 40.6 | TTT | 33.8 | CCCC | 7.0 |

The effectiveness gain from adding an inspector diminishes as a team's size grows. Column 4 in table 7 shows the effectiveness gains for adding CBR readers. The effectiveness gain from 2 to 3 CBR readers (+9.4%) is lower than the gain from two SBR-A readers to a team of 3 that contains all 3 SBR roles (+10.8%, see [Biff00b] for details).

For a team of 6 inspectors, an all-CBR team (59.5%) is on average less effective than a team with 3 CBR readers and the 3 other RT roles (62.5%). Although individual CBR readers perform better than SBR readers, a mix of all 4 RT roles is most effective, since the gains for an additional inspector diminish faster, if a single RT is used.

### 4.3 Team effect on individual contribution

Assembling nominal team results from independent inspector results is no problem with data from the first inspection cycle, since this data is available without team interaction. For reinspection all defects were corrected that a (real) inspection team had found. Thus an inspector started the second cycle with a document whose defect density depended on the performance of his/her inspection team. For the analysis of individual reinspection data we have to consider these defect density differences because members of better teams have fewer defects left.

In data analysis we have to check whether the performance of the team in the initial inspection cycle influenced the average effectiveness of an inspector during reinspection. We aim to test whether differences in the team effectiveness in the first inspection cycle make it harder for inspectors with low effectiveness in the first cycle to improve in the second cycle (since good teams leave fewer and possibly harder to find defects for the second inspection cycle). We divided the teams into three groups with low, medium, and high effectiveness in the first inspection cycle and investigated whether individual effectiveness is different for these 3 groups.

For all 3 groups of teams the mean change of effectiveness of its inspectors between the 2 inspection cycles is not significantly different. Thus we conclude that the team interaction in the first inspection cycle did not have a notable effect on individual defect detection in the second inspection cycle, which we investigate in the following section.

### 4.4 Inspector contribution in reinspection

In this section we look at inspector contribution to product quality in the second inspection cycle on the individual inspector level.

*Figure 2: Individual defect contribution ($A_1$ horizontal, $A_2$ vertical) for both inspection cycles by RT.*

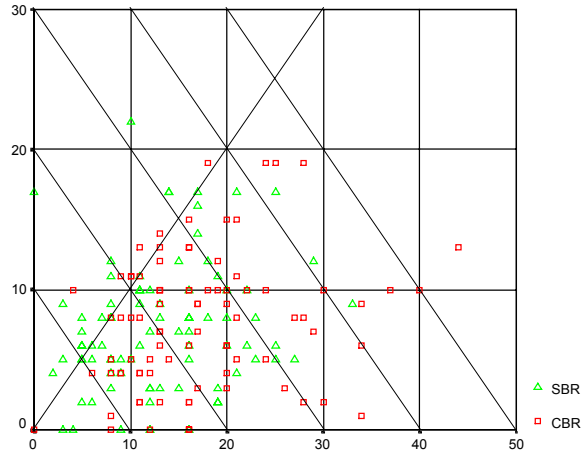



Figure 2 shows the absolute number of defects each inspector contributed in the 2 inspection cycles. Almost all inspectors helped to improve the product quality in the second inspection cycle. 80% of the inspectors contributed less in the second inspection cycle than in the initial inspection.

*Figure 3: Individual detection rate (%) for the first ($F_1$, horizontal) and second inspection cycle ($F_2$, vertical).*

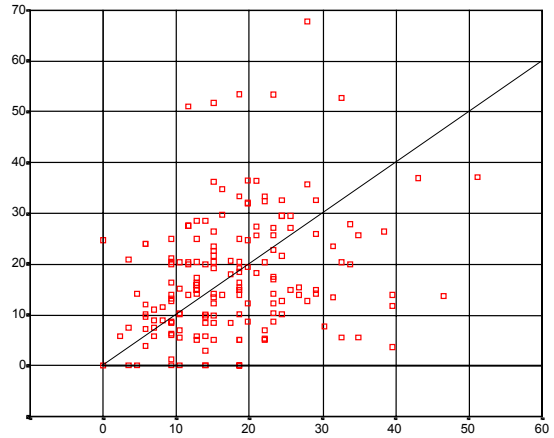



Table 3 shows that the mean individual effectiveness was indifferent to the inspection cycle. Yet in figure 3, that exhibits the individual effectiveness for both inspection cycles, the picture looks different. Half of the inspectors were (much) more effective in the second inspection cycle, while the other half was (much) more effective in the initial inspection (correlation coefficient = 0.296). The distribution of major defects is similar to the distribution of all defects.

### 4.5 Effort and coverage

The variable effort and coverage depend on the RT used and influence the number of defects to be found, since we expect to find more defects with a higher time investment or with more thorough coverage of the inspection object.

*Table 8:Mean effort in inspection cycles by RT role ($H_i$).*

| | Inspection $H_1$ | | Reinspection $H_2$ | |
|---|---|---|---|---|
| | mean | std.dev | mean | std.dev |
| CBR | 5.6 | 2.1 | 3.4 | 1.6 |
| SBR-A | 4.5 | 1.9 | 3.6 | 1.3 |
| SBR-D | 5.0 | 2.3 | 3.5 | 1.7 |
| SBR-T | 4.4 | 2.0 | 3.1 | 1.5 |

In the first inspection cycle CBR readers read longer than SBR readers; SBR roles each read similarly long (see table 8). In the second cycle all readers invest less effort and take similarly long.

*Table 9a: Mean detection rate by RT and effort for the initial inspection.*

| | CBR ($F_1$) | | SBR ($F_1$) | |
|---|---|---|---|---|
| | mean | std.dev. | mean | std.dev. |
| 0 to 2 h | 15.1% | | *8.4%* | 5.2% |
| 2 to 4 h | *17.9%* | 7.3% | 12.1% | 7.2% |
| 4 to 6 h | *23.4%* | 9.3% | 18.8% | 7.9% |
| 6 to 8 h | 19.5% | 11.3% | 20.5% | 4.4% |
| 8 to 10 h | 16.6% | 12.0% | 17.4% | 6.0% |

*Table 9b: Detection rate by RT and effort for reinspection.*

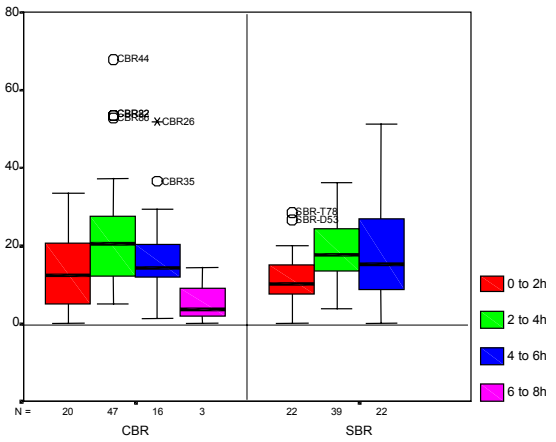
| | CBR ($F_2$) | | SBR ($F_2$) | |
|---|---|---|---|---|
| | mean | std.dev. | mean | std.dev. |
| 0 to 2 h | 12.7% | 10.5% | 11.3% | 7.8% |
| 2 to 4 h | 22.5% | 13.9% | 18.3% | 8.9% |
| 4 to 6 h | 17.9% | 12.4% | 17.5% | 12.3% |
| 6 to 8 h | 6.0% | 7.4% | | |

Tables 9a and 9b show the detection effectiveness by reading effort and RT (see also figure 4). As expected the defect detection rate ascends with the effort invested, but surprisingly decreases after a peak. This peak suggests a number of different approaches by inspectors to use their RT, which result in different levels of effectiveness regardless of the time invested.

Together with the lower effort invested during reinspection the coverage of the inspected document was significantly lower during reinspection. Yet there was no

significant relation established between document coverage and defect detection effectiveness.

*Figure 4: Detection rate for reinspection by RT and effort (%).*




### 4.6 Efficiency

Inspection efficiency, the number of defects found per hour of reading, is an important criterion for the project manager, since it determines the effectiveness return on an invested resource unit.

*Table 10: Defects found per hour of reading ($I_i$).*

| | Inspection $I_1$ | | Reinspection $I_2$ | |
|---|---|---|---|---|
| | mean | std.dev | mean | std.dev |
| CBR | 3.5 | 2.0 | 2.4 | 1.6 |
| SBR-A | 3.6 | 2.5 | 2.2 | 1.6 |
| SBR-D | 2.8 | 1.8 | 2.2 | 1.7 |
| SBR-T | 2.9 | 1.8 | 2.3 | 1.5 |

As we found out that CBR readers found more defects and also read longer than SBR readers, we compared the efficiency of these RTs (see table 10). In the initial inspection CBR and SBR-A readers both were more efficient than SBR-D and SBR-T readers. During reinspection all inspectors were less efficient and all readers exhibited a similar mean level of efficiency.

### 4.7 Inspector qualification

Before the experiment we let the workshop supervisors grade the inspector candidates according to their development and problem-solving performance during the entry test of the workshop as A/B/C which meant above average/average/below average (see also [Biff00c]).

This coarse measure, which is not specifically tailored to inspection, is still significant for dividing the inspectors into groups who differ in their inspection performance. Inspector groups A and C perform on average significantly (p=0.04) different for reinspection (see table 11).

*Table 11: Detection rate ($F_i$) by inspector qualification (A,B,C).*

| | A | | B | | C | |
|---|---|---|---|---|---|---|
| | mean | std.dev | mean | std.dev | mean | std.dev |
| Insp. | 19.6% | 8.6% | 17.8% | 8.8% | 15.6% | 10.1% |
| Reinsp. | 20.5% | 13.2% | 17.5% | 11.8% | 14.5% | 10.0% |

## 5 Discussion of Results

In this section we discuss the experiment results with the expectations derived from theory and lessons learned from the use of RTs and further research issues.

### 5.1 Reading technique effects

*Expectation 1 – RT focus: CBR and SBR roles influence the probability of a given defect to be found. Thus the readers of different RTs find different sets of defects regarding defect severity and location.*

This expectation was met. Inspectors using checklists found on average more defects than users of scenarios. Inspectors using scenarios found in the initial inspection more defects, which are important from a project/quality manager's point of view, i.e. more severe defects or defects that are hard to find without a reading technique. The distribution of RT focus on document parts was different.

*Expectation 2 – RT effort and efficiency: RTs encourage different approaches to read the document. We expect these approaches to differ in their associated effort and efficiency.*

This expectation was met. CBR readers read longer and were more efficient than SBR readers in the initial inspection.

Overall, the choice of RTs proved to be able to significantly influence the individual performance of inspectors.

### 5.2 Combination

*Expectation 3 – RT combination effects: We expected a larger gain of effectiveness from an additional inspector for a nominal group with different RT viewpoints than for a group of homogeneous RT readers, due to the combination of different defect sets they are supposed to find.*

This expectation was only partly met. CBR readers were overall significantly more effective than the SBR readers were (which deviates from the results of another experiment, see [Mill98]). Thus the expected effect could only be seen for inspection groups of 6 or more inspectors, where a mix of all 4 RT roles is most effective, since the gains for an additional inspector diminish faster, if a single RT is used. SBR viewpoints find different defects and work more effectively in combination.

### 5.3 Reinspection

The second inspection cycle provided a positive contribution to product quality.

*Expectation 4 – $A_1 > A_2$: The number of defects in the document is smaller during reinspection (T-$B_1$), thus, using the same process the number of defects found during reinspection, $A_2$, is expected to be smaller than during inspection, $A_1$.*

This expectation was met. The benefits of inspection cycle diminish significantly. The inspectors read shorter in the second inspection cycle (-1.4 to –2.2 hours depending on the RT) and found on average less defects: CBR readers on average found half the number of defects they found in the initial inspection, SBR readers less than 70%.

*Expectation 5 – $M_1 <= M_2$: The number of major defects found can be higher, if the inspectors are more familiar with the product and the easy defects have run out. On the other hand ineffective inspectors may not find major defects even in the second cycle.*

This expectation was not met, but the mean effectiveness to find major defects increased somewhat for both groups of RT readers.

*Expectation 6 – $F_1 >= F_2$: Reinspections are not more effective than inspections, since the process stays the same and the number of defects in the document decreased.*

This expectation was only partly met, as the mean $F_1$ and $F_2$ do not differ significantly. Yet half of the inspectors were more effective in the second inspection cycle, which contradicts the expectation and suggests some learning effects.

*Expectation 7 – $I_1 > I_2$: Reinspections are less efficient than inspections, i.e. the average effort to find a defect is higher, since defects that are easy to find are expected to be found during the first inspection cycle.*

This expectation was met. All RT roles were on average less efficient and reached only 60% to 80% of their efficiency levels in the initial inspection.

Overall the differences between RTs regarding effort, efficiency, and the number of defects found, which could be observed in the first inspection cycle, are no more significant in the second inspection cycle. Yet the contribution of the second inspection cycle is lower than from the first inspection.

### 5.4 Use of experiment results for inspection practice

The software engineering students in the experiment probably match novice software developers or customers with limited inspection experience better than experienced software engineers. We suggest the following guidelines for introducing a focused reading technique to inspectors from this population.

- *Preparation:* If there is an inspection process in current use, find out the effectiveness and efficiency of this process. Characterize the typical inspection objects (type, size, complexity) and list specific goals for product quality and the resources available (see also [Basi96]).
- *Introduction:* Use or tailor RTs in your specific environment (defect and document types, inspection goal and inspector viewpoints) that have been empirically tested for a comparable surrounding. Improve team inspection effectiveness with a mix of RTs.
- *Analysis:* Analyze inspection tasks for task-oriented selection and training of inspectors. Consider inspector qualification in general and specific for the RT tasks applied. If you have the choice, use fewer inspectors who match the inspection tasks in the mix of RTs well than a lot of people who do not really fit to the inspector profile of the inspection at hand.
- *Monitoring:* Measure the inspection quality and update your choice of the RT mix accordingly.

### 5.5 Lessons learned

Main findings of the experiment are that the RTs focused inspector attention on specific aspects of the inspection object. They made a difference in the probability that inspectors would find minor and major defects and groups of defects in different document parts.

Consequently building inspection teams of 6 or more persons from inspectors using a mix of all RTs was more effective than taking any mix that excluded one or more RTs.

The introduction of a second inspection cycle on average improved product quality but not inspection quality compared to the initial inspection. The experiment results suggest improving defect detection in the first inspection cycle rather than systematically conducting reinspections.

As a by-product of the inspection process investigation we found a simple measure of inspector qualification that significantly separated inspector groups with different mean defect detection effectiveness in the experiment. Further work will be to identify and improve tasks from RTs for better inspection planning as well as for selection and training of inspectors in general and for a given RT.

## REFERENCES


IESE/ISERN technical reports can be downloaded from *http://www.iese.fhg.de/Publications/*.

[Basi96] V. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Soerumgaard, and M. Zelkowitz, "The Empirical Investigation of Perspective-Based Reading", *Empirical Software Engineering: An International Journal*, vol. 1, no. 2, 1996, pp. 133-164.

[Biff00a] S. Biffl, M. Halling, M. Köhle, "Investigating the Effect of a Second Software Inspection Cycle", *Proc. APAQS 2000 Asia-Pacific Conference on Quality Software*, Singapore, IEEE Comp. Soc. Press, Oct. 2000.

[Biff00b] S. Biffl, M. Halling, W. Gutjahr, "Investigating the Inspection Effectiveness of Nominal Teams Varying Team Size and Defect Detection Methods", *Technical Report 00-25, Dept. Software Engineering, Vienna Univ. of Tech., Austria*, May 2000.

[Biff00c] S. Biffl, "Analysis of the Impact of Reading Technique and Inspector Capability on Individual Inspection Performance", *Technical Report 00-22, Dept. Software Engineering, Vienna Univ. of Tech., Austria*, April 2000.

[Ciol97] M. Ciolkowski, C. Differding, O. Laitenberger, and J. Münch, "Empirical Investigation of Perspective-based Reading: A Replicated Experiment", Fraunhofer Institute for Experimental Software Engineering, Germany, *Tech. Report Int. Software Engineering Research Network*, 1997, ISERN-97-13.

[ElEm99] K. El Emam, O. Laitenberger, and T. Harbich, "The Application of Subjective Estimates of Effectiveness to controlling Software Inspections", Fraunhofer Institute for Experimental Software Engineering, Germany, *Tech. Report Int. Software Engineering Research Network*, 1999, ISERN-99-09, accepted for publication in the Journal of Systems and Software.

[Gilb93] T. Gilb, and D. Graham, *Software Inspection*, Addison-Wesley, 1993.

[John98] P.M. Johnson, "Reengineering Inspection," *Comm. of the ACM*, vol. 41, no. 2, February 1998.

[Lait99] O. Laitenberger, and C. Atkinson, "Generalizing Perspective-based Inspection to handle Object-Oriented Development Artifacts", *Proc. of the Int. Conf. on Software Engineering*, 1999.

[Mill98] J. Miller, M. Wood, and M. Roper, "Further experiences with scenarios and checklists", *Empirical Software Engineering: An International Journal*, vol. 3, no. 1, 1998, pp. 37-64.

[Port94] A. Porter, and L. Votta, An Experiment to Assess Different Defect Detection Methods for Software Requirements Inspections, *Proceedings of the 16th International Conference on Software Engineering,* 1994, pp. 103-112.

[Saue00] C. Sauer, D. Jeffery, L. Land, and P. Yetton, "The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research"; *IEEE Trans. on Software Engineering* vol. 26, no. 1, January 2000, pp. 11-14.

[Trav99]G. Travassos, F. Shull, M. Fredericks, and V. Basili, "Detecting defects in object-oriented designs: Using reading techniques to increase software quality", *Proceedings of the Conference on Object-oriented Programming Systems, Languages & Applications (OOPSLA)*, 1999.

[Vott93] L. Votta, "Does every Inspection need a Meeting?"; *ACM Software Eng. Notes* 18(5): 107-114; 1993.

# The Darker Side of Metrics

Douglas Hoffman, BACS, MBA, MSEE, ASQ-CSQE
Software Quality Methods, LLC.
24646 Heather Heights Place
Saratoga, California 95070-9710
doug.hoffman@acm.org

## Abstract

There sometimes is a decidedly dark side to software metrics that many of us have observed, but few have openly discussed. It is clear to me that we often get what we ask for with software metrics and we sometimes get side effects from the metrics that overshadow any value we might derive from the metrics information. Whether or not our models are correct, and regardless of how well or poorly we collect and compute software metrics, people's behaviors change in predictable ways to provide the answers management asks for when metrics are applied. I believe most people in this field are hard working and well intentioned, and even though some of the behaviors caused by metrics may seem strange, odd, or even silly, they are serious responses created in organizations because of the use of metrics. Some of these actions seriously hamper productivity and can effectively reduce quality.

This paper focuses on a metric that I've seen used in many organizations (readiness for release) and some of the disruptive results in those organizations. I've focused on three different metrics that have been used and a few examples of the behaviors elicited in organizations using the metrics. For obvious reasons, the examples have been altered to protect the innocent (or guilty).

## Biography

Douglas Hoffman is an independent consultant with Software Quality Methods, LLC. He has been in the software engineering and quality assurance fields for over 25 years and now is a management consultant specializing in strategic and tactical planning for software quality. He is Section Chairman for the Santa Clara Valley Section of the American Society for Quality (ASQ) and is past Chairman of the Silicon Valley Software Quality Association (SSQA). He is also a member of the ACM and IEEE, and is certificated by ASQ in Software Quality Engineering and has been a registered ISO 9000 Lead Auditor. He has earned an MBA as well as an MS in Electrical Engineering and BA in Computer Science. He has been a speaker at dozens of software quality conferences including PNSQC and has been Program Chairman for several international conferences on software quality.

# The Darker Side of Metrics[1,2]

## Introduction

Software measures and metrics have been around and used since the earliest days of programming. I have studied and used software measures and metrics with varying degrees of success throughout my career. I might even be labeled a reformed measurement enthusiast[3]. During the 25 years or so that I have studied and used software metrics I have been surprised by some of the effects the metrics have had on the organizations, and often I have been extremely distressed over the negative impacts I have seen. Even though I have touted software metrics and successfully begun several metrics programs, every software organization I have observed that has used metrics for more than a few years has had bizarre behaviors as a result. There is a decidedly "dark side" to these metrics programs that impacts organizations all out of proportion to what is intended. In the last year Kaner[4,5,6] has provided a framework for understanding why this might occur. One source comes from a lack of relationship between the metrics and what we want to measure (Kaner's 9th factor)[7] and a second problem is the over-powering side effects from the measurement programs (Kaner's 10th factor)[8]. The relationship problem stems from the fact that the measures we are taking are based on models and assumptions about system and organizational behavior that are naïve at best, and more often just wrong[9]. Gerald Weinberg provides excellent examples in his *Last Word* article analyzing some benign software inspection metrics[10]. Weinberg shows how counting defects found during preparation and at code inspections gives metrics relating mostly to the number of inspectors and telling almost nothing about the product or process it proports to measure.

It is clear to me that we often get what we ask for with software metrics. Whether or not our models are correct, and regardless of how well or poorly we collect and compute software metrics, people's behaviors change in predictable ways to provide the answers management asks for when metrics are applied. Don't take me wrong; I believe most people in this field are hard

---

[1] This information was first generated for presentation and discussion at the *Eighth Los Altos Workshop on Software Testing* in December, 1999. I thank the LAWST attendees*, Chris Agruss, James Bach, Jaya Carl, Rocky Grober, Payson Hall, Elisabeth Hendrickson, Bob Johnson, Mark Johnson, Cem Kaner, Brian Lawrence, Brian Marick, Hung Quoc Nguyen, Bret Pettichord, Melora Svoboda,* and *Scott Vernon,* for their participation and ideas.

[2] I differentiate between the measures of an attribute and metrics computed from the measures. Ultimately we should take measures to compute metrics.

[3] Lawrence, Brian "Measuring Up," *Software Testing and Quality Engineering* vol. 2, no. 2 (2000)

[4] Kaner, C. "Rethinking Software Metrics," *Software Testing and Quality Engineering* vol. 2, no. 2 (2000)

[5] Kaner, C. "Yes, But What Are We Measuring?," 1999 PNSQC

[6] Kaner, C. "Measurement of the Extent of Testing," 2000 PNSQC

[7] ibid.

[8] ibid.

[9] Many models go so far as to ignore mathematical truths. Many times we categorize based on ordinal scales; Defect Severity, for example. We assign numbers to the categories and depict the order based on the values we chose. We know that a "Severity 1" isn't ½ as much as a "Severity 2," and we can't claim that all "Severity 3" defects are the same. We could just as well use colors and call the categories as Green, Yellow, Orange, and Red. Doing arithmetic with them (e.g., the Priority is Severity times Likelihood) is as absurd as multiplying colors.

[10] Weinberg, G. "How Good Is Your Process Measurement," *Software Testing and Quality Engineering* vol. 2, no. 1 (2000)

working and well intentioned. Although some of the behaviors cause by metrics may seem funny or even silly, there are potentially serious consequences to organizations because they use metrics. The specific observations I make here are based on real companies using software metrics in their product development. I have taken some care to change enough of the details that the innocent (or guilty) cannot be easily identified. In some instances, I have combined observations from multiple organizations. But, you wouldn't be alone if you think you recognize your organization in some of the situations. I've noticed that people often recognize their own experiences here.

## Three Metric Examples

I've selected three examples of metrics used to decide when a product is ready to release. There certainly are other examples and other metrics, but this has been a particularly ripe area of examples from my experience. The three metrics used to show a product's readiness for release are:

1. Defect find/fix rate
2. Percent of tests running/Percent of tests passing
3. Complex model based metrics (e.g., COCOMO)

Briefly, each of the metrics is used to describe an attribute of project status (how far along is the project, is it ready for release, are we meeting our milestones, etc.). These attributes were applied by management to monitor and adjust project plans and member behaviors in order to keep the project on schedule. I haven't a clue about the attributes' scales and don't think anyone else can, either. The variation in the attribute and the measures is all over the map – a few projects run like clockwork (or so I've heard), but most don't run as planned, and some I've worked with were just out of control. (Out of control is a term I use for software that has progressively much worse

**Kaner's Ten Measurement Factors**

1. The purpose of the measure. What the measurement will be used for.
2. The *scope of the measurement*. How broadly the measurement will be used.
3. The *attribute to be measured*. E.g., a product's readiness for release.
4. The appropriate *scale for the attribute*. Whether the attribute's mathematical properties are rational, interval, ordinal, nominal, or absolute.
5. The *natural variation of the attribute*. A model or equation describing the natural variation of the attribute. E.g., a model dealing with why a tester may find more defects on one day than on another.
6. The *instrument that measures the attribute*. E.g., a count of new defect reports.
7. The *scale of the instrument*. Whether the mathematical properties of measures taken with the instruments are rational, interval, ordinal, nominal, or absolute.
8. The *variation of measurements* made with this instrument. A model or equation describing the natural variation or amount of error in the instrument's measurements.
9. The *relationship between the attribute and the instrument*. A model or equation relating the attribute to the instrument.
10. The *probable side effects* of using this instrument to measure this attribute. E.g., changes in tester behaviors because they know the measurement is being made.

quality as developers try to patch it up, followed by project cancellation or quick turnover of most of the management and staff.)

I've neaver heard of any direct measure of project status, program readiness for release, or progress toward meeting milestones, etc. Instead, we've used surrogate measures and metrics; the instruments we used to measure are:

1) counters of new and resolved defect reports,
2) percents of tests running and passing, and
3) a "Mulligan's stew" of metrics (including cyclomatic complexity, defect counts, defect find/fix rates, defect severities, estimated size of programs, experience levels of developers, past projects' metrics, and others) combined and mixed thoroughly in an arithmetic equation (such as COCOMO).

## Defect find/fix rate

The first two metrics use pairs of measures to determine convergence on the planned project completion. The ratio of defects found to defects fixed intuitively feels like a reasonable way to see the end. When we find more than we fix (ratio greater than 1) during a specified time period, we are discovering problems faster than fixing them. When the ratio equals 1, we are not gaining ground or losing it in terms of fixing problems. When the ratio gets below 1, the developers are reducing the number of known problems. For the life of the project, the ratio of all defects found to all defects fixed should approach 1 as we fix all known problems. This model is based on several assumptions that don't hold:

1) all defects that are found are reported,
2) there is a goal of fixing (or resolving) all known defects,
3) when all known defects are fixed the product is ready to release,
4) there are reasonable resolutions for all fixed defects.

Defect counts don't naturally vary, given a consistent definition of defects. One has either been found and reported or it hasn't. The count of the number of reported defects can easily be done, and several people are likely to come up with the same counts given the same defect database. However, human nature makes hash of the numbers by accident and with purpose. Reducing the effort to hunt new defects or withholding reports will directly and immediately improve the ratios. Developers and testers can become extremely creative in recategorizing defects as enhancement requests, problems in other products, duplicates, unassigned, etc. in order to resolve them without fixing the underlying problems.

A few examples of observed behaviors of these sorts may serve to clarify:

- To reduce the number of defects, twenty-five reports against a subsystem were all marked as "duplicates" of one new defect. The new defect report referred to each of the twenty-five for a description of the problem (because the only thing the twenty-five had in common was that they were reported against the same subsystem).

- In an organization where defects didn't get counted before initial screening and assignment, a dozen defects that hadn't been resolved in more than four weeks were assigned to the developer "Unassigned," and thus were not counted.
- In one case the testers withheld defect reports to befriend developers who were under pressure to get the open defect count down. In another case the testers would record defects when the developers were ready with a fix to reduce the apparent time required to fix problems.
- A test group took heat for not having found the problems sooner (to give the developers more time to fix the problems).
- Developers only reported problems after they had been fixed (thus never making the ratio worse).
- I've seen defects fall in the crack, get lost, pushed in circles, or be forever deferred.

One general reaction to found/fixed metrics was the creation of "Pocket lists" of defects by developers and testers. Developers kept these unofficial lists of defects and action items to themselves. If they reported the defects, it created a negative perception about the code and they also needed to address the problems. One manager went so far as to publicly criticize individuals for having more than five defects against their modules. The project (with 40 modules) now *never* has more than 200 defects reported (and seldom fewer than that). The pocket list has been as benign as not reporting problems observed and fixed in code, and as blatant as knowing about existing problems others were likely to encounter.

There are also ways that management can make this situation much worse (such as a Dilbertian "bug bonus" for the number of bugs found or fixed), so the developer and tester are encouraged to report and fix large numbers of defects kept in pocket lists so they create the appearance of a flurry of last moment heroic activity. Indeed, it is quite difficult to take software measures without creating significant side effects.

## Percent of tests running/Percent of tests passing

The second case (percent of tests running and percent passing) also may intuitively feel like a reasonable way to see the end. If 100% of our tests run, and 100% pass, we're done, right? The percentage of tests running can be interpreted two ways; either the testers haven't had time to run all the tests, or the software isn't complete enough to fully test. Likewise, the percent of tests that pass may feel like a reasonable way to measure progress. The terms themselves are difficult to pin down, and no matter how they are defined and enforced, there are simple ways to manipulate the percentages. Also, this model of testing makes several false assumptions:

1) all tests are known before testing begins,
2) whatever constitutes "a test" is well understood and agreed upon,
3) whatever constitutes "running a test" is well understood and agreed upon,
4) test outcomes are clearly pass or fail, and
5) release occurs when all tests (or a specified percentage) pass.

Very few organizations I've worked with do only pre-defined tests. Most test groups mix regression tests with exploration and continue to create new tests until the product escapes. Unless we know the exact number of tests we will run, we don't know the denominator. And, the definitions of a test, running a test, and passing and failing are subject to debate and manipulation. If a test is used in several configurations, is it a separate "test" in each? If it's only counted as a test once, do failures in several (but not all) configurations count as one failure, several, or a fraction of one? Does a test fail twice if it detects more than one defect? Does a long exercise count as multiple tests? Do we only count those tests that are for completed features? Do we have to run tests we know will fail? Do we have to report defects against those failures?

Some examples of observed behaviors due to these metrics:

1) the redefinition of what a "test" is in order to increase the number to be counted and increase the percentage passing. (Each test is divided into sections because having one of ten (1000 line) tests that can't run looks much worse than three of two hundred (50 line) tests.)
2) top management declares victory and releases because "All four of the tests that could run were tried, and 100% passed. (The code just wasn't complete for the other 5,768 tests.)"
3) replacement of expected results with actual (bad) results because a problem was "known." (Development demanded that testing remove the test from the count of tests running and not passing for those defects that weren't going to be fixed. Management would not let the testers reduce the count of tests running, so they compromised…) [I call this "institutializing a defect" – making sure it stays in the product forever.]

## Complex model based metrics

The last situation uses complex multivariate mathematical models to describe the project. A "mathematical" model is applied to decide ahead of time how many defects there should be in modules. The models are also used to predict the rate of defect reporting, so the progress and readiness of a project can be discerned simply by counting the defects found to date. "According to the model, we should find 50% of the defects through our testing. Since there are 200 defects in the project (according to the Function Point computations), the project should be ready to release when we've found 100 defects." These predictions then become self-fulfilling prophecies, with sometimes crippling side effects.

For these measures, there isn't any real relationship between the measured or subjectively assigned attributes and the meaning assigned. There might be a relationship between the mathematical model and organizational behavior or project status, but I doubt it. The amazing thing I've observed is the near religious fervor that goes into defending the validity of the model based on the fact that on the last *N* projects, the equation has yielded a precise estimate of release on the day of release. (Never mind that it wasn't correct any of the 52 weeks previous to that, and the subjective values and equation fudge factors were changed every one of the past 52 times the equation was used.) This is what I call the "you always find your keys at the last place you look for them" effect. When they look back at the project, they conclude they now know the numbers for "experience factors," "product complexity," and all the other elements that plug into the

equation. They only really know that numbers can be chosen for the equation to show what they now know – its ready for release. The sad thing is the number of managers and engineers who don't realize that given any moderately complex equation with multiple variables, values can be selected to generate any particular result. For example, given the equation 5 * X + 3 * Y + Z, we can pick values of X, Y, and Z to yield any positive result.

Some examples of observed behaviors due to these metrics:

- Managers demanding sign off by testers without testing because the model showed release should occur in spite of testing not being complete. ("We've followed the curve precisely for eight months – obviously it's ready for release.")
- Punishment of testers for not finding enough defects quickly enough through lowering their rating (and thus their pay).
- Reporting of minor problems, variations on a defect, and seriously questionable tests, to increase defect counts.
- Testers not reporting defects (or reporting trivial problems) in order to keep the defect counts corresponding with the predicted values from the model. (Because management had such faith in the models, people were expected to perform exactly as predicted. Having too many defects was interpreted as meaning the project was poor quality and too few was interpreted as the tester doing a poor job.) [I've been given the argument that "the exactly predicted number of defects was reported every week on an 18 month project." This is a statistical impossibility for a real world process. W. Edwards Demming described the real world effect as being due to normal random variation. My chemistry instructors called too perfect data in an experiment "dry-labbing," documenting the predicted results instead of taking accurate measurements (even subconsciously).

## Other Side Effects

In response to noticing some of the above side effect behaviors, the rules can change. In some situations, defect fixes cannot be deferred to future releases because opening a project with defects already reported skews the statistics (and anyway, the argument goes, the defects weren't put in on this project; they were put in on previous projects). But, since products cannot be released with known defects and the defects cannot be deferred, they are resolved ('not a defect' or 'no fix') in the current project and may be reentered if someone remembers them during the next project.

Deferral was used as a technique to reduce the number of defects, so management mandated justification in person for all deferred defects. Consequently, the number was reduced through consolidation of defects (25 marked as duplicates of a new one that references each of the 25). Then all duplicates required management review. Defects were then resolved en mass as "no fix intended." The rules changed then to force management review of all defect reclassification. Then defects became stacked into "submitted/need assignment" to keep them from getting into the statistics until resolution was ready.

## Conclusions

Software metrics have been successfully employed for decades to understand, monitor, and improve products and processes. There are volumes of literature describing successes and methods, and organizations regularly implement metrics programs. In software development and quality assurance there is almost blind acceptance of the value of such programs, even though in many of these same organizations the metrics program is secretly causing lower productivity and quality.

It is imperative for any organization interested in quality to be alert and careful about metrics. Even organizations that have well established programs, especially organizations with long established metrics programs, ought to consider whether the metrics have the desired meanings and identify what side effects are caused. Where efforts are diverted without improving the product or its quality, some questioning should be made as to the appropriateness of the measures and metrics. The unintended side effects may be slowing rather than streamlining the organization, and can even serve to obscure our understanding of test results and reduce the overall product quality.

Cem Kaner has provided a framework for understanding and rethinking software metrics, but observations of behaviors within organizations is often sufficient to recognize unintended side effects. By reassessing the meanings of our metrics and recognizing their limitations we can potentially reduce the negative impacts.

# A UNIVERSAL METRICS REPOSITORY[§]

Warren Harrison
warren@cs.pdx.edu
Department of Computer Science
Portland State University
Portland, OR 97207-0751
http://www.cs.pdx.edu/~warren

## ABSTRACT

A neglected aspect of software measurement programs is what will be done with the metrics once they are collected. As a consequence, databases of metrics information tend to be developed as an afterthought, with little, if any concessions to future data needs, or long-term, sustaining metrics collection efforts. A metric repository should facilitate an on-going metrics collection effort, as well as serving as the "corporate memory" of past projects, their histories and experiences. Within this context, four important limitations of contemporary metrics repositories are: obsolescence; ambiguity; augmentation (or lack, thereof); and focus. In order to addresses these issues, we have suggested a transformational view of software development which treats the software development process as a series of artifact transformations. Each transformation has inputs (artifacts) and produces outputs (artifacts). The use of this approach supports a very flexible software engineering metrics repository.

**Key Words/Phrases**: Software Metrics, Software Engineering Repositories, Software Engineering Data Collection

Warren Harrison is a Professor of Computer Science at Portland State University and Adjunct Associate Professor of Medical Informatics and Outcomes Research at Oregon Health Sciences University. His software engineering research includes models of return on investment for process improvements, software quality assurance, software measurement, formalized decision-making and empirical studies of software engineering. He is currently Editor-in-Chief of the *Software Quality Journal* and co-Editor-in-Chief with Vic Basili of the *Empirical Software Engineering Journal*. Warren received his B.S. in Accounting from the University of Nevada – Reno, his M.S. in Computer Science from the University of Missouri – Rolla, and his Ph.D. in Computer Science from Oregon State University.

# A UNIVERSAL METRICS REPOSITORY[§]

Warren Harrison
Department of Computer Science
Portland State University

## Introduction

When metrics are collected pertaining to a software product or process, the measurements are usually stored for later retrieval. Such a collection of metrics data is known as a *metrics repository*.

The importance of repositories has been recognized in the past. A recommendation from the Workshop on Executive Software Issues held in 1988 (Martin, et al [1989]) stated that "*Software organizations should promptly implement programs to: Define, collect, store in databases, analyze, and use process data*". Twenty years ago Basili [1980] wrote "*All the data collected on the project should be stored in a computerized data base. Data analysis routines can be written to collect derived data from the raw data in the data base. The data collection process is clearly iterative. The more we learn, the more we know about what other data we need and how better to collect it."* Even then he clearly envisioned a data collection process that would change over time. As users learn more about the products, processes and metrics, we should expect the information they desire to change.

The significance of a persistent collection of software engineering data becomes more obvious when we note that it represents the only objective source of an organizational memory. When did project X finish? How many programmers worked on it? How reliable was it?

Unfortunately, a neglected aspect of software measurement programs is what will be done with the metrics once they are collected. As a consequence, databases of metrics information – we hesitate to call some of these efforts *metric repositories* – tend to be developed as an afterthought, with little, if any concessions to future data needs, or long-term, sustaining metrics collection efforts. For instance, in what was described as a study in "the process and methods used, experience gained, and some lessons learned in establishing a software measurement program" the associated metrics database was simply a collection of ten separate spreadsheet files [Rozum 1993]. We feel that this state of affairs has a significant affect on the viability of measurement programs and software engineering research in general.

Researchers in other fields depend upon the existence of major archived collections of empirical data. For instance, in medicine, health insurance databases are a rich source of information about the characteristics of the population, the frequency of diseases and in some cases, the effectiveness of various treatments. Economists utilize archived stock market databases to test hypothesis and search for insights. However, no single

significant source of data similar to a major insurance company's database or a national stock market exists for software engineering research. Instead, we must piece together data from a variety of sources in order to obtain generalizable results. Unfortunately, due to the lack of attention given to the issue of flexible metrics repositories, this task appears insurmountable. Few results exist today that are based on a single comprehensive analysis of multiple data sets. At best, we may analyze one small data set and then confirm the results using additional small datasets. The Software Reuse Metrics Working Group at REUSE '97 (Baldo et al[1997]) pointed out the lack of industry-wide repositories and the difficulty of normalizing data across domains, projects, and organizations.

This document describes a prototype repository that addresses these issues and proposes a plan to implement, evaluate and populate such a mechanism.

**Past Efforts at Metrics Repositories**

Over the past two decades, there has been some interest within the community to make software engineering data available. In this section, we consider some illustrative examples. While we do not claim these examples represent every metrics dataset, we feel that they are notable in that they were ostensibly designed to *foster* distribution and sharing of data among a variety of unrelated organizations. We can only imagine that most corporate metrics repositories are even less accessible than these examples.

*The DACS Productivity Dataset*

One of the earliest examples of a publicly available "metrics repository" is one that was (and continues to be) disseminated by the DACS (Data and Analysis Center for Software) at Rome Airforce Base. The repository is known as the *DACS Productivity Dataset* (http://www.dacs.com/databases/sled/prod.shtml) and consists of summary data on over 500 software projects dating from the early 1960s through the early 1980s. The DACS Productivity Dataset (DPDS) was intended to support research in cost modeling and estimation, and as a consequence includes most of the parameters that were in vogue within the estimation community at the tine the repository was created. The parameters maintained by the DACS Productivity Dataset include:

- *Project Size.* Number of delivered source lines of code (DSLOC) in the delivered project.
- *Project Effort.* Effort in man months required to produce the software product.
- *Project Duration.* Duration of project in total months derived from begin and end dates of projects, less any "dead time" in the project.
- *Source Language.* Programming languages used on the project recorded by name and expressed as a percentage of the total DSLOC written in each different language.
- *Errors.* The number of formally recorded Software Problem Reports (SPRs) for which a fix has been generated during the period covered by the project.

- *Documentation.* Delivered pages of documentation including program listings, flow charts (low and high level), operating procedures, maintenance procedures, etc.
- *Implementation Technique.* The implementation techniques used on the software project expressed as a percentage of the DSLOC built using specific techniques of Structured Coding, Top Down Design and Programming, Chief Programmer Teams, Code Reviews or Inspections, and Librarian or Program Support Library.
- *Productivity.* Ratio consisting of DSLOC to Total man-months
- *Average Number of Personnel.* Ratio consisting of Total Manmonths to Total Months
- *Error Rate.* Ratio consisting of Errors to DSLOC

The repository consists of fixed columns containing each of these pieces of information on every project for which information was captured. This rigid, inflexible format has serious ramifications on later software engineering data collection.

First, by mandating specific measures, it codifies metrics that may or may not be the ones we really want to collect. Boehm, Brown and Lipow [1976] suggest that "… *the software field is still evolving too rapidly to establish metrics in some areas. In fact, doing so would tend to reinforce current practice, which may not be good*." This is just as true in 2000 as it was in 1976. Secondly, some metrics may be perfectly fine for past projects, but obsolete for more recent ones. Development techniques such as Chief Programmer Teams and Structured Coding are unknown to many contemporary software engineers. Likewise, for some very popular languages such as Visual BASIC, the concept of "Delivered Source Instructions" may be inapplicable.

For purposes of illustration, assume an organization collects Logical Sources Statements ala' the *IEEE Standard for Software Productivity Metrics* [IEEE 92] rather than delivered source lines of code as is done in the DPDS. The size parameters between the earlier and later data are no longer comparable. Does this mean that we create yet another repository to maintain the post-DSLOC data? If so, it makes it difficult to easily combine data that does continue to be comparable from the two repositories – for instance, average number of personnel or average defects for a project. This difficulty is compounded when one operates within a culture that promotes a "metric of the week" mentality. Existing metric repositories can fall by the wayside erasing all previous objective organizational knowledge. Conversely, an organization may resist modifying their metrics collection efforts, even when they should because it will result in the loss of a significant investment of time, effort and organizational knowledge.

*The Software Reliability Dataset*

The DACS also maintains a dataset of failure data from 16 projects compiled by John Musa in the 1970s (http://www.dacs.com/databases/sled/swrel.shtml). The dataset consists of failure interval data to assist software managers in monitoring test status, predicting schedules and assisting software researchers in validating software reliability models. It represents projects from a variety of applications including real time command and control, word processing, commercial, and military applications.

- *Failure Number.* A sequential number identifying a particular failure.
- *Failure Interval.* The time elapsed from the previous failure to the current failure.
- *Day of Failure.* The day on which the failure occurred in terms of the number of working days from the start of the current phase or data collection period.

As with the Productivity Dataset, the focus of the Software Reliability on a specific application, results in very specific (and era-specific) data being maintained. As a result, the applicability of the data is limited.

The dataset in its current form is not compatible with most defect datasets, even though the information itself on these 16 projects may be of interest, and comparable to other datasets.

*The Architecture Research Facility (ARF) Dataset*

This dataset was produced by Weiss (1979) in the late 1970s. It contains information on a 192 man-week software project developed at the Naval Research Laboratory and the 142 errors that were discovered in the code. As are the other datasets we discuss, the ARF dataset is available from the DACS (http://www.dacs.com/about/services/pdf/Data-Brochure.pdf). This represents an improvement in terms of schema sophistication over the Productivity and Reliability Datasets in that the data is separated into two tables, one containing component information and one containing information on software problem reports.

- *Software Components*
  - Component Code
  - Total Statements in Segment
  - Number of Comments in Segment
  - Number of Pre-Process Statements in Segment
  - Subjective Complexity (Easy, Moderate, Hard)
  - Function (Computational, Control, Data Accessing, Error Handling, Initialization)

- *Software Problem Reports*
  - Form Number
  - Project Code
  - Programmer Code
  - Form Date
  - Number of Components Changed
  - Number of Components Examined
  - More than One Component Affected
  - Date Change was Determined
  - Date Change was Started
  - Effort for Change (Less than 1 hour, 1 hour to 1 day, 1 day to 3 days, more than 3 days, unknown)

- Type of Change (Error Correction, Planned Enhancement, Implement Requirements Change, Improve Clarity, Improve User Service, Develop Utility, Optimization, Adapt to Environment, Other)
- Code of Changed Component
- Type of Error (Requirements Incorrect, Functional Specs Incorrect, Design Error, Several, Misunderstand External Environment, Error in Language Use, Clerical Error, Other)
- Phase When Error Entered System (Requirements, Functional Specs, Design, Code and Test, System Test, Unknown)
- Data Structure Error (Y/N)
- Control Logic Error Flag (Y/N)
- Error Isolation Activities (Pre-acceptance Test, Acceptance Test, Post Acceptance Test, Inspection of Output, Code Reading by Programmer, Code Reading by Another, Talk with other Programmers, Special Debug Code, System Error Message, Project Specific Error Message, Reading Documentation, Trace, Dump, Cross Reference, Proof Technique, Other)
- Time To Isolate Error (Less than 1 Hour, 1 Hour to 1 Day, More than 1 Day, Never Found)
- Work Around Used (Y/N)
- Related to Previous Change (Y/N)
- Previous Form Number
- Previous Form Date

Because of the increased schema sophistication, we can retrieve a much richer set of information than with the earlier “flat file” datasets. However, the dataset still suffers from a lack of generality. Clearly Weiss was interested in collecting information about errors. As such, much potentially useful information is missing (length of time to develop, hours of design, etc.). This raises serious doubts as to the ease of combining the information with other datasets. However, we would argue that under some circumstances, this is exactly what an analyst may wish to do.

For the sake of argument, assume that the definition of a defect in the Productivity Dataset is comparable to a "problem" in the ARF Dataset and the definition of DSLOC in the Productivity Dataset is comparable to the definition of a statement in the ARF Dataset. Given this assumption, we may find that it would be desirable to roll the component data from the ARF dataset up to the level of a "Project" as represented by the Productivity Dataset, so we could fully utilize all our data to compute statistics about defects per DSLOC for a project. However, this would require the integration of at least two databases If the ARF dataset actually contained data on multiple projects, we would find this to be a non-trivial task - in fact, we might very well limit our analysis to only one dataset or the other, thereby ignoring a significant information asset, and making decisions with less than all the information available to us. In order to be able to do this easily, not only must the definitions be compatible, but also the schema.

*The NASA/SEL Dataset*

The DACS also maintains a more general dataset, the NASA/SEL Dataset (http://www.dacs.com/databases/sled/sel.shtml), which contains information collected since 1976 at the NASA Goddard Software Engineering Laboratory [Basili, 78], (also see http://sel.gsfc.nasa.gov/). One of the major functions of the SEL is the collection of detailed software engineering data, describing all facets of the development process, and the archival of this data for future use. The SEL maintains a tremendous amount of such information (the dataset is distributed on a CD-ROM containing 424 MB of data and consisting of over 100 tables)

This dataset is a good example of a more flexible repository. Rather than being developed to fulfill a specific function such as reliability estimation or cost estimation, the SEL Dataset is meant to provide a much more flexible source of data for use by researchers. While the size of the schema (the schema requires a number of pages to fully describe the tables and their relationships) prohibits a complete listing of factors here, the dataset contains the following information:

- General project summary (completed at each milestone)
- Component summary (completed during the design phase)
- Programmer-Analyst Survey (completed once by each programmer)
- Resource summary (completed weekly)
- Component status report (completed weekly)
- Computer program run analysis (completed each time the program is run)
- Change report (completed for each change made)

Unlike the other datasets, the NASA/SEL dataset enjoys on-going maintenance, last being updated (at DACS) in 1997.

The NASA/SEL schema is much more flexible than the earlier attempts. By virtue of the organization of the data as a collection of tables "connected" via foreign keys, the dataset can be augmented by future data that may contain additional characteristics. However, it still suffers from some limitations as described below.

**General Limitations of Contemporary Metrics Repositories**

If the only purpose of a metrics repository is as a place to maintain a "tool dump" of a single project or set of completed projects, then the limitations of most repositories are minor. However, if the repository is expected to serve as a home for ongoing collection efforts, then most contemporary metrics repositories are seriously deficient. In our discussions we assume not only that the metric repository is intended to support an ongoing metrics collection effort, but we additionally consider the metrics repository as the "corporate memory" of past projects, their histories and experiences. Within this context, four important limitations of contemporary metrics repositories are:

1. **Obsolescence** – In most examples of contemporary metrics repositories we have surveyed, specific metrics are "built-into" the schema. For instance, consider the

measure of “Delivered Source Lines”, or “Cyclomatic Complexity”. What happens when a specific metric goes out of vogue? Do we continue to collect the obsolete metrics in order to remain compatible with the schema? If new metrics become popular, do we avoid collecting them because “there’s no place to put them?” Or do we simply chuck what we have and start over?

2. **Ambiguity** - It may often be the case that users of a particular repository do not know what a specific field within the repository means unless they were involved in the original collection efforts. Florac [1992] points out the importance of *Communicatio*n (will others know precisely what has been measured and what has been included and excluded?) and *Repeatabilit*y (would someone else be able to repeat the measurement and get the same results?). Layout documentation of most repositories is limited (at best) to often out-of-date documentation files. When a user sees the word "Design" in an error report, does that mean the error was injected, found or fixed during the design phase? Similar issues exist for almost every other property of a project we may wish to retain - for instance Goethert, et al [1992] propose a definition for counting staff hours, Park [1992] suggests a method for counting source statements.
3. **Augmentation** – It is expected that as we gain experience with the use of metrics, we’ll wish to augment the information set they provide us. A manager who becomes used to obtaining defect information might then desire information on design events or maintenance effort. If the repository does not embrace such augmentation of data, we’ll find groups of additional repositories springing up to meet the information needs of various stakeholders. The danger in this is that multiple repositories are terribly difficult to keep consistent and multiple repositories represent significant inefficiencies in effort to both capture and utilize the information. Additionally, the difficulty of mining information from diverse repositories makes it less likely that the entire store of organizational knowledge will be accessed.
4. **Focus** – Most repositories are historical. That is, they reflect occurrences and properties from past projects. In addition, because they focus on what has happened as opposed to what is currently happening, they tend to be product-centric. As such, they are quite good at reflecting information about products, but not so good at reflecting information about processes or events. While product information is important, much decision-making deals with process and event information more than product information.

The current state of repository technology leads to an inflexible, product-centric view of software engineering decision-making. In the following section, we describe a method of viewing the software development process that integrates product and process data in a natural, flexible manner.

The goal of our work is to define, implement, evaluate and populate a "universal" metrics repository that will address these issues.

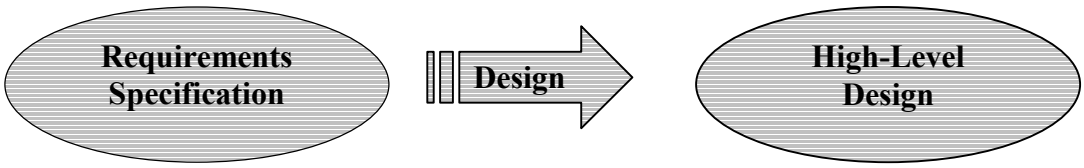
## A Model of Development to Support Data Collection

In order to address the issue, the repository design must view a software project as a dynamic, growing collection of artifacts as opposed to a static, monolithic bucket of code. Unfortunately, the current view of metrics repositories is exactly that. The general view

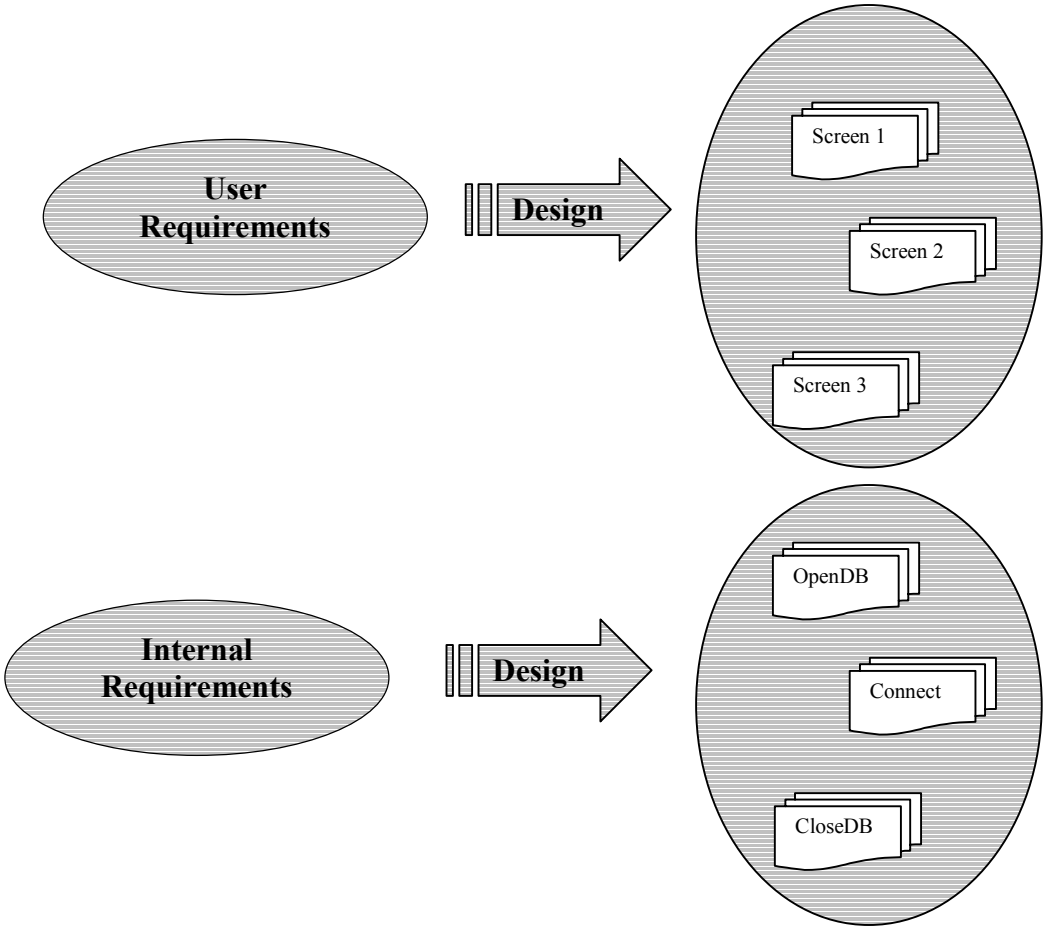
considers the software development process as simply a collection of intermediate products. For instance, we might characterize a software project as consisting of a requirements specification, a design, and finally a collection of code models.

*A Transformational View of Software Development*

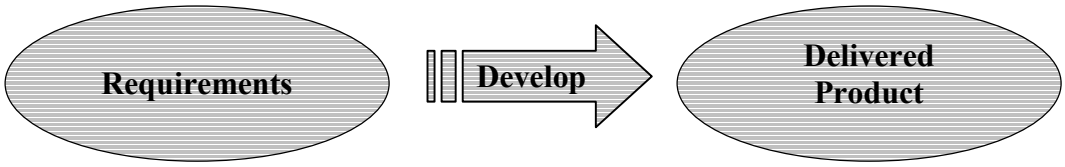
The transformational paradigm of software development views the software development process as a series of transformations of artifacts. An artifact is some identifiable product of an activity that takes place during the software development process. For example, the needs analysis phase of a software development project produces an artifact we sometime refer to as a Requirements Specification. In turn, these artifacts are used as inputs to other transformations to create new artifacts. That is, an activity transforms one or more artifacts into a new kind of artifact or collection of artifacts. Each transformation has inputs (artifacts) and produces outputs (artifacts). For example, the process we usually refer to as "design" transforms a requirements specification into a design document:




The transformations can be described at arbitrary levels of abstraction. For instance, the requirements could actually be logically (and perhaps physically) separated into two pieces – say the user interaction and internal processing segments. This implies that the "Design" transformation actually creates two artifacts: the user interaction and internal processing artifacts. This situation can be described in the following manner:

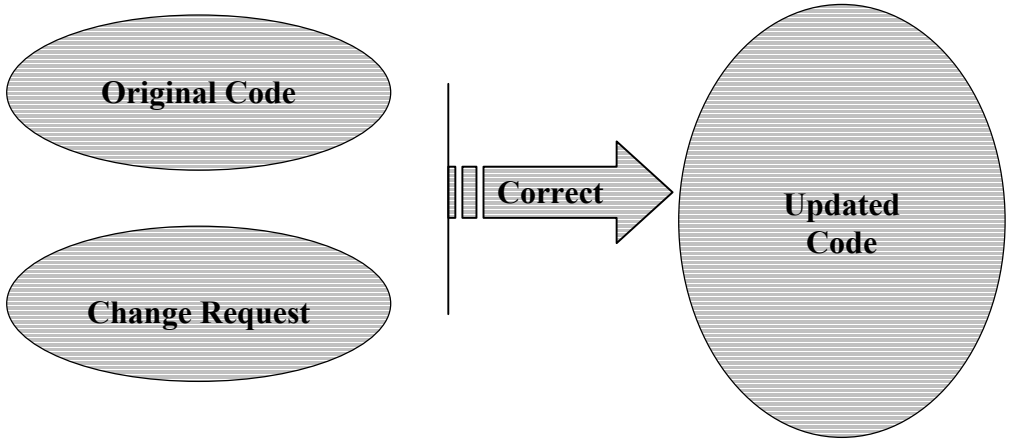


Of course, the paradigm supports any level of abstraction desired. So for instance, the requirements artifacts could be paragraphs, pages, chapters, documents, or the entire set of requirements depending upon the needs of the user. To illustrate an even more abstract representation, consider the entire software development process that transforms a requirements specification into a delivered product:




*The Transformational Approach to Iterative Software Development*

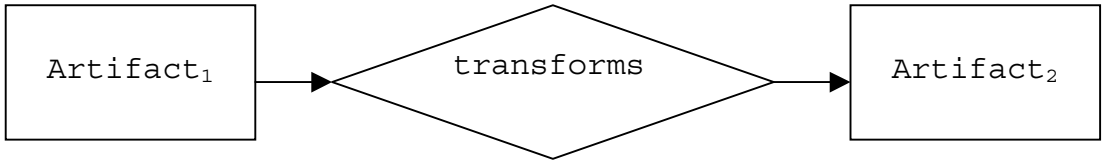
A major weakness of the traditional "intermediate product" view of repositories is its inability to accommodate iterative development. When a product is returned due to a change request, it may be awkward to represent the change in a new version of the product. For instance, should an error be found during testing which results in a code change, either the revised code unit will not be represented in the repository, or the changed code unit will replace the original design product. On the other hand, the transformational view accommodates this situation quite naturally: In this example, the original code artifact and a specific change request are transformed into an "updated code artifact".




Lest concerns about storage space be raised, what is being maintained is a record of artifact characteristics, not the artifact itself.

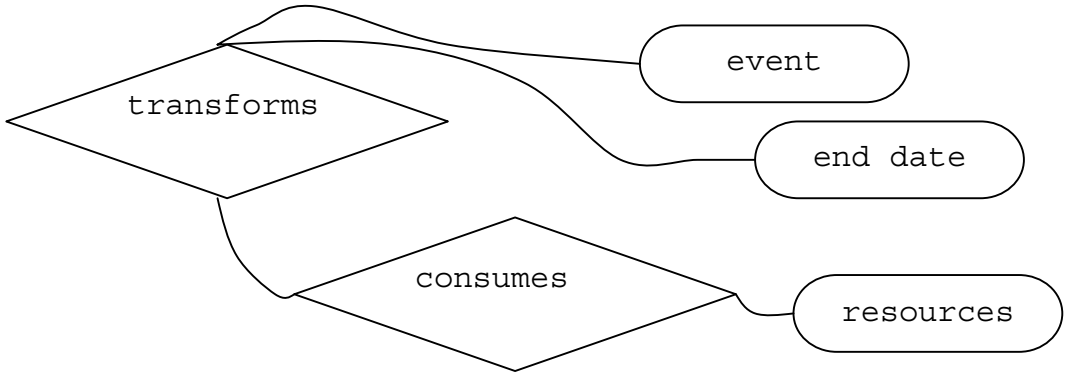
**Schema Issues**

The transformational approach to software development gives rise to the organizational aspect of our proposed repository, which consists of `Artifact` entities connected via `transforms` relationships:



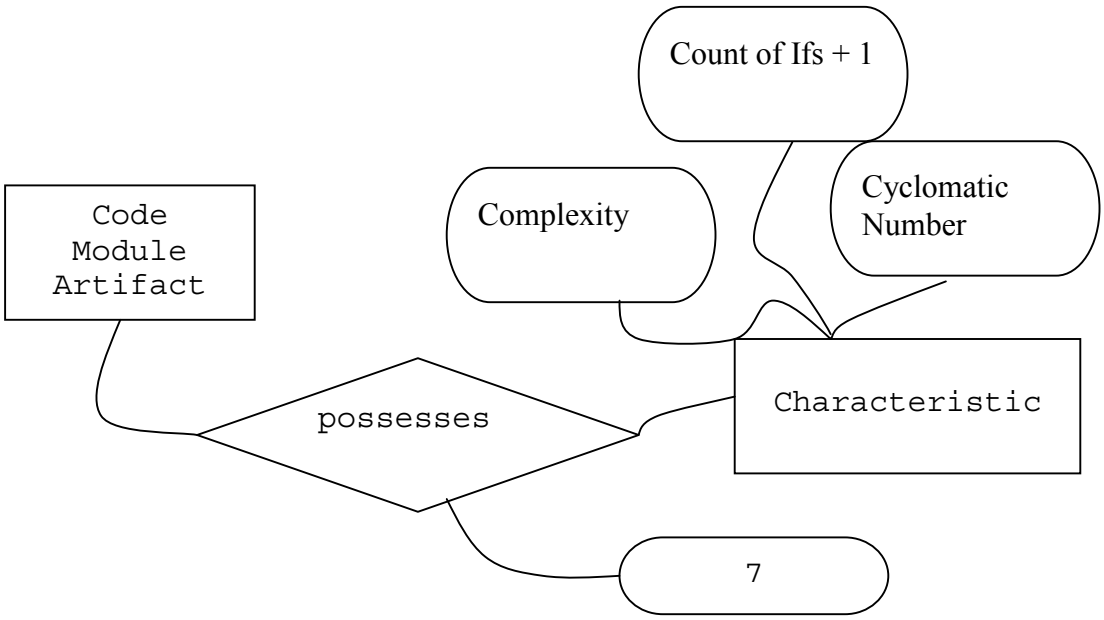

This provides the flexibility to represent an arbitrary flow of artifacts through a software development process, regardless of process model or level of granularity.

In order to maintain appropriate information about the transformation, each must be annotated with descriptive items. A transformation is associated with an `event` that denotes the sort of activity or event that took place to effect the transformation, a count of resources consumed during the transformation, and a completion date. This gives rise to the second aspect of our proposed repository, which annotates `event` and `end_date` of each transformation and associates it with resources consumed:




Each artifact also is associated with certain specific information. However, rather than enforcing a standard selection of characteristics (which is what most previous repositories have chosen to do) the third aspect of the proposed repository is that artifacts are linked via relationships to as many or as few characteristic entities as data permits.

For instance, an artifact such as a code module may be data rich or poor. It may be linked to dozens of characteristics, or it may be linked to a single one. Each characteristic includes a `name`, `type` and `description` and the `possesses` relationship is annotated to reflect the quantity of the characteristic. This provides maximum flexibility by avoiding mandating specific metrics for a given artifact. For example:




Thus, an artifact characteristic has a property describing the "domain" of interest, such as *complexity*, *size*, *application area*, etc. In addition, each characteristic has a "working name", such as *Cyclomatic Complexity*, *Software Science Effort*, *Lines of Code*, etc. Because the characteristics that can be associated with an artifact may include measures made over many different time periods, by many different people for many different reasons, the schema includes the `descriptions` property. This meta-data allows each data item to be defined in order to avoid inappropriate combining of data (or discover opportunities to appropriately combine data which have different names) simply because the characteristics have similar names (e.g., "Lines of Code" could mean a variety of different measures which should not be combined).

**A Prototype Implementation**

We have implemented a proof-of-concept prototype using MySQL. The project homepage can be found at `http://www.cs.pdx.edu/~reposit/`. This site describes the schema and provides limited access to the prototype implementation. The Prototype Repository consists of the following tables or relations (currently we assume the repository consists of metrics for a single project, thus we omit a Project entity, future versions of the repository will support multiple projects):

Entities:

- Artifact (**id**, name)
- Event (**id**, name)
- Resource (**id**, name)
- Characteristic (**id**, name, type, description)

Relationships (all are *n*-to-*n*, and the entire tuple comprises the key):

- transforms (Tid, Artifact.id, Artifact.id, Event.id, EndDate)
- possesses ({Artifact,Resource,Defect,Event}.id,Characteristic.id, qty)
- consumes (transforms.id, Resource.id, quantity)

Each entity instance is assigned a unique unsigned integer identifier (id). This is done by assigning consecutive, unsigned integers to each entity instance as it is added to the database. In addition, each Transforms relationship is also assigned a unique id from this pool of unsigned integers.

The intention is to capture the transformation of one or more artifacts (say a design specification) to another artifact (say a piece of code), and the resources necessary to perform the transformation.

*Artifacts* represent the specific "trackable" items that make up a project – code units (files, modules, packages, functions – the level of granularity is aribtrary), specification documents, design documents, test cases, etc. *Events* represent types of activities that occur, such as an inspection, a modification, an error correction, etc. *Resources* represent the types of things valued by the project that are consumed in the process of creating the artifacts – effort, calendar time, computer time, etc. *Characteristics* represent types of information about the entities – size, color, weight, etc., as appropriate and available.

The relationships connect various entity instances to other entity instances. The most significant relationship is the *TransformsTo* relationship. *TransformsTo* represents the transformation of one Artifact into another Artifact because of an Event that occurs. We are also interested in the *Resources* that are consumed by a particular transformation, so the *Consumes* relationship associates a particular transformation with a particular resource type as well as the quantity of this type of resource that was consumed during the transformation. Every Artifact, Event, and Resource *Possess* a certain set of Characteristics.

A partial example follows:

```
Artifact(001,A1)
Artifact(002,A2)
Artifact(005,A3)
Artifact(013,A4)
Event(003,Bug Fix)
Resource(004,Programmer Effort in Hours)
Characteristic(007,Cyclomatic Number,Complexity,V(G) per XYZ metric
tool)
Characteristic(008,LOC,Size,Number of non-blank lines in module)
Characteristic(009,Pages,Size,Number of non-blank pages in document)
Characteristic(010,C Code,Type,File consisting of C source code)
Characteristic(011,Requirements,Type,Document consisting of features
the product must implement)
Characteristic(015,Bug Report,Type,Record of a bug)
Characteristic(016,Bug Description,Info,Missing Reply Feature)
TransformsTo(012,002,005,003,11-01-99)
```

```
TransformsTo(012,013,005,003,11-01-99)
Possesses(001,011,0)
Possesses(002,010,0)
Possesses(005,010,0)
Possesses(002,008,49)
Possesses(005,008,59)
Possesses(001,009,29)
Possesses(013,015,0)
Possesses(013,016,0)
Consumes(012,004,10)
Recorded(002,006)
```

This example represents the following facts (among others).

1. Artifact 1 is a 29 page requirements document
2. Artifact 2 is a 49 line C file
3. Artifact 5 is a 59 line C file that resulted from a modification of Artifact 2 in response to a Bug Fix to correct a "Missing Reply Feature", which took 10 hours of programmer effort
4. Artifact 13 is a bug report

**Summary and Next Steps**

We currently have a tentative schema and working prototype for a Universal Metrics Repository. Our next effort will be to expand the repository by populating it with data from several other sources. This exercise will address two interesting questions:

> (a) can a single repository schema in fact represent data from a variety of heterogeneous schemas and
> (b) can the schema successfully integrate data from the various sources to seamlessly base answers to queries across multiple sources.

To this end, we are currently working at populating the repository with data from other sources, including other DACS datasets as well as data from industrial repositories.

**References**


[Basili 1980] Basili, Victor R., “Data collection, Validation, and Analysis”, *Tutorial on Models and Metrics for Software Management and Engineering*, IEEE Computer Society Press, 1980).

[Baldo 1997] Baldo J., Butcher, D., Nada, N, Poulin, J., Quinones, Y., Trump, D. Scoy, F. and Wu, Z., "Software Reuse Metrics Working Group Summary", *Proceedings of Reuse '97*, 1997.

[Boehm, Brown and Lipow 1976] Boehm, B. W., Brown, J.R., and Lipow, M., “Quantitative Evaluation of Software Quality”, *Proceedings, Second International conference on Software Engineering,* 1976, pp. 592-605.

[DACS 1990] *DACS Productivity Dataset* (http://www.dacs.com/databases/sled/prod.shtml)

[Florac 1992] Florac, W., "Software Quality Measurement: A Framework for Counting Problems and Defects", SEI Technical Report CMU/SEI-92-TR-022 , 1992.

[Goethert, etal 1992] Goethert, W., Elizabeth, K., Bailey, E, and Busby, M., "Software Effort & Schedule Measurement: A Framework for Counting Staff-hours and Reporting Schedule Information", SEI Technical Report CMU/SEI-92-TR-021, 1992.

[IEEE 1992] *IEEE Standard for Software Productivity Metrics,* IEEE Std 1045-1992.

[Martin 1989] Martin, R., Carey, S., Coticchia, M., Fowler, P. and Maher, J., "Proceedings of the Workshop on Executive Software Issues August 2-3 and November 18, 1988", SET Technical Report CMU/SEI-89-TR-006, 1989,

[Park 1992] Park, R., "Software Size Measurement: A Framework for Counting Source Statements", SEI Technical Report CMU/SEI-92-TR-020 ADA258304 , 1992.

[Rozum 1993] Rozum, J. , "The SEI and NAWC: Working Together to Establish a Software Measurement Program", SEI Technical Report: CMU/SEI-93-TR-07, December 1993.

[Van Verth 1992] Van Verth, P., "A Concept Study for a National Software Engineering Database", SEI Technical Report CMU/SEI-92-TR-023, 1993.

# Testing Web-Applications—
# Effective Techniques for Analyzing and Reproducing Errors

Hung Q. Nguyen

This talk focuses on the characteristics of Web-Application errors to derive key issues to consider in the error analyzing and reproducing process. It helps you isolate application errors from configuration and technical support issues. Effective techniques to make errors reproducible are shared. You will also learn from examples of other common Web-Application error types beyond broken links that your testing might miss.

**Biography**

Hung Q. Nguyen is the president and CEO of LogiGear Corporation, a Silicon Valley company he founded, whose mission is to help software development organizations deliver the highest quality products possible while juggling limited resources and schedule constraints. Today, LogiGear has expanded its reach to dot-com companies in a myriad of industries. Specializing in testing of e-applications including e-commerce sites, portals, database-access applications and web sites, *Logi***Gear** also has extensive testing expertise in e-mobile products such as wireless communication products and hand-held devices, as well as traditional end-user applications such as productivity and edutainment software titles. LogiGear also offers a comprehensive "Practical Software Testing Training Series" and TRACKGEAR™, a Web-based defect tracking solution. In the past two decades, Hung has held management positions in engineering, quality assurance, testing, product development, and information technology. Hung is the author of Testing Applications on the Web (Wiley, 2000) and co-author of Testing Computer Software, 2nd edition (VNR, 1993/Wiley, 1999). He holds a Bachelor of Science in Quality Assurance from Cogswell Polytechnical College. He is an ASQ-Certified Quality Engineer and an active senior member of the American Society for Quality (ASQ).

# Testing Web Applications

## Effective Techniques for Analyzing and Reproducing Errors

## Eighteenth Annual PNSQC

**Hung Q. Nguyen**
**LogiGear™ Corporation**
hungn@logigear.com
www.logigear.com

# The Themes

- The notion of a bug—Is it a failure or an error?
- Web error examples
- Considerations in failure analysis
- The checklist for failure hypothesis

# The Notion of a Bug

- **The distinction between errors, conditions, and failures.**
  - **An *error* is a design flaw, an implementation mistake or a deviation from a desired or intended state.**
  - **An error won't yield a failure without the *conditions* that trigger it. Example, if the program yields 2+2=5 on the tenth time you use it, you won't see the error before or after the tenth use.**
  - **The *failure* is the program's actual incorrect or missing behavior under the error-triggering conditions.**

# The Notion of a Bug

## Tester's Objectives:

1. Finding failures in the product
2. Reporting failures including how to reproduce them
   - Steps: The exact sequence of activities that leads to a failure
   - CONDITIONS: The operating environment variables required to expose a failure
3. Focusing on the quality risk from the customer's perspective

## Developer's Objectives:

1. Locating errors in the code
   - Which line of code contains the error?
   - Under which CONDITIONS does the error cause failures?
2. Fixing errors
3. Focusing on the quality risk from the technical perspective

# Why Analyze a Bug?

- **Analyze bugs in order to**
  - **Make your communication effective**
    - **make sure you are reporting what you think you are reporting**
    - **make sure that the triggering conditions required for exposing a failure is thoroughly investigated and recorded**
    - **make sure that questionable side issues are thoroughly investigated**
  - **Support the making of business decisions**
  - **Avoid wasting the time of the programming and management staff**
  - **Find more bugs**

# Condition Dependencies

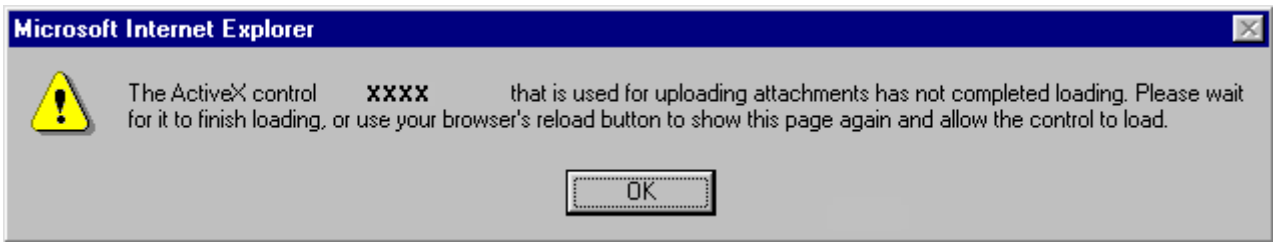
- **Application-specific conditions**—The application only exhibits failures in certain conditions. For example, cut and paste work correctly in normal mode but not in zoom-in mode.
- **Environment-specific conditions**
  - **Static operating environment** (i.e., configuration and compatibility-related errors) in which incompatibility issues may exist regardless of dynamic conditions such as processing speed and available memory.
  - **Dynamic operating environment** (i.e., resource and time-related errors) in which otherwise functional components may exhibit failures due to memory-related errors and latency conditions.

# The Notion of a Bug

| | One Failure | Multiple Failures | No Failure or Non-Reproducible / Hard-to-Reproduce Failures |
|---|---|---|---|
| **One Error** | Simple and isolated condition | Common code or module | **Missing operating environment variables** |
| **Multiple Errors** | Dataflow, combinations of interactivity, or sequences of state-transition | Various combinations | |
| **No Error** | Affected by uncontrollable external environment (e.g., DLL conflict, SQL is not running, read-only volume, DNS problem, etc.) | Common code, module or condition | |

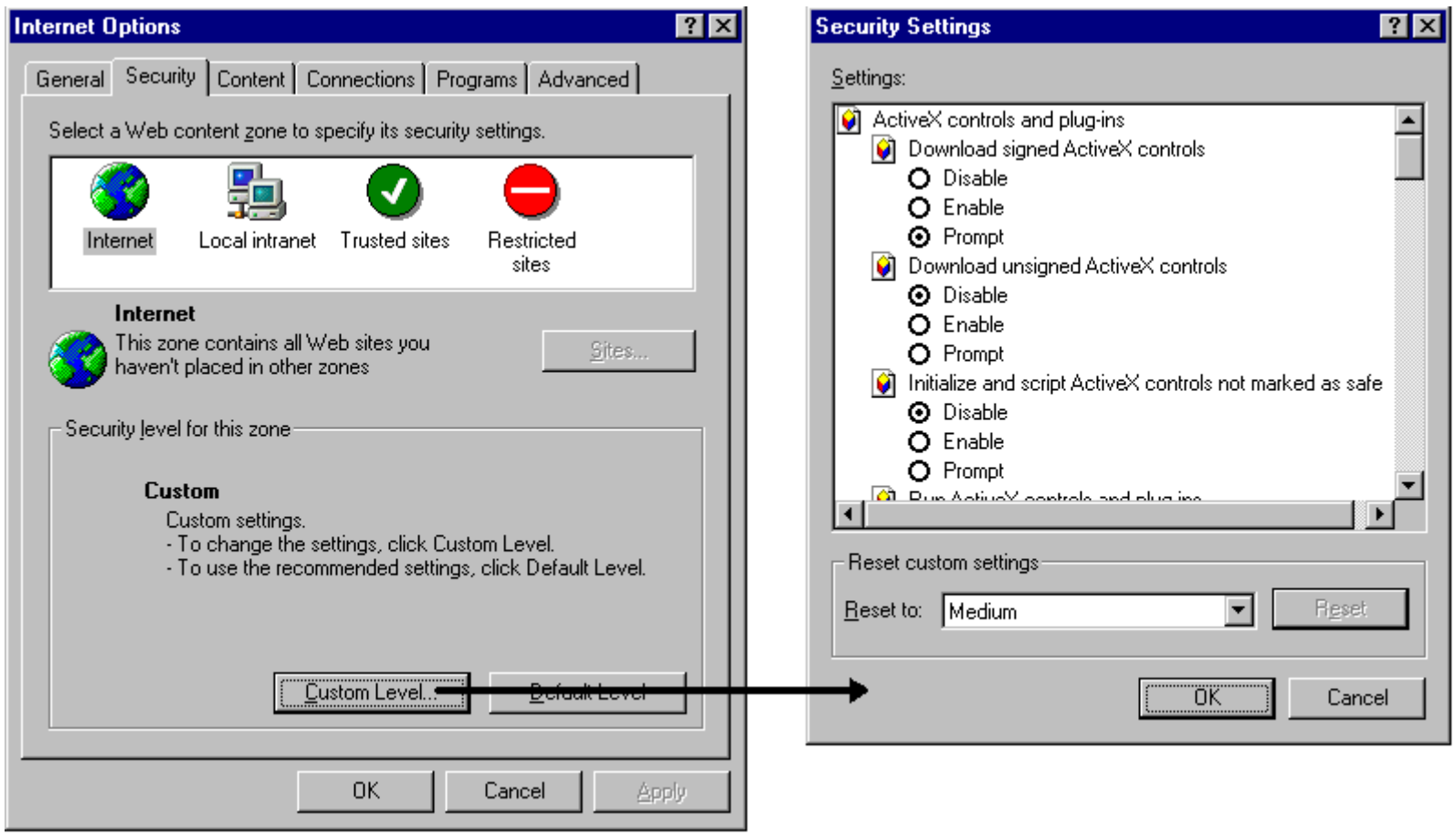
# A Browser Configuration Error Example

**A browser-based error message was raised upon downloading an ActiveX control. The failure is sometimes reproducible using one browser but not in another.**



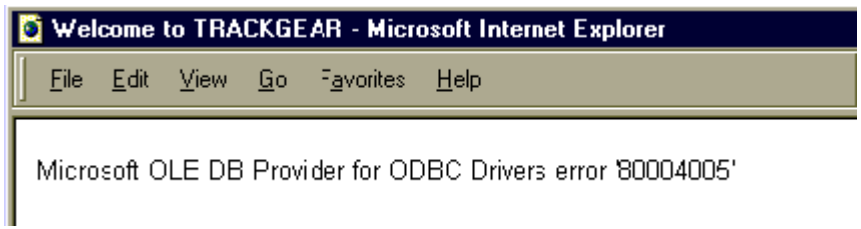
# A Browser Configuration Error Example

**As it turns out, this failure depends on a security setting. The ActiveX is unsigned and "Download unsigned ActiveX controls" is disable in the browser configuration. Therefore the ActiveX control cannot be downloaded.**



# A Server Configuration Error Example

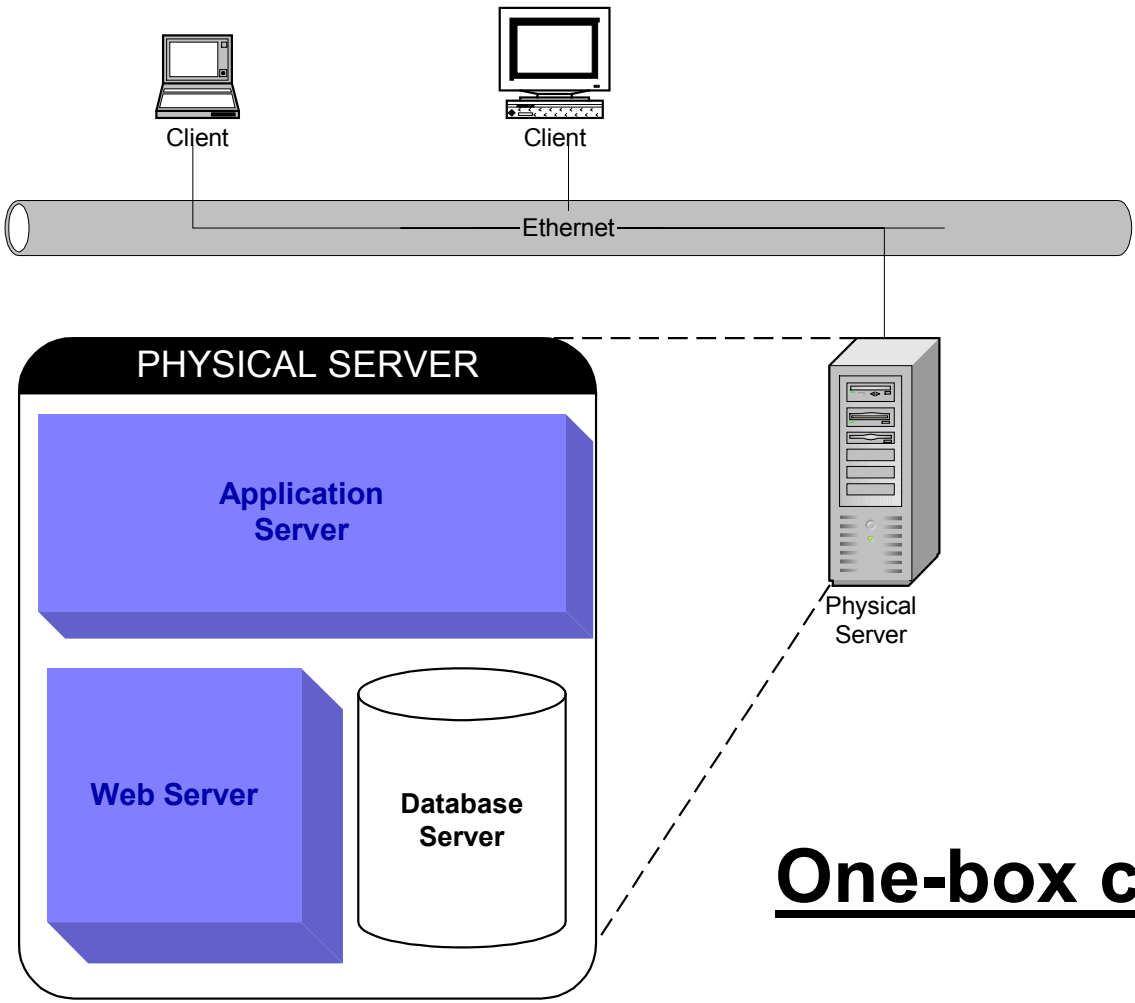
## An error message was raised due to a failure during the log-in procedure.




**The Web server's application directory has not been configured properly to execute scripts**

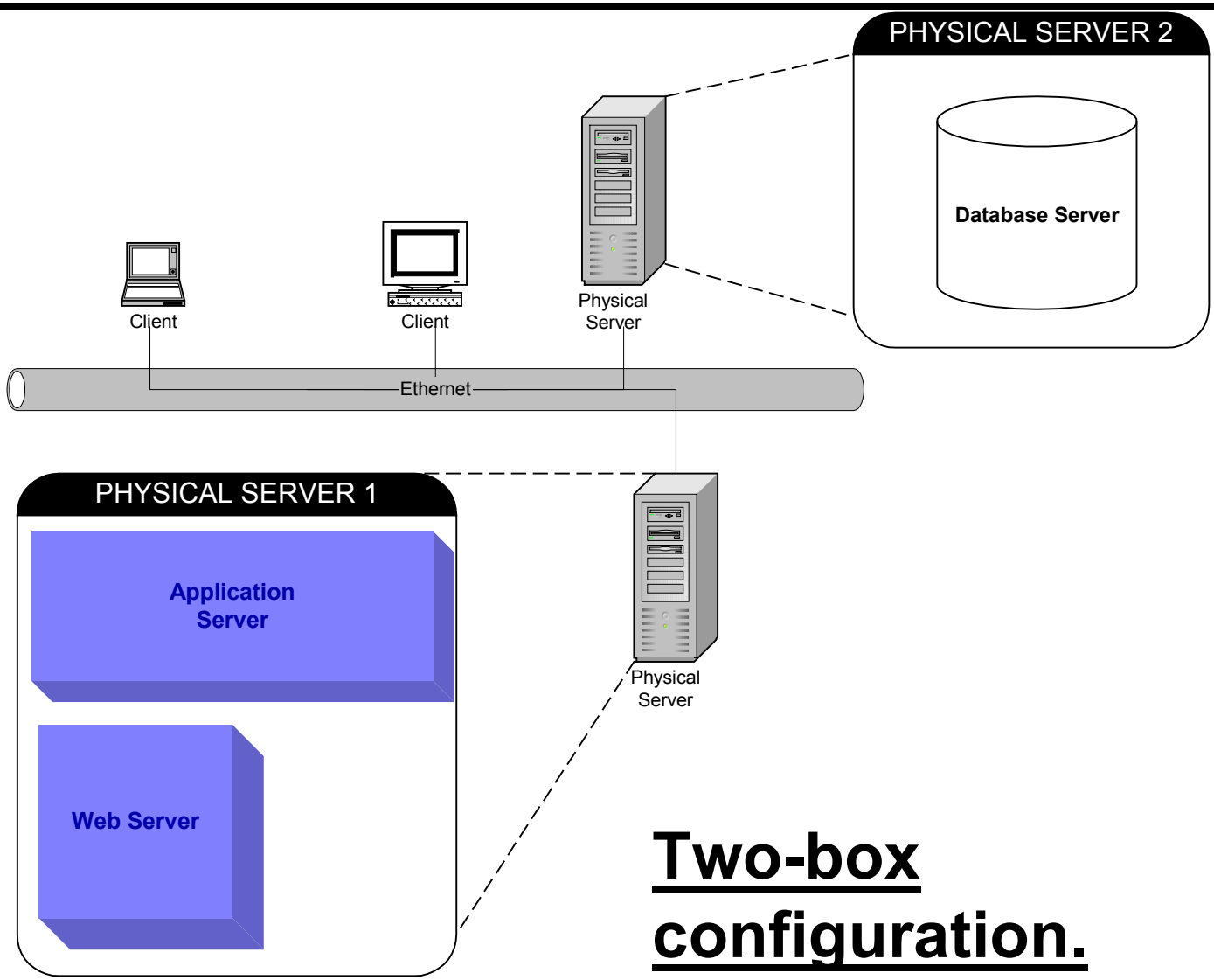
This is a server configuration issue. However, if the installation program failed to programmatically configure the Web server according to specification, then this is a *software error.* If the system administrator fails to properly configure the Web server according to specification, then this is a *user error.*

# A Distributed-Server Error Example




**One-box configuration**

# A Distributed-Server Error Example




**Two-box configuration.**

# A Distributed-Server Error Example

**The implementation:**

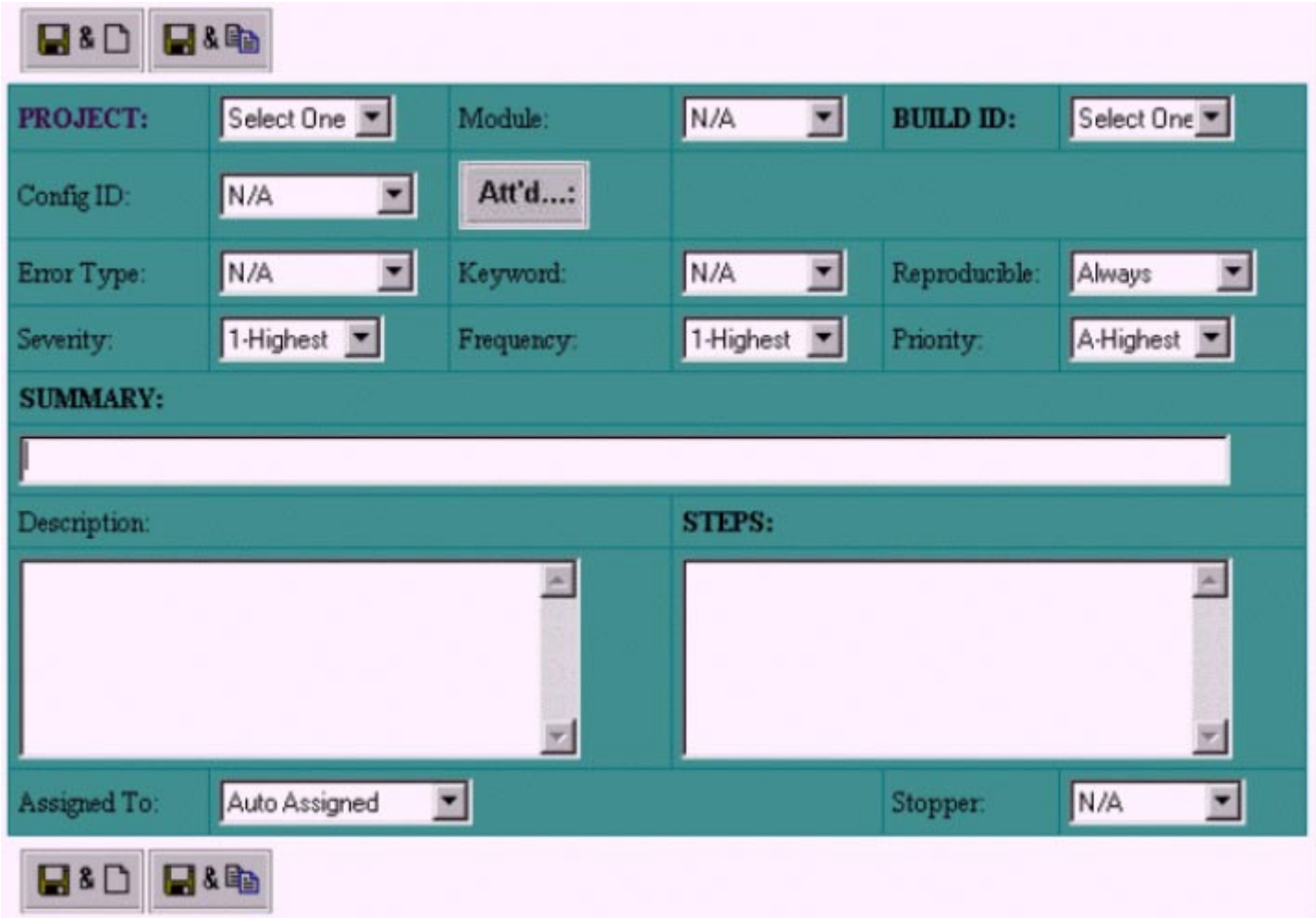
1. **Connect to the database server and run a stored procedure.**
2. **Write the query result to a file named `c:\temp\data01.txt`.**
3. **A separate process will open the text file and read from it.**

**On which server is the `data01.txt` file expected to be found?**

- **There will be no issue expected in the one-box configuration.**
- **The two-box configuration causes a failure because `data01.txt` is saved on the database server box, and the process that attempts to open the file expects it to be in the application server physical box.**

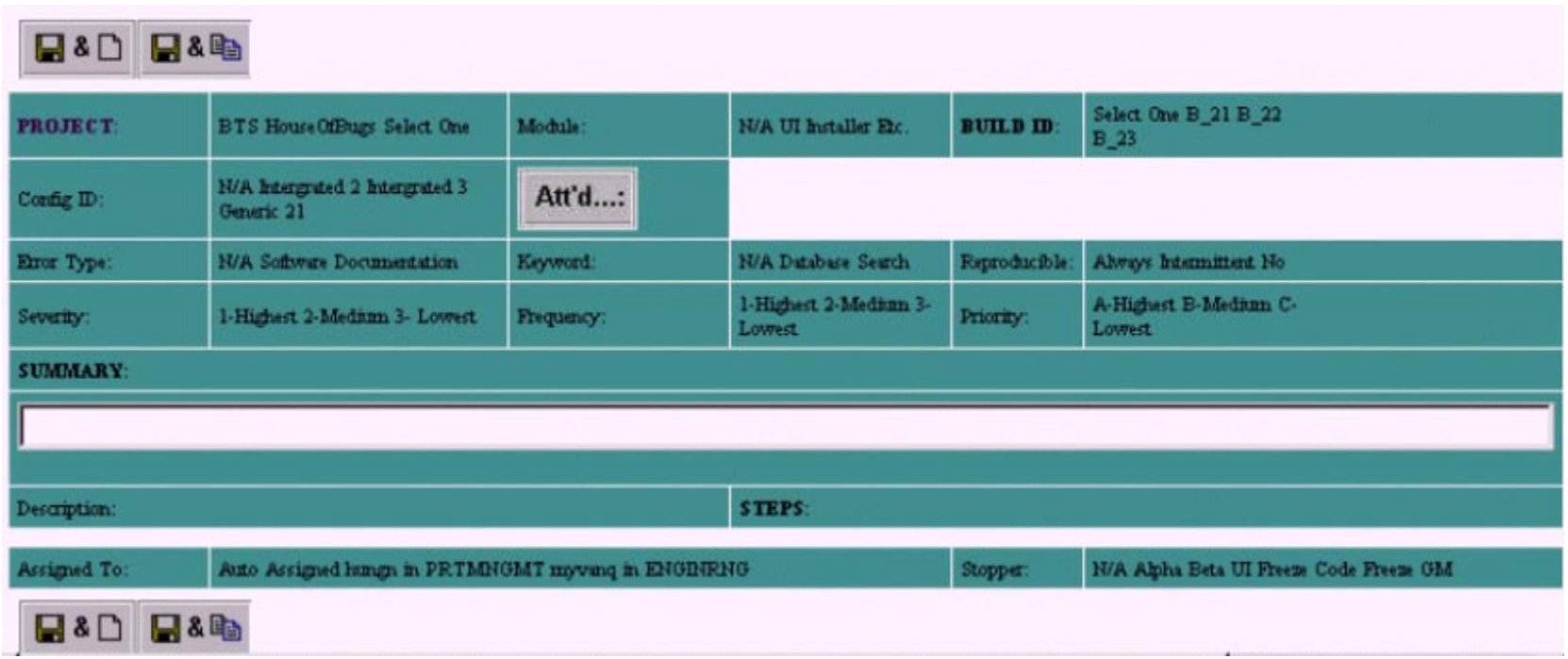
# A Browser Incompatibility Error Example

**Browser A is displaying an HTML page.**



# A Browser Incompatibility Error Example

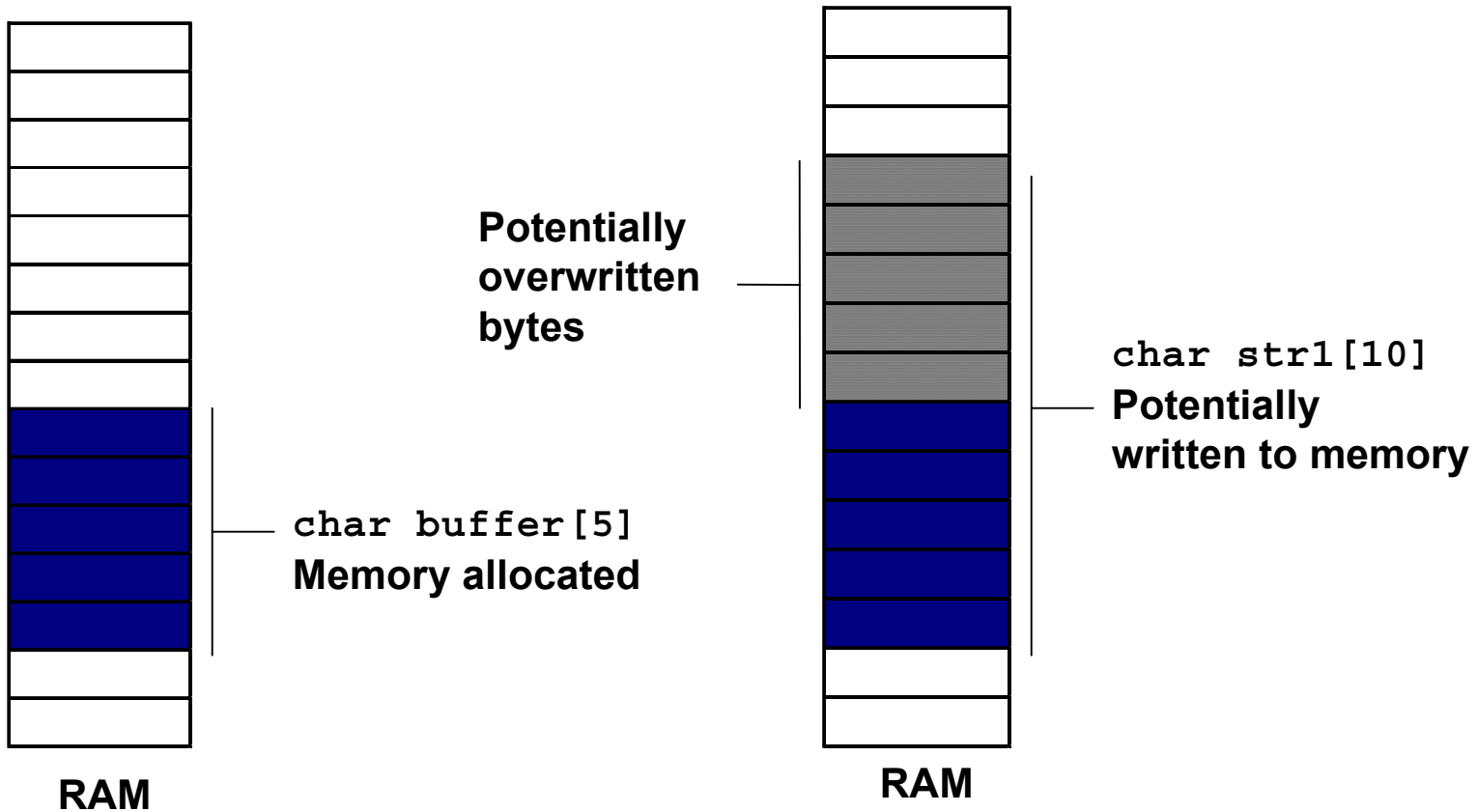
**Browser B is displaying the same HTML page as browser A. However, browser B is incompatible with certain HTML tags, therefore, the controls and text are not formatted correctly.**



# A Memory-Overwrite Error Example

```
#include <stdio.h>
void show_string(char* str2)
{
   char buffer[5];
   strcpy(buffer, str2);
   printf("Your string is: %s\n", buffer);
}
main()
{
   char str1[10];
   gets(str1);
   show_string(str1);
   exit(0);
}
```

# A Memory-Overwrite Error Example



# A Memory-Overwrite Error Example

**What failures do we expect to see?**

- **It depends on the existence of data or code in the overwritten bytes.**
- **If the overwritten bytes are not currently used by any process, then we won't see any failure.**
- **If the overwritten bytes contain data or code, we might see anything from minor failure such as display glitch to fatal failure such as application or system crash.**

# A Multi-Threaded Deadlock Error Example

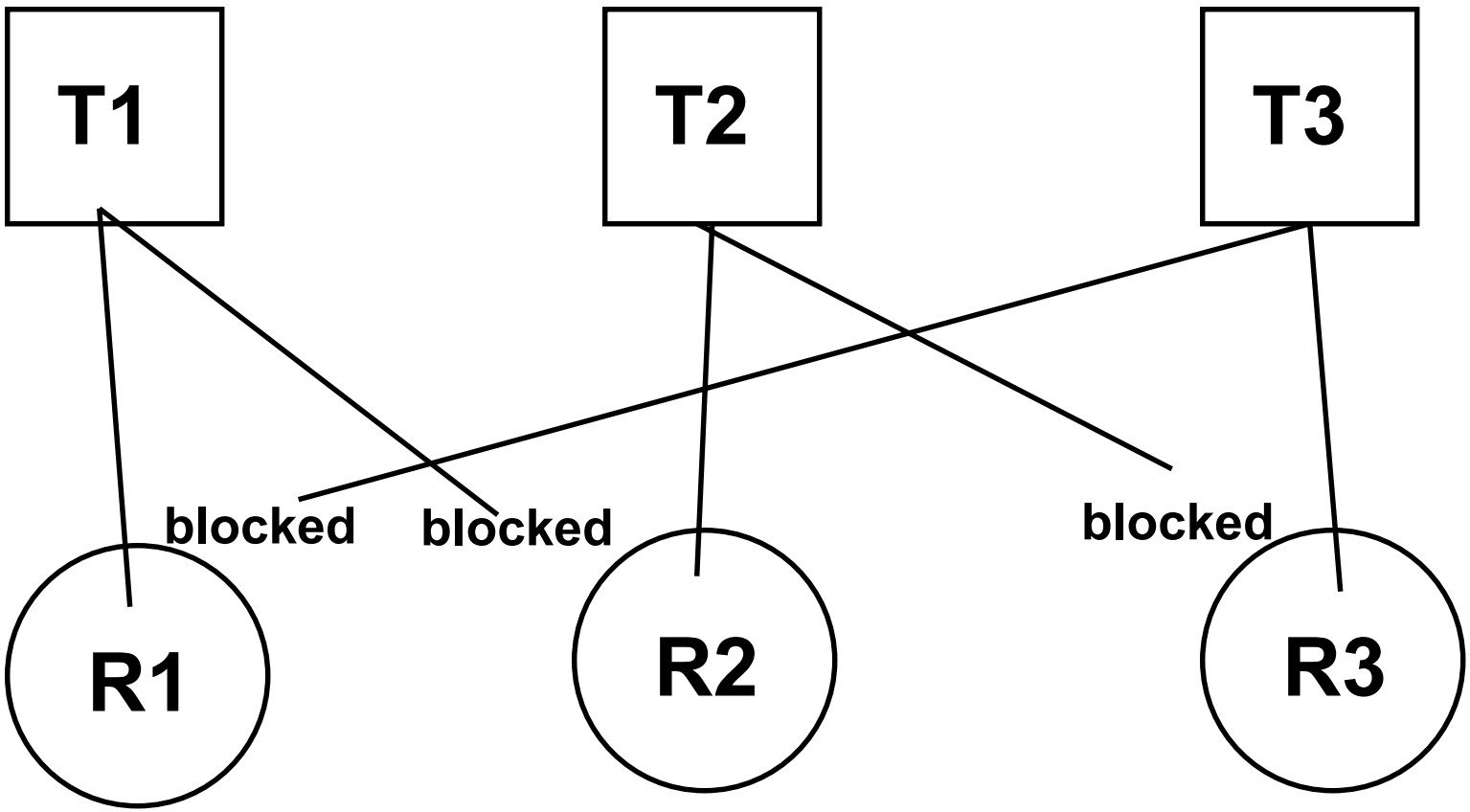
- A deadlock situation always involves the need of resources and the process of acquiring and releasing them.
  - Resources can be hardware devices such as printers, terminals, display devices, processors, and storage devices, or stored information such as subroutines, files, file handles, and data.
  - Software applications need the resources to do their tasks.
  - When a resource is needed, a thread acquires it by asking for it, uses it and then releases it.
- Deadlock occurs when all of the following conditions are true
  - Mutual Exclusion: A thread can have exclusive control of an object.
  - Hold and Wait: A thread can hold locked resources while acquiring the remaining needed resources.
  - No Preemption: While a resource is held by a thread it cannot be reassigned to another thread.
  - Circular Wait: Each thread is holding some resource needed by others while waiting for resources held by others.

# A Multi-Threaded Deadlock Error Example

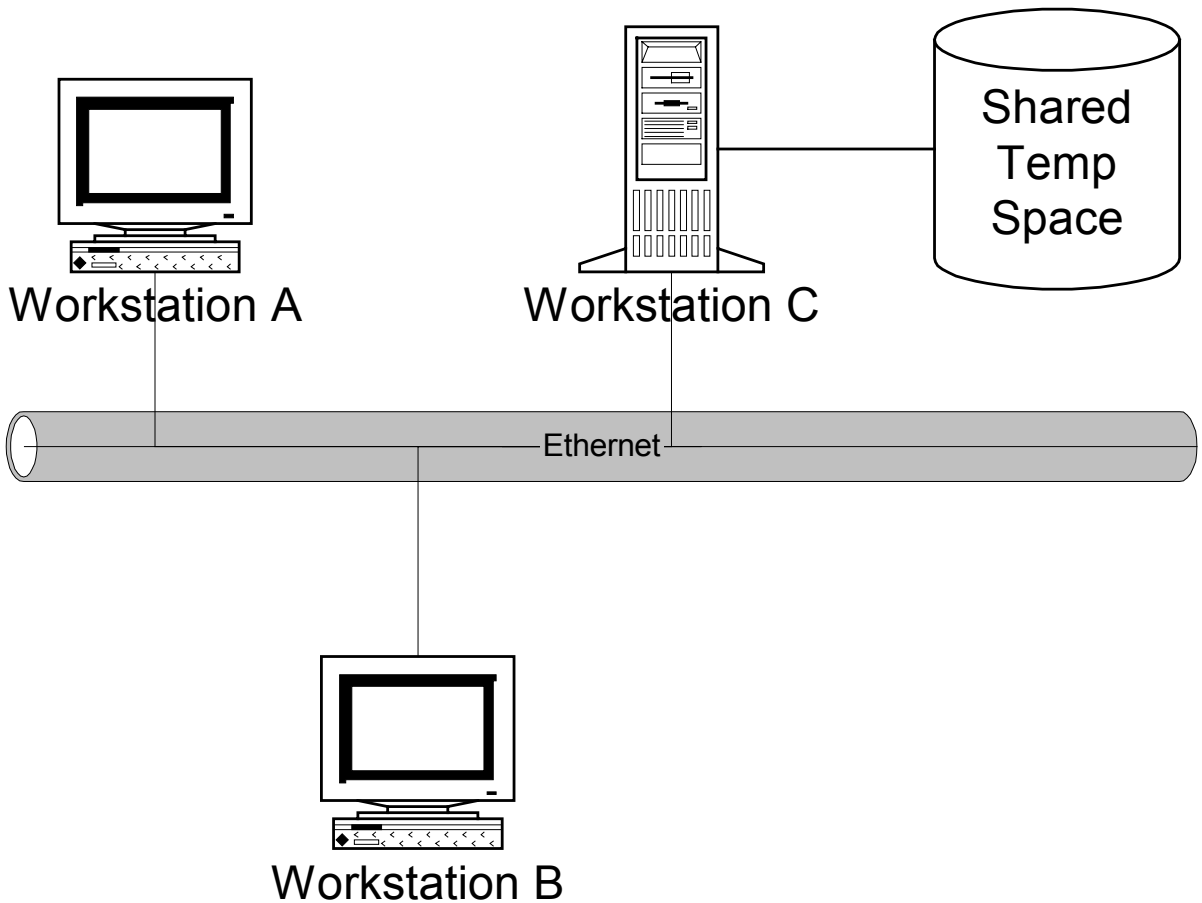
- T1 needs R1 and R2
- T1 holds on to R1
- T1 waits for R2

- T2 needs R2 and R3
- T2 holds on to R2
- T2 waits for R3

- T3 needs R3 and R1
- T3 holds on to R3
- T3 waits for R1



# A Multi-User Error Example



# A Multi-User Error Example

| | | | Before | After |
|---|---|---|---|---|
| Step | Workstation A | Workstation B | Workstation C: Shared Temp Space Available Memory | Workstation C: Shared Temp Space Available Memory |
| 1 | Workstation A needs to write 200 MBs of data to the shared temp space on Workstation C. Workstation A asks Workstation C if the needed space is available. Workstation C tells Workstation A that it has the available memory space. Note that Workstation A did not reserve the space. | | **400 MEGs** | **400 MEGs** |
| 2 | | Workstation B needs to write 300 MBs of data to the shared temp space on Workstation C. Workstation B asks Workstation C to GIVE it the needed space. Workstation C tells Workstation B that it has the available memory space and it reserves the space for Workstation B. Workstation B writes the data to Workstation C. | **400 MEGs** | **100 MEGs** |
| 3 | Workstation A finally gets its response from Workstation C and begins to write 200 MBs of data. Workstation C however now has only 100 MBs of temp space left. Without proper error handling, Workstation A crashes. | | **100 MEGs** | **0 MEG** |

# Considerations in Failure Analysis

- **Bugs don't just miraculously happen and then go away. Unsuccessfully replicating a failure may or may not indicate sloppiness. It may instead reflect a failure dimension or condition that you're not thinking about.**

# Five Fundamental Analyzing Considerations

- When you see a failure on the client side, we are seeing the symptom of an error — not the error itself.
- Errors may reside in the code or in the configuration.
- Errors may reside in any of several layers.
- Errors may be environment-dependent and may not appear in different environments.
- Examine the two classes of operating environments — *static* and *dynamic* demand different approaches.

# Four Fundamental Failure Considerations

- **James A. Whittaker draws a useful analysis on why software fails:**
  - **Caused by *inputs* from the operating environment**
  - **Caused by *outputs* to the operating environment**
  - **Caused by *storing* and *manipulating* internal data**
  - **Caused by performing *computation* using inputs and internal data**

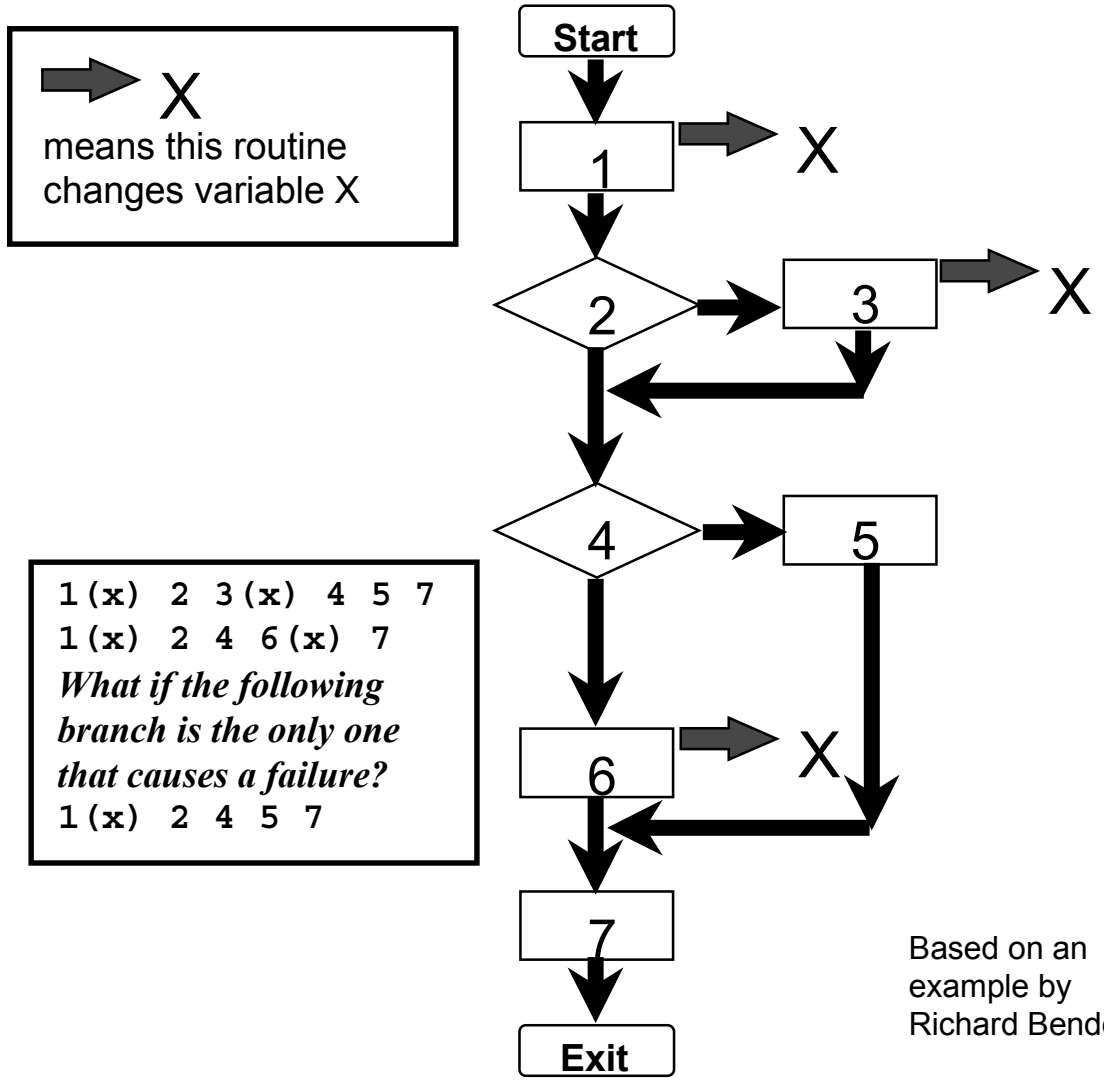
# The Analyzing Process

- **When we find a bug, we are looking at a failure, a set of symptoms of an underlying error. We hypothesize the cause, then we try to recreate the conditions that make the error visible. Sometimes, these conditions are subtle.**
  - **Every attempt to replicate involves a theory.**
  - **The analyzing process — *Hypothesize, test, observe and evaluate.***

# Characteristics of Hard-to-Reproduce Failures

- Memory dependent
- Memory run-time errors
- Predicated on corrupted data (e.g., a corrupted registry key in Windows® environment)
- Configuration dependent
  - Software and hardware
  - Client-side and server-side
- Time dependent
- Initialization issues
- Data flow dependent (see slide 28 for an example)
- Control flow dependent
- Error condition dependent (e.g., a failure occurs only after the first time an error condition is detected and handled)
- Multi-threading dependent
- Multi-user dependent
- Special cases (e.g., algorithm, dates, etc.)

# A Data Flow-Dependent Error Example



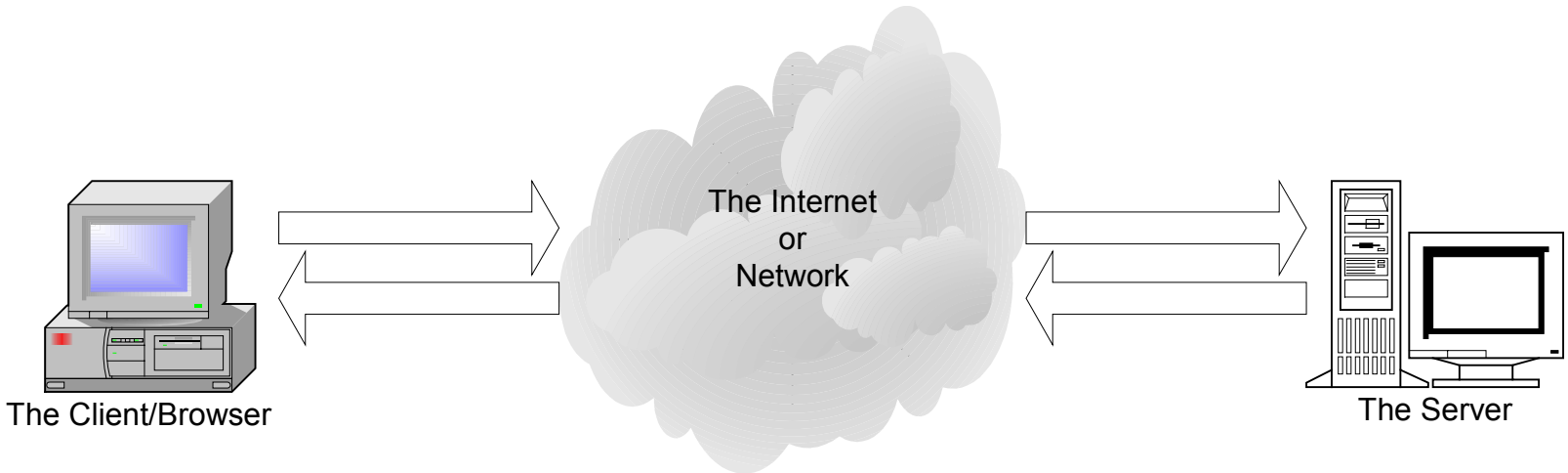

Based on an example by Richard Bender

# The Checklist for Failure Hypothesis

- **Ask relevant *"what-if"* questions**
- **Investigate environment-specific conditions**
- **Investigate program state issues**
- **Investigate data-flow issues**
- **Investigate input issues**
- **Investigate output issues**
- **Investigate logic, rule, and computation issues**
- **Investigate internal data storage (read/write) and manipulation issues**
- **Think in terms of client-server processing**
- **Think in terms of multi-layer operation (component-based architecture, see slide 30-32 for more information)**
- **Think in terms of multi-user processing**
- **Review the characteristics of hard-to-reproduce errors**
- **Review "Making Your Web Application Test Report More Reproducible"**
- **For more information, go to:**
  **www.logigear.com/training/weberror.html**

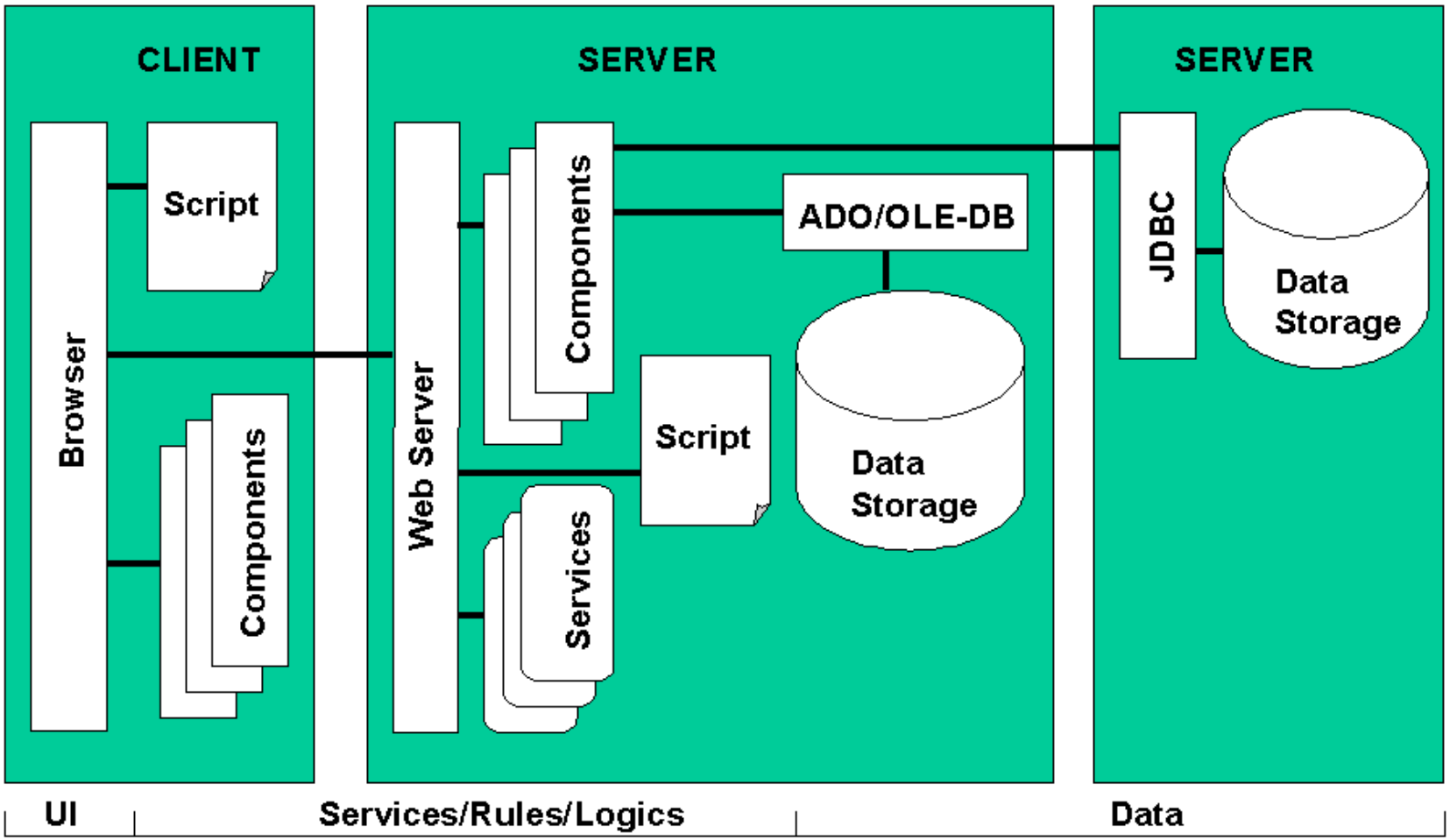
# Three Usual Suspects

**In a Web environment, a failure might be caused by an error on the client side, the server side or the network. Investigate each area thoroughly for potential errors and error-triggering conditions.**



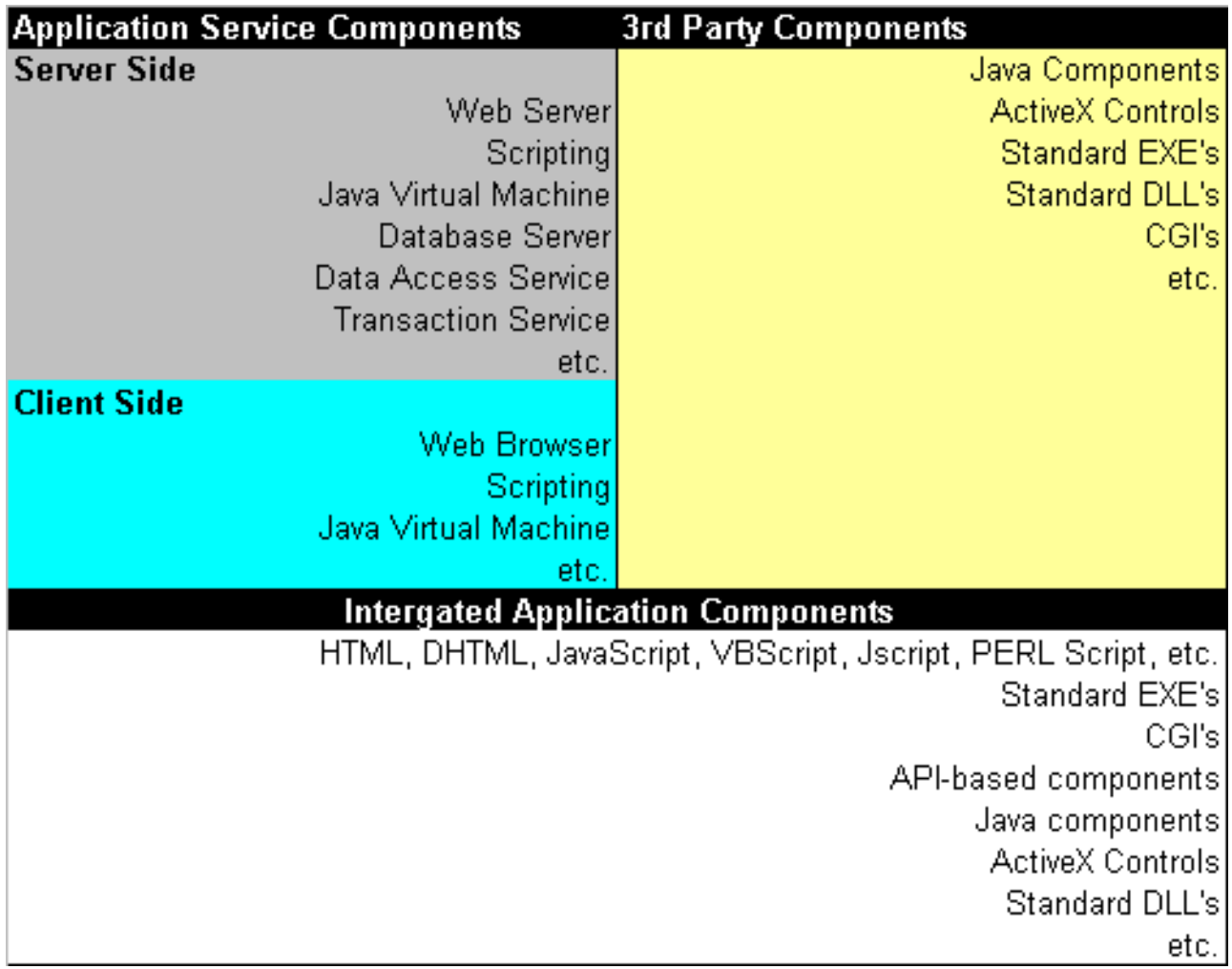
# Multi-Layer Operation (Component-Based)

**An error may reside in any of several layers on the client side or the server side.**



# Multi-Layer Operation (Component-Based)

**An error may reside in any of several components on the client side or the server side.**



# Other Considerations

- Consider conducting design-walkthrough
- Consider conducting code-walkthrough
- Consider using static analyzers
- Consider using run-time memory error detection tools
- Consider using error logging to assist bug analysis

# References


- Leveson, Nancy, “Safeware: System Safety and Computers.” Addison-Wesley, 1995.
- Nguyen, Hung, “Testing Applications on the Web.” New York: Wiley and Sons, 2000.
- Nguyen, Hung,“Testing Computer Software.” LogiGear Corporation Training Handbook, 2000
- Nguyen, Hung, “Testing Web Applications.” LogiGear Corporation Training Handbook, 2000
- Nguyen, Hung, “Testing Web-based Applications--Analyzing and Reproducing Errors in a Web environment.” Software Testing and Quality Engineering, May-June 2000
- Pettichord, Bret, “Beyond the Bug Battles.” STAR East Conference, May 4th, 2000
- Pham, Thuan and Pankaj Garg, “Multithreaded Programming with Windows NT.” New Jersey: Prentice Hall PTR, 1995.
- Whittaker, James, “How to Break Software Applications: A Case Study.” STAR East Conference, May 4th, 2000

# Solving the Software Quality Management Problem in Internet Startups

**James L. Mater and Bala Subramanian**
**QualityLogic Inc.**
**Beaverton, Oregon**

## ABSTRACT

This paper is based on direct experience with the software quality management problems of Internet-based startup ventures. These companies must deal with quality processes and organizational issues at the same time they are experiencing rapid growth and constant pressure to perform. We have developed a model and process that allows the profit and loss manager to overcome inherent problems in managing the software quality assurance function and the costs of poor quality while dealing with rapid growth. The learning shared in this paper can be very critical to the success of other such startup companies.

## Introduction

The Internet phenomenon challenges the quality community to address issues that it has never before encountered. The advent of hyper-growth "dot com" companies, fueled by an abundance of capital looking for spectacular returns, is the entrepreneur's dream come true and the software engineer's worst nightmare. These companies are characterized by extremely short time-to-market windows (measured in weeks instead of months or years), a scaling problem unprecedented in human history, poorly developed infrastructures and processes, inexperienced technical and business staff, and competition that is but a "click" away.

The consequences of operating in this environment are pronounced. Software and system quality are at a low point in the brief history of the software industry[1]. There are daily reports of catastrophic system and company failures. Now, there is a new threat of class-action lawsuits for failure to deliver on express or implied promises to consumers.

The attempts to address quality issues in this new economy are feeble at best. Indeed, the very term "quality assurance" invokes a reaction of uselessness in the Internet world – it only gets in the way of getting product to market ahead of the competition. A lack of experienced quality assurance personnel trained in this environment hampers the efforts of those trying to deal with the problem. The shortcomings of the historic organizational view of software quality – i.e., that software quality is owned by the VP or Director of Development and is implemented by a test and/or quality assurance group

[1] See for instance *Business Week*, December 6, 1999, "Software Hell: Special Report" and the keynote talk by Howard Rubin, 9th International Conference on Software Quality, ASQ, October 4-6, 1999.

that reports to him/her - become apparent in this hyper-growth atmosphere. And the basic premises and focus of traditional software quality methods are inadequate to address the quality issues that incorporate the entire Internet startup: development, IS, operations, marketing and customer service.

Our own experience in this New World has taught us some key lessons that are shared in this paper. First, we must think of software quality in a different way than in traditional software organizations. Secondly, the positioning of this function in the Internet organization is absolutely essential to its success and survival. Thirdly, the job description and caliber of leadership of the quality function have to be redefined. Lastly, the software quality function is not a critical core competency in this environment and can therefore be outsourced successfully.

The software quality community needs to step up to these challenges or become obsolete.

## The Quality Problem in Internet Startups

The quality problem in Internet startups can best be illustrated by a case study. One of the fastest-growing Web sites today is 800.COM, an E-Commerce site that sells consumer electronics, cameras, DVD movies, and similar products over the Internet. Launched in October 1998, the company has become the leading online retailer of home entertainment products.

Although 800.COM is extremely successful now, it took a gutsy move by the President and CEO, Greg Drew, to get the site launched and operational. Shortly after the company was formed, Drew decided the Web site had to be operating on the Internet by Christmas. He poured all the financial resources of the company into the project, investing everything in a sink-or-swim effort typical of Internet startups.

The project required tremendous effort by the company's programmers, but the gamble paid off. The site was up in time to take advantage of the Christmas season.

Software and operational problems often plague Internet companies experiencing such phenomenal growth. In many cases, web software is unable to handle unlimited traffic, and response time slows. When thousands of people access a site, they expose conflicts in the system because every possible option is exercised using every imaginable combination of PC, operating system, and browser. Severe software conflicts can cause the site to crash. Recent crashes of auction and securities trading sites are examples of what can happen.

800.COM suffered only minor bugs in its race to Christmas, but one in particular was costly. Software that printed UPS shipping labels failed to push the address up high enough on the label and 3,000 packages were returned at a cost of $5 each. The problem was easily fixed, but it pointed out the need for a better way to find and solve software problems before they became an embarrassment.

At 800.COM and other Internet "e-tailers", the web site cannot be brought down for maintenance or new code rollout. It has to operate 24 hours a day and 7 days a week. The quality issues extend beyond the traditional development group responsible for web

site features and functions. They include backend systems that deliver products, the creative team that designs new web pages, the IS organization that worries about keeping the system 100% up and reliable, and the marketing team that is constantly changing the requirements for the entire system. [2]

For the "e-tailing" industry in particular, quality is a major issue. A recent study highlights the problem with the explosion of online stores. Andersen Consulting went shopping at 100 of the biggest and best-known online stores. Out of 480 gifts it tried to buy, it was able to complete only 350 purchases.

The study found that more than one quarter of the top Web sites either could not take orders, crashed in the process, were under construction, had entry blocked, or were otherwise inaccessible. The study, originally designed to measure order fulfillment times, found that the real issue today is not speed of delivery but reliable web sites.[3]

Another major issue for the new business form called "e-Commerce" is that companies move so fast and the technical community is so inexperienced that major business risks are not even understood until it is too late. For instance, the Federal Trade Commission issued a 1975 Rule that covers mail order companies. The Rule requires companies to notify the consumer if an order cannot be shipped on time. The customer must then be given an opportunity to cancel the sale, although the company can presume that the customer wants the order completed if the customer says nothing after being notified.

If the company cannot meet the revised shipping date, the Rule requires the consumer to be notified a second time, and unless the consumer expressly consents to the new delay, the company must cancel the order.

Many e-Commerce "e-tailers" are not even aware of this rule. The FTC has already taken action against Geocities for violating it.

In addition, companies are exposing themselves to class-action lawsuits that can be brought on the basis of breach of contract. Although such suits are not intended to go to trial, the cost of settling and the accompanying publicity can be very damaging to an electronic retailer.[4]

It's not just the e-tailing community that has quality problems, but the infrastructure community as well. Major outages of the telecommunications systems

---

[2] Interviews with 800.com executives during 1999.

[3] "One in Four Online Buys Thwarted: Andersen Consulting finds that many top Web sites stumble," Reuters, December 21, 1999.

[4] "E-tail Failures Could Trigger Federal Legal Action" By Stephen Caswell, E-Commerce Times, December 29, 1999.

occur on a regular basis, and more than one vendor of Internet infrastructure components[5] has failed due to quality problems with their product[6].

## Current Quality Management Efforts in Internet Companies

Traditional software engineering approaches are inadequate for the challenge of the Internet economy. There are significant differences between the historic software engineering environment and this new economy that include:

1. *Time to scale technology, organization and processes.* While the software engineering community evolved in a business environment that spends months or years developing products, the Internet economy requires companies to not only develop products much more rapidly, but to do so while growing their organization and processes at the same time.[7]
2. *The pervasiveness of software in e-Commerce companies.* Software is both the enabler and the life-blood of these new enterprises. Software underlies almost every aspect of e-tail and infrastructure companies and is in constant flux as the companies grow and change to meet their changing competitive and market demands.
3. *The scope of the software quality issues.* The lines of software development have become blurred. Development is not just the purview of the engineering or IS department; firms are likely to have software being developed or integrated in two or more functional organizations. One organization may build key software components, another may integrate them with third-party packages, still another group may develop the web pages, and a fourth organization may own and operate the fulfillment technology.
4. *Software quality cannot be addressed in a vacuum.* For example, e-tailers succeed or fail as a result of an entire customer shopping experience. This experience includes everything from the ease-of-use of the web site to the speed of fulfillment and even the interface with customer support. All are quality issues that need to be addressed by rapidly emerging, inexperienced organizations of artists, technologists, and business people.

---

[5] These include packaged software for functions like "shopping carts", "personalization" of web sites, order management, etc.

[6] See for instance *Business Week*, December 6, 1999, "Software Hell: Special Report" and *The Software Conspiracy*, Mark Minasi, McGraw-Hill, 2000.

[7] Typical examples are a music e-Commerce company growing from 10 to 80 in four months; an Internet advertising company growing from 60 to 120+ in a year; an e-tailer growing from 60 to 200 in 6 months. Growth in web traffic and commerce is even more spectacular.

An organization like this is not anything like the environments that subject themselves to SEI CMM audits[8] or classical process improvement programs. In our experience, the problems and solutions are, in fact, similar, but the time scale, organizational models, skills, and tools required are significantly different.

Although much of the discussion in the software quality community includes issues such as "ease-of-use" and "customer experience", we almost never encounter a quality assurance organization that is not an engineering function reporting to a Development Manager or VP. This same model has been translated to Internet startups with minimal success in our experience.

The Internet startup environment is so fast-moving and experienced staff so scarce that we observe the following in many startups:

1. Quality assurance teams are not formed until products are ready to ship or go live. The basic habits and processes of software development are already in place and are difficult to change.
2. Quality assurance teams are asked only to be test teams and expect that this is their only role. They become consumed with attempting to just keep up with the pace of development and are preoccupied with building methods and processes for testing products or web sites already in development.
3. The staff members hired for quality assurance are typically inexperienced, lack formal software engineering training, are technophiles (e.g., love the technical aspects of the job), and have no business experience. This is due in large part because quality assurance is the name typically applied to testing and the skills sought are those of entry level testers. This is also due to the lack of skilled quality assurance professionals with e-business experience.
4. Company leadership does not ask nor expect the quality assurance group to do more than test product but blames them if something breaks.
5. Quality assurance teams are not trained nor encouraged to take on the wider role of managing quality for the company as a whole.

It is not hard to understand why the results we see are less than desirable. Even in companies that have seasoned technical leadership at the CEO/COO level, there is difficulty focusing on this issue and solving the problems raised above. One company that we follow attempted to solve this problem by hiring a Manager and QA engineers who were willing to leave a larger, well-established software company for the startup environment. After a few months of getting the team skilled in the products and culture of the company, the whole team left for "sunnier" parts and the company had to start over again. The VP of Engineering lost his job and the problem is still unsolved as of this writing.

---

[8] The Software Engineering Institute has developed a model called the Capability Maturity Model that allows software organizations to assess their own ability to produce quality software repeatably on time and on budget.

## Solving the Quality Problem in Internet Startups

The keys to providing necessary quality oversight are to 1) elevate the software quality assurance function to a corporate quality function reporting to the CEO or President/COO and 2) upgrade the skills of the group to match this increased influence and responsibility. Companies can do this themselves (although we have observed that it is a very difficult transition to make), or they can outsource the function to a specialized firm.

The CEO and President/COO (especially of an Internet start-up) need to know that their organization is scaling up and implementing the tools and processes that will ensure continued product or service success. What they need is a professional and independent view of and impact on the quality of the processes and products as the company grows.

By elevating the quality function to a strategic one reporting to the CEO or COO level in the Internet startup and by upgrading the skills of the quality team, the company is able to address these major issues. The charter for the quality team becomes one of risk assessment and management and organizational influence.

Risk assessment and management is a term that seems to resonate with the leadership of Internet startups[9]. While it is basically classical "quality assurance," it takes on additional responsibilities in an Internet startup (relative to how QA is typically chartered in most software companies). The key risk assessment and management activities are focused throughout the company so that the customer experience is "delightful". These activities include:

1. Understanding key failure conditions. Common ones include scalability, robustness, and speed of changes of technology and in the organizations themselves.
2. Planning for complete robustness, i.e., never be the cause of a failure (to a browser, to the OS, to anything). Customers in the Internet world are not tolerant and competitive solutions are just a "click" away.
3. Providing traditional testing that is critical to risk management and provides the assessments and data necessary to make go, no-go decisions.
4. Leading the company relative to process. It needs to assess the right amount of process, understand the right processes to implement when, and then influence the company to adopt and use these processes. Is there a process for understanding customers (needs, satisfaction)? How does information flow from the customer to appropriate groups in the company? How are projects planned and managed? How are products verified so they won't crash and burn when delivered? Can the company repeatably and reliably provide new content and functionality to customers, at Internet speed?

---

[9] Reactions from seasoned Venture Capitalists, advisors and CEOs of Internet startups in discussions with QualityLogic staff during January 2000.

All of this has to be done in the context of a rapidly growing company; doing it greatly increasing stress on the systems and services the company provides. The sophistication of processes and tools must migrate at the same speed as the company.

Organizational influence is the second critical function the quality team takes on. In Internet companies this entails helping establish a learning culture that can adapt and change as technology, markets and competition change. It requires the team to be an effective change agent, constantly monitoring risks and behaviors and causing changes in the behaviors necessary to mitigate risks. This is not the usual software QA role but becomes possible with the influence available from the CEO sponsorship and the upgraded skill sets needed for this role definition.

The key issues to focus on in this role include:

1. Influencing the organizational culture to become one that learns what's going on around them and takes the appropriate action to continue to be successful. Influence individuals and help mold them into a team that propels the company towards its ambitious goals.
2. Influencing behaviors of individuals in the company. The quality team needs to understand what behaviors are required for success. It needs to influence and change behaviors to ensure that everyone in the company (including themselves and the executive team) understands and acts accordingly. The real challenge is that people react to behaviors of others, not to what is said or written.
3. Realizing that the QA team can't do everything itself (nor can other teams) in Internet time. The ability to leverage internal and external resources is the key to success.

   By bringing the resources of a specialist company to the quality problem, companies can leverage seasoned management experience and an infrastructure of labs and technical resources. Not only can this impact the bottom line by improving the quality of products and services but it can also be done very cost-effectively.

What we find most interesting about this redefinition of the software quality assurance role is that it becomes a corporate quality assurance role. The organizational positioning, the skills required, and the basic corporate functions are well defined for traditional industries.[10] Since software is the basic technology that drives Internet companies, it is not surprising that the definition of the software quality function should look very much like a corporate quality function.

## A Case Study: Unicast

A case in point is Unicast Communications Corporation[11] in New York City. The COO is Berkeley Merchant, a past President of the Software Association of Oregon. The

[10] See *Juran's Quality Control Handbook* McGraw-Hill.

[11] See www.unicast.com for more information.

company provides both the technology and creative services to create, serve, and report rich media ads on the Internet, called The SUPERSTITIAL™. This is the Holy Grail of Internet advertising, and Unicast is the leader in this business.

When Berkeley took over as COO, he inherited an emerging organization that was ready for the next step in its evolution. The company was getting product shipped but with significant issues that caused significant problems with customers. These customers included major advertisers such as HP, Macy's, and Universal Studios, and any hiccups in their Internet advertising reflected directly on Unicast. Unicast had one test engineer (who lacked formal training in software quality assurance and engineering) and loose, informal processes for development and test.

One option was to outsource the quality function, and this is what Unicast did. The company hired a specialist firm in quality function outsourcing. Within two weeks the vendor took full charge of the next milestone product release for Unicast. A professional QA Manager was put on-site. The vendor implemented the test infrastructure and test effectiveness and managed product quality assurance, including technology integration testing with third parties through production release. The result was a very robust and quality product release.

By the end of the third month of outsourcing the quality organization, Unicast had a comprehensive quality program, including a process improvement program. The vendor immediately began to leverage its other labs to supplement the testing activities (saving significant dollars over doing work in New York City). A defect management process was implemented quickly, and a life cycle model was installed for development – all while increasing the testing effectiveness and product quality without hampering shipment schedules.

The results were dramatic. The timely release (the first major activity of the quality function vendor) helped Unicast to move ahead with its business strategy of strategic alliances with other industry players. The outsourced quality function at Unicast provided a significant and almost immediate benefit in professional Quality expertise, quick scaling up to meet the Internet business's growth and leveraging of available supplemental expertise and resources.

It also helped create a strategic competitive advantage for Unicast. The executive team was able to start viewing product quality as a competitive advantage in an environment which tends to focus more on features, functions and time-to-market than customer satisfaction.

## A Process for Quality in Internet Startups

An ideal solution is to establish a quality culture, an appropriate platform of processes, and a quality function very early in the life cycle of an Internet startup. In the current venture capital-backed model, these startups have a primary problem getting to market and a secondary problem of scaling an organization quickly. The availability of budget is not a major issue, but money must be spent wisely to get to succeed quickly.

In this environment we have the opportunity to establish things right from the outset. The key steps are:

**Initial Risk Assessment.** This process looks at the basic requirements process, the plans for technology, vendors, integration and organization. Based on these factors, recommendations are provided as to where risk exists and what can be done to mitigate it. For instance, if there is a plan to purchase and integrate several technologies that have never been integrated before, the assessment would point out these risks and suggest some plan for managing the risk. Some examples of mitigation plans for risks include: identifying a single project manager, implementing strong integration testing program, reconsidering the strategy, etc. A close look at the requirements process would itself be invaluable, as many startups haven't done a very good job of documenting and communicating requirements to staff and vendors.

Another key process for Internet startups is an assessment of the scalability risks of the basic architecture and design. This forces a clear statement of requirements in order to assess the capabilities of the architecture and design in light of these requirements.

**Business Planning.** A key requirement of the CEO of a startup is continual fundraising. One of the keys to convincing investors to support an emerging venture is demonstrated ability to deliver quality products and services. An independent quality assurance vendor can ad immeasurable credibility to a new venture and can assist in developing the sections in the business plan that relate to quality execution and budget.

**Quality Infrastructure Definition and Implementation Assistance**. This step defines and establishes the key (and appropriate) processes and tools to enable management of the technical aspects of the business. These processes should include a life cycle model, configuration management, release management, defect or issue tracking, and a basic metrics program to measure both risk and quality on an on-going basis. The philosophy and value are to put in place early the right processes, culture, and tools to ensure successful scaling of the organization. It must be stressed that these may be only rudimentary processes at this early stage, but they provide the basis for scaling and formalizing the organization and implementing supporting tools as appropriate.

**Test Infrastructure**. The infrastructure includes the processes and tools for testing the early and on-going implementations of the product or service architecture. This stage includes the test planning process, test case development, use of vendor resources and labs for test execution, recommendations for engineering design to support test, reporting, etc. The intent is to absolutely meet the needs for startup testing while building the basic infrastructure for on-going test of the product/service.

The skill sets required to implement and manage this process are significantly different from those of traditional test or even QA functions in software organizations. We touched on these skills earlier in defining this new role. The key attributes of the team leader for this function include excellent: credibility with senior executives; organizational influencing skills; business risk analysis skills; quality assurance and test experience and skills.

## Benefits

If done well, the results of these activities can be profound for the success of the company. The activities can save dollars later on and more importantly ensure that products/services get to market quickly and work when they get there.

The real advantage of this model is that it puts a high value on the early definition of infrastructure that we believe to be critical to future success. It is also palatable and indeed attractive to investors and executives alike, especially those who understand that if this is done right, time-to-market will actually be shortened and product quality will increase while the organization is being architected for future growth.

## Conclusions

In many ways, the Internet is a new frontier for software quality assurance. For the first time, software quality and corporate quality are almost synonymous, and demands on the software quality function are expanded to meet a strategic corporate role rather than a technical role.

In this paper, we have discussed the quality problems faced by Internet startups, the changing role of software quality in Internet startups and the solution to these challenges. The keys to success in the Internet world are to elevate the quality function to a corporate level and to upgrade the responsibilities and skills of the quality team.

# A Pragmatic Software Configuration Management Model for the E-World

**Michael K. Jones**
**Quality Assurance Director at**
**Integrated Information Systems**
**Tempe, Arizona, USA**
**And**
**Assistant Professor**
**Western International University**
**Phoenix, Arizona, USA**

## ABSTRACT

Software configuration management must be practiced for any software system from the moment any work on the software system begins to when support ceases for that system. In the E-World, which is an expression used to describe the environment in which web applications are developed and utilized, effective and time-efficient software configuration management must be practiced such that baselines are identified and changes are tracked to those baselines. However this software configuration management effort must not impede development and must be clearly understood from a functional basis. The overwhelming time to market requirement for E-World development efforts can not tolerate any impediments to development time and responding to client needs. This high emphasis on time efficient processes has not been true for many software development projects in other environments in the past as a rule, and pragmatism has surely not governed many efforts that had to observe numerous government regulations.

The theoretical model that is most commonly used in reference to software configuration management is one that has a typology of configuration identification, change control, status accounting, and configuration audit. This typology is not functionally oriented and is consequently either misapplied with gaps in coverage or impedes the development of the software system by being oppressive and bureaucratic. This model was developed with hardware in mind as its primary target and consequently must be interpreted for software projects.

A better model for software configuration management that is clearly understood and is scaleable is the subject of this paper. Software configuration management can be functionally broken out into the areas of 1) version control, 2) document control, 3) change management 4) build management, and 5) release control.

By discussing each of these functional areas of software configuration management at length, a realistic and doable model of software configuration management will be seen from the explanations and examples supplied in the paper. The reader of this paper will then, hopefully, leave with a better appreciation of what to expect in software configuration management efforts in the future, both as a developer and as a client.

# INTRODUCTION

Software configuration management is a crucial activity for any software development effort. The software configuration management activity, however, must not delay or impede the rapid software development schedule necessary to meet the harsh time to market needs of the E-World. Consequently, effective and time-efficient software configuration must be practiced with all efforts having a justification in functional value. The software configuration management theoretical model that is most commonly referred to in literature does not easily correspond to the functions that must be accomplished through software configuration management activities. A better model that is functional in derivation, and that will be clearly understood and will be easily scaleable is the subject of this paper.

A functional model of software configuration management is organized into the areas of 1) version control, 2) document control, 3) change management, 4) build management, and 5) release control. This typology corresponds directly to the functional tasks that must be performed for a project and also agrees with the typology of the major software configuration management tool vendors. Each area will be defined and the practices or techniques that must be implemented for web applications will be discussed.

# THE THEORETICAL MODEL

The theoretical model of software configuration management arose out of post World War II efforts in the aerospace industry for the United States Defense Department and were a spin-off of previous successful efforts for hardware configuration management according to Berlack (1992). This model has continued to be referred to by textbooks on configuration management, Buckley (1996), and by textbooks on software engineering, Peters (2000). The model consists of four elements: configuration identification, change control, status accounting, and configuration audit. Although this model still stands head and shoulders above the Microsoft model of the "daily build" as discussed by Pfleeger (1998), the traditional model still remains encumbered by a focus on bureaucracy instead of project functionality.

The Microsoft daily build technique, for those who are unfamiliar with it, consists of declaring the baseline by whatever was "thrown over the fence" to the build manager at either the beginning of the day or the end of the day, depending on the project. Why this technique is not held in high regard by at least some in the software engineering community, is that builds are not a result of conscious decisions about what functionality should be included in a baseline, but only a result of frenetic programmer activity that is personality dependent.

Definitions for the traditional model as provided by Buckley (1996) provide some insight into this conclusion:

Configuration Identification – "Configuration Identification includes the selection of configuration items; the determination of the types of configuration documentation required for each configuration item; the issuance of numbers and other identifiers affixed to the configuration items and to the technical documentation that defines the configuration items configuration, including internal and external interfaces; the release of configuration items and their associated configuration documentation; and the establishment of configuration baselines for configuration items"(p.259).

Change Control – "The systematic proposal, justification, evaluation, coordination, approval or disapproval of proposed changes and the implementation of all approved changes in the configuration of a configuration item after formal establishment of a baseline"(p.259).

Status Accounting – "The recording and reporting of information needed to manage configuration items effectively, including:

a) A record of the approved configuration documentation and identification numbers.
b) The status of proposed changes, deviations, and waivers to the configuration.
c.) The implementation status of approved changes.
d.) The configuration of all units of the configuration item in the operational inventory (p.260).

Configuration Audit – "The verification of a configuration item's conformance to specifications, drawings, and other contract requirements (p.258).

From these definitions, there is a strong inference, substantiated by the author's personal experiences on past projects that used the traditional model ( dictated by DOD or NASA standards), that extraneous paper documentation becomes the overriding "end" as opposed to the "means". Along with this paper, came a bloated personnel count with "checkers" of the "checkers". Hence, large amounts of extraneous paper documentation and superfluous personnel equate to needless bureaucracy.

## THE FUNCTIONAL MODEL

The functional model, in contrast, that this paper advocates is strictly based on the correspondence of software configuration tasks to the tasks found in the basic software development model. The basic software development model, according to Sommerville (1996), who is the author of one of the most commonly used textbooks on software engineering and consequently one that most people might be familiar with, consists of four fundamental activities which are common to all software development processes: software specification, software development, software validation, and software evolution. The functional model of configuration management maps the functions of version control, documentation control, change management, build management, and release control to the development model.

The model is called a "functional" model because the typology is based on tasks that are commonly called out on a WBS (Work Breakdown Structure) and there is no need for interpretation of task versus configuration management area type. A functional emphasis is important for the E-World, i.e. web applications, because all actions must be as time-efficient as possible to meet deadlines and yet control of baselines and changes must still occur. There is not the luxury of pondering over what configuration management means according to some abstract model, but rather a driving concern with getting tangible tasks performed quickly. A "functional" model is focused on getting those tasks done, not with generating paper.

Mapping the functional software configuration model to the development model, version control takes place at the conclusion of development with formal software baselines prior to validation. Documentation control takes place at the conclusion of specification and then continues throughout with traceability of documents to software baselines. Change management is initiated immediately following the first instance of the use of version control or document control. Build

management occurs with the initial repeatable documentation of how to construct the first formal baseline with updates then being a constant necessity. And release control is performed at the conclusion of validation such that all versions of the system that are released to outside parties are approved, recorded, and tracked against requests for defect resolution and enhancements. A description of typical tasks and activities for each functional area follows and is organized on the basis of "What", "Why, "When", "Where", "Who", and "How" for the purposes of clarity and definition. In addition, the special needs and consideration of web applications will be discussed.

## VERSION CONTROL

What:
Version control combines procedures and tools to manage different versions of configuration objects that are created in the software development process, according to Pressman (1997). Configuration objects are identified, for internet applications, by completing a physical architecture diagram during the design process. It is only at that point that objects may be discerned in a configurable form. The physical architecture diagram had to be preceded by the creation of the domain model and the object model for successful design derivation.

Why:
Version control provides for visibility, control, traceability, and monitoring of software components and systems, and of documentation. It is a necessary function for successful project management and the implementation of a "planned" system architecture.

When:
Version control is instituted for software systems after development is complete, so that baselined units can be employed in system integration and validation testing. Version control is then repeated with new versions for those items as changes are introduced to those baselined items.

Where:
Version control is performed through version control utility software located on a system that is accessible by all participants on project team ( i.e. developers, testers, project managers, and quality assurance.

Who:
Version control is performed by multiple parties with different roles. Unit developers check units in and out for their work. System developers construct and identify specified groups of units as configured systems. Testers check systems out for testing. Project managers approve these configurations at project milestones. All of these activities are performed by project personnel with the appropriate privileges granted in the version control system. The administrator of the system, however, should be a configuration management analyst who is responsible for the administration of version control throughout the organization, in addition to other areas of responsibility such as change management and release control,.

How:
Version control is accomplished mainly by software tools with some set of the following capabilities:

Version any type of file
Forward/reverse deltas

File compression
Branching
Automatic branch creation in certain situations
Version labels
Visual differences
Visual merging
Parallel development
Support projects as well as files
Support sub-projects
Unlimited directory hierarchy
Recursive commands through hierarchy
Track changes to project structure
Project/file sharing between multiple applications

In the preceding nomenclature, a definitive description of version control for the functional model was presented. This description provides the foundation for a discussion of special E-World needs and considerations.

For web applications, tools are a necessity as they greatly increase efficiency over manual methods with hard media and much personnel. The drawback to using automated tools is the cost of sufficient licenses to support all the users of the configuration management process.

Other considerations for web applications are the necessity to baseline not just source code, but also executable code, graphic images ( GIF and JPEG files), data files, and any type of digital article necessary to create the finished system. In many cases today, content management is simply a new name for version control of web applications in production, but the tool utilized has been optimized for control and delivery of controlled items from the repository to the production server.

One special concern, due to web applications, is the need to check for the legal right to use and archive the images received for version control. With very creative and eccentric designers, copyright authorization may have been neglected. It is recommended that metadata of copyright permissions be captured with any images at the same time that they are accepted into version control. Additionally, all HTML code, should have a copyright protection statement inserted so any unauthorized use can be prohibited if internet users cut and paste the code.

## DOCUMENT CONTROL

What:
Documentation control combines procedures and tools to manage and preserve versions of all identified documents in the development process model.

Why:
Documentation control provides a means to preserve and retrieve documents used in the development model so that traceability to software baselines for decision-making is maintained and a basis for litigation and negotiation is available.

When:
Documentation control is performed whenever a document in the development model is approved by the required parties. The master of the document is archived in a controlled area and a copy is used in daily work.

Where:
Documentation control is performed through documentation control utility software located on a system that is accessible by all participants. Signed hard-copy masters are kept in a controlled area.

Who:
Documentation control is performed by all individuals who create documents throughout the development process. After the designated approval of their documents, they are responsible for seeing that those documents are placed in documentation control. The administration of the documentation control function should be the responsibility of the project configuration management analyst.

How:
Documentation control is accomplished mainly by software tools with the following capabilities:

Forward/reverse deltas
File compression
Version any type of file
Branching
Automatic branch creation in certain situations
Version labels
Visual differences
Visual merging
Parallel development
Support projects as well as files
Support multiple threading
Unlimited directory hierarchy
Recursive commands through hierarchy
Track changes through business cycle
File sharing between multiple applications

Documentation control is broken out separately in the functional model, due to tool and organization issues. In the realm of tools, it is certainly possible to use most of the version control tools on the market for document control, if they will accept the file type that the document was written in. However, some document control utilities are sold with server licenses strictly and not by user licenses, so that they may make more sense in providing a larger group of users access and/or supporting in a cost-efficient manner access to a corporate knowledge management repository. Version control tools, in contrast, are usually sold by user licenses and may be tied to other tools, that the document users may not need.

One special consideration for E-World applications is that the prospective document control system must have the capability to keep hyperlinks embedded in the documents. Other issues may be globalization concerns that revolve around the use of non-Western alphabets in web pages and documents, and the consequent need to have the ability to utilize Unicode, a new font mapping standard.

# CHANGE MANAGEMENT

What:
Change management is the systematic proposal, justification, evaluation, coordination, approval or disapproval, and implementation of all approved changes to the configuration of a system after the initial baseline of that system.

Why:
As uncontrolled change leads to chaos, controlled change enables changes to the system to be accommodated efficiently and effectively both in time and cost, and permits a planned resolution of reported defects and requested enhancements for that system.

When:
Change management may be initiated at any point in the project life-cycle after the first baseline, through the creation of some form of a change management request that is submitted to a configuration control board for resolution. Changes are always targeted against specific baselines, whether those baselines are versions of code, versions of design, or versions of requirements. It is very possible that a requested change to a code version will have a backward effect on both design and requirements. Changes include both reported defects and requested enhancements.

Where:
Change management is accomplished usually through the use of two configuration management tools: 1) a change control tracking system, and 2) a version control system. The change control system must be linked to the version control system. The change control system registers the initial appearance of the change request and routes the request to analysis review, after which the request awaits the decision of the configuration control board as to disposition. The system then routes the request to the designated implementer, records a description of the implementation, and routes the request back to the board for approval of implementation and incorporation into a release baseline. All changes must have links to specific items in the configurations captured in the version control system.

Who:
Change management involves the entire project team and clients. A change management request may be created by any member of the team or any client. The configuration control board is usually made up of the project manager, a technical lead, a configuration management analyst for support and administration, and quality assurance. The board determines the resolution of all requests and defines all releases.

How:
A successful change management system is attained through cost-effective and time-efficient processes, trained and professional personnel, and well-chosen configuration management tools. A configuration management system should not result in a burdensome bureaucracy, but in a documentation of decision paths that show that when changes were made to the system, the changes were analyzed before implementation and the implementation itself was reviewed prior to incorporation in the client load. All changes link directly to items in a controlled version system, and conversely all ensuing versions after the initial version have links to change management requests. Those links to the version control system are both links to the previous version where the defect was found or the enhancement needed, and to the new version where the change was implemented.

For web applications, with time always of the essence, a time-intensive change management system based on paper and numerous meetings is not probable and in many cases possible. Successful change management in the E-World hinges around two things. The first factor for successful change management is automating the change process using a tool with automatic notifications and forwarding once system approved decision-makers have made a choice of actions. The second factor is empowering the Project Manager ( or Product Manager depending on whether the project is a custom application or a product) to act as a one person change control board. It is up to the Project Manager to solicit and obtain the views of others towards the Project Manager's approval of changes.

## BUILD MANAGEMENT

What:
Build Management is the function that either performs or verifies the build of all configured baselines through controlled documentation.

Why:
Build management provides for traceability and repeatability for all configured baselines such that deliverables to clients are traceable to source articles.

When:
Build management is performed when any configured baseline is produced.

Where:
Build management is performed at any location whenever an original configured baseline is produced.

Who:
Build management may be performed by any designated project personnel, but the record of how that build was accomplished, as well as a documented process that details the technical steps to perform the build, must be reviewed and archived by the configuration management analyst.

How:
Build management may be accomplished by either through an automated utility that interacts with a version control tool, or through a recorded script that captures the commands and references necessary to repeat the build with no differences found in the products of builds performed separately. The script, upon version control of the baseline, would also be placed in an archive and linked to that baseline.

Build management for traditional software configuration management individuals is an unfamiliar subject, but it is a major challenge to be dealt with for web applications. The reason for this is the complexity of web applications due to the interdependencies of varied software technologies and the hardware environments they operate within. For the theoretical model a simple version description document was sufficient for a later configuration audit.

For web applications, not only must all software and hardware be listed in sufficient detail to be clearly identified, but even the sequence of how that hardware and software is set up and installed must be recorded if the successful building of a production server is to take place. For web applications that the author has been responsible for, a document defined as a "Configuration

Specification" was created that records the identification of all hardware items, the identification of all software items, the setting of any switches and breakers, all commands entered into the system, and the proper sequence of the preceding actions. This documentation is far beyond what is usually found in a Version Description Document. Furthermore, this document is validated by being utilized to create from scratch the test servers and then finally the production servers.

## RELEASE CONTROL

What:
Release control is the function that collects, documents, and transmits all deliverables to points external to the company. A configuration audit of all deliverables must be performed to ensure an accurate record of configurations and validated traceability to build procedures and system documentation. (It must be noted here that the term configuration audit is misunderstood by many new to software configuration management as a process audit. It is not a process audit, such as would be performed by Quality Assurance auditors, but a product audit against the software to be prospectively delivered to outside parties.) For web applications, the configuration audit must be conducted against the "Configuration Specification." The audit is the validation of the "Configuration Specification," and is the build of a production server from scratch. This is conducted in a controlled environment, usually a staging or test lab.

Why:
Release control ensures that traceability as to configuration, authorization, and destination for all deliverables is kept and auditable. Release control records are a starting point when answering client questions as to what configuration the client currently has when changes are discussed.

When:
Release control is performed in the life-cycle of custom code and product code as the delivery interface to the client. When changes to deliverables have to be implemented during installation, those changes must be reported to Release Control so that a true and accurate record of client configurations is kept.

Where:
Release control is performed at the company, prior to delivery to clients. When changes occur at the client site during installation, those changes are required to be reported back to Release Control for accurate recordkeeping. From release control, all changes are then incorporated into the version control and change management system. Approval is implicit due to the involvement of the project management team during client installation. An important point to make is that these five functions all support and interact with each other.

Who:
Release control is performed by a configuration management analyst who supports multiple projects for their release of deliverables to customers. The analyst performs this function, in addition to also providing administration of version control and change management systems. In the case of emergency transmissions due to the immediate needs of the client, when Release Control is unavailable, project managers are always empowered to deliver the appropriate fix to the clients. The next working day, however, copies of that transmission and the proper records must be given to Release Control so that the organization has records of that delivery.

How:
Release control is accomplished through the use of the version control utility by keeping a record of the configuration of all external deliverables. The configuration management analyst should also check for agreement of appropriate personnel as to whether the deliverables should be transmitted to the outside parties. Also, the analyst should perform a configuration audit, if one has not been performed prior to this point, to validate the agreement of software to the system documentation. This would be the "Configuration Specification" for internet applications. The audit will assure that all items of the delivery are in agreement and the traceability of the software and hardware is to the correct build procedure, i.e. the "Configuration Specification" for internet applications. Finally, hard copies of all release approval signatures and dates should be kept for audits and possible future litigation.

Release control is identified as a specific activity under the functional model due to its critical nature in the E-World. All changes to a production web application must be tracked and have traceability to controlled items in a repository, so that effective content management can occur. Due to the need to expedite defect removal and in some cases accelerate enhancements to site functionality, the project manager (or product manager) is always empowered to make decisions for releases without involving others.

## MIGRATION STRUCTURE

For any software configuration management model and to optimize the functional configuration model, a compartmentalized physical facilities structure for the controlled migration and implementation of software systems has been found to be effective, particularly so for the E-World. This migration structure for web applications consists of a defined development environment with control of the project configuration decided by the project manager, a staging environment for the testing of the developed systems with formal controls in place for changes to the system under test, and a production environment where no changes are permitted as the system operates in the critical 24 by 7 by 52 time frame of reliability and non-down time mode. As a software system moves from development to staging, entrance criteria must be met as to the completion of required documentation, a defined baseline, and the preparation of test procedures. Similarly, as a software system moves from staging to production, entrance criteria must be met as to complete system documentation, and completed test documentation that provides a basis for risk management as the reliability of the software system in the production environment.

With the total revenue stream of pure play (all business taking place on the web) e-commerce businesses dependent on their web sites, a comprehensive risk reduction strategy for down-time of the web site is a prime necessity. Consequently, hosting operators are now usually contractually liable for down-time as a result. All of this is driving more web application development organizations to a controlled migration structure for their projects.

## SYSTEM AUTOMATION

For the successful utilization of the functional model of software configuration management, most especially in the E-World, automation of the software configuration processes is essential. Providentially, numerous software tool vendors have met this need with suites of software configuration management tools. These tools are organized along the functional model and not the theoretical model, with specific tools in each suite focused respectively on version control,

documentation control (in some cases the same tool as the version control tool), change management, build management, and release control. There is no suite or individual tool that corresponds to the typology found in the theoretical model. In fact, the functional typology is what the vendors address in their specifications not the theoretical model. A prime source for information about these tools is the "Configuration Management Yellow Pages" as found on the World Wide Web with the URL of http://www.cmtoday.com/yp/configuration_management.html. Automation of the software configuration management functions provides, among other things, for the automatic requirement of approved change requests for every baseline once the initial software baseline has been achieved, the mapping of baselines to change requests at release control reviews, the linking of build procedures to every baseline, and the automatic traceability of system documentation to software baselines.

For web applications, automated software configuration management accelerates process decisions. If the decision was between a time expensive manual system or not doing configuration management, it is possible that the latter choice might be taken, even with a consequent loss of control and the lack of change records. The use of automation makes sure that the possible decision to not do configuration management is never considered.

## SYSTEM INTERFACES

Having discussed system automation above, it is necessary to discuss the need for several required system interfaces for a software configuration management system in an E-World environment. All of these interfaces must be either integral with the system or an API must be written for back and forth transmission of information. First, by linking the change management utility with the organization's e-mail system, automatic notification of required actions or decision can be made and recorded, with the change request then being forwarded on to the next decision point for further action. Second, due to the need for the smooth and expeditious reception and response to client or customer requests for defect resolution or enhancement, an API is necessary between the customer support automated software and the software configuration management tools. By this means, a customer request may be received and then transferred to the software configuration management system, with real-time status available to the customer support representative at any time for the decisions or progress made on requests. Third, the content management delivery software system must be linked to the software configuration management system with an API for the necessary adjustments of the site software in the E-World that are necessary to support new data and image needs that must be displayed and manipulated. Fourth, the system site software must be controlled through the software configuration system to avoid conflicts between operating systems and third party software that have been purchased to operate with the web sites.

## CONCLUSION

This paper has tried to discuss the recognition of a new standard model for software configuration management and how that model makes possible pragmatic software configuration management in the E-World. The paper has described the old theoretical model of software configuration management and defined the new functional model as version control, documentation control, change management, build management, and release control. Individual considerations for web applications have been reviewed for each area of the functional model. The paper has discussed

each of these areas in detail through using a “What”, “Why”, “When”, ”Where”, “Who”, and “How” typology to provide clarity and definition. Ramifications of the functional model of software configuration management have been discussed as regards to migration structure, system automation, and system interfaces that are necessary for success in the E-World. In particular, it has been noted that the functional model of software configuration management matches the organization of the software configuration management tool suites currently available for software development.

In conclusion, the paramount principle for the guidance of performing software configuration management in the E-World should be one of pragmatism. In the past, pragmatism has not been the paramount principle in software configuration management for many projects as exhibited by their paper and personnel bloat. Definition and communication should be the ultimate goals for any software configuration management activity and not barricades and forms. Through taking a functional view of the tasks of software configuration management, efficiency can be improved by seeing rational reasons related to functions for the tasks to be performed. Furthermore, scaleability can be matched to that of the development model more congruently. The E-World will tolerate no paper for paper’s sake, and so consequently a functional view of software configuration management will better ensure that the work of software configuration management always brings value.

## 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.

Peters, J. & Pedrycz, W. Software Engineering: An Engineering Approach. New York: John Wiley & Sons, Inc.

Pfleeger, S. (1998) Software Engineering: Theory and Practice. Upper Saddle River, New Jersey: Prentice Hall, Inc.

Pressman, R. (1997) Software Engineering: A Practitioner’s Approach. New York: McGraw-Hill.

Sommerville,I. (1996) Software Engineering. Harlow, England: Addison-Wesley Longman Limited.

# Software Six Sigma ($S^3$)
# Level 4 Made Easy

## Presented by:


## Mike Palmer

Software Engineering Process Group Lead, Honeywell
11100 North Oracle Road
Tucson, Arizona 85740
Tel: (520) 469-5053
Fax: ((520) 469-5065
Email: Mike.Palmer.Tucson@Honeywell.com

## Tom Lienhard

Software Engineering Process Group, Honeywell
11100 North Oracle Road
Tucson, Arizona 85740
Tel: (520) 469-5166
Fax: ((520) 469-5065
Email: Tom.Lienhard@Honeywell.com

# ABSTRACT


Software Six Sigma ($S^3$)
"Level 4 Made Easy"

**Software Six Sigma ($S^3$)?**

Many practitioners feel statistical process control cannot be meaningfully applied to software projects. This paper will clearly demonstrates that they are wrong.

**What is Software Six Sigma ($S^3$)?**

Years of software development characterized by missed schedules, cost over runs, burnt-out engineers, poor quality products, made it imperative to find a better way to produce software products.

While it is usually helpful to launch process improvement programs, many such programs get bogged down in detail. They either address the wrong root cause or they keep beating on the same solutions wondering why things don't improve.

Software Six Sigma ($S^3$) uses the traditional Six Sigma tools and applies them uniquely to the software world to understand process behavior and to bring stability, predictability, and improvement to software processes. The emphasis is on the use of statistical process control (SPC) methods

The benefits of SPC have been so evident in manufacturing that it would be foolish to ignore their potential for improving software products and processes. $S^3$ allows the software organization to realize these benefits

**What will $S^3$ give you?**

$S^3$ will demonstrate how a software organization can measure and analyze characteristics of software products and processes using SPC, so that the performance of activities that produce the products can be:

- managed,
- controlled,
- predicted, and
- improved to achieve business and technical goals.

Traditional software measurement and analysis methods, those that provide status at a point in time and compare against the "plan", are not sufficient for determining past performance or for predicting process performance. Focus with $S^3$ is toward acquisition of quantitative information and the use of SPC methods to help identify the problems and opportunities present in the process. Then the organization can confidently use the data to control and predict the process behavior and guide the improvement activities - Decisions based on data.

# Introduction

Honeywell's strategy for success starts with great people employing Six Sigma Processes which in turn will drive growth and productivity resulting in a premier company. Great strategy, as long as one not only understands what is Six Sigma but more importantly, how to apply it to the software development processes.

I have heard Six Sigma described as; a metric; a benchmark; a vision; a philosophy; a goal; a pipe dream; even a disease. Just what is Six Sigma?

In twenty words of less, Six Sigma is a customer-driven management methodology to significantly increase customer satisfaction through reducing and eliminating defects.

Peeling back the layers of the Six Sigma onion:

- Sigma is a letter in the Greek alphabet.
- The term "*sigma*" is used to designate the distribution or variation of any process or product characteristic.
- The "*sigma value*" is a metric that indicates how well a process is performing or the capacity of the process to perform defect-free work.
- The "*sigma level*" of a process is the number of standard deviations between the center of the process distribution and the closest specification limit.

Why would a company care about their sigma value? In today's world good enough is not any more. A company must run just to stand still. Competition is not relaxing, customers want better, faster, less expensive. If a company is interested in staying in the software business in the long run, they must learn to work smarter not harder.

Years of software development characterized by missed schedules, cost over runs, burnt-out engineers, poor quality products, made it imperative to find a better way to produce software products. Most software organizations are fighting fires.

If we cannot express what we know in numbers, we don't know much about it. And if we don't know much about it, we cannot control it. And if we cannot control it, we are at the mercy of chance. The key to process improvement is "Information Based on Data"

While it is usually helpful to launch process improvement programs, many such programs get bogged down in detail. They either address the wrong root cause or they keep beating on the same solutions wondering why things don't improve.

Software Six Sigma ($S^3$) uses the traditional Six Sigma tools and applies them uniquely to the software world to understand process behavior and to bring stability, predictability, and improvement to software processes. The emphasis is on the use of statistical process control (SPC) methods. The premise for Six Sigma does not differ significantly from anything heard before:

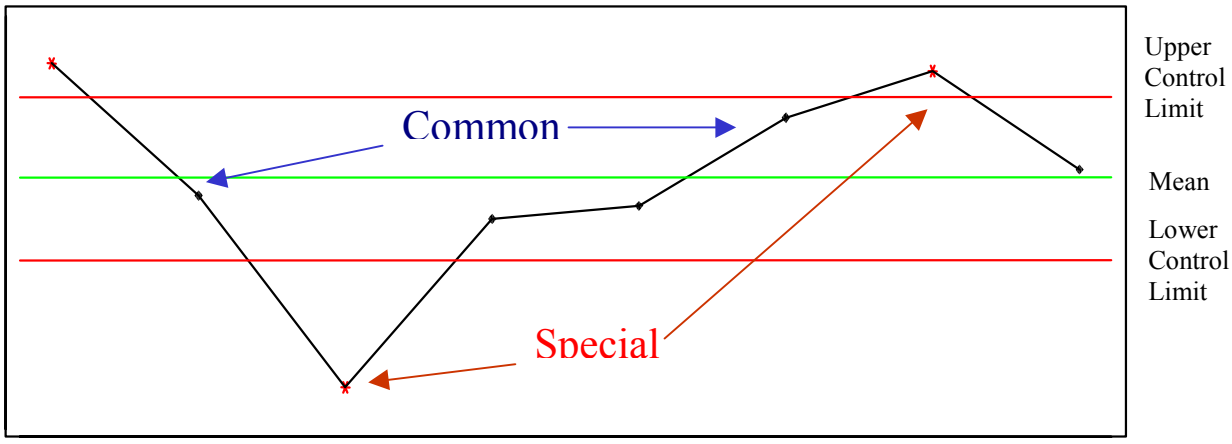
- Management involvement is key;
- Focus on the process;

- Continuous improvement is a philosophy of doing business; and
- Reduction of variation

Variation is the enemy. Variation exists in everything. It is safe to assume that everything is a result of some process, then it is rational to conclude that the variation in the process introduces variation in the product. Therefore, product variation is due to variation in the process. Sources of variation can be identified, quantified, and mitigated by control or prevention. Variation comes in two forms; common variation and special cause variation.

Common cause variation is continuously active in the process and is predictable. Special cause variation is due to extraordinary events and a specific reason can be assigned to the cause. Think about driving to work. The drive is predictable. You know it takes between 18 and 23 minutes to get from your house to the parking lot, depending on how many lights are red. It is predictable. But if there is an accident or you run out of gas it make take 45 minutes. That 45 minutes is a special cause, you can assign a specific reason to why it took so long. But a commute of 22 minutes is due to noise, there may to be a specific, extraordinary event that caused you to take 2 minutes longer than yesterday. It was within your prediction.

It is imperative to first know if variation is due to common or special cause. This is where to control chart comes into play. The control chart is the cornerstone to understanding what type of variation is influencing the process. The control chart looks like a run chart but it also has the process mean and upper and lower control limits plotted.




Control Chart

The control limits reflect the variation "built" into the process, the common cause variation. Control limits are not related to standards or specifications! They are a measure of what the process does/has done. They are "The Voice of the Process" or "The Process Capability Baseline". Control limits identify the extent of variation that

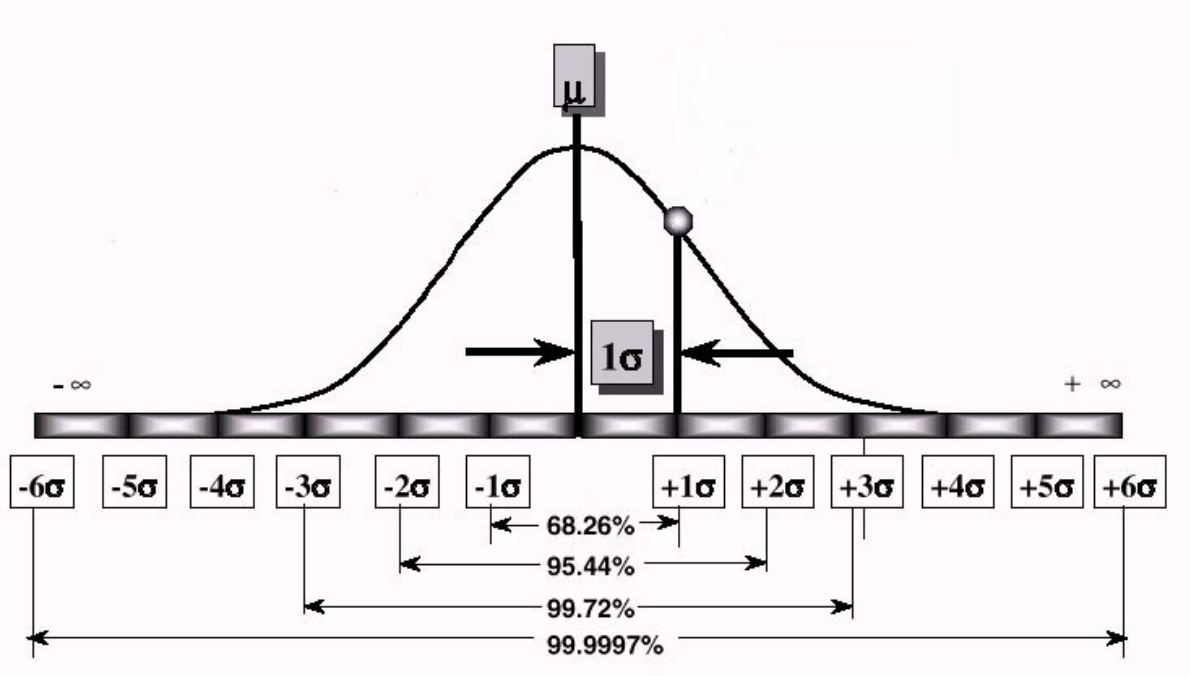
now exists so that there is not an overreaction to common causes. Extending the control limits allow predictions of process performance if no changes to the process are made.

The control chart is instrumental in identifying the type of variation within the process. This is the starting point on the Six Sigma journey, other tools in the Six Sigma toolbox will then identify where in the process allows the largest source of variation, what factors have the greatest influence o the variation, and where to set those factors to minimize the variation.

Once the variation has been identified, characterized, and minimized it can be assured that less defects will be delivered (here defects are defined as anything which may result in customer dissatisfaction) which results in a satisfied customer!

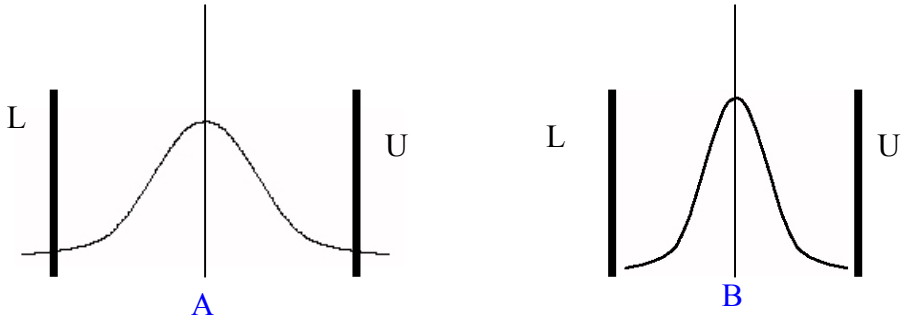
Six Sigma is based on statistical process control (SPC). Most data tends to follow the normal distribution or bell shaped curve. One of the key properties of the normal distribution is the relationship between the shape of the curve and the standard deviation. The following "rules" apply to most distributions found in the real world:

1. 60 – 75% of the data are within a distance of one standard deviation on either side of the mean.
2. 90 – 98% of the data are within a distance of two standard deviations on either side of the mean.
3. 99 – 100% of the data are within a distance of three standard deviations on either side of the mean.
4. This +/- 3 standard deviation range is used as representative of natural process variation.



Variation isn't bad if we satisfy the specification, right? The answer to this question depends on three other questions. How large is the process variation I relationship to the specification? Is the process variation due to common or special causes? And, is data collected and measured to answer these questions?

Lets look at two pilots landing on a runway. The specification is to land on the tarmac. Pilot A lands ten times and Pilot B lands ten times. All twenty landings are within specification (called a good landing). The average of both pilots is the exact middle of the runway. Their distributions are below




Both pilots are within specification and both have the same average. Which pilot would you want to fly home with tomorrow? Why?

Analyzing Pilot A's distribution shows a 4 sigma process. This means 6,210 times for every 1,000,000 landings, Pilot A will be in the dirt. At Phoenix Shy Harbor Airport, that would be 9 times a day! But he was within specification for our sample!

Analyzing Pilot B's distribution shows a 6 sigma process. This means 3.4 times for every 1,000,000 landings, Pilot B will be in the dirt. At Phoenix Shy Harbor Airport, that would be 1 time every 200 days! But Pilot B is Six Sigma, 1 time every 200 days is not good. Yes, sometimes being Six Sigma is not good enough.

Lets look at another example. A publishing house is looking at misspelled words wondering what is an appropriate sigma level.

- At 1 sigma there is 170 misspelled words on a page – unacceptable in any scenario
- 2 sigma, 25 misspelled words per page – still unacceptable
- 3 sigma, 1.5 misspelled words per page – might be acceptable for a newspaper (customers would still buy the paper)
- 4 sigma, 1 misspelled word per 30 pages – about the average paper back (I would not even be able to find the mistake)
- 5 sigma, 1 misspelled word in a set of encyclopedias – I definitely would never find this one!
- 6 sigma, 1 misspelled word in all the books in a library – overkill

A publishing house might drive itself out of business if it demanded Six Sigma for all of its publications. The customer (You and I) would not buy any more or less paperbacks with a misspelled word every 30 pages. The Voice of the Customer needs to be taken into account when setting sigma goals.

As can be seen, sometimes Six Sigma is not enough and sometimes it is too much. But the data always needs to be collected and analyzed to make the determination. The decision must be based on data.

The benefits of SPC have been so evident in manufacturing that it would be foolish to ignore their potential for improving software products and processes. $S^3$ allows the software organization to realize these benefits. But the questions always come up: Does Six Sigma replace the CMM? Can Six Sigma be used with the CMM? What CMM level would an organization have to be before Six Sigma can be used?

Six Sigma does not replace the CMM, Six Sigma is not a framework to define a process. It is a toolbox that contains various tools to solve a variety of problems. Six Sigma needs to be used with some process framework and the CMM is definitely applicable. Six Sigma can be used at any CMM maturity level assuming some underlying infrastructure is in place – more on this to come.

Lets look at how Six Sigma can be uniquely applied to software at the various CMM maturity levels. A quick reminder of the CMM levels. There are five levels in the model:

- Level 1 – Ad hoc, anything goes
- Level 2 – Best practices reside at the project level
- Level 3 – The organization has a defined common process which includes the collection of product and process measures
- Level 4 - all the process assets accumulated from Level 2 and 3 are used to quantitatively understand the software project
- Level 5 - the organization concentrates on improving the software process to gain an advantage in quality, productivity, and timeliness

At Level 1, lets assume the required infrastructure is not in place and that measures are not being collected. Therefore, only a limited number of Six Sigma tools would be applicable and none of the SPC tools could be used.

At Levels 2 and 3, the organization has the infrastructure – process are repeatable, verified, and measurements are being collected. Any metric that is being collected can now be statistically understood using a combination of tools available in the Six Sigma Toolkit. The software organization can measure and analyze characteristics of software products and processes using SPC, so that the performance of activities that produce the products can be:

- managed,
- controlled,

- predicted, and
- improved to achieve business and technical goals

As the organization moves to Level 3, best project sub-practices can be statistically chosen based on fact rather than who speaks loudest. The organization can chose the best sub-process from a variety of projects to indeed have the best process the organization has to offer.

Level 4 gets much more interesting. Level 4 has two Key Process Areas (KPA) – Quantitative Process Management and Software Quality Management. Lets look at the KPAs in detail and determine how they maps to Six Sigma and visa versa.

Quantitative Process Management focuses on removing special causes of variation from the software development process. This is an exact quote from the "Introduction to the Capability Maturity Model" class. Special causes of variation from the process, this sounds just like the control chart we heard about in the Six Sigma Overview. Software Quality Management looks at the software products quantitatively and the achievement of specific quality goals. The text in the CMM goes on to say "Projects achieve control over their products and processes by narrowing the variation in their process performance to fall within acceptable quantitative boundaries…". Again, this sounds just like what was discussed in Six Sigma and we can even replace acceptable quantitative boundaries with calculated control limits.

When performance falls outside normal range of process performance (or control limits - mapping to Six Sigma this would be a special cause), identify the reason (since Six Sigma taught explained a special cause has an assignable cause) and take corrective action when appropriate. There is definitely a direct correlation to Six Sigma with these KPAs. Historically, this is the level most people reserve for SPC. However, as has been shown SPC can be very beneficial at the lower levels, also.

Defect Prevention (Level 5 KPA) aims to control common causes of variation in the software development process. The text in the CMM goes on to say "Software processes are evaluated to prevent known types of defects from recurring". This is continuing the reduction of variation from the special or assignable causes to the common or predictable causes. Straight from the depths of the Six Sigma world. And some people think SPC is not applicable to software!

Process Change Management and Technology Change Management (Level 5 KPAs) involve continuous improvement to the software processes with the goal of improving quality, productivity and decreasing cycle time. The Six Sigma tools allow the organization to statistically analyze whether a process change was actually an improvement to the normal range of process performance or whether the change actually make the process less efficient. (I have seen this happen a few times before, maybe even been responsible for one or two)

The Six Sigma toolbox provides the tools to implement the Level 4 and 5 KPAs. A Level 5 organization is one that has implemented, among other things, statistical process control

(SPC). Now, if anybody says SPC cannot be applied to software – it is known they are wrong!

So, how can an organization start applying what Honeywell calls Software Six Sigma ($S^3$)? Three simple steps:

1. Develop Infrastructure
2. Statistical Process Control
3. Continuous Measurable Improvement

**Develop the infrastructure** – Before SPC can be applied a few prerequisites must be established.

- First and foremost, there MUST be a commitment from management. This has been stated over and over by just about anybody involved with process improvement. If there is not a commitment from management, process improvement is doomed!
- A repeatable and consistent process must be being used.
- Defined, consistent process measures
- Data collected and available from above process
- Assurance of process compliance

Is this implying that the organization must be CMM Level 3? As discussed earlier, no. But what it is saying is that for the sub-process which is desired to be under SPC all the level 3 "characteristics" must be in place. So, for example, if a level 1 organization happens to perform peer reviews which are repeatable, metrics are collected, and there is assurance of process compliance, then SPC techniques can be applied.

Why must there be this infrastructure in place? Lets assume that the process performed is not repeatable. The measurements taken would be charted in a control chart. Some points will be shown as outside the control limits. Investigation will be done to find the assignable cause. Most likely, that will be that a different process was performed. Time was wasted on a wild goose chase. The same is true if there is no process compliance verification. The assignable cause would be a different process. Lack of data is obvious.

**Statistical Process Control** – Use Six Sigma tools to identify sources of variation and then act to eliminate or reduce those sources. Measurements of process performance on project are collected and analyzed, usually via a control chart.

Statistically understand the process capability. If variation is determined to be a problem, use Six Sigma tools to discover sources of variation and to determine the magnitude of the effects due to process changes. Implement the improvements and monitor the impact, again using the various Six Sigma tools.

**Continuous Measurable Improvement** – Repeat the cycle. Measure, Analyze, Improve, Control. Can be next biggest hitter, the next quickest, the next logical problem. It can be anything the organization wants. Just don't stop the cycle. There is always something to improve. Control chart the data and see if there is a problem. If not, control chart some other data until an area that needs to be worked is identified.

**S$^3$ Toolbox** – some tools contained in the S$^3$ toolbox are described below

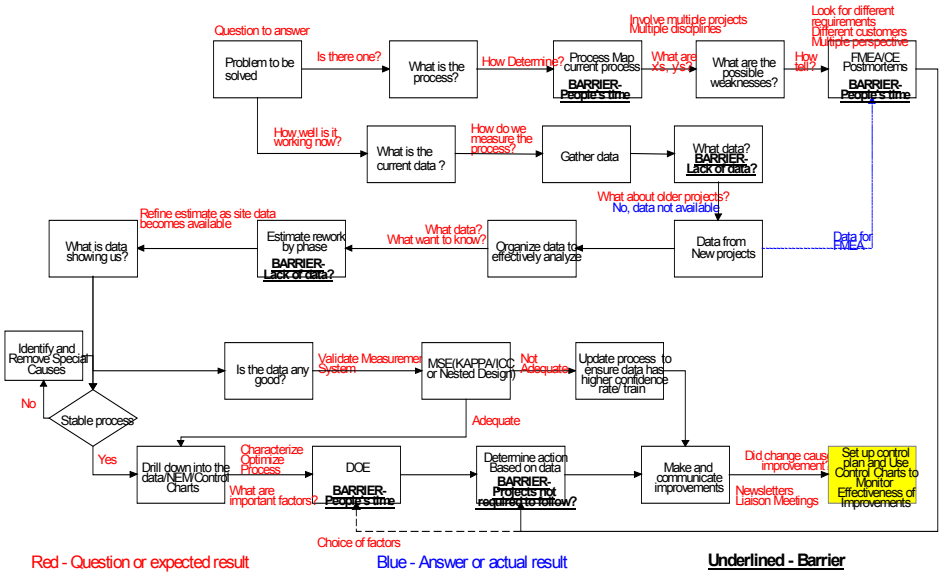
Control Chart – as mentioned earlier, the control chart will be the starting point for most of the S$^3$ activities. Control charts are techniques for quantifying process behavior and were developed by Walter A. Shewhart in the 1920s to gain control of production costs and quality. Shewart's control charts are the statistical tools of choice to se when determining whether or not a process is in control.

> *"A phenomenon will be said to be controlled when, through the use of past experience, we can predict, at least within limits, how the phenomenon may be expected to vary in the future. Here it is understood that prediction within limits means that we can state the probability that the observed phenomenon will fall within the given limits"*
>
> Walter A. Shewhart 1931

With control charts, a picture is painted that tells how the process has behaved in respect to a particular characteristic over time. By using control charts, conditions that must be satisfied for a process to be considered in control is graphically illustrated.

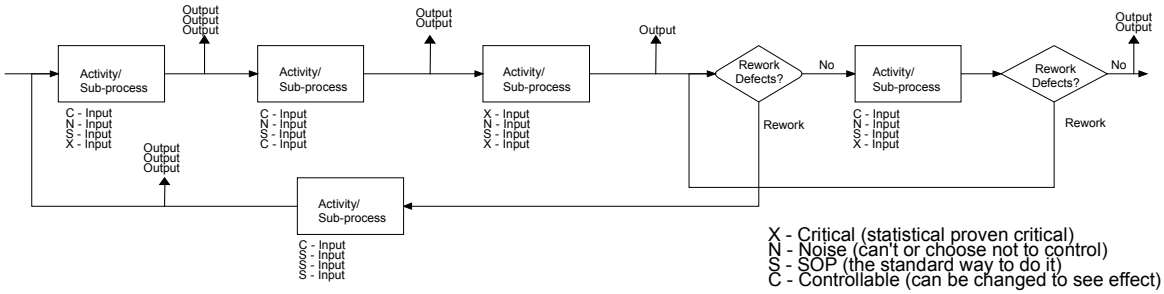
Thought Process Map – is the documented logic behind a series of decisions made to solve a problem. The TPM is used after a problem has been identified that needs to be solved. It is a tool that guides you through the Define, Measure, Analyze, Improve, Control (DMAIC) process and helps determine the specific problem solving approach to use, the resources needed, and potential barriers that will be encountered. The TPM is a living document which contains :

- questions that originated at each stage of the project,
- answers to those questions,
- techniques used to arrive at those answers, and
- barriers that were encountered.




Thought Process Map

Process Map – is a graphical representation of a process flow identifying the steps of the process, the inputs and outputs and opportunities for improvement. The PM provides a step-by-step picture of the "as-is" situation for discussion and/or communication. They highlight non-value added steps that can be eliminated or needed steps that must be added. This is most likely the first tool from the toolbox to be used after the TPM documented the problem and approach.




Process Map

Failure Mode and Effect Analysis – is a structured approach to

- identify the ways in which a product or process can fail,
- estimate the risk of specific causes with regard to these failures,
- prioritize the actions that should be taken to reduce the chance of failure, and
- evaluate the current detection mechanisms for preventing these failures from occurring.

**Process/Product**
**Failure Modes and Effects Analysis**
**(FMEA)**

| | | | |
|---|---|---|---|
| Process or Product Name: | In-Phase Peer Review | Prepared by: Tom Lienhard & Team | Page 1 of 3 |
| Responsible: | Tucson Software Engineering Process Group | FMEA Date (Orig) 23 Sept 99 (Rev) 29 Sept 99 | |

| Process Step/Input | Potential Failure Mode | Potential Failure Effects | SEV | Potential Causes | OCC | Current Controls | DET | RPN | Actions Recommended |
|---|---|---|---|---|---|---|---|---|---|
| What is the process step/ Input under investigation? | In what ways does the process step go wrong? | What is the impact on the Key Output Variables (Customer Requirements) or internal requirements? | How Severe is the effect to the customer? | What causes the process step to go wrong? | How often does cause or FM occur? | What are the existing controls and procedures (inspection and test) that prevent eith the cause or the Failure Mode? **Should include an SOP number.** | How well can you detect cause or FM? | | What are the actions for reducing the occurrance of the Cause, or improving detection? **Should have actions only on high RPN's or easy fixes.** |
| Peer Review Planning | No Review Team identified upfront w/review package(roles & responsibilities) | Product not reviewed by appropriate disciplines | 6 | SW plans do not require this | 1 | SEPG/SQA and peer review of plans | 1 | 6 | None |
| | | | 6 | Lack of process awareness | 6 | Moderator to ensure review package complete (contains reviewers) | 6 | 216 | Re-train moderator and conduct process evaluations |
| | No product evaluation criteria identified with review package | Product not reviewed to customer and/or process requirements | 9 | SW plans do not require this | 1 | SEPG/SQA and peer review of plans | 1 | 9 | None |
| | | | 9 | Lack of process awareness | 6 | Moderator to ensure review package complete (contains review criteria) | 6 | 324 | Re-train moderator and conduct process evaluations |
| | No notice or agenda with review package | Team not able to give adequate review time | 6 | SW plans do not require this | 1 | SEPG/SQA and peer review of plans | 1 | 6 | None |
| | | | 6 | Lack of process awareness | 6 | Moderator to ensure review package complete (contains notice and agenda) | 6 | 216 | Re-train moderator and conduct process evaluations |
| Product Review | Product not reviewed | Defects not found | 9 | Adequate time not given to review | 6 | Moderator and SQD ensure adequate time was given to review product | 3 | 162 | None - cultural thing |
| | | | 9 | No or inadequate evaluation criteria | 6 | Moderator to ensure review package complete (contains evaluation criteria) | 9 | 486 | Create generic checklists for site (to highest level) and conduct process |
| | | | 9 | Inappropriate reviewers | 6 | Moderator to ensure appropriate reviewers | 9 | 486 | Update command media to identify required participants on notice & agenda and conduct process evluations |
| | Metrics not captured | Organization quantitative data incorrect/incomplete | 3 | SW plans do not require this | 1 | SEPG/SQA and peer review of plans | 1 | 3 | None |
| | | | 3 | Lack of process awareness | 9 | Moderator to ensure review process followed (metrics captured) | 6 | 162 | Re-train moderator and conduct process evaluations |

The FMEA tools is best used in a proactive manner, but may certainly be used in a reactive situation. The FMEA is used early in the process improvement investigation, but after a process map has been developed.

Measurement System Evaluation – The use of this tool will quantify the quality of the measurements taken. It will show whether the observed variation is real, and how much of the observed variation is due to the way the variables are measured. Measurement system variation can make the processes appear to be better/worse than they actually are. It is important to get an accurate picture of the variation due to both the process and measurement system so the correct improvement can be put into place. The tool will indicate if the measurement system is accurate and/or precise (repeatable).

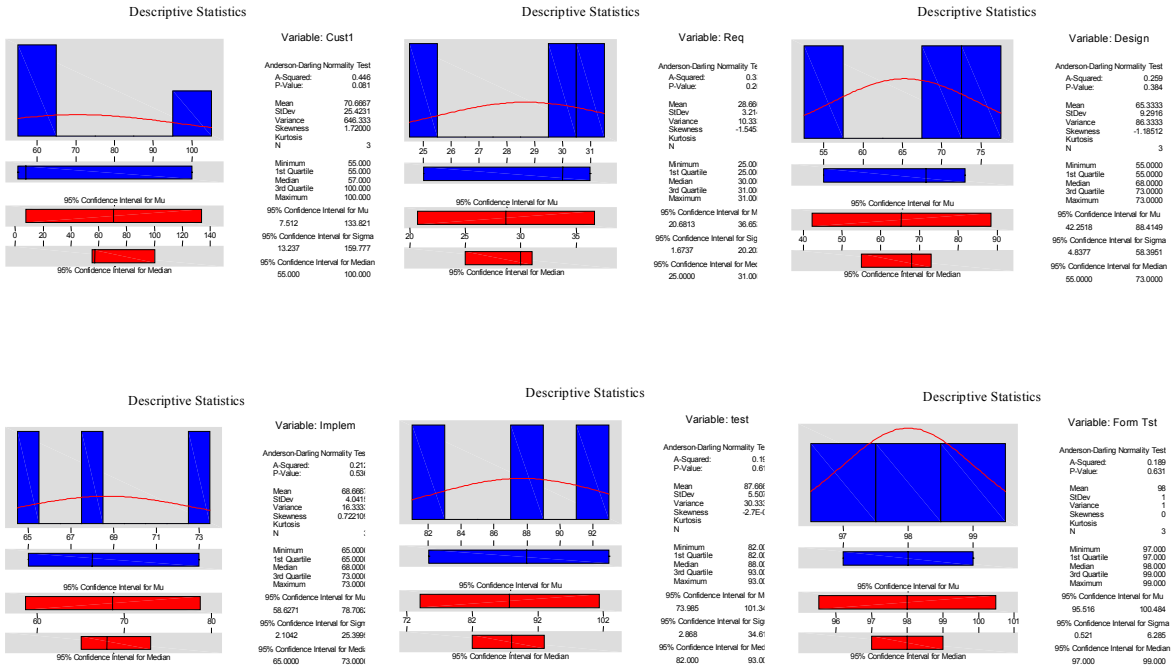
Quality Function Deployment - Often called the "House of Quality", this tool focuses on what the customer wants, not what the process provides. It is a detailed system for translating the needs and wishes of the customer into design requirements for products and services.

Cause & Effect Matrix - A simple QFD matrix, used to ensure that the customer requirements are considered in determing the key characteristics of the product/process. It relates the Key Inputs to the Key Outputs (customer requirements) using the process map as the primary source of input information. Key Outputs are related in importance in importance to the customer and Key Inputs are scored as to their relationship tot he Key Outputs.

Multi-Vari Studies - Characterizes the baseline capability of the process (a quick snapshot of the process). Determines whether major variation is positional, sequential, temporal or due to continuous inputs.

Components of Variation - Quick and effective method of determining where the major sources of variation are to determine where to go work first. Tis tool will quantify how much variation is present at each level in the process hierarchy.

Confidence Intervals - This tools will determine the likelihood that the sample mean and standard deviation (distribution) will be exactly the same as the true population. Confidence intervals quantify our uncertainty and provide a range of plausible values for the population parameters

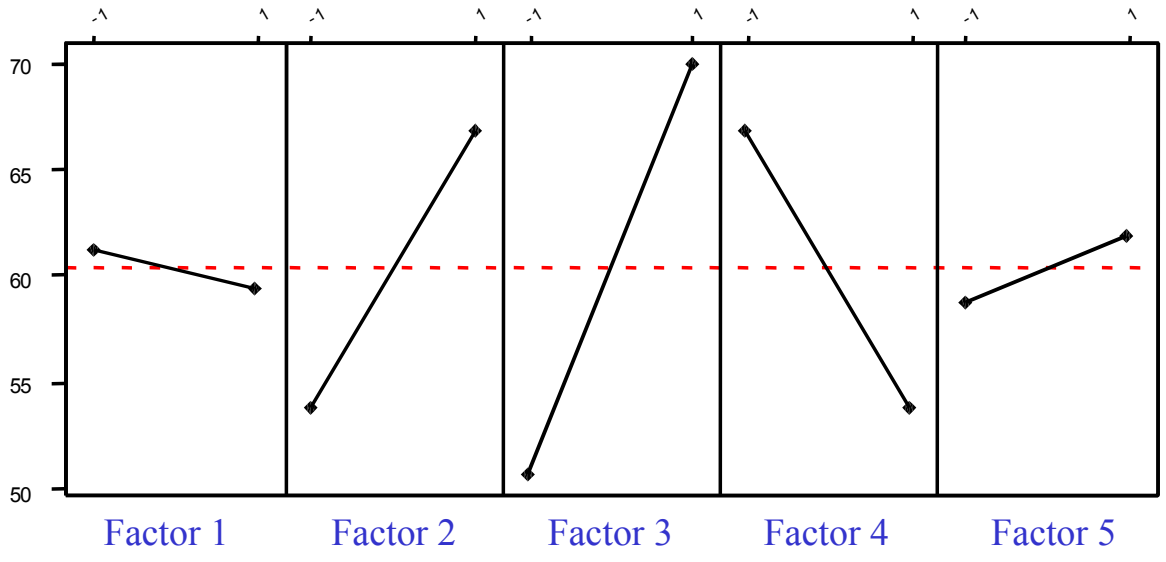



Confidence Intervals

<u>Design of Experiments</u> – A systematic series of tests in which various input variables are directly manipulated to "kick the process" and the effects on the output variables are observed. This tool will determine:

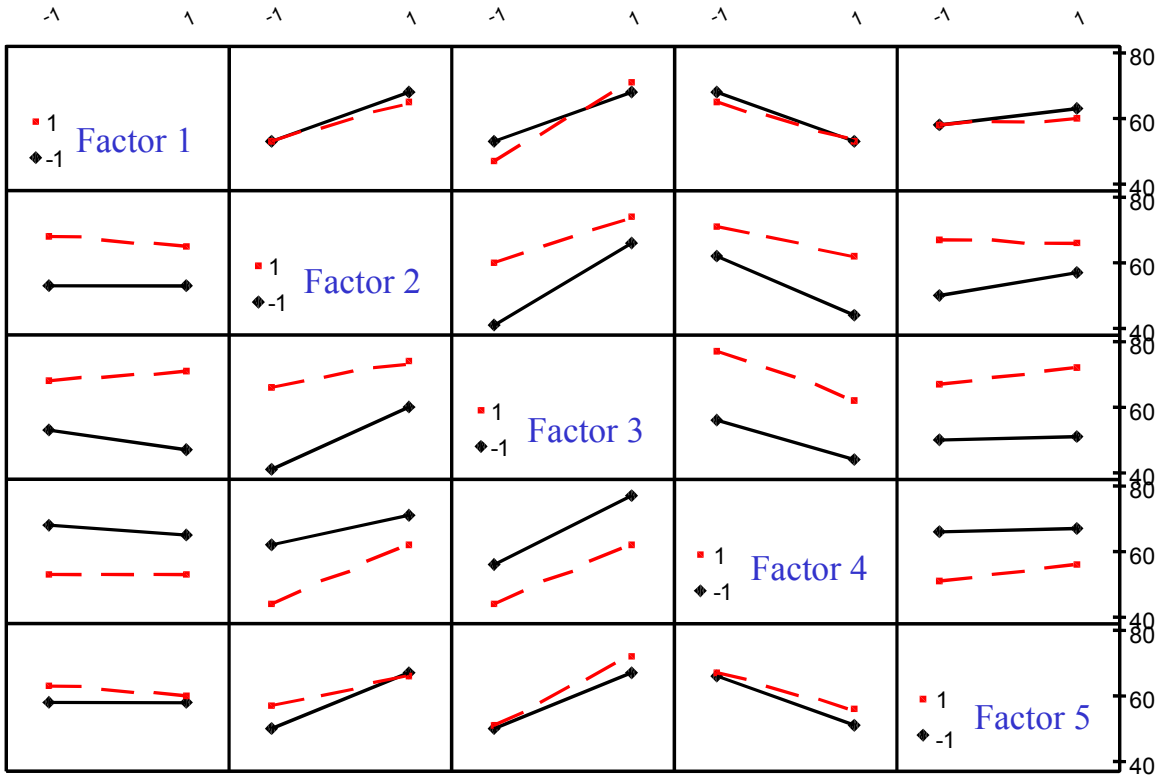
- inputs which influence the output,
- where to set the influential inputs to center the output on the target,
- where to set the influential inputs to minimize the variability of the output, and
- where to set the influential inputs to minimize the effect of the noise variables.

<u>Design of Experiments; Main Effects Plot</u> – This tools will show what impact the inputs have on the output.




Main Effects Plot

Design of Experiments; Interaction Effects Plot – This tools will show what impact any interactions between the inputs have on the output. (Two factors interact when the effect that one factor has on the output depends on the setting of another.)




Interaction Effects Plot

**Conclusion**

Traditional software measurement and analysis methods, those that provide status at a point in time and compare against the "plan", are not sufficient for determining past performance or for predicting process performance. Focus with $S^3$ is toward acquisition of quantitative information and the use of SPC methods to help identify the problems and opportunities present in the process. Then the organization can confidently use the data to control and predict the process behavior and guide the improvement activities - Decisions based on data.

As was demonstrated throughout this paper, SPC can be applied to the software development process, aligns itself with the SEI CMM, and will provide valuable information for all process improvement activities.

**Title:** **A Practical Roadmap for CMM implementation**

**Abstract:**
The paper captures the mechanisms evolved in a global Information Technology (IT) services company to achieve compliance to the Software Engineering Institute's Capability Maturity Model (SEI-CMM). A roadmap is provided to facilitate institutionalization of CMM Key Process Areas (KPA).

The paper also describes the transitioning of a software development team from the ISO 9000 model to SEI-CMM model using a building-block approach that synchronizes the organizational business objectives with the path of continuous improvement. The various methods and metrics that ensured CMM level IV-compliant practices within a Quality Management System (QMS) defined for ISO 9000 standards are explained. The role of the Software Engineering Process Group (SEPG) and quality team in initiating, defining and supporting a change of practitioner mindset from a task- and delivery- oriented mode to a quality driven mindset is discussed. The specific challenges faced in fostering a quality mindset at various management and practitioner levels and the solutions that worked are enumerated. The possible contributors to the (varying degrees of) process compliance and key motivating factors are analyzed. The objective is to ensure that the discussion helps identify possible impediments to the quality journey and pointers to solutions. A comparison of the CMM model and ISO 9001: 2000 (Draft International Standard to be released towards the end of 2000) is made where appropriate. This is to enable software organizations define a QMS that complies with both ISO 9001:2000 and CMM models.

**Author(s):**

**M.S. Ramkumar**
Asst. Manager – Quality, HCL Technologies
Email address: Ramkumar@hclt.com
Work Phone: 91-44-8290755
Dr M.S. Ramkumar is a Doctorate in Neurobiology and has six years of IT experience in areas pertaining to QC and QA. He is a CQA, ISO 9001 Lead Auditor and trained in CMM methods. He is part of the Quality team responsible for Quality system definition and implementation.

**K.R.Gopinath**
Quality Manager- HCL Technologies
Email Address: krg@hclt.com
Mailing address: 50 – 53 Greams Road, Chennai – 600006, India
Work Phone: 91-44- 8290755
K.R. Gopinath is a postgraduate in Physics with over 19 years of experience in process control systems, networking and software engineering. He heads the Organizational Quality Team and the SEPG leading the CMM initiative. He is a CQA, ISO 9001 Lead Auditor and trained in SEI-CMM methods and holds ASQ membership since 1995.

Mail address: 50 – 53 Greams Road, Chennai – 600006, India

## 1 Introduction

The Software Engineering Institute's Capability Maturity Model (CMM) is fast becoming the Industry standard for process improvement in organizations providing Information Technology (IT) products and services.There are many authoritative reports available (Paulk 93, Paulk 94) which map the similarities and differences between the CMM and ISO 9001 models. While there is a strong overlap between them there are also areas that are addressed in a detailed manner in one model but not the other. The ISO model addresses control of customer-supplied product and handling ,storage, packaging, preservation and delivery in greater detail than the CMM. ISO 9001:1994 standards implemented in letter (and not fully in the spirit) seem to address the minimum requirements of an adequate quality system with a focus on preventing non-conformities rather than ensure improvement. The CMM focuses in greater detail on continuous process improvement using quantitative means to verify the quantum of improvement in various management and technical areas.

This paper is based on the real-life experiences of a large IT service provider (seven hundred plus employees) catering to multiple global customers straddling diverse business domains. The division taken up for discussion is organized as vertical Lines of business, each catering to a specific technology area, e.g. Automotive Software, Telecom, Client Server etc. The various constituents had a chequered history regarding process focus with one of the groups being certified as possessing ISO 9001-compliant systems, while others followed ISO 9001-compliant processes without being formally certified. Once the Upper Management mandated the SEPG to ensure compliance with the CMM model the challenge facing the Organization was to propel groups with varying levels of process "maturity " to a common, higher level of process rigor.

The challenges that were converted into opportunities and the mechanisms evolved to accelerate the Quality journey have been captured in this paper to provide inputs to other Organizations traversing a similar path to higher levels of the CMM. The approach is also relevant to Organizations without a formal ISO certification that are interested in a CMM –Based Appraisal for Internal Process Improvement. The ISO 9001:2000 standard expected to be released around the end of 2000 also exhibits a significant level of convergence with the practices mandated by the CMM for software development organizations. The authors have also highlighted such areas in order to facilitate the development of a quality system that would comply with both models using the same building blocks.

## 2 Challenges to process compliance

The challenges faced by a software development organization with demanding schedule and delivery objectives are common irrespective of the domain or technology platform. These commonly include

- Ownership for quality activities and process improvements are laid at the door of the quality Department. This is also true for a ISO 9001-certified group where the quality team essentially drives the process effort. In CMM parlance this situation shows the absence of an "institutionalized" process.
- Quality-related activities are often limited to the mandatory processes defined by the quality management system and improvement efforts are isolated phenomena executed using managerial "discretion"
- The developers consider process related activities as an overhead and a "necessary evil" to be performed once the "real" work of development is complete
- The project managers face a virtual oxymoron of on-time, high-quality delivery

## 3 Approaching the Challenges

The Process Improvement effort was mounted through a two-pronged approach comprising
a) Institutionalizing the process mindset and
b) Improving the Software Engineering Processes.

### 3.1 Institutionalising the process mindset

It was realised (based on earlier experiences) that direct championing of the process improvement initiatives by the upper management was vital to ensuring the momentum of the quality journey. This section details steps taken to prove to the rank and file that the management commitment to quality would translate to vigorous action on all fronts.

- Top Management exhibited direct and visible commitment to the quality journey. Project reviews by upper management which were earlier held only for firefighting purposes were replaced by planned management reviews of all projects. These reviews focused on the execution of project plans including adherence to commitments given to external and internal customers, process rigor in terms of peer review and testing activities, customer feedback if any, and issues pertaining to lack of resources (infrastructure and personnel)
- A massive organization-wide training and reorientation effort was initiated to involve and align employees at all levels with the organizational goals This training was aimed at introducing the CMM model and explaining how the QMS was designed to meet the requirements of the model.
- Ownership of "Quality " was shouldered by frontline managers at all levels with the quality team being the facilitators for the journey.
- Information sharing methods were formalized to ensure that best practices are shared rapidly and implemented consistently.

### 3.2 Improving the Software Engineering Processes

While one part of the challenge was to bring about an attitudinal change the other task was to provide alternate, quality-centric methodologies for performing the development tasks in a more efficient manner.

- The ISO 9001-certified group took up process improvement steps that went beyond the "expected levels" in terms of strengthening metrics, requirement management and Software Quality Assurance (SQA) focusing on defect prevention
- It was decided to leverage on practices used in this group to identify gaps with respect to CMM level IV requirements.
- Dedicated groups were identified for enhancing the usage of automated tools for testing, configuration management etc. that would help accelerate the evolution to a higher level of process maturity.
- The use of process automation tools helped institutionalize a strong and tightly controlled process especially in areas like configuration management and testing. This helped in two ways: firstly, the process could be executed only in one and the correct way ; and manual reporting could be avoided because of the built-in features present in several commercially available tools
- A major mindset change was encouraged when testing by an independent SQA team was initiated
- One of the development team members (in each project) was designated the project software quality analyst and was responsible for ensuring process adherence and process improvements within the project group.
- Technical Working Groups (TWG) were set up with members from different domain areas who could contribute to the Organizational Process Definition and provide inputs for tailoring the process. This helped foster interaction between Managers spanning several technical area and helped nurture a sense of "ownership" over the process improvement initiative.

## 4 Key Motivators for Process Improvement Efforts

For the upper management : The upper management normally views all decision through the prism of growth and/or revenue. Any process improvement effort to gain support at this level needs to be presented in terms of productivity gains and product quality improvements. The successful growth and revenue generation of the ISO 9001 certified group exploded the myth of the quality-productivity tradeoff.
For the middle management (including frontline project management) : This group is very crucial to the success of any process improvement effort. They are driven by objectives defined by the upper management. The visible commitment of the upper management and an opportunity to "own" the process improvement effort provided the right ambience for this group to participate wholeheartedly in the Quality journey. The success of the project teams on the process front was one of the key result areas for the managers to deliver.
For the practitioners : This is the group that (prior to proper orientation) often perceives process initiatives as "additional work". Our approach was to prove quantitatively (using data from the team with ISO 9001

certification) using productivity and rework data , that process improvements in fact save time and reduce firefighting. One additional factor that helped was introducing elements of the Personal Software Process that helps overcome the "what's in it for me? " factor. The Personal Software Process(PSP), developed by Watts S. Humphrey, provides a roadmap for the individual developer to scale increasingly higher peaks of performance. This is done by enhancing the ability of the developer to estimate the task at hand, plan well and execute with discipline using sound quality management principles.The training programs on the CMM also included inputs on how such a process focus would ensure increasing efficiencies for the individual developer.

## 5 Gaps Identified between ISO 9001 and CMM Level IV

The gap analysis was performed using a detailed questionnaire which tracked various project activities against the key practices required at various levels of the CMM. . (The entire questionnaire can be provided by the authors on request) .
Usually, the gap analysis is performed by the quality team (with or without an external consultant) and the findings are communicated to the rest of the organization.
In our case the exercise was performed in two steps.

- At the project/ group level the project Managers and senior team members were asked to answer the questionnaire. This helped bring the awareness within the team of areas in which corrective actions or other steps were required.
- The quality team performed a "formal" gap analysis with an external consultant by mapping the practices of the ISO 9001 certified group with CMM Level IV requirements.

The formal gap analysis revealed the following major areas for corrective action :

1. The organizational policy had to be formally defined and communicated for all KPAs.
2. The peer reviews area had to be formalized and implemented consistently.
3. Existing metrics programs had to be synergised into an organizational program.
4. Formal SQA procedures for all project deliverables had to be implemented consistently.

In addition, fine tuning had to be done at the practice levels of some key process areas to ensure compliance to Level IV Requirements
The various areas that require strengthening in an organization with ISO 9001-compliant processes are listed in the table below. Please note that these have been identified for a team with a customized QMS and may require some customization to suit other organizations hoping to achieve a CMM Level IV-Compliant process.

We have provided comments on how some of these gaps would be addressed by the ISO 9001:2000 standards to be released by the end of 2000. These inputs are based on the Draft International Standard (DIS) available currently and could undergo minor changes in the release version of the standards.

### 5.1 Level 2 KPAs

(Note : Software Subcontract Management has been left out since our Organization does not subcontract development work)

| KPA | Suggested Areas of Improvement | Areas addressed in ISO 9001:2000 (Draft International Standard) |
|---|---|---|
| **Requirements Management** | • Training in RM Processes and RM Audit<br>• Senior Management review of Change request<br>• Guidelines for RM, Impact Analysis, Support Group Involvement | • Reinforces role of upper management in ensuring customer and statutory requirements are met with a view to providing customer satisfaction<br>• Section 7.2 provides details of areas to be covered in requirements management |
| **Software Project planning** | • Proposal Review Checklist for all items listed in respective practices<br>• Support groups to be involved in reviews<br>• Documented procedures for size, effort and cost estimates and schedule estimation<br>• Support group plans to be negotiated and agreed to<br>• Data to be captured on time spent in planning activities<br>• Guidelines to include assumptions | • Upper management is responsible for ensuring tht quality levels and objectives are established at appropriate levels in the organization<br>• Section 5.4 mentions planning for resources, quality management procedures and their continual improvement<br>• Section 8.1 on Measurement, analysis and improvement mandates the definition, planning and implementation of measurement activities to assure conformance and achieve improvement |
| **Software Project Tracking and Oversight** | • Detailed procedures for tracking size, effort, cost, etc<br>• Event-driven formal project reviews at selected milestones<br>• Data to be captured on effort spent on tracking activities and changes to Software Development plan | • Section 7.0 on Product Realisation details various tracking activites to be performed at various phases of the product development<br>• All activities need to be formally reviewed and approved<br>• Section 5.6 makes Management Review a mandatory activity to ensure continual process improvement<br>• Data on various activities also needs to be captured to verify if processes are fulfilling their objectives and resultant product conforms to the defined requirements |

| | | |
|---|---|---|
| **Software Quality Assurance** | • Clear definition on the role of SQA group<br>• Training for SQA roles<br>• Documented procedure for SQA Plan<br>• Audit of Software Work products<br>• Interaction with Customer SQA to be planned<br>• Periodic audits of SQA function | • Section 7.0 on Product Realisation has mandated the formal definition of acceptability criteria and calls for the definition of verification and validation activities for both process and product<br>• Section 7.2.3 delienates areas where formal communication with the customer is required and how customer feedback including complaints needs to be addressed<br>• Section 8.2.2. provides details on the scope and objectives of Internal Audits |
| **Software Configuration Management (SCM)** | • SCM policy to be formally stated and mention repository for storing Configuration items<br>• Documented procedure for preparing the SCM Plan<br>• Audit of SCM group to be formalized | • Section 5.5.6. mentions control of documents including review and approval prior to release<br>• Section 5.5.7. mandates control of Quality records<br>• Sections 7.5.2. and 7.5.4 mention details of Product identification and traceability |

### 5.2 Level 3 KPAs

| KPA | Suggested Areas of Improvement | Areas addressed in ISO 9001:2000 (Draft International Standard) |
|---|---|---|
| **Organization Process Focus (OPF)** | • Procedure for identification and dissemination of best practices from different groups of the organization<br>• Periodic assessments to be made on the effectiveness of the Organizational process and corrections made as required<br>• Metrics on CMM time spent, milestones, project compliant status etc. to be collected and sent to Senior Management | • The upper management is mandated with monitoring the effectiveness of Organizational processes and ensuring continual improvement (section 5.6- Management Review) |

| | | |
|---|---|---|
| **Organization Process Definition** | • Guidelines for developing software process assets including change management and configuration control<br>• Descriptions of life cycles to be documented and maintained under configuration control<br>• Design of software process database and keeping under configuration control to be formalised<br>• Tailoring Guidelines<br>• Audits to ensure usage of standards and controlled access of Process assets | • Covered by section 5.6 on Management Review. Outputs of review to ensure that QMS is effective<br>• Section 6 mentions that adequate resources should be provided to implement and improve the QMS |
| **Training program** | • Policy for TP to be drafted with a documented process for developing a Training Plan<br>• Organization-wide Course Design standards to be developed<br>• Waiver procedures to be developed<br>• Skills database to be used as input for resource allocation<br>• Training activities to be reviewed by Senior management | • Section 6.2 dealing with Human Resources provides details on how personnel need to be qualified on the basis of education, training, skills and experience<br>• The organization also needs to address competency needs of personnel and provide training, the effectiveness of which shall also be monitored |
| **Integrated Software Management** | • Policy to cover Tailoring and process assets<br>• Estimation guidelines to include use of Software process assets/database<br>• Risk Management to be formalized and contingency plans to be developed<br>• Software Plans and work products to be traceable to Software Requirements so that consistency is ensured<br>• Metrics to be collectedon both functional and technical Software process engineering activities<br>• Checklist to be prepared for facilitating QA audits | • Addressed by the Management responsibility section |

| | | |
|---|---|---|
| **Intergroup Coordination** | • Policy statements to cover all affected groups<br>• Team building and other soft skill trainings to be imparted for all Project leaders and above<br>• Work products from other groups to be reviewed based on developed acceptance criteria<br>• Metrics to include time spent on Intergroup Coordination activities<br>• Audits to cover process for negotiating critical dependencies and handling of issues. | • Section 7.3.1. mentions that interfaces between different groups shall be managed to ensure effective communication and clarity regarding responsibilities |
| **Peer Reviews** | • Training to be a pre-requisite for leading a review<br>• Peer Review process to be strengthened to include development of checklists and tracking corrective actions to closure. | • Section 7.0 on Product Realization mandates a review for every phase in the product development<br>• The review process also need to be verified for effectiveness |

### 5.3 Level 4 KPAs

| KPA | Suggested Areas of Improvement | Areas addressed in ISO 9001:2000 (Draft International Standard) |
|---|---|---|
| **Quantitative Process Management (QPM)** | • Written policy for Quantitative analysis of all Organizational functions<br>• Projects to have QPM plan as per a documented procedure<br>• Milestone tracking and time spent on QPM planned vs actual to be collected<br>• Audits to cover QPM activities | • Section 8 on Measurement, analysis and improvement mandated collection of data and its analysis using applicable methodologies including statistical technologies<br>• Such analysis needs to validate the effectiveness of defined processes in meeting their objectives |
| **Software Quality Management (SQM)** | 1. Documented procedure for develping Software Quality Plan<br>2. Product Quality goals to be defined, monitored and revisedcontinuously<br>3. Also to be reviewed on event driven bases, analyzed and corrective action taken<br>4. Senior management reviews to include SQM reports | • Product quality data also needs to be measured at every phase of development to ensure that they meet the acceptance criteria<br>• Objective records need to capture details of the above including authority responsible for product release<br>• Release of a non-conforming product is not permissible except with the formal approval of the customer |

## 6 Action Plan for Institutionalisation- Activities

a) **Closing the Process Chasm –** This activity largely consists of performing the gap analysis to identify the gaps between existing organizational processes and requirements of the CMM Model. Once the gaps are identified Technical Working Groups were set up to define improved procedures that ensure all the gaps are plugged.

b) **Metrics Data** – This activity consists of collecting data from completed projects on details of effort, cost, schedule variance etc. Data also needs to be collected on the execution efficiency of various processes used for software development. This activity has to start early in the improvement effort in order to ensure the collection of sufficient data.

c) **Building the process scaffolding-** The degree of process Insitutionalization in an Organization is largely dependent a support structure that nurtures a quality –first approach. The elements of this structure include Organizational ownership of the process by upper management and SEPG. A crucial differentiator is the change from management by deadlines to management by data. This ensures that project tracking is proactive rather than reactive.
Quality champions, well-versed in process and CMM were selected from across the Organization and function as internal consultants for the CMM initiative. A massive training program was conducted to convey the redefined working model of the organization. Those employees required to play specific roles such as SQA, internal auditors etc underwent specialized training in these activities. Upper and middle management received orientation on soft skills required to foster change at all levels. Practitioners received specific training on project –related areas and peer review procedures . In the case of the latter a formal software inspection method was implemented for two reasons: it provided a high-visibility impact at the practitioner level and encouraged a positive attitude towards process improvement efforts.

## 7 Benefits obtained

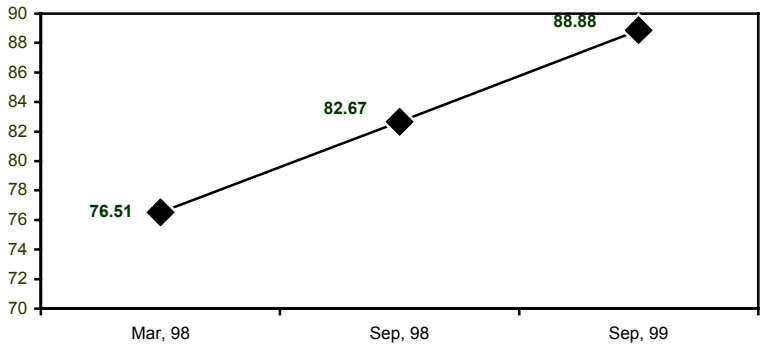
The changes perceived by various section of the organization and customers are as follows-

Employees : Employees at all levels including frontline managers underwent a massive training program. The logistical challenges involved in training a group of over seven hundred employees were huge. These were overcome by having a phased training mode that covered the group over a five month period. The practitioners and frontline managers received on an average about four days of focused training that covered details of CMM model, changes to organizational procedures . The organization invested in about 1480 person days of training over a six month period.

Upper Management : Management reviews became proactive and covered details of product quality, process compliance, opportunities for improvement.
The table shown in Annexure I was used to generate data on process compliance within a project which was then reviewed by the senior management.
The trends revealed by the data was validated by the findings of internal audits which were conducted every month. The followup was in the form of specific process improvement plans that were used to target areas where impact on customer satisfaction were highest.

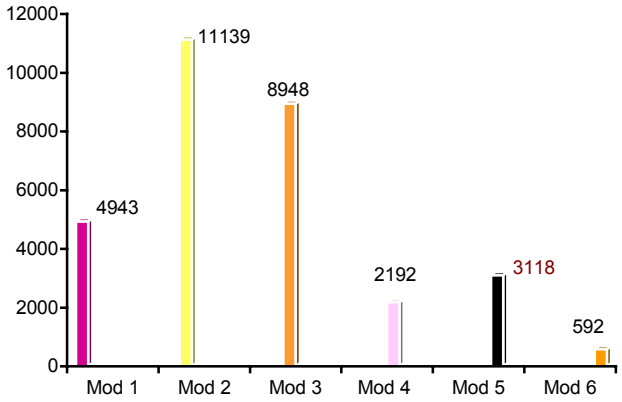
Customers: The process improvement efforts were also vindicated by customer perception of the organizational ability. One of our customers had in place a rigorous audit procedure using which a supplier's ability to provide high quality products and services were evaluated. The evaluation paradigm included weightages for quality of product, process capabilities of the team, cost of the engagement, logistic support available and customer satisfaction.
Our organization was audited by this customer thrice over a eighteen month period and the final rating improved from 76.51 to 88.88 over the time (see figure below).
Consequent to this improvement the Customer reduced the frequency of audits to once a year instead of the earlier practice of an audit every six months. This also lead to an improvement in

the quality of work obtained from customers which took the form of more development projects than maintenance activities.
The performance improvement obtained convinced upper and middle management about the benefits of the process improvement initiatives.

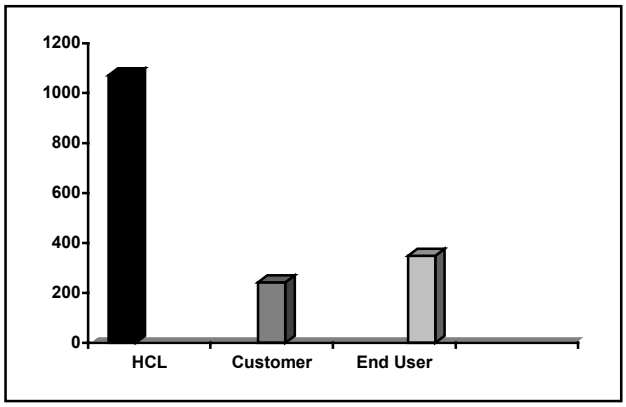



**Improvement in performance rating based on Customer Audit**

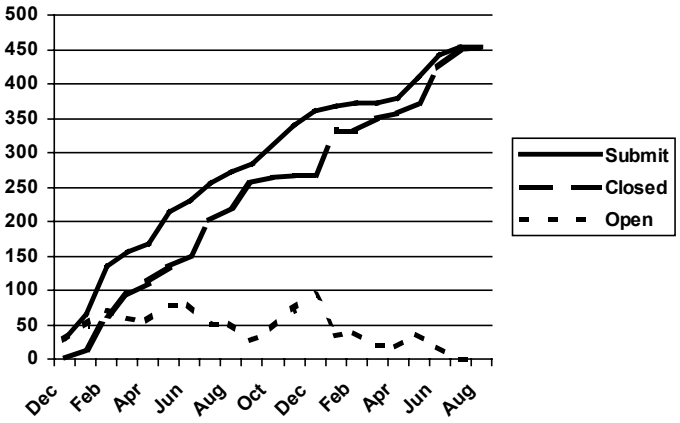
Improvements seen in Project Execution
We have provided data collected from a reference project to provide a glimpse of the nature of improvements obtained. This project commenced before the ISO 9000 and CMM initiatives were formally implemented. At the outset the project execution was beset with problems including lack of requirements clarity and absence of a coherent process.
This project spanned about twenty eight months with an average team size of about twelve team members. The final product delivered was about 60,000 Lines of Code (LOC) . This product has been in the field for about six months now without any major problems resulting in high customer satisfaction.The details of the submodule size are shown in the graph below.

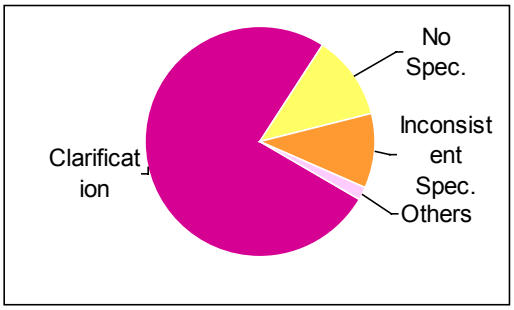



The tracking of defects in the project revealed that the quality system had succeeded to a large extent in providing a high quality product. While a majority of defects were captured prior to delivery others were detected by the customer and end user. The analysis of the latter revealed that the roots for these errors lay in the incorrect and/or incomplete capture of requirements. In order to eliminate this problem specific training needs were recognized and addressed. The graphs below provides details of defects identified by the development organization, customer and end user respectively.




Apart from the defects there were issues discussed with the customer where requirements lacked clarity. These led to frequent discussions with the customer and resulted in several changes which had to be controlled and implemented after due impact analysis. The graph below (left) captures the trend of issues throughout the life cycle of the project. The graph below (right) provides details of the issues as identified by subsequent analysis.




As the above details emphasize the process improvement initiative using the ISO 9000 and CMM models succeeded in not only institutionalizing best-in-class project management processes but also resulted in visibly improved customer satisfaction levels even during the execution of the projects. The organizational morale and enthusiasm for continuous process improvement increased tremendously .

## 8 Keys to Success

The following would appear to be pre-requisites for a successful attempt at institutionalizing a CMM-Level IV compliant process in an ISO 9001 compliant organization :-

- Targeting the Level IV requirements through formal management commitments in terms of resources , orientation and tracking.
- Ensuring a reasonable time frame with enough stretch required from the Organization to make it a challenging task
- Ensuring processes have specific goals that are tied-in to the business objectives of the Organization
- Ensuring that the redefined procedures are simple and unambiguous with metrics being used to verify the attainment of objectives
- Using tried and trusted project management methods to plan and track the CMM initiative. This would ensure that the implementation process is under senior management scrutiny and any mid-course correction is quickly carried out to ensure that Level IV compliance is institutionalized.

The flow chart in Annexure 2 depicts the sequence of steps that were performed to transition from ISO 9001 model to a CMM Level IV-compliant process model.

## 9 Summary

The impending changes to the ISO 9001 standards (ISO 9001:2000) considerably narrow the gap that were perceived to exist between ISO 9001 and CMM models for a software organization. The implementation of the ISO 9001:1994 standard in the letter rather than the spirit has undoubtedly contributed to this negative perception of the ISO model. The new ISO standard also reflects a change of focus to ensuring customer satisfaction and working for continual improvement. This marks a departure from the earlier perception that the focus is on prevention of non-conformity to the exclusion of all else .

The new ISO 9001 standards place specific emphasis on a ” process approach “ supported by four pillars of management responsibility, resource management, product realisation, and measurement, analysis and improvement .These are in remarkable coincidence with the common features of the CMM with emphasis on commitment to perform, ability to perform, activities performed, and measurement and analysis. This convergence between the models would greatly help organizations that may require to follow both models for strategic reasons as well as customer requirements. Finally, both models are an eloquent testimony to the vision of Dr W. Edwards Deming whose principles find a clear echo in the basic concepts underlying both models. These include focus on defect prevention, continuous improvement through process refinement and training for all levels to ensure systematic approach to knowledge management in the ever-learning organization.

## Annexure 1

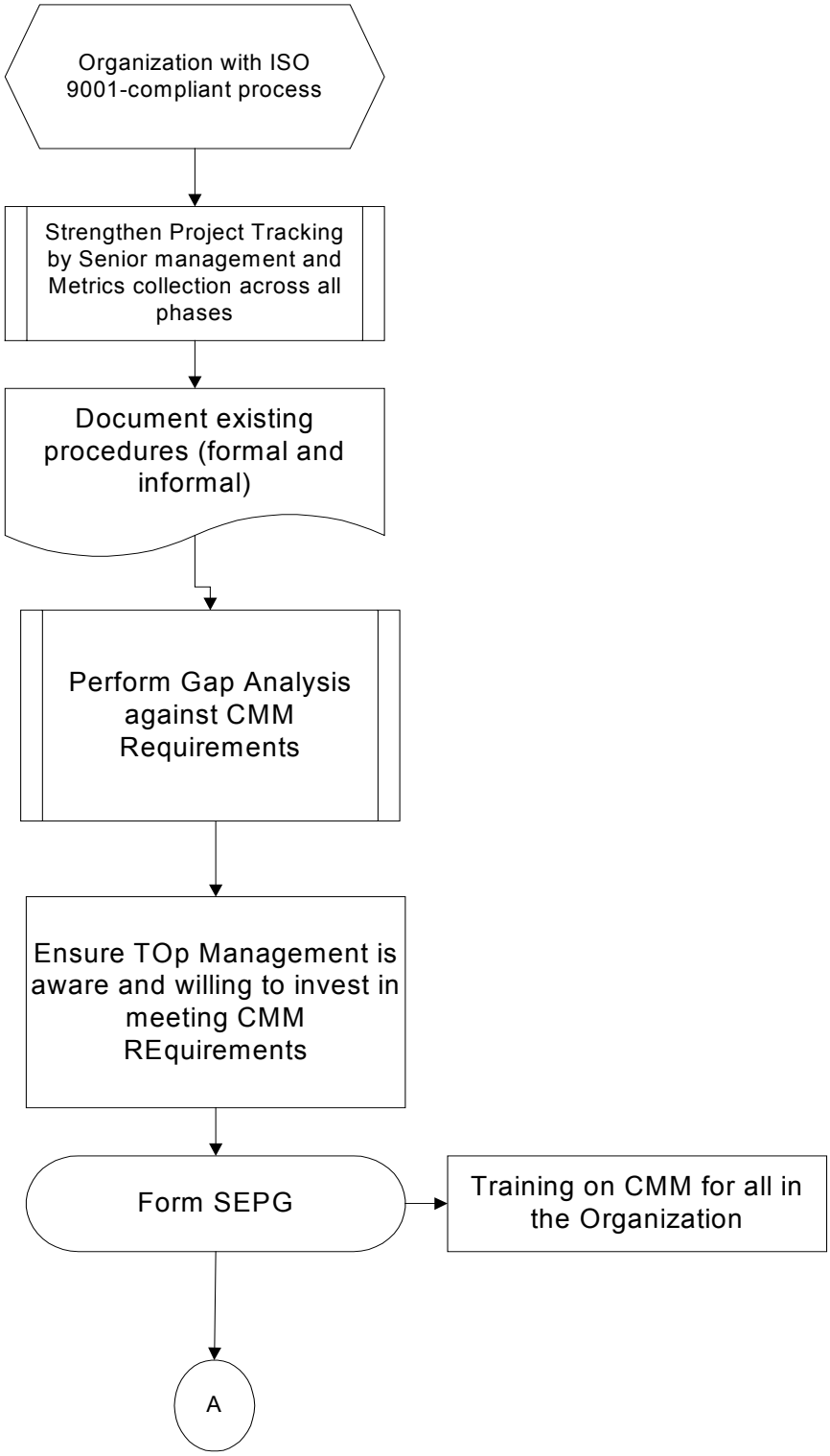
**Process Review template used for Upper Management Review of project**

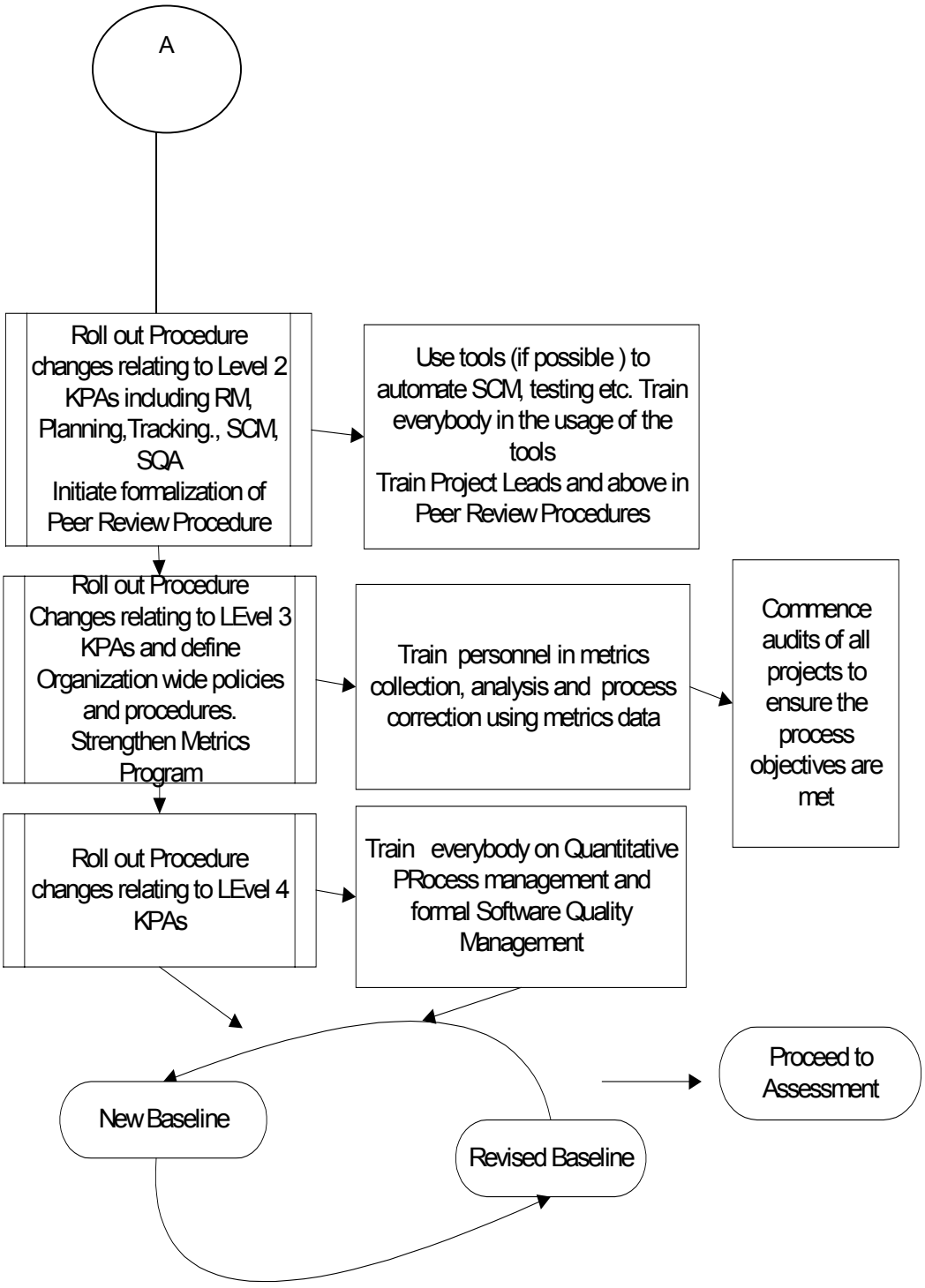
| Internal Process | Max | Current | External Process | Max | Current |
|---|---|---|---|---|---|
| PMSR | 2 | | Status reports | 2 | |
| Billing details maintained | 2 | | Conference calls | 2 | |
| Team meetings | 2 | | Minutes of Meeting | 2 | |
| Meeting Issues logged & tracked | 2 | | Change evaluations | 2 | |
| Contract review | 2 | | Change requests raised | 2 | |
| Management review | 2 | | **Total** | **10** | |
| MOM for management review | 2 | | | | |
| Audits conducted | 2 | | | | |
| SQA Reports | 2 | | | | |
| Project plan | 2 | | **Testing Process** | **Max** | **Current** |
| Quality plan | 2 | | Test Plan available | 1 | |
| Organization chart | 2 | | Test case defined | 1 | |
| Configuration management plan | 2 | | Testing is planned | 1 | |
| Risk management plan | 2 | | Testing is conducted | 1 | |
| Training plan | 2 | | Test logs maintained | 1 | |
| Effort metrics collected | 2 | | Test metrics collected | 1 | |
| **Total** | **32** | | Test Time planned (Hours) | | |
| | | | Test time actual (Hours) | | |
| | | | **Total** | **6** | |
| **Review Process** | **MAX** | **Current** | **Knowledge Management** | **Max.** | **Current** |
| Reviews planned | 1 | | Time planned for KM | 1 | |
| Reviews conducted | 1 | | Time spent for KM | 1 | |
| Review record maintained | 1 | | **Total** | **2** | |
| Review metrics collected | 1 | | | | |
| Review time Planned (Hours) | | | | | |
| Review time spent (Hours) | | | | | |
| **Total** | **4** | | | | |

**Note:** Rating is 2 for on time, 1 for done but not on time and 0 for not having done

**Annexure 2**

**The following Flow diagram details the milestones along the road for CMM Implementation**




A
Roll out Procedure changes relating to Level 2 KPAs including RM, Planning,Tracking., SCM, SQA
Initiate formalization of Peer Review Procedure
Use tools (if possible ) to automate SCM, testing etc. Train everybody in the usage of the tools
Train Project Leads and above in Peer Review Procedures
Roll out Procedure Changes relating to LEvel 3 KPAs and define Organization wide policies and procedures.
Strengthen Metrics Program
Train personnel in metrics collection, analysis and process correction using metrics data
Commence audits of all projects to ensure the process objectives are met
Roll out Procedure changes relating to LEvel 4 KPAs
Train everybody on Quantitative PRocess management and formal Software Quality Management
New Baseline
Revised Baseline
Proceed to Assessment

**References**


1. M.C. Paulk. Comparing ISO 9001 and the Capability Maturity Model for software. Software Quality Journal, 2: 245-256, 1993.
2. M.C. Paulk. How ISO 9001 compares with the CMM. IEEE Software, Jan.1995
3. W.Humphrey. Managing the Software Process. Addison-Wesley, 1989.
4. P.Jalote. CMM in Practice: processes for executing software projects at Infosys. Addison-Wesley, 1999.
5. International Standards Organization. Draft International Standards. 2000.

# Self-Assessment using the CMM - An Empirical Study

Robert F. Roggio
Professor of Computer and Information Sciences
University of North Florida
Jacksonville, FL

Pratibha Kashyap
Project Manager
PSINet Consulting Solutions
Irving, Texas

Ava S. Honan
Associate Professor of Information Systems and Decision Sciences
Auburn University Montgomery
Montgomery, AL

Abstract: This paper analyzes how software development organizations in a large metropolitan area assess their own software process improvement efforts via self-assessment using features and practices in the Capability Maturity Model (CMM). A number of graphs are produced based on surveys, questionnaires, and personal interviews (instruments included) with both managers and developers working in small, medium and large IT organizations. This paper assumes familiarity with the basic features of the CMM.

## I. Introduction

Many software development organizations are undertaking software process improvement (SPI) strategies as they attempt to deliver reliable, robust, high-quality complex software. Software that provides needed functionality in a highly-competitive, ever-changing marketplace presents a constant challenge of conflicting goals, where the goals are simply producing this reliable, cost-effective, highly-reliable software but in a severely constrained budget-time framework. While software has historically rarely been delivered both on time and within budget, it is becoming even more difficult to meet ever-rising client expectations. The real problem becomes even more exacerbated as organizations attempt to apply the new or improved methodologies and technologies: managers are having difficulty effectively managing the software process - if they can manage it at all. Many software professionals nowadays strongly assert that quality operational software is directly related to the process used to manage the development and maintenance of the software.

This paper is organized in six sections. After the Introduction, the second section briefly discusses SPI and the CMM. The third section presents the survey instruments and the response statistics. The fourth section consists of the analysis of each of the target organization's efforts directed toward software process improvement. The fifth section consists of analyses in meeting the KPAs that includes an assessment of each organization's position in the CMM framework. The last section assesses the impact software processes have on organizational goals.

The authors are not connected to the SEI nor are they attempting to provide a CMM-based software process assessment (SPA). The analysis is derived from answers to questionnaires and interviews. The questions are based on knowledge of the CMM and were developed from publicly-available documentation.

## II. Software Process Improvement

While customer satisfaction is the maxim of many of today's software development organizations, the inability to manage the software process continues to cause problems with the delivery of reliable and usable software that meets schedule and cost constraints. This is true even as organizations apply new software methodologies and technologies. A software process can be defined as a set of activities, methods, practices, and transformations that people employ to develop and maintain software and the associated products. [4] One of the objectives of software process management is to produce products according to plan while at the same time improving the organization's ability to deliver better products. An important step in addressing software problems is to treat the entire software project as a process that can be controlled, measured, and improved. Process improvement does not happen overnight. It does , however, yield consistent results that accrue steadily over the long term.

Software process improvement must have measurable improvement goals. For example, an organization may want to improve their customer satisfaction rating or reduce the number of defects reported

during production. The organization must determine how to quantify these goals so they can be measured before and after an improvement initiative is undertaken. The need to establish measurable goals links directly to executive management's need to understand the return on investment for process improvement. Management's support of a process improvement initiative is critical to the success of that undertaking. Equally critical to the success of such an initiative is identifying the return, justifying the initiative, setting goals, and measuring the results.

Historically, the U.S. Department of Defense (DOD) formed a joint service task force to review software problems in the DOD in 1982. This examination resulted in the creation of the Software Engineering Institute (SEI) at Carnegie Mellon University and related activities. The SEI developed the Software Process Maturity Model to address the need for improved software. In 1991, SEI produced the Capability Maturity Model (CMM) for Software Improvement, which identifies the key practice areas (KPA) for each maturity level and provides an extensive summary of these practices.

## III. SPI Efforts in Development Organizations

According to Herbsleb and Goldson [2], a carefully conducted survey may be an effective way to examine a broad cross section of a product. A survey may offer the possibility of examining results experienced by both successful and less successful organizations.

**Select the Organizations**.

In order to get a broad and balanced perspective six organizations were selected for this analysis. From these six, two were considered large, two medium and two small in size. This basis for classification is rather arbitrary: the size of the IT departments. The large organizations exceed 200 IT members, the medium ones have 100-199 members, and the small ones have fewer than 100 IT members.

The study started by identifying these organizations and appraising selected members of these organizations of the effort about to be undertaken. This was an attempt to ensure a high response rate for the questionnaires would take place. Papers on similar SPI efforts were also studied.

**Create Questionnaires**

Questionnaires were prepared by identifying the goals and objectives to achieve from this study. Two sets of questionnaires were created: one for management and one for developers. Management typically has a broader and higher level perspective of software improvement efforts, while development teams usually have first hand knowledge of whether company-wide processes are practiced and implemented as planned. The same questionnaires were sent to each organization. While the instrument is not included in this paper, one may get a copy by email: broggio@unf.edu.

Prior to construction of the questionnaires, several people from each organization were targeted and appraised of the project goals. They were requested (and agreed) to undertake the questionnaire and telephone if they had any questions. Phone call follow-ups were undertaken to ensure questions were understood. In order to get the correct information, respondents were assured that their information would only be used when averaged with other people's responses and their demographics would not be divulged to anyone as an individual set of information. They were also assured that their name and their company's name would not appear in the analysis in any way.

Seventy-two questionnaires were sent to managers, team leaders, analysts and software engineers of all six organizations. Out of 72 questionnaires 25% were sent to managers and team leaders and the remainder to developers. Sixty-five responses were received! These were obtained by a vigorous schedule of reminders, phone calls, emails, and re-mailing the responses. Out of these 65, 56 met the selection criteria (discussed ahead). Nine were discarded. Of these nine, six were only partially filled out and three could not justify the information.

To ensure higher credibility of the data, five to six individuals were interviewed from each organization (included two managers/project leaders) to obtain additional information on the organization's

focus. These individuals were asked open-ended questions whose answers were used to validate the information on the responses from their organizations. For example, if the responses show that the organization has rated 4 and 5 in Quality Assurance area, these individuals were queried as to the process used for QA, how defects were tracked and how these analyses have been used for preventing defects in the future.

| Type of Organizations | No. of Questionnaires Sent | No. of Response Returned | No. of Good Responses |
|---|---|---|---|
| Org-1 (L) | 14 | 14 | 12 |
| Org-2 (L) | 13 | 12 | 10 |
| Org-3 (M) | 11 | 11 | 10 |
| Org-4 (M) | 12 | 9 | 8 |
| Org-5 (S) | 10 | 9 | 7 |
| Org-6 (S) | 12 | 10 | 9 |
| **Total** | **72** | **65** | **56** |

Figure 1. Questionnaire/Response Matrix

## IV. Organizational Efforts for Software Process Improvement

The first analysis centers on process data collected from each of the six organizations.

Software development can be very complex, and there are often many alternative ways to perform the various tasks. The reason for defining the software process is to improve the way the work is done. When organizational responsibility for the software process activities is established, that organization's overall software status can improve.

In order to focus on SPI efforts, it is important to understand which improvements are critical for an organization, what are the good and weak practices of the organization, and what organizations are doing to improve them. In some organizations software processes are improvised by practitioners and their managers during the course of the project. In some other cases, even if the processes are specified, they are not rigorously followed and enforced. These organizations are usually focused on solving an immediate crisis. Whenever hard deadlines are imposed, they compromise on product quality and functionality to meet the deadline. There is no objective basis for judging the product quality or for solving process problems. For such organizations, it is important to have an organizational focus to develop and maintain a standard set of process which can be used across the organization. The dedicated resources are required to coordinate and enforce these processes.

In some organizations, quality of software product and processes are monitored by the managers. Schedule and budgets are based on prior performance and are realistic. These organizations follow the disciplined process consistently, since all the participants understand the value of doing so.

The organizational process focus involves developing and maintaining an understanding of the organizations' and projects' software processes and coordinating the activities to assess, develop, maintain, and improve these processes. [4]

A defined process can help direct software professionals in a systematic way. Process models can

be used either to describe what is done or what is supposed to be done. Most organizations have at least some policies, procedures, and standards. To be fully effective, however, these process models should be explicit.

In order to analyze the SPI efforts, questions were asked regarding organizational efforts, commitment and ability to improve the software process. Organizations 1 and 2 have focused their SPI efforts on requirements management, project planning and tracking, and configuration management. They have developed their own improvement methodology and assigned dedicated people with specific roles and responsibilities to enforce these standards. However, these practices are not always followed because managers lack of personal involvement in overseeing them. Also, management needs to facilitate rather than provide direction. The organizational focus for software engineering efforts is comparatively lower because it is up to the middle/lower management to enforce these practices.

Organizations 3 and 4 have defined processes but they are not kept up-to-date and no dedicated group enforces these standards. It is up to each individual to follow them. Management is not held responsible for improving and implementing these defined processes and they are not committed since senior management does not perceive SPI as high priority. Organizations 5 and 6 have rarely defined and documented processes.

It appears as if Organizations 1 and 2 have stronger commitments to see that the process is established and will endure by defining organizational policies and leadership. Also they have the ability to perform because of their higher resource level, funding, organizational structure and training program. These organizations have plans and procedures to perform the activities and also track them and take corrective actions if necessary. They have setup measurement practices to control and improve the processes. These activities were verified to have been performed in compliance with the processes that have been established.

In order to understand the comparative status of defined, documented, and practiced processes for different organizations, let's look at Figures 2 and 3 more closely.

**Requirements Assessment**

Requirements management establishes a common understanding between the customer and the software project of the customer's requirements that will be addressed by a software project [1].

Organizations 1 & 2 have very well defined and documented requirements processes, which are used for documenting the requirements and getting the customer's sign-off. Organization 1 has been rated lower than Organization 2, because sometimes changes to the allocated requirements are not reviewed and

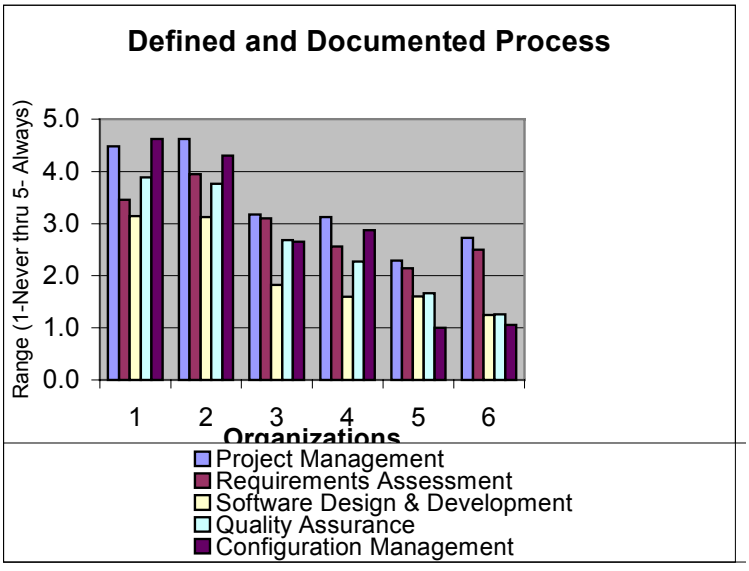



Figure 2. Defined and Documented Process Results

incorporated into the software project plan. Since requirements keep changing, it is important to adjust the design accordingly by incorporating them at planned intervals. For Organizations 3, 4, and 5, the level of defined and documented process decreases with size, with the exception of Organization 6. The reason is that management started focusing on requirement management and trying to control the requirements by enforcing the processes at project level

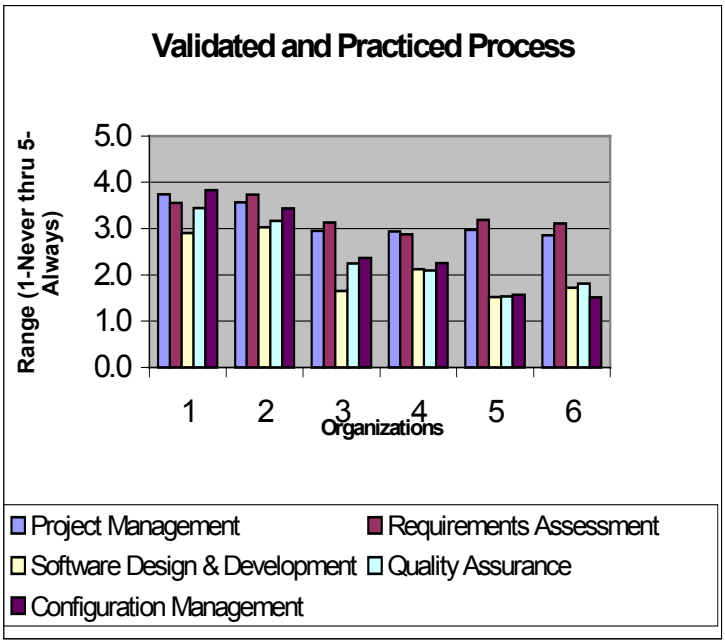



Figure 3. Validated and Practiced Process Results

Figure 3 shows these processes are practiced frequently by Organizations 1 and 2. Also, the rest of the organizations don't have well defined and documented processes for requirements management, but requirements validation is still done frequently and processes are continuously improved. Organizations 5 & 6 have validation and a practiced level higher than their documented level. The reason is middle or lower management forces validation of requirements with the customer. Also they try to establish a common understanding of requirements by sending them via emails etc.

**Project Planning and Management**

According to [4] the purpose of project planning is to establish reasonable plans for performing the software engineering and for managing the software project, which involves developing estimates, establishing necessary commitments, and defining the plans to perform the work. This plan provides the basis for performing and managing the software project activities.

Organizations 1 and 2 have very well defined and documented planning processes. As we move from Organization 3 to Organization 5, we observe the level of defined and documented process for planning decreases. Again Organization 6 has a better project management process, despite being a smaller organization. The reason still remains the same that management took special steps to define and document the project planning process, so it can be used by other projects as well.

Figure 3 shows that project planning and management methods are practiced frequently by Organizations 1 and 2. Also, the remaining organizations don't have well defined and documented organization wide processes, but sometimes project management and planning is done at the individual project level. However, the level goes down as organization size decreases.

**Software Design and Development**

Design and development standards provide the uniform means for developing software, which in turn increases productivity and improves readability and maintainability.

Organizations 1 and 2 have better documented processes in comparison to the rest of the organizations. The level decreases as organization size decreases.

This area is relatively low in defining as well as practicing the design and development methodologies because it appears that management pays inadequate attention and too many activities depend on how individuals feel on any particular day. Organizations 1 and 2 rarely practice these procedures and rest of the organizations almost never practices them.

**Quality Assurance**

QA refers to a set of activities designed to evaluate the process by which software products are developed and maintained. This involves reviewing and auditing the software products and activities to verify that they comply with the applicable procedures and standards.

Organizations 1 and 2 have well-defined and documented QA processes. Unfortunately, this practice seems to decreases with the size of organization. Organizations 3, 4, and 5 rarely developed and documented QA processes. Organization 6 does not have any documented QA processes at all.

Figure 3 shows that Organizations 1 and 2 occasionally audit and review the software activities. They keep an error log and analyze the patterns and try to prevent them from happening in the future.

**Configuration Management**

Configuration management is used to establish and maintain the integrity of the products throughout the project's life cycle. Changes to the software products are controlled and affected groups and individuals are informed of the status and contents of software baselines.[4]

The study reveals that Organizations 1 and 2 have well-defined and documented processes for configuration management. Again, the maturity level appears to drop as organization size decreases. Organizations 5 and 6 do not have any configuration management process at all. Understandably, the larger the organization, the larger the need to have more effective configuration management.

Figure 9 indicates that Organizations 1 and 2 have very good configuration management. They have configuration management software installed for version control and a dedicated group to handle these activities. Therefore, the practice level is very high. Organizations 5 and 6 do not have any configuration management.

It is noticed that Organizations 1 and 2 have better defined and documented configuration management policies and they are validated and practiced very frequently because the dedicated process management group is present to enforce them. As shown in Figure 3, the level of practiced processes for software design and development is very low. The reason behind this poor practice was that there was no dedicated group to enforce the design and code reviews. However, it was noticed that the small organizations have rarely documented processes for project management and requirements assessment, but the practiced level is higher than documented level. To validate this information a few IT members from these organizations were personally interviewed to discover that the practice of these processes is dependent upon how the managers feel it is. Sometimes managers even define their own processes. The practice level for software design and development and quality assurance also depends on project managers.

## V. Organizational Positions in the CMM Structure: Examination of KPAs

Section V centers on organizational strengths and weaknesses in particular KPAs. Spreadsheets were created for data pertaining to key process areas for levels 2 -5 of the CMM for each of the six organizations. Information for CMM Level 2 was entered from the responses of project managers and project leaders (KPAs

for level 2 refer to project management, project planning and tracking, requirements management, software quality assurance and configuration management). Developer responses provided . information for CMM Levels 3, 4 and 5. Personal interviews with managers and developers supplemented these questions.

The nature of continuous process improvement implies that the process itself must consist of small, precise steps in gaining maturity over time. As higher levels of maturity are obtained, essential components of the software process become established and stabilized. When a specific set of these process goals have been achieved, a higher "maturity level" is reached, with an attendant improvement in the process capability of the organization.

While there are many ways to define and implement processes that contribute to the construction and maintenance of high-quality software, the KPAs exhibit a fundamental set of behaviors that all software organizations should display regardless of their size and/or products. [2] Herbsleb and Goldson cite that key practices must be interpreted in the light of a project's or organization's business environment. Further, this interpretation should be based on an informed knowledge of both the CMM and the organization and its projects. In order to understand the organization's business environment and goals, individuals from all levels were interviewed. This information was used for interpreting the KPAs.

The KPAs of level-2 relate to establishing basic project management controls. The analysis shows that Organizations 1 and 2 have frequent commitment and ability to perform the requirements management and software project planning. They do this by measuring and controlling the activities to make sure that they are implemented in compliance with the processes. However, the level of these practices decreases with the size of the organization. It can be seen that key practices for software project tracking and quality assurance are followed only at times even for Organizations 1 and 2, which are considered large. In these areas a dedicated process group does not have much control over enforcing these practices.

Figure 4 reveals that Organization 6 has comparatively better requirements management and project management practices even though the process are not well defined and documented. Organizations 5 and 6 lack configuration management practices. Software quality assurance is very rarely practiced in these organizations

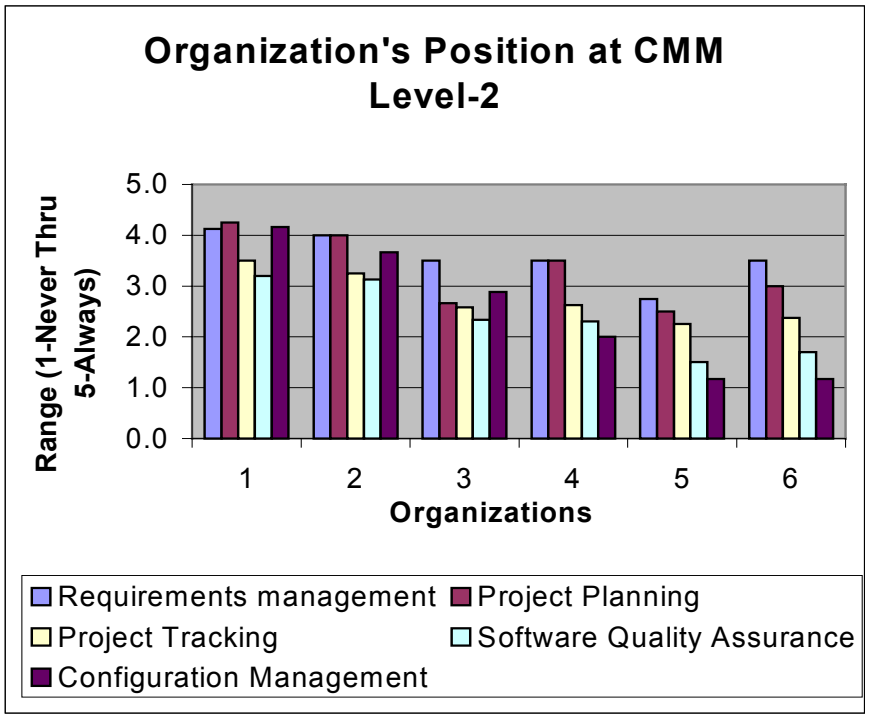



Figure 4. KPAs – Level 2

**KPAs for Level 3**

For KPAs at Level 3, a standard process (software engineering and management) for developing and maintaining software is documented and used across the organizations. This assists and guides both management and technical staff in performing more effectively. Software process activities are assigned to a process group within the organization.

Organization Process Focus

Organizations that demonstrate process focus have a software engineering process group (SEPG), which is dedicated to providing the long-term commitments for developing and maintaining the software processes across current and future software projects (Figure 5). Organizations 1 and 2 have better process focus than the other organizations because Organizations 1 and 2 have dedicated resources to enforce the standard processes. Organizations 3 and 4 have the processes established but funding and resources are not allocated to improve and enforce these practices. Organizations 5 and 6 do not have organization process focus.

Organization Process Definition

According to J. Herbsleb et al [5], the purpose of organization process definition is to develop and maintain a usable set of software process assets that improve process performance across the projects and provide a basis for long term benefits to the organization. In our analyses, Organizations 1 and 2 exhibit better performance here, and, the level appears to almost steadily decrease as we progress from Organization 3 to Organization 6.

Training Program

Training necessitates the identification of specific needs within the organization and then implementing the training to address these specified needs. Organizations 1 and 2 have stronger training programs to provide on-the-job training, classroom training or guided self-study to obtain the skills required for performing the roles effectively. Organizations 3, 4, 5, and 6 have training programs; they are not implemented according to the documented process.

Integrated Software Management

Herbsleb et al [5] go on to point out that the purpose of integrated software management is to integrate the software engineering and management activities into coherent processes. This involves managing the software project using the project's defined software process. The management of the software project's size, effort, cost, schedule, staffing, and other resources is tied to the task of the project's defined software process. The level of integrated software management activities steadily decreases as organizational size decreases from Organization 1 to Organization 6.

Software Product Engineering

Software Product Engineering involves performing the engineering tasks to build and maintain the software using the project's defined software process and appropriate methods and tools.[5] At times, software product engineering is followed to produce the correct product. Figure 11 shows that these practices are not followed frequently. Everyone uses their own way of performing the software engineering tasks.

Inter-Group Coordination

The inter-group coordination is rarely seen for Organization 1, where software engineering groups coordinate with other engineering groups to better satisfy the customer needs. Inter-group coordination is better in smaller organizations where it is easy to coordinate and track the critical dependencies and negotiate among the groups. Figure 5 shows that Organization 6 has frequent inter-group coordination.

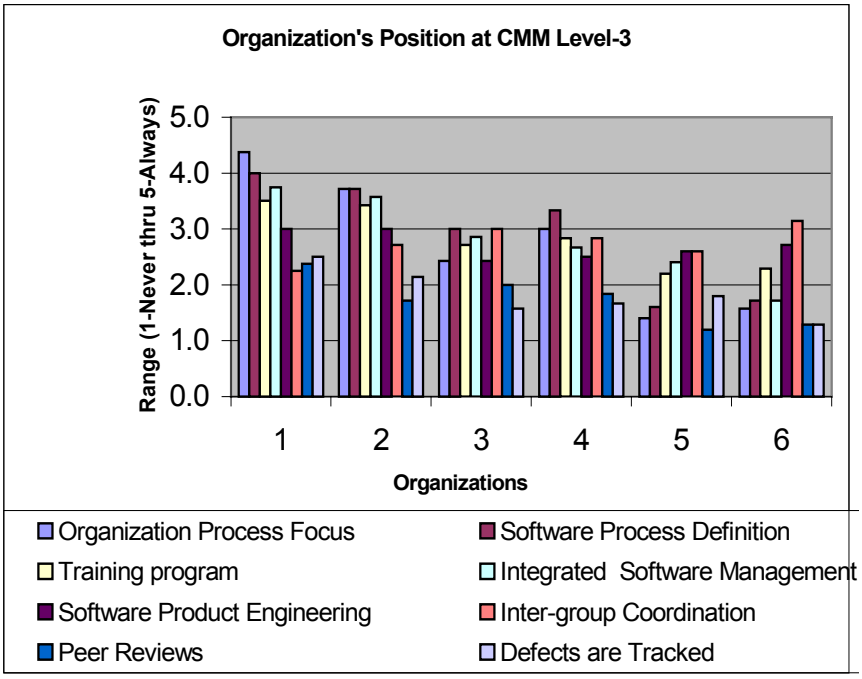



Figure 5. KPAs for Level 3

Peer Reviews

Peer reviews can remove the defects in early stage of a software project. In general all the organization appear to be lacking in the peer review area. Trained review leaders do not lead the peer reviews. Middle management's focus is required to enforce these activities.

**KPAs for Level 4**

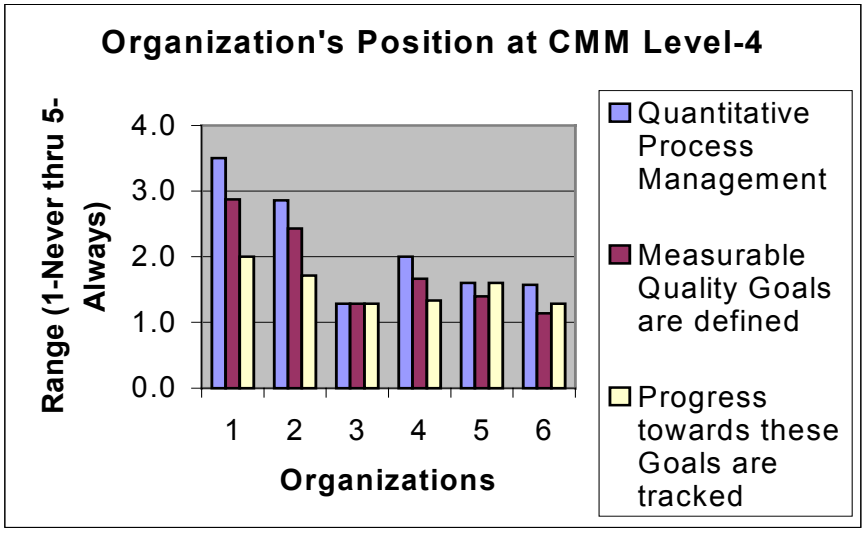



Figure 6. KPAs for Level 4

According to [4], in KPAs for level 4, quantitative quality goals are set for software products and processes. Projects achieve control over their product and processes by narrowing the variation in their process performance to fall within acceptable quantitative boundaries.

Quantitative Process Management

Figure 6 shows that Organization 1 has the best quantitative process management among all other organizations. The quantitative goals are set for the customer satisfaction. Software process performance appears to be tracked by comparing the planned goals against the actual results achieved from following a particular software process. Emphasis is placed on defining the specific causes of the defect within a process that is "measurably stable" and then correcting those circumstances that caused the defect to occur. Organizations 1 and 2 have a process to plot these deviations and bring them to executive management's attention if it goes beyond the certain limits. This is used for the big projects only. Organizations 3 to 6 have very rarely defined and implemented practices for quantitative process management

Software Quality Management

The purpose of Software Quality Management is to develop a quantitative understanding of the quality of the project's software products and achieve specific quality goals.[4] Quality goals in terms of functionality, maintainability, reliability, and usability for each life cycle stage are defined and documented. High-leveraged quality goals for the software products are those that provide the greatest customer satisfaction at the least cost, or the "must-haves" from the customer. Characteristics of the product quality that describe how well the software product will perform or how well it can be developed and maintained can be identified.

Figure 6 shows that sometimes Organizations 1 and 2 have defined measurable software quality goals. The rest of the organizations do not have any defined measurable quality goals for the software products. All the organizations rarely track the progress towards these goals.

**KPAs for Level 5**

A Level 5 maturity [4] focuses on continuous process improvement. The organization analyzes defects to determine their causes and evaluates software processes to prevent known types of defects from recurring. Improvements occur both by incremental advancements in the existing process and by innovations using new technologies and methods.

Defect Prevention Activities

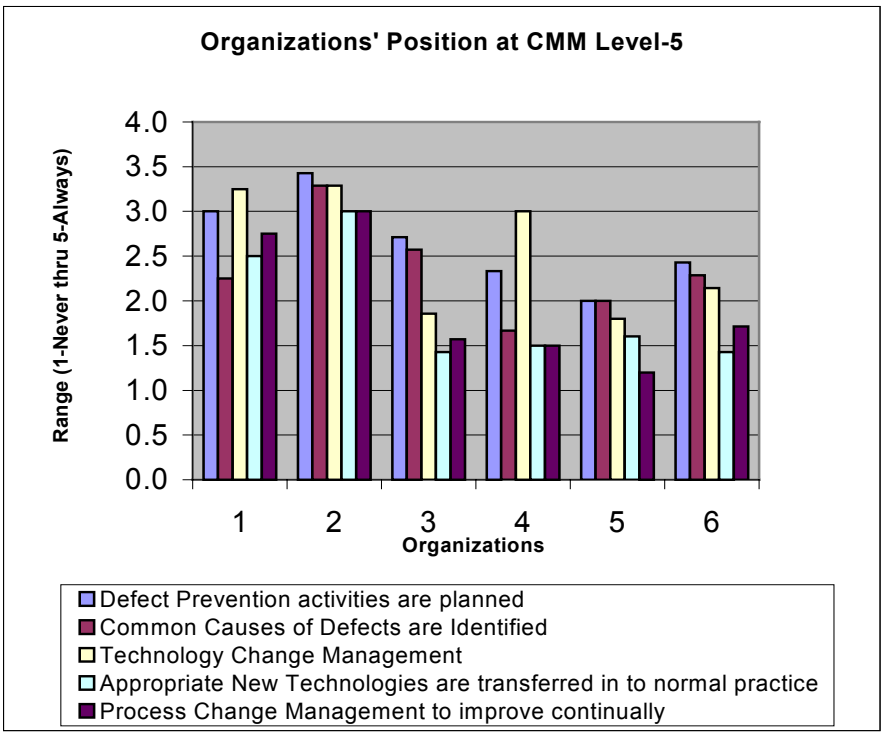
Figure 7 shows that Organization 2 has the best defect prevention activities planned. Defect prevention activities include task kick-off meetings, causal analysis meetings, reviewing and planning of proposed actions, and implementing actions. In the team meetings these defect prevention activities are shared. Organizations 1 and 3 claim they sometimes have defect prevention activities planned; Organizations 4, 5, and 6 assess their defect prevention activities near "rarely."

Common Causes of Defects

Defects are identified and analyzed to determine their root causes by drawing cause/ effect diagrams. The causes could be inadequate training, breakdown in communications, mistake in manual procedures, and unidentified details of the problems. Organization 2 sometimes has defection prevention activities; Organizations 1, 3, and 6 rank their defect prevention activities slightly above "rarely" and Organizations assess their defection prevention activities between "never" and "rarely."

Technology Change Management

The purpose of technology change management is to identify new technologies (i.e., tools, methods, and processes) and transition them into the organization in an orderly manner.[4] This involves identifying, selecting, evaluating, and incorporating new technologies into the organization. The objective is to improve software quality, increase productivity, and decrease the cycle time for the product development. Figure 7




shows that Organizations 1, 2, and 4 have better technology change management processes. Organizations 3, 5 and 6 are rarely able to transfer these new technologies due to so many other constraints.

Figure 7. KPAs for Level 5

Process Change Management

The purpose of process change management is to continually improve the software processes used in the organization with the intent of improving software quality, increasing productivity, and decreasing the cycle time for the product development. Figure 7 shows that Organizations 1 and 2 have processes to improve software quality.

## VI. Impacts of Processes on Organizational Goals

Aggressive SPI activities help in achieving the organizational goals for quality and productivity. To achieve higher quality, organizations need to know what to improve. Assessment and improvement of software processes are important and critical for any organization reliant upon its software as its products and services. The software processes affect scope, budget, quality, productivity, project deadline, and customer satisfaction. Figure 8 shows that change in project scope is mainly due to poor estimates, and poor planning. Organizations 3, 4, and 5 sometimes meet the customer requirements. Poor estimates, planning and requirements are major reasons for late product deliveries and budget overruns. Organizations that are rated low (Organizations 3, 4, and 5) in project planning and tracking report that they have projects that rarely get completed within budget and on schedule. Organizations 1 and 2 have better project management practices and indicate that their projects are frequently completed within budget and on time.

Other significant reasons for poor quality are ineffective peer review and design. Since all the organizations lack effective processes for both software design and development, product quality is met only marginally. Since Organizations 1 and 2 have more mature processes in general, customers' requirements are frequently met. In contrast, Organizations 3, 4, and 5 are only sometimes able to meet customer requirements. All organizations report additional rework caused by ineffective or complete lack of peer reviews, inspections, designs, and planning.

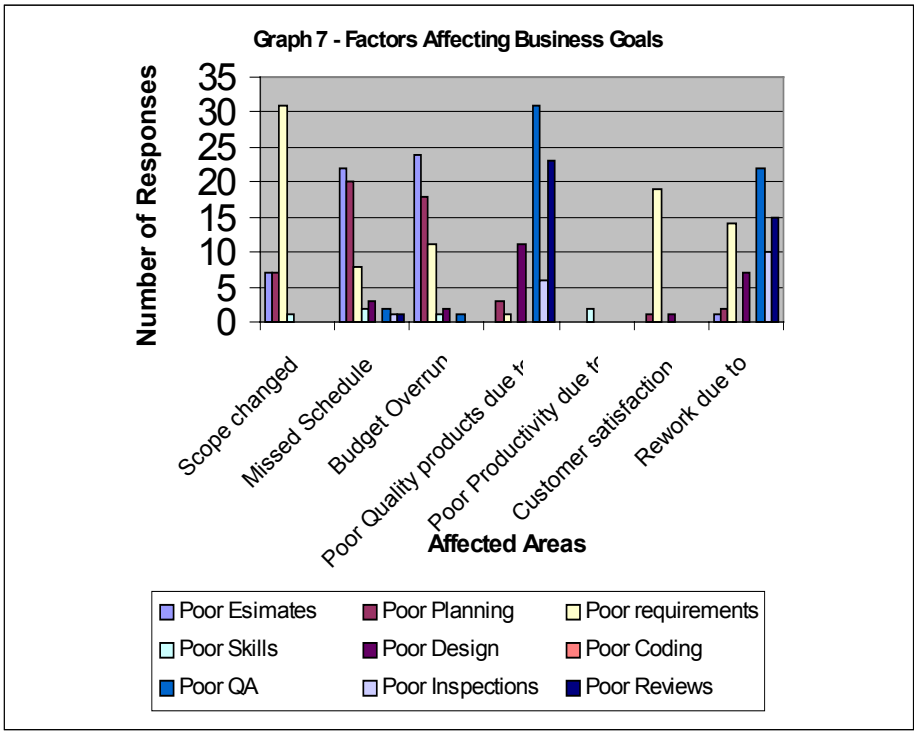



Figure 8. Factors Affecting Business Goals

## VII. Conclusions

In the last two decades, key buzzwords tossed about by many software practitioners include "quality," "process," "assessment," and "improvement." SPI provides guidance, metrics, and a set of additional tools to address the challenges that software quality, complexity, and competitiveness present to the modern software professional. As one of those tools, CMM is a framework describing the key elements in an effective software process, and the quality of the process is an important predicator of the organization's ability to deliver high-quality products on schedule and within budget. The CMM guides the organizations in selecting the improvement strategies by determining their current process status, and identifying the issues, which are critical to improving their software quality and software process.

Organizations 1 and 2 have developed there own SPI methodologies and they have assigned the dedicated resources to enforce the defined processed by designating roles and responsibilities. They have defined improvement goals; higher management is committed to these goals. Sometimes, however, these practices are not 100% followed because of both a lack of middle management's personal involvement and commitment and the entire staff's buy-in.

Organizations 3 and 4 do have their own methodologies but they are not validated, practiced and improved. No dedicated resources, lack of management commitment, and inappropriate organizational structure all appear to contribute to this deficiency.

Organizations 5 and 6 sacrifice quality to meet project schedule. Organization 6 is exploring some of the features and benefits of SPI.

It is difficult to determine the maturity level of each organization. Organizations 1 and 2 have good practices in Requirements Management, Project Planning, and Configuration Management, which are KPAs for Level-2. Additionally, they have satisfied Organization Process Focus and Organization Defined Processes that are KPAs from Level-3. Organizations 1 and 2 appear to lie between CMM levels 2 and 3 while Organizations 3 and 4 fall short in meeting KPAs in Level 2. Organizations 5 and 6 appear to be CMM Level 1 organizations.

Software process improvement is a long-term incremental activity. It requires investment, risk, time, and the pain of cultural change. Management's commitment and staff buy-in are absolutely essential. Improvement goals should be set and management should monitor the progress. Processes should be tailored according to the business environment and goals. Studies show that SPI pays off.

## References


[1] Bill Curtis, "Software Process Improvement: Methods and Lessons Learned," TeraQuest Metrics, Inc., Austin, Texas 78720-0490 USA,l-512-219-286

[2] Herbsleb, J. D. and Goldson, D. R. "A Systematic Survey of CMM Experience and Results, *Proceedings of ICE '96*, Berlin, Mar. 25-30.

[3] Humphrey, W.S., *Managing the Software Process*, Reading, MA: Addison-Wesley, 1989.

[4] Paulk, M.C., Weber, C., Curtis, B., & Chrissis, M.B., *The Capability Maturity Model: Guidelines for Improving the Software Process*, Reading, MA: Addison-Wesley 1995.

[5] J. Herbsleb, D. Zubrow, D. Goldenson, W. Hayes and M. Paulk, "Software Quality and the Capability Maturity Model", *Communications of the ACM*, Vol 40, No 6, Jun 1997 pp.30-40.

[6] H. Wohlwend and S.Rosenbaum, "Software Improvements in an International Company," *Proceedings of 15th International Conference on Software Engineering*, Los Alamitos, CA, 1993, pp 212-220.

[7] Watts S. Humphrey, Terry R. Snyder, and Ronald R. Willis, "Software Process Improvement at Hughes Aircraft," *IEEE Software*, Vol. 8, No. 4, July 1991, pp. 11-23.

[8] Bach, J. "Enough About Process: What We Need are Heroes", *IEEE Software*, Vol12, No2, 3/95, pp:96-98

[9] Bamberger, J. "Essence of the Capability Maturity Model", *Computer*, Vol 30, No 6, June 1997, pp. 112-114.

[10] Brown, N., "Industrial-Strength Management Strategies", *Software*, Vol 13, No 4, July 1996, pp 94-104.

[11]M. Diaz and J. Sligo, "How Software Process Improvement Helped Motorola," *IEEE Software*, Vol 14, No 5, September/October 1997, pp. 75-81.

[12] M. Daskalantonakis, "Achieving Higher SEI Levels", *IEEE Software*, Vol:11, No:4, July 1994 pp:17-24.

[13] R. Dion, "Process Improvement and the Corporate Balance Sheet", *IEEE Software*, Vol:10, No:4, July 1993, pp: 28-35.

[15] Fayed, M and M. Laitinen, "Process Assessment Considered Wasteful", *Communications of the ACM*, Vol 40, No 11, November 1997, pp:125-128.

[16] K. Fowler, "SEI CMM Level 5: A Practitioner's Perspective," *Crosstalk*, September 1997.

[17] T. Haley, "Software Process Improvement at Raytheon", *IEEE Software*, Vol 13, No 6, Nov 1996, pp:33-41.

[18] Bollinger, T. and C. McGowan, "A Critical Look at Software Capability Evaluations", *IEEE Software*, Vol 8, No 4 July 1991, pp: 25-41.

[19] W. Humphrey, T. Snyder, and R. Willis, "Software Process Improvement at Hughes Aircraft", *IEEE Software*, Vol 8, No 4, July 1991, pp:11-23.

[20] D. Johnson and J. Brodman, "Realities and Rewards of Software Process Improvement", *IEEE Software*, Vol 13, No 6, November 1996, pp: 99-101.

[21] Jones, C. "Our Worst Current Development Practices", *Software*,Vol 13, No 4, March 1996, pp: 102-104.

[22] S. McConnell, "Software's Ten Essentials", *Software*, Vol 14, No 2, March/April 1997, pp: 143-144.

[23] H. Wohlwend and S. Rosenbaum, "Schlumberger's Software Improvement Program," *IEEE Transactions on Software Engineering*, Vol 20, No 11, November 1994, pp. 833-839.

[24] G. Wigle and G. Yamamura, "Practices of an SEI CMM Level 5 SEPG," *Crosstalk*, November 1997.

[25] G. Yamamura and G. Wigle, "SEI CMM Level 5: For the Right Reasons," *Crosstalk*, August 1997.

## Title of Submission: Software Process Improvement in Small Companies

Primary Contact Author Name: Dr. Ita Richardson
Affiliation: University of Limerick
Email Address: ita.richardson@ul.ie
Mailing address:
Department of Computer Science and Information Systems
University of Limerick
National Technological Park
Limerick
Ireland

Work Phone: Tel: +353-61-202765
Fax: +353-61-330876

Other Authors of Contact Names: Prof. Kevin Ryan
Affiliation: University of Limerick
Email address: kevin.ryan@ul.ie
Phone numbers (Wk): +353-61-202405

Key Words/Phrases:
Software Process Improvement, Small Companies, Action Research, Implementation of Processes, Software Process Matrix, Quality Function Deployment

Author Biography (25-50 words):

Ita Richardson received her PhD in Computer Science from the University of Limerick in 1999. A lecturer at the University of Limerick, her lecturing responsibilities include Software Process Improvement, Systems Analysis and Design and Computer Graphics. Her PhD research was in the application of Manufacturing Quality Techniques to Software Process Improvement in Small Software Development Companies. She is a founder member of the Small Firms Research Unit at the University of Limerick, whose research is specifically concerned with the growth and development of small firms.

Kevin Ryan received his BA, BAI (Engineering) in 1971 and Ph.D. in Computer Science in 1978 all from Trinity College Dublin. In 1990 he became Professor and head of the Department of Computer Science and Information Systems at the University of Limerick. From 1994-1999, he was Dean of the College of Informatics and Electronics. He has been programme chair of RE'99, organisation chair of ICSE2000 and is on the editorial board of Requirements Engineering Journal and the Software Process Journal. He currently holds the position of Vice-President Academic at the University of Limerick.

# Software Process Improvement in Small Companies

**Ita Richardson**
**Kevin Ryan**
Department of Computer Science and Information Systems
University of Limerick
National Technological Park
Limerick
Ireland

Telephone: +353 61 202765
Fax: +353 61 330876
email: ita.richardson@ul.ie
kevin.ryan@ul.ie

## Abstract
The derivation of Software Process Improvement plans is an area of Software Process research where it is accepted that more work is needed. This is especially true for smaller companies which tend to have limited resources. The authors are part of a major effort to model and validate cost-effective generation of software process improvement plans. A variety of research approaches were used including self-assessment, external assessment, action research and control studies. One of the action research studies is described, beginning with an outline of its pre-existing development process. The researchers then identified a prioritised set of actions and proposed them to the practitioners. An agreed action plan was devised and put into effect. The results, after one year, are summarised. A short comparison is made with other companies and some general conclusions are drawn.

## 1.0 Introduction
### 1.1 Motivation and Approach
Following any software process assessment, it is important that an organisation implements a process improvement strategy to produce a well-defined software process. In theory, this is simple. It is much more difficult in practice.

Authors have recognised that the models currently available do not provide an improvement strategy [1], [2]. The IDEAL model for the Capability Maturity Model was presented in 1995 [3], but as recently as 1998, Debou and Kuntzmann-Combelles note that "due to lack of documentation on the post-assessment phase, assessments are often being performed as a one-shot event without connection to any improvement strategy" [4]. For the large company, it may be possible to devise the required strategy by investing in the development of action plans. However, for the small company, this requires yet another investment from a much smaller purse.

The authors of this paper have developed a generic method to be used by small software development companies, allowing them to devise Software Process Improvement (SPI) strategies. This method, based on Quality Function Deployment, provides a Software Process Matrix (SPM) which can be used to quantify the impact of a large number of possible process improvement actions. (Full details of the derivation, structure and use of the SPM are given in [5] and [6]). The authors implemented the method – using the Software Process Matrix - in two small software development companies in Limerick, Ireland. The paper discusses this implementation in one of those companies.

The researchers used action research with control groups to investigate the usefulness of the Software Process Matrix which they had developed [7]. Two companies, Computer Craft and DataNet were involved in the action research where one of the researchers intervened over a period of one year. Another two companies, Software Solutions and Ríomhaire were treated as control companies. These are pseudonyms for four small companies – each with less than 50 employees - which were located in the mid-west of Ireland. At the commencement of the research period, personnel within the action research companies completed a self-assessment questionnaire based on the BootCheck assessment tool which came

from the BootStrap Institute. The results from this questionnaire were entered into the Software Process Matrix, and from this data, an action plan was devised within each company. The majority of these actions were implemented in both companies. The implementation and outcome within Computer Craft, one of the action research companies, is discussed in detail in this paper. The results are compared with those achieved in the other three companies.

### 1.2 Research in Computer Craft

As this was an action research project in Computer Craft, employees were interviewed and observed in their work. Access to complete documentation on two projects was obtained. One project was developed in the initial stages of the research; the other was developed towards the latter end of the research. Four processes which were changed as a result of the SPM implementation, Organisation Process, Customer Management, Implementation and Project Management, are discussed in this paper.

The researcher was permitted to interview the software development personnel, who also answered the self-assessment questionnaire for input to SPM. The results were presented at a meeting chaired by the researcher and there was open and frank discussion by the engineers on what approach to take. The researcher was given the opportunity to observe processes and had access to all documentation on two projects, Data Function and Banking Organiser. This included versions of many documents and internal memos. She was also given open access to their bugs database (stored on Lotus Notes), and to the company's private intranet.

## 2.0 Software Process at Computer Craft Prior to Intervention

### 2.1 Organisation Processes

Prior to the researcher visiting the company, there was no particular emphasis on software process improvement. There had been some work done on the development of standards within the software development group, but this was done on an individual basis by the manager of the group rather than it being promoted within the organisation. The software development manager was a software engineer with thirteen years experience. While he was not a founder of the company, he joined it when it was very young. All other software engineers within the software development group were also graduates, bringing experience from a variety of other companies with them to Computer Craft. Training of employees was not prevalent within the organisation.

Software for the group was being updated during initial visits by the researcher. Members of the group were installing a world wide web browser, updating their word processing package and investigating the possibility of using a CD-based telephone directory. The group also maintained a Lotus Notes database. Included in this database was the option for the software engineers to circulate information to each other. There were also many memos on different thoughts about specifications, problems within modules, fixes found etc.

Although interaction within the software development group was relaxed and informal, it was felt by the software engineers that there was not enough interaction between the group, customer support and sales. They also felt that, at times, sales were changing specified product requirements.

### 2.2 Customer Management

Customers were dealt with exclusively by the sales representatives in Ireland and the U.K. It was not normal for developers in the software development group to meet the customer until installation time.

The sales representatives were the intermediate contact between the customer and the software development group. Although the software developers were aware that the customers were continually reviewing specifications, from their point of view "*our customers are the sales people"*. They recognised that "*the final say will probably be (the Customer Services Manager)*". In many ways, "*contact with the (actual) customer by the PC Development Group is non-existent*", and in the words of one engineer "*the development group's strength is in computing"*.

For customised product, the customer services project manager wrote up a requirements specification. There was no particular standard in use: *"each customer services project manager identifies the need in their own way*", and complete information was not always received from the client: "*the client may not know how to address particular issues*". Customers, those in the U.K. in particular, went through this specification in detail, checking out costs and deadlines, as this was the basis of the contract. Once completed, this document was signed-off by the customer and was passed to the software development group. However, there were cases where "*questions were left unanswered*" and "*information was lacking*".

Customer requirements often changed as developers proceeded through the design stages. One developer stated: "*There is a huge amount of information from the customer, but it is changing – moveable goalposts*". When changes came from customers, there was no formal means of dealing with them. The difficulty faced by the group was that features not previously included in the requirements, and subsequently the functional specification, impacted the final project schedule.

Developers sometimes found problems at later stages in development, such as system test and installation, which could have been avoided if requirements had been collected properly. This was a frustrating situation for the software engineers, particularly as they had deadlines to meet.

**2.3 Implementation**

Decisions about implementation of software were made by the customer services project manager. They needed to examine such issues as: could the product be installed? could it do the critical tasks? were there any show stoppers? or cosmetic bugs? The final decision was usually based on the answer to "*is it a major concern if the software goes with this bug?*" As there was no quality shipment policy defined, "*there have been a few bad blunders*". The existing quality policy only required that, prior to shipment, all software was signed off by three people and there was a stamp on delivery media.

Following test completion, the software was shipped to the customer services project manager who installed the software for the customer. In some cases the software development manager or software engineer helped with the implementation. When there were changes made during implementation the customer compared the product with the requirements specification, which was unambiguous as far as the company staff were concerned. Software installation was done either from diskettes or a laptop computer.

Because of the nature of the product, implementation was usually done in a number of phases, depending on the number of modules required by the customer. This could take up to three months. A template was defined by the customer services manager for two of the implementation phases. He also defined what should be in the pack when software was implemented. This documentation was available to the software engineers on an internal Notes database. During implementation of a product, the company did not always have documentation available for the customer, but this would be sent to them as soon as possible.

**2.4 Project Management**

Project deadlines were driven by the customer, giving the software development group no leeway or contingency on time. At the start of a project, the software development manager drew up an initial schedule, breaking this into defined phases and milestones, but these were not always adhered to. The main difficulties arose when a schedule was set, and then new features, which disrupted the schedule, were added. For example, the manager initially drew up a Gannt chart for each project. This was usually kept up to date until the project was approximately 60% complete when it fell victim to pressure of work. Nevertheless the software development manager was ultimately responsible for the completion of the project.

Prior to the implementation of projects, it was not unusual for developers to work long extra hours. In general, the time constraints caused problems and "*the developers do not expect to have enough time*". In another case, a software engineer had not completed the scheduled module in time, despite having put in extra hours the previous few days, and cancelled an installation on the day it was supposed to take place.

## 3.0 Researcher Intervention

The researcher intervened in the software process in Computer Craft, fulfilling the role of participant-observer as identified by Burgess [8]. Questionnaires which were used as input to the Software Process Matrix were completed by the software engineers. The SPM generated an ordered list of actions, with the action expected to have the highest impact given first. The following top 10 practices were then proposed as the basis for a software process improvement action plan:

1. Identify this organisation's product items
2. Establish product baselines for each product supplied
3. Verify all changes to requirements are monitored
4. Specify and document system requirements
5. Collect, identify, record and complete new customer requests
6. Assign a person with SQA responsibilities
7. Identify the initial status of the product
8. Define delivery contents (media, software, documentation, documentation) to customer from subcontractor / software development group
9. Define quality criteria and metrics for the project deliverables
10. Assign responsibility for software development plan, work products and activities.

The software development group in Computer Craft was not willing to accept the proposed actions at face value. They felt that some of the actions should be combined, and they would not accept the priorities given without some discussion. After a lengthy group meeting, chaired by one of the authors, it was decided that the company should concentrate on the following set of practices, combined into six action items with the following priority:

Action 1:

- List all the company's products and their dependencies. This information was not formally available in Computer Craft, a fact that was the subject of discussion at that time. (Based on practice 1.)

Action 2:

- Set up a procedure to specify and document system requirements. (Practice 4 above.) The group wanted to have a template for the specification but not a procedure as they felt this would inhibit developers too much.
- Set up a procedure to collect, identify, record and complete new customer requests when new requests come in, prior to development. (Practice 5 above.)

Action 3:

- Set up a procedure to define delivery contents (media, software, documentation) for the customer from the subcontractor / development group. (Practice 8 above.)
- Set up a procedure to verify that all changes to requirements are monitored once the requirements specification has been signed off (Practice 3).

Action 4:

- Set up a procedure to define quality criteria and metrics for the project deliverables. (Practice 9.)
- Assign a person with SQA responsibility and also define what the SQA responsibility is. While individual software engineers were expected to be responsible for their own quality, the SQA person had a responsibility of letting people know the quality criteria. Examine the job description, define what software quality means, and expand the SQA role to include the software process rather than just the correction of problems. (Based on practice 6 above.)

Action 5:

- Set up a procedure to establish product baselines for each product supplied so that Computer Craft would know what was the minimum product that they could ship. (Practice 2.)
- Set up a procedure to assign responsibility for the software development plan, work products and activities. (Practice 10.)

Action 6:

- Set up a procedure to identify the initial status of the product. This procedure could be based on the current handover document. (Practice 7.)

## 4.0 Software Process following Intervention

### 4.1 Approach

As the engineers within the software development group had many years work experience in other companies, the researcher did not become involved in drawing up procedures, based on the agreed actions, within the company, but instead met with the software development manager on a regular basis to discuss progress. The software development group worked on the implementation of actions from SPM within Computer Craft. The changes were facilitated by the installation of an intranet, which was used to share documentation and procedures, and by the recruitment of a quality assurance engineer.

Intervention by the researchers within Computer Craft was evaluated twelve months after the presentation of the SPM action plan. This section of the paper discusses the processes at this time. Much of the discussion centres on the Banking Organiser project, the main project being worked on by Computer Craft, to which updated software processes were applied. This project consisted of a series of software modules which extracted data from Data Organiser, outputting it in formats which interfaced with other software used in the customer company. It was seen as "*unique in Computer Craft*" in that it was a first-time development. Much of the recent work the developers had been doing was customisation of the Data Organiser software.

### 4.2 Organisation Processes

One software engineer and the quality assurance engineer had been recruited nine months previously. Initially, formal training was not provided for the new personnel. Both people were expected to learn from sifting through manuals – "*a thick file of stuff done before*" – and by asking questions. Neither person had difficulty with doing this, although when discussing a particular task, one engineer felt that "*it was a small bit of being thrown into the deep end"*. The software development manager was aware of the need for formal training within the group, and had arranged for the quality assurance engineer to attend a training course.

The members of the software development group interacted informally on a regular basis. This was helped by the layout of office space, as they had moved their offices to the main company building. They were located in an open plan area which they shared with people from the customer services help desk. They could also interact easily with the customer services help desk and secretarial support. The software development manager also held group meetings each Monday morning. These were used to review the previous week and to discuss what would be worked on the following week.

The company had installed an intranet, which was used extensively by the software development group for storage of standards, procedures and documentation relating to individual projects. This intranet allowed all company employees to view documentation. This was especially useful for overseas employees.

### 4.3 Customer Management

The main project worked on by the software development group was Banking Organiser. In this project, the software development manager dealt directly with the customer. Initially, the customer produced a document listing their requirements. Following this, the software development manager spent some time on the customer site, met their project manager, and attended meetings about their requirements. The Banking Organiser project manager became the point of contact for the software development manager, who then passed requirements to the software developers. If the software development manager could not answer the software developer's questions, then he talked to the project manager. As the project progressed, the developers had direct contact with the customer, and recognised that their customer was the company for whom the product was being developed.

One section of the Banking Organiser project suffered from "*feature creep*". Because of this, the engineer working on the software found that certain changes "*should be easy but because of this are relatively difficult*".

### 4.4 Implementation

The software development manager implemented the software for the project in the customer site.

Documentation was written for the installation of Data Organiser, but this could never be complete because problems specific to the customer site may occur. However, when errors were detected, there was a list available to the developer of possible solutions that had been used in previous implementations.

### 4.5 Project Management

At the start of the Banking Organiser project, the software development manager produced a table of the project phases and the time it would take to complete each phase. He also considered what personnel were required for the project, taking into account other projects being worked on within the company. Using an automated system, he created a project plan, showing planned project tasks, responsibilities and duration. Developers were expected to enter actual time spent on the project, allowing the software project manager to track project progress. One software engineer stated that "*project management has improved*" and consequently, the Banking Organiser project "*was a tightly controlled project from the start*". Project progress was updated on a regular basis, as the manager was now treating the software development group as a "*business unit*". Therefore, emphasis had to be placed on both income from customers and the cost of the group to the company.

Software development group meetings were held at the start of each week. At this meeting, the group reviewed the work done and action items closed during the previous week, the target for the current week, action items open, daily work done by individuals within the group for the previous and current weeks, and any off-site visits to be carried out during the current week. This allowed the group to be updated on the status of projects, and gave a formal forum for discussion around problems that existed within any projects.

## 5.0 Analysis of Changes due to Intervention

### 5.1 Organisation Processes

At the end of the research period, a number of procedures, specifications, guidelines and templates had been written in an effort to streamline the software processes within the organisation, following the actions recommended by using the Software Process Matrix. There was still an emphasis on quality assurance and testing of the product. There was no interest shown in the attainment of formal recognition of the process, as there was no market requirement for ISO9000 or for any other measurement such as the Capability Maturity Model or SPICE.

When new personnel were recruited, it was normal for them to be trained on the job. The software development manager had recognised the need for formal training and, although it was being organised on an 'as needed' basis, training was being provided for some members of staff. There were no training plans available for individual employees.

The software development group had moved premises into the main company building. In these new offices, they were located in the same open plan area as the customer support group. This new arrangement had a positive effect on the communication flow within the software development group and between them and the customer support group. Discussion of customer problems was no longer an effort. Prior to this move, the lack of communication between the groups had been a source of complaint by the software engineers.

At the start of the research period, a Lotus Notes based system was used extensively within the organisation. The instigator of the system told the researcher:

> *"So, I wanted to use Notes because it is good for sharing information and I could get at the information. One thing I noticed as a difference between us and (previous company) was that everyone had the information and it was always there. Even just little discussion documents."*

One of the difficulties with using such a system was that, even though a lot of useful information about maintenance and development tasks existed and was available to everyone, there was no format where one could see the full picture within a project. This situation had changed with the implementation of the intranet, which put less emphasis on Notes within the company. It was being used extensively to store documentation, including procedures and test plans. The intranet was structured in such a way that all

documentation on each project was filed together, and people could access what was needed easily. As access was available to all company employees, including those overseas, installing it was a major advantage to Computer Craft.

**5.2 Customer Management**

Customer management had changed significantly during the research period in Computer Craft. Initially, the software development group had little contact with the customer, and regarded the sales representatives as the customer. The developers had some reservations about not meeting the customer earlier in the requirements process. Because of action 2, in the course of the Banking Organiser project, the software developers in Computer Craft had direct contact with the customer. The software development manager had visited the customer site and the project manager in the client company had been actively involved in specifying the product. Comments from the software development group indicated that this changed way of working contributed to the success of the final installation of the product.

During initial research, the company had neither procedures nor guidelines for the collection and documentation of customer requirements. Documentation and collection methods varied between software engineers, even when they were working on the same project. By the end of the research period, a template for the program specification had been introduced, containing a section on requirements. This replaced the previous requirements, functional and technical specifications. Software engineers working on one project were providing and working from consistent information and the customer was satisfied with the output.

The usefulness of the guidelines was evident to one of the developers who took over a specification when it was about 25% written, and was able to complete it successfully. This was not the case with a previous project that he was assigned upon joining Computer Craft. On that occasion he "*needed to sift through a file and find out what has to be done*". As there were no specifications he had found that he was "*asking the customer questions which were asked before*". The developer had found this a very difficult situation to be in.

The existence and use of the specifications did not prevent "*feature creep*", but contributed to its reduction. Unlike the Data Function product, there were no modifications required after implementation of the Banking Organiser.

**5.3 Implementation**

In the initial stages of the research project, the customer services manager installed software within the customer company, sometimes involving the software development group. For the BO project, the software development manager carried out the implementation, and no problems were experienced. A procedure for installation had been written, again by the SQA engineer, an appointment had been recommended in the output from the Software Process Matrix. One of the difficulties faced by the company was that they did not specify the release criteria for a product. This had been discussed, and the feeling of one software development meeting, which a researcher attended, was that a release criteria stating the minimum amount that should be contained in a released product was needed. When asked a measure of failures being shipped, the Software Development Manager suggested that "*one failure in every three is bad quality*".

**5.4 Project Management**

The focus of the software development group had changed throughout the research period. By the end of the research, the software development manager was considering his group as a business unit, and needed to identify income and expenditure. This, in turn, had an effect on the way in which he managed projects. Schedules were set in conjunction with the customer rather than being dictated by the customer services group. Engineers were expected to give the manager feedback, and thus, he was able to keep a tighter control on the project. This was also a recommended action from the Software Process Matrix. The only difficulty was that the software engineers would have liked to know their deadlines and deliverables earlier in the project timeframe.

Software development group meetings continued to be held, and these were seen as a useful forum for discussion. While the manager kept the team updated on the detailed status of the Banking Organiser project, they would have preferred to have a global overview of the project.

Risk analysis had not been used when the researcher first visited the company, but had since been included in an unapproved procedure available on the intranet. In the specification document, there was a section for discussion on risks, but this had not been completed in all specifications.

### 5.5 Analysis of Software Process Matrix Implementation

Following the intervention by the researcher at the beginning of this research project, Computer Craft were to implement six action items based on the top 10 practices which had been identified from the Software Process Matrix. Because a number of key personnel left the company soon after these were identified, some of them were not implemented.

Reviewing the six priority actions we can say that two were implemented in full (numbers 2 and 3), two were partly implemented (numbers 4 and 5) and two had not been implemented at all (numbers 1 and 6). In the terms of the ten practices identified and proposed by the SPM, three were not completed during the research period and another three had been implemented but not formalised with the organisation. Of course it is important that all be implemented, as the evidence from the Banking Organiser project is that the actions that were implemented aided the improvement of the software process within Computer Craft.

### 5.6 Analysis of Change in Computer Craft

Overall, many changes were made in Computer Craft during the research period. Probably the most significant of these was the manner in which the customer was dealt with. The software development manager worked closely with the project manager in the client company, and the group produced specifications which contained the customer requirements. Changes to these were discussed with the software engineer, modifications were normally made to specifications and 'feature creep' was not prevalent on the project. This in turn improved both the testing and implementation of the product. Testing was another process which changed significantly. Plans were written based on the specification, were detailed, and were followed closely by the software engineers. User acceptance testing was done. Again this had a positive effect on the implementation of the product.

One disimprovement which occurred was the reduction in emphasis on code reviews. These had been used when the researcher first visited the organisation, but had been all but eliminated for the Banking Organiser project. This should be reviewed by the company. The researcher also observed that there were no standards for screen design, although these could be easily written based on the U.S. product, Data Organiser.

It is evident that the implementation of the practices identified when using the Software Process Matrix was partially responsible for the improvements in Computer Craft's software process. However, there was a need for greater management support for the project, particularly when the company was going through recruitment difficulties. The identification of the required practices was not sufficient. It was important that the exercise was followed through and the practices be implemented within the organisation. However, the software development manager had recognised that changes were required and had used the Software Process Matrix as a basis for identifying the most relevant changes.

The company has not identified any market requirement for the implementation of any particular standards such as ISO9000 or for being assessment using SPICE or the Capability Maturity Model. It is possible that customers may look for this in the future, and the company is now better placed for proceeding with such actions.

## 6.0 Comparison with other Companies

While the researchers were intervening within Computer Craft, a similar intervention took place within DataNet. This company had been founded two years previously. It employed nine people, three of whom

developed software. At the same time, the software processes within two control companies – Ríomhaire and Software Solutions – were also investigated.

In both action research companies, Computer Craft and DataNet, the successful implementation of actions from the Software Process Matrix caused positive improvement to the software process, particularly to customer management and project management. What SPM provided in both companies were the actionable first steps and pressure for change which were combined with other factors previously existing in the companies: leadership and vision, capable people and effective rewards. In DataNet, pressure for change had also arisen from the focus to become ISO9001 certified. Positive improvements were also evident within Software Solutions; in this case, pressure for change was there because the company wanted to become ISO9001 certified. In Ríomhaire, there was little evidence of improvement within the organisation, mainly because the company was already ISO9001 certified. Their current goal was not improving the current process, but rather ensuring that the process would continue to attain this level during audits. However, in the future, this lack of change in the software process could cause difficulties in Ríomhaire, particularly if customers require a software-based standard such as the Capability Maturity Model or SPICE, as, according to Paulk, "ISO9001 describes only the minimum criteria for an adequate quality-management system, rather than addressing the entire continuum of process improvement" [9].

In the context of the small indigenous software development company, the success of the software process is crucially dependent on the quality of the developers. During their study of the four software development companies, the authors noted that it was not very common for developers to be checked as to whether they carried out a process correctly. Given the resource constraints in a small company this is not surprising, but it implies a high level of trust between the software development manager and the employees. A quality culture must be based on good processes, professionally implemented. Such a small group can not afford the overhead of checking and re-checking that would otherwise be needed. Capable people are indeed fundamental to success within a small company.

## 7.0 Summary

The objective of this research was to effect change, change that would be long-term and remain after the completion of the research. In Computer Craft, change occurred because the company used the Software Process Matrix to identify prioritised action items. Four processes which changed significantly were organisation processes, customer management, implementation and project management. These processes were improved because of the emphasis given to them by using SPM, and procedures were written to encapsulate these improved processes. While it cannot be stated emphatically that the change that has occurred is long-term, procedures which had not previously existed within the action research companies were written as a result of this research project. There is also a requirement within the company to use the procedures and evidence that they are being used to good effect. Only time will tell whether the initial impact will be sustained.

## Acknowledgements

The authors would like to thank their colleagues at the University of Limerick, particularly Prof. Eamonn Murphy, and all of the participating companies who made the research possible. The reviewer's comments were very helpful and the final version of this paper was prepared with the facilities of Linköping University, Sweden.

## References

[1] Peterson, Bill, Transitioning the CMM into Practice in *Proceedings of SPI 95 - The European Conference on Software Process Improvement, The European Experience in a World Context*, 30th Nov-1st Dec, 1995 Barcelona, Spain, pp. 103-123.

[2] Draper, Lee, Kromer, Dana, Moglilensky, Judah, Pandelios, George, Pettengill, Nate, Sigmund, Gary, Quinn, David Use of the Software Engineering Institute Capability Maturity Model in Software Process Appraisals, output from *CMM v2 Workshop*, February, 1995 Pittsburgh, Pennsylvania, U.S.A.

[3] Peterson, Bill, Software Engineering Institute, *Software Process - Improvement and Practice*, Pilot Issue, August, pp 68-70, 1995.

[4] Debou, Christophe, Kuntzmann-Combelles, Annie, Linking Software Process Improvement to Business Strategies: Experiences from Industry in *Proceedings of SPI 98, The European Conference on Software Process Improvement*, 1-4th December, 1998, Monte Carlo.

[5] Richardson, Ita, 1999. "Improving the Software Process in Small Indigenous Software Development Companies using a model based on Quality Function Deployment", PhD Thesis, University of Limerick, November 1999

[6] Richardson, Ita and Ryan, Kevin, "Software Process Improvements in a Very Small Company", in preparation.

[7] Richardson, Ita, 1997. "Quality Function Deployment – A Software Process Tool?" in *Proceedings of the Third Annual International QFD Symposium, Vol II*, Linköping, Sweden, October, 1997. pp 39-50.

[8] Burgess, R.G., 1982. "Some Role Problems in Field Research" in *Field Research: a Sourcebook and Field Manual*, Robert G. Burgess, editor, George Allen & Unwin (Publishers) Ltd., London, pp 32-45.

[9] Paulk, Mark C., 1995. "How ISO9001 Compares with the CMM", *IEEE Software*, January, pp 74-83.

# Using XML as a QA Foundation Technology

**Richard Vireday**
Sr. Software & Release Engineer

**Steven B. Augustine**
Sr. Technical Marketing Engineer

Managed by
Intel® Internet
Authentication Services

## *Abstract*

We began exploring using the structured format of XML to capture and pass on the output of build logs and regression tests into project databases. But it also shown promise in reducing the mish-mash of file formats, tools, spreadsheets, Post-its and other information any project generates. XML is easily displayable to humans, as well as parsed and used for automation activities.

Key Words and Phrases: XML, Regression Test, HTML, Web, Unit Test, Component Test, Integration Test, COM, Portability.

**Richard Vireday** is a Sr. Software and Release Engineer with Intel Internet Authentication Services. He has worked at Intel for 16 years. Some of the hats he has worn include CAD/CAE Developer, PLD/FPGA hardware and software designer, Release and Installation Engineer, and QA Team Lead. He has a Masters of Computer Science and Engineering from OGI (Oregon Graduate Institute), holds three patents, and is a long time member of both ACM and IEEE. Email: *Richard.Vireday@Intel.com*

**Steven B. Augustine** is a Sr. Technical Marketing Engineer with Intel Internet Authentication Services. Before coming to Intel, he co-founded and served as VP of Product Design and Marketing for Deschutes Digital, Inc., a provider of wireless agribusiness solutions.

# Using XML as a QA Foundation Technology

**Richard Vireday**
Sr. Software & Release Engineer

**Steven B. Augustine**
Sr. Technical Marketing Engineer

## 1 Introduction

A fundamental source of problems with QA and Testing is the high volume and wide scope of data that must be generated and reviewed on a constant basis. The product QA and Test methods and tools that our department had been using for years certainly provided abundant evidence of this class of problems. Most of the tools produced output logs in clumsy or proprietary formats, our primary report generation tool was a text editor, and developer-written unit or component tests were virtually nonexistent. When we were planning for our next cycle of products, we decided to use the opportunity to address. Of particular importance to us was the creation of new methods and tools for product QA and Test that would allow us to be less reliant on proprietary and clumsy legacy output data and files while automating all redundant or trivial tasks to make more efficient use of our (always) limited manpower.

We initially began looking at using XML for Build and Integration results. Sending the build logs to our project server, we could easily make information available that was often buried in build scripts and logs. Scaling up to include test results was a logical progression.

Before continuing further in the year 2000, we anticipated multi-platform development needs, as well as the ability to test products in umpteen different configurations and flavors. Our testing needs have been exploding with each new release. With the cooperation of one development team to actually help create Regression Tests, if they only had templates, we explored the regression testing requirements of our existing tools, and that of some vendors as well.

We have just started using a Regression Test Harness for Unit, Component and even Integration testing. One of the abilities we wanted to exploit with XML was storing all of the possible test results into a project server database. Then the results of local runs could be compared against whatever a Developer or Tester determined what was best. Multiple PASS/FAIL scenario results for different platforms could be kept in the database.

In addition, testing C++ COM components in Windows™[1] environments was a particular requirement we had to address. To help tackle this need, we developed a COM interface to components. This dumps the component's internal state to an XML database, and can be used by Developers, Build and Integration and sometimes by final Validation team members.

The final result is that our current plans are for extensive use of XML as a foundation technology in the entire project development lifecycle. Our efforts are just beginning in 2000, and by the time the paper is presented in the fall 2000, we will be able to report better on the successes and failures of this changeover to XML in our support infrastructure.

[1] All Trademarks and Copyrights are properties of their respective owners.

In a tiny bit of irony, this paper was done with Microsoft Office 2000 Word. XML/XSLT enabled tool. See Section Office 2000 section below. All **PNSQC 2000** submissions had to be in Word format this year, so the entire conference is using XML.

# 2 An Introduction to XML

XML, which stands for the e**X**tensible **M**arkup **L**anguage, is derived from the Standard Generalized Markup Language (SGML) standard. As such, XML is similar in structure and format to its cousin, the HyperText Markup Language (HTML). The two, however, serve very different yet complimentary purposes – HTML is a language designed to communicate the *display* and *layout* of a document, whereas XML communicates the *semantics* of a document.

More completely put, XML is designed to communicate complete, detailed descriptions of objects or documents between business partners across the Internet. However, because XML was designed for maximum flexibility, it is well suited for any application demanding platform-neutral, human-readable, machine-parseable structured messaging or data storage. (See "XML Applications," below.)

Another key difference between XML and HTML is that XML defines the concept of document *well-formedness* – a *well-formed* XML document must strictly adhere to the XML standard, which is both rigorous and unambiguous. In contrast, HTML defines no such requirement, which, coupled with HTML's loose and often ambiguous structure, leaves the exact interpretation of an HTML document up to the parser. Clearly, this behavior is undesirable when attempting to communicate a complete, detailed description of an object or document in an electronic message – the message must be parsed correctly and predictably regardless of platform, implementation, or communication protocol. Well-formed XML attempts to answer this issue.

In addition to being well-formed, a document may also be considered to be *valid* XML, meaning that, in addition to adhering to the XML standard, the document also conforms to a given *schema (or validation specification)*. By agreeing on a message schema and requiring all XML messages to be valid with regard to that schema, business partners can insure that their XML messages are parsed correctly and predictably *and* understood fully and correctly. In summary, a *well-formed* document is syntactically correct, and a *valid* document is both syntactically and semantically correct.

Perhaps the best selling point of XML is that it is free, having been created for the public domain by the World Wide Web Consortium's (W3C) XML Working Group.

## *2.1. XML Syntax*

A well-formed XML document is composed of a *prolog*, one and only one root *element*, and an optional *footer* largely reserved for future functionality. The root element may contain zero or more child elements, as long as no part of the root element appears in the content of any other element. (In other words, the root element must be unique within the document.)

An XML prolog consists of an *XML Declaration* and an optional *Document Type Declaration*. The XML Declaration declares the version of XML that the document's syntax conforms to and, optionally, the character encoding type of the file and whether or not the document relies on any external documents. The Document Type Declaration is used to associate a Document Type

Definition (DTD) with the document. (See "XML Schemas," below) Since the XML Declaration is always stored at the physical beginning of a file, XML parsers need only read the first few lines of an XML file to determine how to interpret the rest of the document.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<myElement>
</myElement>
```

A complete XML document with p prolog and one empty element

In the above example, the prolog defines two optional attributes: "encoding," which tells the parser that the file is stored using the "UTF-8" 8-bit Unicode Transformation Format standard, and "standalone," which indicates that this document does not rely on any external documents. The "version" attribute must always be included and, at the time of this writing, can only have a value of "1.0" as no other versions of XML exist. Note that the XML Declaration is bracketed by "<?" and "?>"; this syntax is reserved for parser directives.

An XML element consists of an opening tag containing zero or more attributes, an optional body, and a closing tag. The body of an XML element may contain text, other elements, or a combination of both.

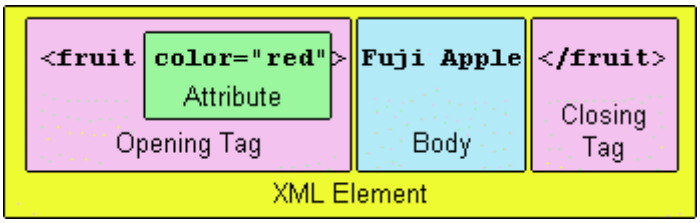



**Figure 1** – XML Element and Tags

At first glance, the element syntax of XML appears to be identical to that of HTML. There are, however, four key differences that are important to note:

1. XML is case sensitive. Unlike HTML, a ROSE is not a RoSe is not a rose. Case counts, so non-UNIX folks take note and watch that shift key. (The W3C recommends that developers use lower case for element and attribute names to increase the compressibility of the XML document.)
2. Elements must nest completely, or not at all. Syntax like "<B>Bold <I>Bold Italic </B>Italic </I>Plain" is not allowed and must be converted to "<B>Bold <I>Bold Italic </I></B><I>Italic </I>Plain".
3. Closing tags are not optional. Unlike HTML, XML requires each element to have a closing tag. Fortunately, there is a shorthand defined for single tag elements: the closing bracket of the opening tag becomes "/>" and the closing tag is omitted, so that "<BR>" becomes "<BR />".
4. Attribute values must be single- or double-quote delimited.

Another important difference between HTML and XML is that there are no pre-defined element types in XML. It is left up to the developer to decide which elements to define and what the meanings of those elements are. In many ways, XML, like SGML, is a "build your own markup language" language.

## 2.2. Overview of XML features

The XML standard already supports a variety of sub-languages designed to define document schemas, perform document translation and filtering, and help link and organize multiple documents distributed throughout the Internet, and even more "feature" sub-languages are on their way to standardization. The following are descriptions of some of the more important of the existing XML feature languages.

### 2.2.1 XML Schemas

When the XML 1.0 standard was originally created, its authors were pressed to create a syntax for the declaration of schemas that could serve as the basis of valid XML. The result was the creation of the XML Document Type Definition language, or XML DTD. The DTD language is a subset of SGML, and as such, is too cryptic and difficult for mere mortals to read and understand. Other failings of the DTD include the inability to specify element data types, lack of support for element inheritance, and a near-zero extensibility.

Enter XML-Structures and XML-Data, two aspects of the XML Schema language proposed initially by Microsoft and nearing final standardization by the W3C. XML Schema is a subset of XML, which means that, in addition to being valid XML itself, it is a great deal more user-friendly than the DTD. Additionally, XML Schema supports element inheritance and stronger namespace support through XML-Structures and the ability to declare element data types through XML-Data. Extensibility is greatly increased through the ability to link to and inherit from other XML Schema documents on the Internet.

In many ways, the evolution from the DTD to XML Schema is like the evolution from C to C++. Both are still equally effective at most tasks, but XML Schema provides better, faster, more powerful, and easier ways to get there. Other functionality, such as data type declaration, is only supported through XML Schema. Like C, the DTD isn't going away any time soon, although XML Schema is quickly becoming the schema definition language of choice. If you are already supporting a DTD, it will still be useful, but you can mostly ignore the DTD chapters out of any new XML books you may buy.

### 2.2.2 The XML Document Object Model

Central to the goal of XML being an easily machine-parsed language is the XML Document Object Model (DOM). The XML DOM defines a standard model and set of interfaces for accessing an XML document using an XML parser, thereby giving programs that need to be XML-aware easy access to a document's underlying object structure. Using the DOM, a client application can traverse, search, translate (see XSL-Transformations) or modify an XML document at the object (element) level, working with object-oriented classes and interfaces instead of lines in a flat file.

By exposing an XML document as a tree of objects, the DOM allows applications to treat an XML file as a structured-storage object that, unlike most other structured storage schemes, has the distinction of conforming to a standard that can be read and understood by any other program running on any platform.

#### 2.2.3 XSL-Transformations (XSLT)

As stated earlier, XML is designed to communicate what a given data set is, not how to display it. The fact is, however, that most data will eventually have to be displayed in a user interface at some point during its lifetime. This separation between data structures and display logic is not at all unfamiliar to any computer programmer; typically, each view that a user interface supports requires its own custom filter to display the program's data structures in some meaningful way.

XML doesn't radically change this paradigm, but it does greatly simplify the process of filtering, or *transforming* data for display through the use of XSL-Transformation (XSLT) documents. XSLT is part of the Extensible Stylesheet Language (XSL) and a subset of XML, meaning that all XSLT documents are valid XML.

XSLT uses special elements called *transformation templates* to define transformations to be applied to elements appearing at a given position within a document hierarchy, or whose labels match a given regular expression, or both. A transformation template contains a series of rules that are used to transform a single XML element or element set into any other textual format, be it HTML, comma-delimited text, or even another XML or XSLT document.

An XSLT document that transforms XML into HTML is generally referred to as an *XSL Stylesheet* as, for any given input XML document, all HTML output documents generated will share the same page formatting and graphical layout. Only the semantics, or content, of the output documents will differ, as this is the only information drawn directly from the XML input.

**Example:** An excellent example of this technology at work is MSNBC (msnbc.com), the Microsoft-NBC News Network. MSNBC accepts news articles in XML format from many external sources, including the Associated Press, Reuters, and local news agencies. While every XML document accepted by MSNBC must conform to a news industry standardized schema, MSNBC may choose not to use all of the information the XML document contains, and, as it is an XML document, the binary order of the elements that it contains is not defined. Additionally, MSNBC needs to maintain control of the news document's final graphical presentation to ensure the consistency of their site's layout, look and feel.

Their solution was to employ an XSLT document to perform a server-side, real-time transformation of the XML news data stored in their database into a standard HTML 1.1 document in response to each user request for a news article. In this manner, MSNBC is able to control which fields of a news article to display and how they should be displayed, even though they have no direct control over the formatting and content of the source data.

#### 2.2.4 XML Query Language (XQL)

The XML Query Language is relatively young, and has a long way to go before becoming an official part of the XML standard. Nevertheless, the needs that XQL addresses are so important that some XML parsers are already including limited XQL support based on the information provided by the XQL Working Group's preliminary XQL documentation.

XQL is a data querying and filtering language, functionally similar to the Standard Query Language (SQL) used by many databases, but conforming to the XML syntax. Using XQL, XML documents containing data sets and conforming to a given schema can be manipulated in much the same way that a database table might be, thereby enabling XML to be a highly portable, if severely performance-limited database language and storage format.

XQL is planned to integrate tightly with XSLT, allowing the former to use complex querying of input documents to find elements to transform, rather than merely relying on simple pattern matching. (XQL+XSLT would, for example, allow the XSLT document to order an array of similar elements by the value of a certain member attribute, or calculate mathematical sums and means of a data set, grouping the results by an attribute or body value.)

#### 2.2.5 XML Links and Pointers (XLink, XPath and XPointer)

Because the author of a given schema determines the meaning of every XML element in that schema, we can no longer rely on the HTML Anchor ("A") element to create links between documents. We could define an XSLT document that would transform some XML element that we want to be a link into normal HTML Anchor elements, but we cannot ensure that the resultant elements will be correctly recognized by every XML application that might try to make use of our document. This is where XLink, XPath and XPointer come in.

XLink provides basic document linking functionality similar to the HTML Anchor, allowing one document to reference another related document residing elsewhere on the Internet. XPath provides a syntax for addressing individual parts of an XML document by element label, hierarchical position, ID, or other search criteria. XPointer allows an XML document to reference individual parts of another document (addressed by XPath), enabling the author to link to (via XLink) or simply include the external content.

Collectively, XLink, XPath, and XPointer allow authors to save bandwidth and increase readability by linking to small sections of target documents by accessing the target document's object model instead of relying on its physical structure (as with the HTML Anchor Label element). Additionally, these three languages further improve on HTML's linking model by providing detailed meta-information about a linked document drawn from the content of that document.

This past year, the definitions of these standards have been greatly changing. Look for even more changes before the end of year 2000.

#### 2.2.6 Embedding within XML

Within each XML document, it is not required to, but it can reference or physically contain the XSLT, XML Schema, and displays it requires for certain output, but also all of the other XML specification technologies. This self-referential feature can make an XML document absolutely self-contained in terms of describing itself.

In practice, XSL and the other supporting formats are usually referenced externally, so that overall project styles can be propagated, without having to update every document containing a static description.

Also, XML documents can completely contain other documents

# 3 Using XML for QA and Development Activities

The main points to focus on are that XML can act as a repository, but its primary function is as an information storage and transmission technology. XML documents can be created and manipulated by multiple independent or linked processes.

Using XML, the starting areas that we have focused on are **Build and Integration Logs,** and **Unit and Regression Tests**. Other areas we think XML will apply, but have not yet explored are:

- Project Metrics (SLOC, Effort, Code Coverage, Code Reviews, etc.)
- Release Note generation management (automatic defect lists, feature changes, review cycles)
- Configuration Information (replacing initialization files, and registry settings)
- Remote Process control and testing (e.g. via SOAP or other XML technologies. Actually directing remote tests via XML based controls).

## *3.1. Build Logs*

Build Logs should meet several not-so-simple requirements.

1) They should alert the builder quickly when a **new** problem occurs.
2) They should allow the builder to quickly trace any problems.
3) And they should provide an audit record of how the product.

Our first implementation was a simple Perl script that reads the results of the Microsoft® Developer Studio 6 .PLG files, and creates a summary page in XML (see below). The Schema should be obvious.

```
<?xml version="1.0" ?>
<?xml:stylesheet type="text/xsl" href="1.xsl"?>
- <buildlog>
  - <docdata>
      <createdate>Tue Aug 15 02:18:23 2000</createdate>
      <hostname>stinky-build</hostname>
    </docdata>
  - <module>
      CredServiceProxyCOM
    - <file log="CredServiceProxyCOM/CredServiceProxyCOM.plg">
        CredServiceProxyCOM.dll
        <errors>0</errors>
        <warnings>1</warnings>
      </file>
    </module>
  - <module
  … rest of log deleted for brevity
- </buildlog>
```

**Figure 4** – Build Log Raw XML Contents

Applying a simple XSL Style sheet, such as that specified by the xml:stylesheet tag, this Build Log is easily displayed with any IE Browser. This also highlights the use of XML locally on a workstation, not through a Web server.

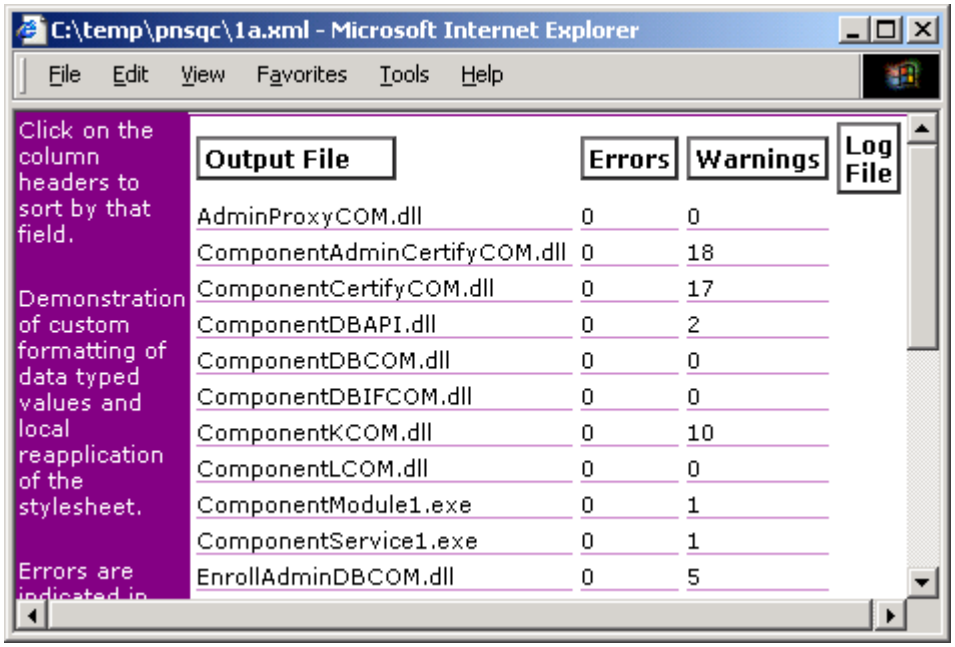



**Figure 5 – Build Log displayed in Web Browser**

The individual .PLG files are collected and then saved with the summary to where the completed build is delivered or archived.

This first demonstration was completed in about 5-6 hours of developer work time, Perl script and XSL formatter included. Over one-third of the development time was spent changing and experimenting with different XML Schemas (information in specific tags vs. attributes, what is the hierarchy). Another third was learning XSL as a new programming format and debugging it. The rest was just getting the coding done.

**Future:** At the time of the PNSQC conference, an updated version of the build log output will be available, with more coloring, and filtering of the build information to show new. For instance, if the warning messages have changed, new ones appear, or old ones disappear. Expectations are that the Schema will change radically as different representations of the data are explored.

#### 3.1.1 Other Display mechanisms

There are several mechanisms that the XML build logs can be displayed besides a Browser (e.g. Microsoft Excel, PowerPoint, or a Web interface). For the time being, the focus is on local Browser displays. Eventually plans are for a Web enabled Project Database and displays.

## 3.2. Regr – Yet Another Regression Test Harness

Long-term project roadmaps created demand in the project groups for a new Test Harness and associated testing support, especially systems that could leverage test harnesses across multiple generations of products.[2] The Top Requirements were determined to be.

1) Works on any Windows™ platform. (9x/NT), and possibly non-Windows as well.
2) Modest overhead for execution
3) Fairly easy to maintain and modify
4) Can drive GUI as well as Command-line tests
5) Interfaces to COM and Web
6) Cheap. No licensing or fee worries
7) Automation of test results back for quick and easy summaries
   - ***This is where XML comes in, as the data transfer mechanism of test results.***

From these listed requirements, we developed **Regr,** and implemented it as a Perl Module. **Regr** relies solely on the ActiveState Perl for Windows™ Platforms, or an equivalent Perl distribution on other platforms.

***Briefly,*** the **Regr** module provides an API (Application Programming Interface) that wraps a single test or test suite within a support harness and environment. The primary output of this support is an XML report, which can then be filtered or fed to a Project Database.

```
# template.pl - Template Example of using Regr Module to create a Test Suite
use Regr;
use Win32::GuiTest;    # optional

 &Regr::Regr;    # Invoke Regr Module system. Handles all command lines, then
                 # returns control to local Tests() function below
 sub Tests {
        # Demonstrate test scripts and scenarios ...
        my (@cmdline_arguments, @attributes) = @_;

        $ret = Regr::TestExe("test1.exe", "-i test1.in –o test1.out",
                "test1.out", "test1.gold", "test1.diff", "xmldiffmrg");
        if ($ret eq "FAIL") {
                Regr::TestResult($ret, "template unable to continue past first test.  If first fails, all of the
rest of these will not work");
                return $ret;
        }
        # Other test code that can be invoked or written in Perl
        ...COM/OLE Interfaces,  GUI Testing, Registry settings, Platform specific, etc.
        # Invoke another set of tests lower down
        $ret = &Regr::Script("shared/src/test.pl", "-x -y -z", 0);

        # Return value of lower-level tests
        &Regr::TestResult($ret, "template finished");
        return &Regr::PASS;
 }
```

**Figure 6 –** Regr API Template example

[2] Although it seems that every product team has a bad case of NIH, often major pieces of systems do sometimes get cut/pasted across to new generations of test support. So it is with Regr, where the functionality is based on an earlier test system of the same name developed in part by the first author.

.

### 3.2.1 Regr Output

The Schema for Regr output is still not settled at this time. Below is a partial working Schema that is going to be the output of Regr.

```
<?xml version="1.0" ?>
<!-- edited with XML Spy v3.0.7 NT (http://www.xmlspy.com) by Michael Jeronimo -->
<Schema xmlns="urn:schemas-microsoft-com:xml-data" xmlns:dt="urn:schemas-microsoft-com:datatypes">
  <ElementType name="Passed" content="textOnly" />
  <ElementType name="TotalTests" content="textOnly" />
  <ElementType name="Successes" content="textOnly" />
  <ElementType name="Failures" content="textOnly" />
  <ElementType name="Success" content="textOnly" />
  <ElementType name="Message" content="textOnly" />
  <ElementType name="File" content="textOnly" />
  <ElementType name="Line" content="textOnly" />
  <ElementType name="FunctionResult" content="eltOnly">
    <element type="Success" />
  </ElementType>
  <ElementType name="Failure" content="eltOnly">
    <element type="Message" />
    <element type="File" />
    <element type="Line" />
  </ElementType>
  <ElementType name="UnitTestSummary" content="eltOnly">
    <element type="Passed" />
    <element type="TotalTests" />
    <element type="Successes" />
    <element type="Failures" />
  </ElementType>
  <ElementType name="TestSuiteSummary" content="eltOnly">
    <element type="Passed" />
    <element type="TotalTests" />
    <element type="Successes" />
    <element type="Failures" />
  </ElementType>
  <ElementType name="UnitTest" content="eltOnly">
    <group minOccurs="1" maxOccurs="*">
      <element type="FunctionResult" />
    </group>
    <group minOccurs="1" maxOccurs="1">
      <element type="UnitTestSummary" />
    </group>
  </ElementType>
  <ElementType name="TestSuite" content="eltOnly">
    <group minOccurs="1" maxOccurs="*">
      <element type="UnitTest" />
    </group>
    <group minOccurs="1" maxOccurs="1">
      <element type="TestSuiteSummary" />
    </group>
  </ElementType>
</Schema>
```

### 3.2.2 GUI Testing

For GUI processing within **Regr** scripts, use of the Win32::GuiTest module[3] has been made the primary method. This module was developed by Ernesto Guisado and is freely available on www.cpan.org.

**GUI Testing** – Is too large a topic for this paper. We continue to explore various methods and GUI Test tools for driving GUI applications (including Browsers). Although we will continue use of various commercial tools for testing tasks (such as Silk, Visual Test, SQA Robot, etc), use of Win32::GuiTest is increasing. It requires seemingly more programming skills than the others, but often not as much. And solutions such as spreadsheet generator tools that non-programmers can use have been successful with other tools.

Eventual goals would be to have a Platform independent version of Win32::GuiTest, for Unix and X and other windowed shells. However, that is not a priority for the team at this point in time.

### 3.2.3 Issues with XML and Regr Test Harness

- XML is lousy for logging - Remember that XML documents cannot be partial or incomplete. For error recovery during tests, most test systems must have a way to clean up the logs, and have provisions for crash and error recovery.

  To address these issues, **Regr** traps errors and recovers like any well-behaved harness, but also produces only **ONE** XML file per test script. Further, the top-level summary is never left in a partial or incomplete state. Each parent XML document links to the rest below it, and the overall summary results are passed back up the tree to the top-level summary. The actual results for each script are stored in multiple XML files. This aids debugging, but has other consequences.

  *Consequences:* When the results are gathered for local analysis, they must either be merged into a single XML document post-facto, or the test directory tree structure for referencing the sub-files must be maintained. (The Project Database can also be used to merge the results, but that is mostly useful for post-facto examinations.)

- Large XML editing. This works better with a specialized Editor. Although RAW XML is fairly easy to correct and modify, it can get involved. There are some decent XML Editors available now, some for free. None can be recommended at this time, because they each have a different set of deficiencies.

- Unit Test output that is not in XML. Existing tests can usually be converted, or a filter created if necessary. If the Unit Test output can be made XML, then guard banding, range checking and some hard validation issues are more easily solved. The hard work is developing the Schemas that developers of the Regression Tests can write to. This is where the IRegrTest COM Interface and other similar regression test tools will come in. (See IRegrTest further on in this paper.)

[3] Used with minor local modifications to the module to add MoveWindow() and SetActiveWindow() calls. Win32::GuiTest versions after 0.6 will have these modifications.

- XSLT - XSL Style sheets – Like DTD's, XSL style sheets are an interim step. Although learning how to use them is useful, there are newer technologies from W3C coming that will generate XSL from a more abstract level. Look for XSL/XSLT to change quite a bit in the coming year. Some time must be spent learning XSL to convert to HTML or Excel or other formats for display. Often the time-trap is continual tweaks to the output for minor formatting issues. This is where providing multiple style-sheets can be very useful.

#### 3.2.4 Future Directions

- XML diff tools – Rational Corporation provides a useful XML Diff+Merge tool with their base ClearCase 4.0 product, and there are other tools now emerging as well. We have identified a need to sometime do inverse merges, only outputting the items that have changed. (Although, this should really not be too difficult to program with Perl or XSLT. There are some XSL scripts already available we have not tried.)
- XML Schemas to Validate Test results – Using Schemas to validate test results may be possible, where certainly just the base values are sanity checked and guard-banded.

  Then also using XSL processing to do some of the difficult difference comparisons would appear possible. In general, the idea would be to provide an XSL style sheet for each Regression Test, and a proper set of outputs and guard bands that would be allowed. Thus the raw results still exist, but regular XSLT processing is used to flag error conditions outside of normal "acceptable" ranges. Values checked and adjusted for Megahertz, CPU cycles, network speed, etc. No results are ready to report at this time.

#### 3.2.5 Problems everyone encounters creating Schemas

What data do you make attributes, and what do you make fields? We follow a guideline of data size and relevance. In general, most everything is in fields.

- Keep large amounts of data in fields. Not attributes.
- Save attributes for important decision or meta information
- Try not to duplicate information.
- Rely on your database. Filter out unneeded junk going to displays, but not to archives!

#### 3.2.6 Regr Summary

**Regr** Test Harness is the basis of the nightly Regression Test system for our software development group. The hope is that it proves stable and expandable for our normal uses, as we intend to continue enhancing the capabilities over time.

And as Code Coverage efforts are expanded, **Regr** and the growing Regression tests will greatly help reduce the time to collect and analyze results on an ongoing basis.

## 3.3. IRegrTest COM Interface

In conjunction with Regr Test Harness, extensive thought was given as to how C++ Classes and COM interfaces be more easily tested with Automation.

The first solution developed has been the **IRegrTest COM Interface.** The basic solution is an interface class that, under program control, can dump the contents of objects. Most important, it is primarily left up to the developers of the Class or Object to define what is the important information to check for normal validation efforts. Massive static data dumps or dynamic traces can produce large amounts of normally worthless information, so the information from an enabled interface must be carefully chosen to find problems with the least amount of information.

IRegrTest provides Static Tracing, which is a test harness-initiated "snapshot" or "x-ray" views of the component under test's state information and member data. Dynamic tracing is left to printf(), OutputDebugString() and debuggers to implement.

### 3.3.1 Interface Definitions

**IRegrTest**

The **IRegrTest** interface must be implemented by all components. This interface provides all basic regression test functionality accessible through the Regression Test Harness.

**ITestHarness**

The Regression Test Harness implements the **ITestHarness** interface. This automation-compliant interface allows external test scripts and programs to control the operation of the Regression Test Harness. The Test Harness is intended for C++ unit and regression test drivers.

### 3.3.2 Test Requirements

All components that support the Regression Test Harness must, at minimum, implement the IRegrTest COM interface and two of the IRegrTest methods: IRegrTest::Dump() and IRegrTest::EnumObjects(). Implementation of the remaining IRegrTest method, IRegrTest::RegisterTracer() is only required if the component will use dynamic tracing.

A component implementing the IRegrTest interface in conformance with the above requirements is said to be a "testable component."

Static tracing, via the component's implementation of IRegrTest::Dump(), must output all global and class member variables and data. Additionally, IRegrTest::Dump() may output all other state information considered relevant by the developer. This additional state information may include special labeling of binary data or data processed in some way as to add interpretive value. It is the developer's responsibility for the contents of IRegrTest::Dump().

### 3.3.3 Implementing IRegrTest

A two-phase process is used to connect a testable component to the Regression Test Harness. The first phase is Enumeration, during which the harness attempts to identify all testable

components contained by the component under test and its children. The second phase is Registration, during which all testable components identified in the enumeration phase are asked to register themselves for dynamic trace output.

During the enumeration phase, the test harness will call the testable component's IRegrTest::EnumObjects() method, which must return an IEnumUnknown object that will expose all components contained by the testable component through an enumeration interface.

It is the responsibility of the developer to implement the IEnumUnknown interface and the enumeration logic it requires. ATL provides a template implementation of this interface for rapid integration with your existing component.

#### 3.3.4 Output Formatting

Static trace output must be formatted as a single well-formed XML element. This element must be tagged "component", must contain the "name" and "version" attributes, and may contain any number of nested elements, each of which may contain any number of attributes, body text, and/or nested elements as long as the well-formedness constraint is met. Static trace output must not contain an XML preamble, DTD or schema declaration.

The output of a static trace containing no data but conforming to the above requirements might look like this:

```
<component name="MyComponent" version="1.0" />
```

The above element is well-formed, and conforms to the schema for the "component" tag (namely, it contains both a "name" and a "version" attribute). However, it contains no data and therefore provides no useful regression test information. Let's add some member data:

```
<component name="MyComponent" version="1.0">
   <object name="CSomeObject">
      <variable name="m_TextString">foo</variable>
      <variable name="m_IntegerNumber">46893</variable>
      <variable name="m_FloatNumber">3.14159</variable>
   </object>
</component>
```

Here, the component element contains three member variables ("m_TextString", m_IntegerNumber" and "m_FloatNumber") belonging to a class (or object) called "CSomeObject". This example meets all previously stated requirements and adds some valuable information.

Please note that the schema shown in the second example is only a guideline; developers may define their own schemas for their components as long as their schemas are well documented, consistent, and adequately describe the component's data, both hierarchically and semantically.

#### 3.3.5 XML Toolset

To assist developers with the formation of static trace output, an XML Toolset Library will be created. This library will completely handle the tasks of formatting a component's data into well-formed XML compliant with the static trace output specifications. Although use of the XML Toolset Library is optional, developers will find it an invaluable aid in the translation of vectors, trees, and other non-scalar variables into XML. Additionally, developers using the XML Toolset Library are not required to document their schemas as the library automatically formats all input to a known schema.

### 3.4. Additional Unit Test Interface

A similar **IRegrTest**–like interface, IASTest, has also been under development in parallel at Intel. Below are the results of the current prototype. In conjunction with the **Regr** test harness, the directions are currently to use this additional test harness for standard unit tests, IRegrTest for COM-based objects, and **Regr** to manage the more complete Integration test suites.

Our thanks to Michael Jeronimo of Intel, for allowing us to show the current output of the IASTest prototypes.

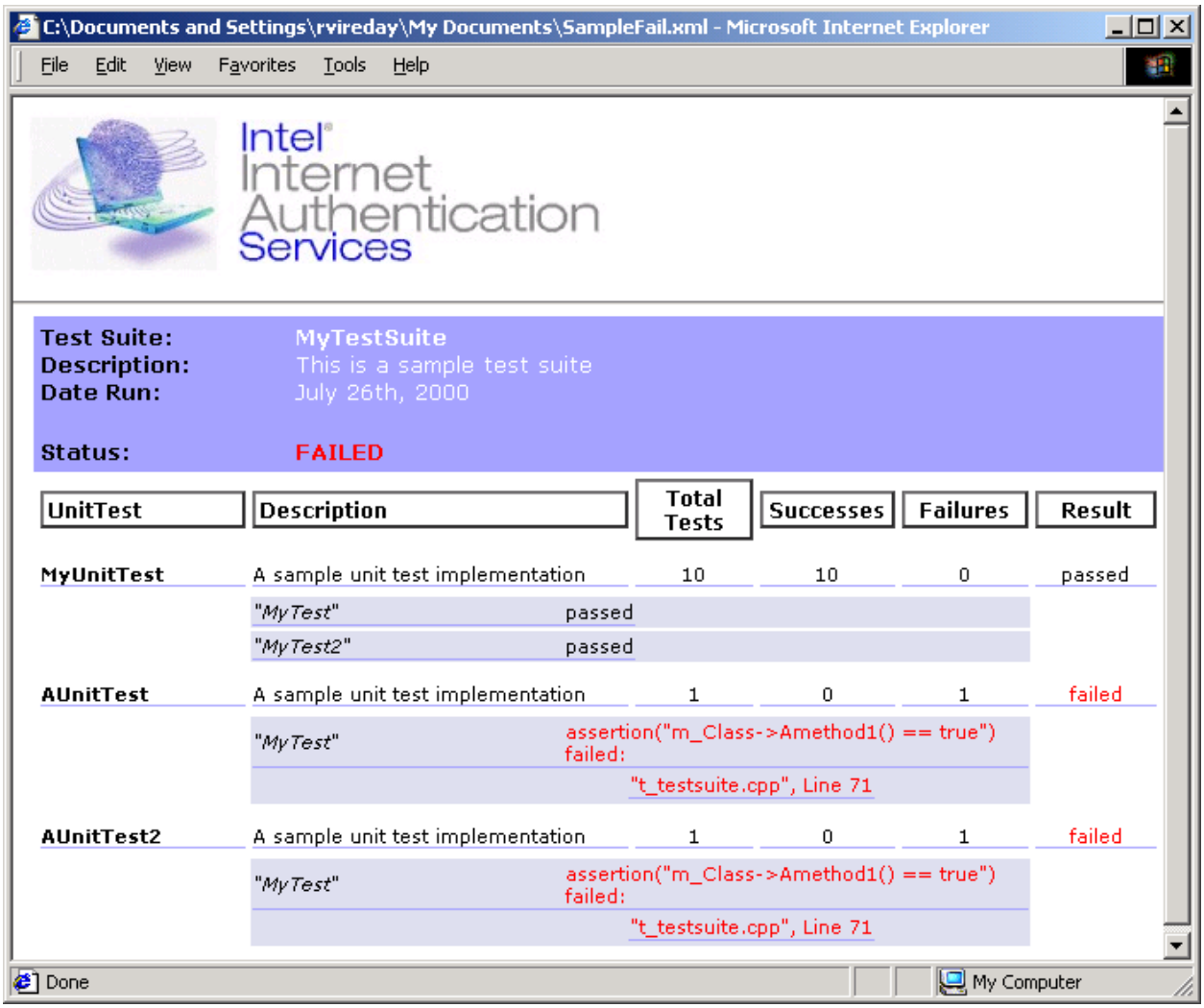
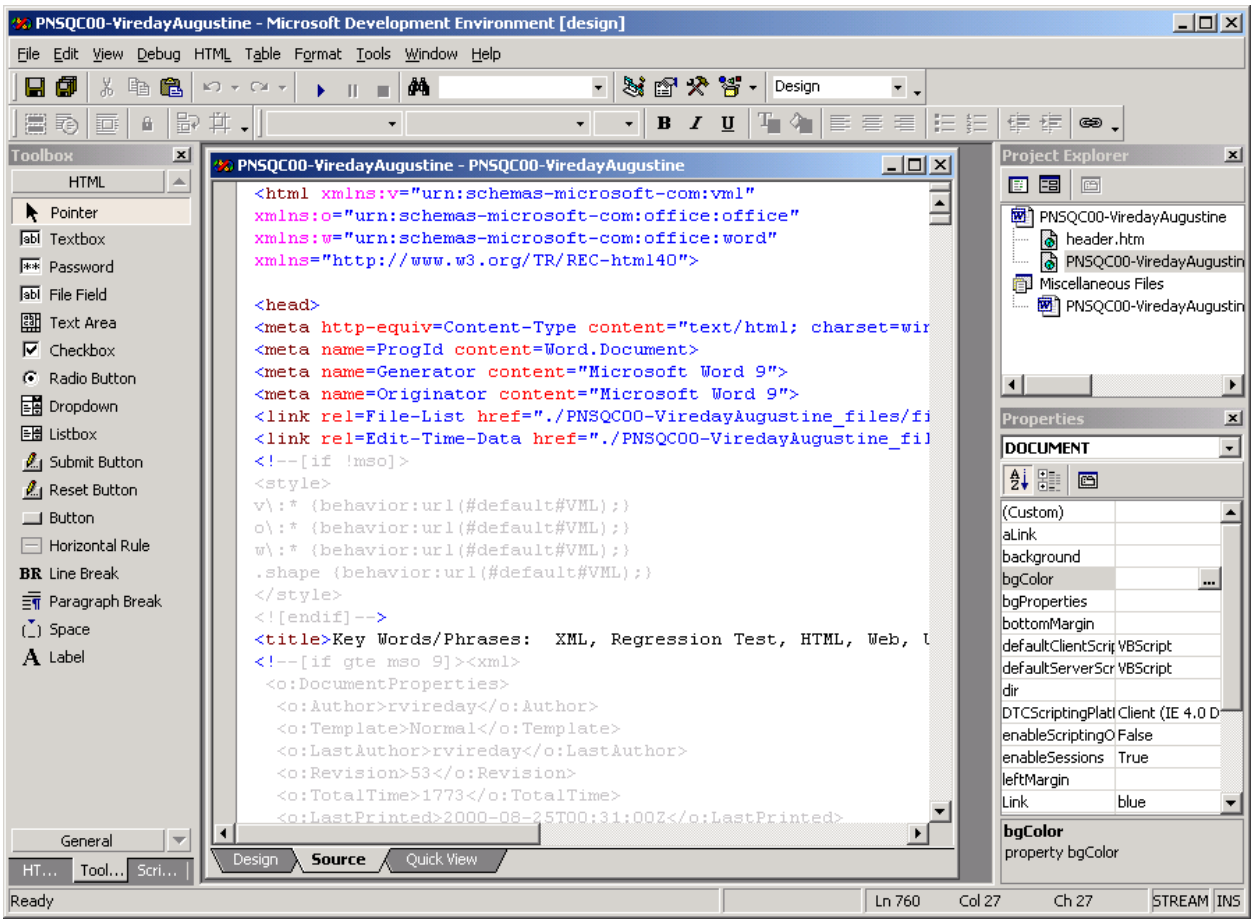



**Figure 7** – Additional Unit Test Output

# 4 Other XML Uses - Office 2000

Opening this Microsoft Office 2000 Word documents with Developer Studio 6.x, it is interesting to view the Schemas that have been used. There are literally hundreds of B2B schemas currently available at http://www.wc3.org, but few so far for QA and Software Development activities.




# 5 Summary

XML-based Technologies were developed to facilitate communication between co-operating Business partners for e-commerce interchange. Further, the technologies seem to allow solutions to common data management processes in the Software Development Lifecycles.

The continued support of XML/XSLT/XHTML and related technologies by W3C and commercial companies appears to have taken a strong movement in the last 12 months.

Our initial experiments and explorations of XML have been encouraging, and we intend to expand our use of this technology.

# 6 Acknowledgements

Thanks to Michael Jeronimo of Intel IAS, for his prototype Unit Test output, and a working Regr schema to use.

Finally, thanks to our families, for their patience in the time taken away to write this paper and do the regular jobs.

# 7 References

www.xml.org - for lots of XML specifications, schemas and vocabularies.

www.xml.com - run by O'Reilly books.

www.w3c.org - the main information store.

www.activestate.com - ActiveState Perl – a very good Win32 distribution. Has all of the core Perl modules, installs and uninstalls cleanly.

www.cpan.org - for Win32::GuiTest and all other things Perl.

*"Getting Started with XML Programming",* URL http://www.xml.com/pub/norm/part1/getstart1.html. This two part introduction has an excellent set of examples for Perl to use XML::Parser and read/write configuration files.

IRegrTest COM Interface, Regr Perl Module – location of all materials referenced in this document will be made at the PNSQC conference.

URL: http://msdn.microsoft.com/vstudio/nextgen/technology/adoplus.asp. Take note that Microsoft® Visual Studio 7 will have much more built-in XML support and usage. Also, this has a good overview picture of XML.

# Trials and Tribulations of Testing a Java/C++ Hybrid Application

**by**

**Steve Whitchurch**

**Mentor Graphics, Inc.**

**Email: steve_whitchurch@mentorg.com**

**503-685-7945**

**Abstract**

The project was to build a viewer (StreamView) that would take a GDSII (Graphical Design Data) Stream File as input, and display it using as little system memory as possible, and as fast as possible. GDSII design files can be Giga Bytes in size, and use a lot of system resources. The underlying code would be written in C++, while the user interface would be written in Java. These two layers would then communicate using the JNI. This combination of C++, Java, and the complexities of an application like this, generated a whole lot of questions on how to test such an application. This paper will talk about some of the issues, and how as a team, we solved them.

**About the Author**

Steve Whitchurch has been in the Software QA arena for 17 years. During that time he has worked at Intel, Mentor Graphics, Summit Design, Tektronics, and is currently the lead QA Engineer for a new product line in the Custom IC Division of Mentor Graphics. Steve has been involved in testing everything from Real Time Operating System Software, Video Editing and Special Effects Software, to Electronic Design Automation Software. Steve has also been active outside of the work environment as a Speaker at PNSQC and STAR. Steve was the creator and publisher of the Software QA Magazine (now known as Software Testing & Quality Engineering Magazine, published by SQE). Steve was also involved in starting the Software Association of Oregon's QA Special Interest Group.

**QA & Development Roles**

As with any new product, StreamView started life as a proof of concept project. The team consisted of two engineers. A very senior Development Engineer, and a senior QA Engineer. For most proof of concept projects, it's very unusual for management to assign a QA Engineer to a project while its in the prototype phase.

While in the proof of concept/prototype phase, QA's role was to perform mostly unit testing. These unit tests were written in C++, as the Java layer was not yet part of the project.

As the project moved from a proof of concept project to a real product project. More people were added to the team. The additions included a GUI Development Engineer, a Middleware Development Engineer, and two part time QA Engineers.

As the project made this transition, most of the QA tasks bubbled up to the GUI layer, as well they should have. With the Development Engineers now focusing on more of the unit testing, QA's tasks needed to focus not only on the wellness of the application, but on the way a user would use the application.

In the case of StreamView, the user test cases were a variety of differing GDSII Stream Files (Customer Designs). These Stream Files would prove to be very valuable during the testing phases of the project., Even with the transition of QA to a more traditional role, there was still some unit testing being performed by QA. This testing was now assigned to a new collage grad that long term, would move from QA to Development. This was a very good fit for Unit Testing, and the new Engineer.

At the start of the project, it made sense to have a QA Engineer take on the role of Unit Tester. It gave QA the opportunity to learn the application, and to have input into the design process. It also made sense to move the Unit Testing responsibility to a new Development candidate later in the project. This was a win-win opportunity for the team and for the new Development Engineer.

As the application moved closer to a release date., Some of the Development Engineers performed testing tasks., Roles were once again changed to fit the needs of the project. Everyone on the project wore different hats at different times. Customer Support was also asked to be involved in the testing cycles. Again, this was a win-win opportunity for the core team and for Customer Support.

When assigning roles, don't just assign roles based on preconceptions such as "QA Engineers only test applications at a high level", "Development Engineers don't test", etc. Each member of the team can have a positive impact at all levels of the project. In fact, a sign of a well functioning team, is where all team members participate at all levels of the project. We all have something to bring to the table. We should not be pigeon -holed just because of our title.

**C++/Java Testing Issues**

The testing issues surrounding this seemingly simple application (simple from a functionality point of view) were huge. The following is a *small sample* of questions that were asked by QA:

- Are there any tools on the market to test a JAVA GUI?
- Can we get at all the underlying C++ functionality from the GUI?
- What about testing the C++ code standalone?
- What about testing the JNI layer?
- What about testing the IPC layer?
- If we are using a Beta version of Java, will this have any impact on the testing tools we use?
- What Java Standard should we follow?

- How do we verify the graphics on the JPannel?
- What about tool tips? How do we validate them?
- What about on-line documentation? How do we test it?
- What about cross -platform dependencies?
- Will the Java GUI really look/act the same on a PC as it does on a Unix box?

A lot of these questions are common questions that should be asked of any project/product that needs to be tested. So from that point of view, this project was not all that unusual. But when ever you add more than one programming language, or more than one supported platform, or more than one what ever. The level of testing complexity increases. It's just a fact of life. Let's take a look at a couple of these issues close up.

*What about testing the C++ code standalone?*
We determined early in the project that we would realize a big benefit by testing the application from the C++ side. This would help us flush out problems like memory usage, database issues, function call problems, etc. This assumption proved to be correct. There were a lot of problems found just by writing test cases in C++ that would exercise the C++ application code standalone. You may say that this is just Unit testing, but that is not true. In many cases our C++ tests would call the C++ code just like the Java user interface would. An example; we had one test that checked the drawing functions that were written in C++. This test used the JNI to make these drawing calls, and then created a JPanel to display the graphics. This test was very useful in finding C++ functions that had problems or that were missing functionality. A simple unit test would just exercise one function at a very low level. What we had here was a kind of test harness for the C++ layer of the application. You could also call this level of testing API testing.

*What Java Standard should we follow?*
We used the Sun Java Look and Feel standard. As long as we followed this standard, we could, for the most part, be assured that the GUI would look and run the same on any supported Java platforms. Following this standard was very useful when questions came up about a look and feel of the GUI. I would recommend this standard to anyone building an application GUI in Java. The book is Java Look And Feel Design Guidelines, Addison-Wesley, ISBN 0-201-61585-1.

*How do we verify the graphics on the JPannel?*
This was a big issue for us because our application is very graphics intensive. There's always the old bit map method. But bit maps have all kinds of problems with platform environmental issues. We did not choose this method.

We decided on two methods of verifying the drawing on the JPannel canvas. The first was an automated way of verifying what we thought should be on the drawing canvas. Since there is a one-to-one correlation between what is in the graphics database and what is drawn on the canvas, we were able to check the contents of the database to verify the contents of the drawing. We did this by writing out the GDS data and doing a compare with the original design file. If the compare was good, then the translated data in the

database was good. And most likely the graphics were good. If the compare failed, then we took a closer look at the graphics on the drawing canvas. Most of the time we found a drawing problem by using this method.

However, this was only part of the answer. As our second method, we also needed to visually check the drawing canvas. There could always be a translation problem between what was in the data base and what was drawn on the canvas. Both these methods proved to work quite well.

Since GDSII Design data can have millions of shapes in one file, visual checking methods, could be a nightmare. To help automate this testing, we created small GDSII design files that focused on one type of shape. For example paths, we created a couple of small design files that had every type of path possible, bases on the GDSII Standard.

With any application that does some type of drawing, there is going to be some amount of manual inspection of the data. I don't think there is any way around it.

*What about testing the IPC layer?*
StreamView has the ability to interface to a IC design debugging tool called Calibre-RVE. The way these two applications talk is through an IPC socket. Calibre-RVE reads in a list of design errors, the user selects one of these errors, and then Calibre-RVE sends a message to StreamView to display and highlight the error on the GDSII design.

We tested this mechanism two ways. The first was by just using Calibre-RVE to send commands to StreamView.

The second was an internal tool that would just send Calibre-RVE commands to Stream-View. This tool proved to be very valuable in debugging problems found. Any time you're testing communications between two applications, it's very helpful to have a test fixture that can simulate the communications between the applications.

*What about cross platform dependencies?*
Even though Java is supposed to be platform independent, we did find a couple of platform dependencies. For the most part the Java GUI worked out well. The biggest problem we had was with system fonts. Java has what is called "font.properties files" that define the fonts used for the platform the application is running on. This does not always work.

The other problem that we saw was with the different windowing environments you can have on one platform. For Example, OpenwWindows and CDE in the SUN environment. Sometimes Java would act different on OpenWindows than on CDE.

For the most part I was very happy with the way Java worked. If you follow the guidelines in the Look-And-Feel book, most of the time the Java application will perform the same on all supported platforms.

A good resource for the known Java bug and general information on Java can be found at "www.javasoft.com". You can find lots of useful information on this web site that can help you test or develop a Java application.

*What about tool tips?*
All Tool Tips and on line documentation were tested manually by inspection. The Tool Tips were included as part of the Testing. This proved to be a good way of verifying the Tool Tips and on-line documentation.

**Test Automation Tools**
One of the problems facing the QA team was test automation. The GUI was based on Java, the underlying code was C++, and the two talk via the JNI. How do you automate this mess? The first step was to find a commercial tool that would fill our needs.

As StreamView's GUI became more robust, we decided to test more of the application from the GUI, the Java side. There would still be some Unit Testing performed, and those tests would be written mostly in C++.

We looked at three tools, JavaStar by Sun Microsystems, QA Partner by Segue, and XRunner by Mercury Interactive. We needed a tool that would work with the latest version of Java, would run on two platforms (to include a 3rd platform in the future), and would be able to effectively test a graphical application. The only tool that fit these requirements was JavaStar by Sun Microsystems. This tools was written in Java, so it would run on any Java supported platform, and it would support the latest version of Java,. It did everything we wanted, with one problem. Part way through the project, I received an email from JavaStar Customer Support that said, Sun was dropping the development and support for JavaStar. Not a good day. This basically left us with no testing tools to automate the testing of our new application. The other two tools (QA Partner / XRunner) either did not support Java, or supported an older version of Java, or did not support the platform we were testing on. That's one of the problems you have when your application is using cutting edge technology.

So now what? In comes the Testing Task List, a pure paper way to automate your testing, and what I consider the most valuable tool any QA/Test Engineer can use. I know most everyone thinks of test automation as a push-button set of tests that run on their own, but that is not the only reason we automate. One of the biggest reasons is repeatable tests. The Testing Task List will give you repeatable test cases. Its also a very good source to drive those push-button test cases when your ready to automate.

| Functionality | Test Information | Completed |
|---|---|---|
| **View Panel** | View Panel Func tions | |
| Pan Functions | | |
| View-All | | |
| Icon (Left Side Palette) | View-All Click Icon | |
| Tool Tip | Check Tool Tip | |
| F1 (Help) | Function Help | |
| Help Key | Help Key Help | |
| Key Board | View-All from Key Board | |
| Ctrl-F | View-All using Ctrl- F | |
| Pan-Up | | |
| Icon (Left Side Palette) | Pan-Up Click Icon | |
| Tool Tip | Check Tool Tip | |
| F1 (Help) | Function Help | |
| Help Key | Help Key Help | |
| Key Board | Pan-Up from Key Board | |
| Up Arrow (Arrow Key Pad) | Pan-Up using Up-Arrow key on arrow key pad. | |
| Page Up | Pan-Up using Page-Up Key | |

As you can see from this example, the Testing Task List is very detailed, and simple to execute. This tool can be given to any team member and you will get the very same level of testing from everyone that uses it. It's a very effective and low-tech way to automate testing tasks.

This is what we ended up using for our test automation for the first release of StreamView. We found a lot of bugs using this method. And it provided us with repeatable test cases that could be handed to anyone on the team to perform.

If you are not using something like the Testing Task List, to drive your push-button automated tests, or to drive your manual testing, you are missing the boat as far as having good, comprehensive test cases.

**Project Documentation**

All our project documentation was written in html. This allowed us to have a web based version of all our project documents on line for anyone to read/review anytime.

This documentation consisted of the following:
1. Project Plan
2. Development Task List. This worked very well. At any time you could see the status of the project. Again a very simple, low tech way to communicate the project status. This task list also had the QA tasks listed.
3. Project Test Plan
4. Testing Task List
5. Problem List

All documents, with the exception of the Project Plan, were living documents. They changed as the project evolved.

One of the things I did as Lead QA was add text links from the Development Task List to the Testing Task List. This way all the QA tasks had examples of what was being tested and how. Anyone reading the test Development Task List could click the QA task link, taking them to the testing Task List for that functionality being tested.

**Problem Reporting within the Project**
Mentor Graphics has a commercial Problem Tracking System. We choose to not use it early in the project. As a small team it made more sense to use an Email/Paper system instead.

Our low-tech system consisted of email and a html document that was updated weekly, or sometimes daily, depending on how frequently updates were needed. This system worked very well for the team. Again a very simple, low-tech solution to a problem.

The system worked like this: QA or Development would find a problem. That reporting engineer would send out an email with info about the bug to a team mail group. The engineer responsible for the code would then respond to the email. Once a week the Lead QA Engineer would update the html document with that week's bugs. The responsible engineer would then respond to the bug as fixed by changing a status field in the document.

When the project hit it’s first major milestone, Code Freeze, the team did switch over to Mentor’s in house Problem Tracking System. By doing this we could officially track bugs. This was important for project management to be effective.

**Alpha/Beta Testing**
The build person for the project was the lead QA Engineer. This was a carry over from the early part of the project. This worked very well until we got closer to the release date, then this duty was transferred to the group's build engineer. Just like the bug reporting system, there comes a time when a project must conform to the companies standards and/or processes. The build person would build and distribute the Alpha/Beta builds.

The request for new Alpha/Beta releases always came from Marketing. The Marketing group was the interface between Engineering and the Alpha/Beta customers.

**Conclusion/Future Work**

Something that can be learned from this project is, that you don't need an arsenal of expensive tools to produce a high quality product. We used things like the Testing Task List (a very valuable tool), a paper/email form of a problem tracking system, and good communications among team members, all low-tech methods of software development that proved to be very effective.

When putting together a project team, look for people that can work well together. This is probably the most important and most forgotten aspect of project management. A well oiled team with a good spec and the right skill levels can produce a high quality product. The StreamView Team worked very well together.

If you are testing or developing an application in Java, there are all kinds of one-line and book form resources available. There is also a local group called the Portland Java Users Group " www.solidware.com/pjug". And don't forget the Sun Java site "www.java-soft.com".

One of the future issues that I'm looking at, is test automation. There comes a time that you just can't keep up with the testing task without test automation. One of the problems that we have, and will face constantly as we are developing new applications using cutting edge technology, is that commercial test tool will not be able to keep up. So what's the answer?

I'm in the process of building my own test driver that is written in Java. By using Java to drive the testing of a Java GUI, I have all the power of the Java language to help with my testing effort.

My test-driver uses a custom script language that enables me to write repeatable test cases. An example of this test scripting language:

```
// Sample Test Case
logfile testlogfile
load design.gds
push viewall
push panup -t
close
exit
```

This test script creates a test log file named "testlogfile", it then loads a GDSII design called "design.gds", it then clicks the "View All" icon on the StreamView View Panel, it then get the Tool Tip for the "Pan Up" icon on the StreamView View Panel, it then closes the design, and then exits the application. While this test is running, all the test results are written to the test log file.

The Java code to do this is very simple. If you take advantage of the Java language all you need is a hook in the application under test that gives you a handle to (in this case a JButton) to ViewAll and PanUp. The test driver code would look something like this:

To push the ViewAll Icon:

getViewAll().doClick();

do.Click() is a JButton method that presses the JButton on the GUI. The result of the button press is the call to the action listener associated with the JButton.

To get the Tool Tip for the Pan Up Icon:

logPrint.println(getPanUp().getToolTipText());

getToolTipText() returns the Tool Tip text for the associated JButton. In this example, the result is sent to the test log file.

One other feature that I have added to my test-driver language is the concept of looping. This allows me to write test scripts that can repeat a set of test commands n times. An example of this looping feature is:

```
// Sample Test Case
logfile testlogfile
load design.gds
// Loop 4 times
loop 4
{
push viewall
push panup -t
}
close
exit
```

This will loop through the “push viewall” and “push panup -t” 4 times then close and exit the StreamView.

I have also added C like comments to my test script language.

This is only a sample of what can be possible using Java to test Java. My plans are to continue to explore the possibilities. The prototype of my test-driven tool has shown a lot of promise.

I think with this new test-driver, a good test coverage tool. we are well on our way to a good test automation solution for the next release of StreamView.

The last thing that I would like to say is, it's been a pleasure working on the StreamView team. The people that make up the team are top notch engineers. I learned a lot. Thanks for having me as a team member.

This paper is dedicated to the StreamView Team:
Alan Sherman
Brent Goodrick
Eric Forsberg
Gary Myron
John Thienes
Mark Meeker
Richard Mallery
Yi Liu

# “What do you mean by that?”

# Ethnographic Interviewing as a Software Requirements Elicitation Technique

Christopher Simmons, Mailstop JF3-204 christopher.a.simmons@intel.com
Erik Simmons, Mailstop JF1-46, erik.simmons@intel.com

Intel Corporation
2111 NE 25th Ave.
Hillsboro, OR 97214-5961

Version 1.1, 8/18/00

**Prepared for the 18th Annual Pacific Northwest Software Quality Conference**

**Key Words**: Requirements Elicitation, Ethnography

**Author Biographies**

Chris is currently a Software Quality Process Engineer with the Software Quality Engineering group of Intel Corporation’s Corporate Quality Network (CQN). Chris is assigned as a CQN representative to the Intel Architecture Labs, Intel’s research and development arm. He holds a Masters Degree in Communication from the University of California Davis. His specialization was in the Ethnography of Communication. Prior to his work in the high-tech industry, he served as a member of the Communications Department faculties of the Universities of California and Minnesota.

Erik Simmons has 15 years experience in multiple aspects of software and quality engineering. He holds a Masters degree in mathematical modeling and a Bachelors degree in applied mathematics from Humboldt State University in California. Erik currently works as Platform Quality Engineer in the Platform Quality Methods group, part of the Corporate Quality Network at Intel Corporation.

**Abstract**

Eliciting software product requirements from customers is challenging and often frustrating. The process has been traditionally approached from a combination of psychological and software engineering perspectives. Over the past five years an alternative approach to requirements elicitation has begun to evolve. Researchers in computing and sociology at Lancaster University, England have begun to apply ethnography; a method which has been occasionally used in industrial design, to the software requirements elicitation process. Ethnography uses specialized interviewing and participant observation techniques to account for situational context and capture information in a culture’s own terms. Research on applying ethnography to requirements elicitation has yielded at least one attempt to structure ethnographic techniques into a design methodology. While the potential benefits of using ethnography in requirements elicitation and product design are well documented, implementing ethnographic tools in requirements elicitation has been given only cursory attention in requirements texts. Researchers admit that very few requirements professionals have made practical use of ethnographic tools. By carefully selecting techniques from ethnography and streamlining them for application to software requirements analysis, we will begin to remedy that gap.

The intent of this paper is to increase software practitioners’ awareness and understanding of ethnographic tools so that requirements practitioners can take advantage of ethnographic tools in actual software engineering. Ethnographic principles and interviewing techniques are presented along with the potential benefits of ethnographic methods for improving requirements elicitation and validation. We provide practical guidelines for applying the material to software requirements elicitation and close by discussing the implementation challenges posed by commercial software development in Internet-time.

## Introduction

Requirements analysts embarking on elicitation efforts are like tourists venturing to a foreign country. Unlike an international tourist, however, the traveler usually remains in his or her home culture. The apparent similarities between the requirements analyst and the customer (such as language, nationality, or even industry background) create an illusion of commonality. This leads to dangerous assumptions that the participants "understand" each other when in reality their differences are significant. The purpose of software requirements is to establish a clear and common understanding of what the software must accomplish. Ambiguity is one of the greatest causes of requirements defects. These defects result in delivery of the "wrong" product, poor customer satisfaction and damaged brand perception.

Even when two participants are aware of the differences between them, requirements elicitation is still perilous. When an analyst is not familiar with a concept from the world of the customer, they often ask the question "What do you mean by that?" This question poses few if any problems when obtaining a simple clarification of terms, but when it is asked in relation to a complex business or technical concept, it triggers a natural tendency for respondents to "translate" their answer into terms that match their perception of the elicitor's background. This translation removes the requirements from the customer's language and terms, dramatically increasing the potential for an unsatisfactory implementation of the requirement. Cultural anthropologists who study subcultures within their own culture have long faced this challenge. The effect culture and subculture have on us is so fundamental that it is hard to identify and often goes unnoticed. Ethnographic methods assist in minimizing cross-cultural translation and the contextual contamination of the culture under study. Ethnography's goal is to capture the "native's point of view" (in this case the "customer's point of view") as accurately as possible using native concepts and language.

One tool used in accomplishing this goal is ethnographic interviewing. Recently, many organizations have realized the importance of hearing the "voice of the customer." Ethnographic interviewing allows us to not only hear the customer's voice, but to comprehend the language, behavior, and values of the customer. Only by comprehending these things can we begin to understand what the customer "means."

## Ethnographic Principles

### *Culture, Artifacts and Meaning*

One of the foremost researchers in Ethnography defines it as "...an explicit methodology designed for finding out both the explicit and tacit knowledge familiar to the most experienced members of a culture." Culture is an idea that is complex and difficult to pin down, and its connection to requirements elicitation is a topic that needs some explanation. Culture cannot be observed directly. Culture is an abstract layer of society that lies beneath our perception and manifests itself only indirectly through artifacts such as symbols, art, language, rituals, customs, and daily practices. When an artifact has associated meaning that only a member of the group understands fully, that artifact becomes cultural in nature. Meaning is significant for software requirement because it organizes behavior. The actions of an individual are largely structured by the cultures to which the individual belongs. The usage patterns for software products, and therefore the product requirements, have their basis in cultural meanings.

The best way to describe the connection of ethnography and culture to requirements elicitation is to overview ethnography and its principles and techniques. The field of ethnography is comprised of a set of methods for gathering and analyzing information regarding other cultures. It was developed during the first half of the twentieth century by cultural anthropologists. The goal of ethnographic methods is to richly describe the culture under study in its own terms. This process seems simple at first glance, but in practice it is surprisingly complex.

### *The Native's Point of View*

The concept of the native's point of view is fundamental to ethnography. The ethnographic process is designed to discover and describe meaning from the contextual perspective of the culture under study. As was discussed above, in our particular context, the "world" is the market or need, and the native is the customer or user.

Implementing ethnographic methods helps to ensure that the requirements analyst approaches the customer with a sense of design humility. Technologists are experts in their fields and can add a great deal of value to product definition by means of their technical knowledge and experience. On the other hand, the technologists experience must be balanced with a respect for customers and a desire to meet their needs as they define them. Success requires that we simultaneously play dumb and remain experts. If a software system provides customers with features never knew were possible,

but fails to meet their basic needs, it will fail in the marketplace. To truly understand our customers' needs we must learn from them on their own terms rather than studying, collecting and documenting data about them from a distant, clinical perspective. We must listen to our customers, but more importantly we must understand them.

### *Walking in Rhythm*

Seeing the world from our customers' point of view requires software requirements analysts to change their own perspective. The process of making this transition is a gradual one. Anthropologists refer to this process of transitioning from an external perspective to the perspective of the "native" as walking in rhythm. The purpose of walking in rhythm is minimizing cross-cultural translation.

#### Translation and Thick Description

Every description of another culture is a translation. This translation is both linguistic and cultural. Requirements analysts often document requirements in a language that is significantly different from that of the customer. In many cases, the requirements document will be written for both the product development team and the customer, it will likely use highly structured natural language to document the system requirements in an attempt to eliminate ambiguity. It may also make use of specialized formats such as algorithms, state diagrams, Planguage or formal specification languages (such as Z). Although the intent is to form a bridge between customer and development team, these structures and formats can be a significant barrier to achieving a common understanding. As a side note, it is interesting that a product requirements document is itself an artifact of the software engineering culture that created it.

Although the software requirements documentation process always requires at least some translation, every effort should be made to minimize it. A requirements document that maximizes use of native (contrast with natural) language and minimizes cross-cultural translation significantly increases the chance of creating a software product that fulfills the customer's expectations.

The type of description yielded by ethnographic methods is referred to as "thick description." This method of description differs considerably from scientific description, which is limited to the documentation of observable facts. The basic premise of thick description is that it contains the contextual information needed to assign meaning to a cultural artifact in addition to its observable characteristics. What this means is that not only do we need to account for the artifacts of a culture, we must understand how they relate to each other and how they fit into the cultural fabric as a whole. Thick description is accomplished through the use of the ethnographic interviewing process.

## Ethnographic Interviewing

James Spradley developed the Grand Tour ethnographic interviewing method in 1979. An ethnographic interview is structured with the specific goals of discovering the native's point of view and native language, minimizing cross-cultural translation and allowing thick description. Ethnographic interviewing is essentially fieldwork. Treating the requirements elicitation process as "requirements fieldwork" has some interesting and beneficial results. As Paul Rothstein of Arizona State University notes, "[ethnographic fieldwork] should be understood as fundamentally iterative and emergent rather than linear and self-evident."

### *Spradley's Grand Tour Method*

Our adaptation of the Grand Tour interview method is composed of five basic steps:

1. Selecting informants
2. Asking descriptive questions
3. Performing a domain analysis
4. Asking structural questions
5. Asking contrast questions

These steps are presented in detail in the sections that follow. For the purposes of explanation, the example of a fire station culture will be used to illustrate each step in the process.

#### Selecting Informants

Ethnographic interviewing is conducted with "informants." Informants are respondents that are competent members of the culture under study. They are sources of information that that use native language in their responses and understand the culture intimately.

The process of selecting the right informants is critical to the results of ethnographic interviews. A guideline is that good informants are not only very experienced in the activities the interviewer seeks to understand, but should be involved in those activities at the time of the interview. A good respondent is also one who is able to devote adequate time to the interviews. In order to capture a broad picture of the culture under study requirements analysts should select multiple informants. This practice helps to prevent obtaining a distorted picture of the culture based on only one member's perceptions.

<u>Example</u>

The informant for our sort and test example will be a firefighter from the station we are studying.

**Asking Descriptive Questions**

Ethnographic interviewing uses descriptive questions to uncover the domains of meaning from a culture in its own terms. Domains are organized categories of symbols that have meaning to the members of the culture. Domains are nested structures with a high-level "cover term" usually containing a number of "included terms." These terms are related by a semantic, or meaning-based, relationship. Descriptive questions are open-ended and designed to allow the informant to identify for the interviewer what they, as a member of the culture feel is important to note.

An interviewer approaching an informant with a pre-determined list of questions regarding the topic has two undesirable effects. The first effect is that the interviewer, not the native has established the priority of what is important to discuss and document. This may seem to be a small point, but informants have a natural tendency to feel in a position of less power than the interviewer and generally go in the direction the interviewer points them.
Without thorough knowledge of the native culture, the interviewer risks framing the discussion around his or her perspective and increases the risk of missing an important topic or artifact entirely. In a requirements elicitation effort, the impact of missing a key requirements will be drastically increased re-work and very likely product or project failure.

The second undesirable effect of the interviewer asking a list of pre-determined questions is that the entire conversation will revolve around the language of the interviewer as a result. The vocabulary of the customer will yield to the vocabulary of the requirements analyst and the conversation will begin to drift away from the native's point of view immediately. This vocabulary shift means that less of the customer's culture will end up in the end product and customer satisfaction will likely be damaged as a result. This damage may or may not be severe, but asking descriptive rather than prescriptive questions could easily have prevented it.

Grand Tour Questions

Spradley's methodology takes its name from the first descriptive question asked. This first descriptive question is a "grand tour" question and sets the stage for the entire interview. Grand tour questions are very broad descriptive question that encourage the informant to begin the discussion at the highest level of abstraction. Grand tour questions can be general in nature or they be formed specifically in relation to a task or time-period. Care must be taken in creating a grand tour question so that it both provides some initial direction for the interview and allows the informant sufficient freedom of response.

Mini Tour Questions

As the informant responds to the grand tour question, the interview is focused on identifying domains of meaning about which to ask follow-on descriptive questions. Ethnographers often referred to these as "mini-tour" questions. The interview process is iterative. Once basic domains have been identified, the interviewer asks descriptive questions about them and additional domains are identified. These domains might be part of the larger domain or they might be additional domains at the same level of abstraction.

Other Question Types

There are three other types of descriptive questions that can be used to help interviewers identify and define domains. They are Example, Experience and Natural Language questions. Example questions ask the informant to provide an example of an artifact that has been discovered in the interview process. This often leads to the discovery of additional domains or artifacts. An example of this type of questions would be "Could you give me an example of a three-alarm?"

Experience questions ask the informant to simply recount the experiences they have had in the culture. In the Intel manufacturing environment an experience question could be: "Can you tell me what you do from the moment you receive a call until the moment you return to the station?"

Native language questions ask the informant what the native term is for a concept the interviewer may be familiar with. Asking native language questions help minimize translation by keeping the interview in the language of the native as much as possible. An example of a native language question would be "How do firefighters refer to the hoses?" The informant's response to this question ["We call them the lines."] would yield a native language term the interviewer could use in the interview from that point forward.

As a guideline, keep in mind that when asking descriptive questions, the goal is simply the identification of domains and the symbols they contain. Analysis of the domains to determine the semantic relationship of the symbols they contain, and consequently cultural meaning, occurs later in the process.

Example

Using our sort and test illustration, a good example of a grand tour question for our Sort and Test Manager would be:

> *"Could you describe for me what you do in a typical day at the station?"*

This questions provides the informant a reference point and structure to answer the question (a typical day) while allowing them to identify the activities, items, processes etc. that they feel are significant. Here is an example of a typical response the potential domain cover terms that can be identified in their response are in bold:

> *"Well, we come in and **BS** for a while, then we usually have **scheduled maintenance** which lasts until lunch, we either grab something at **a local pit** or eat in **the galley** here at **the station** with **the Cap**. We typically get 3 to 6 **calls** a day, but sometimes we'll get **a three or four alarm** and **work it** for most of the day. If we have **a quiet day** we focus on **deferred maintenance**. If it's a **shift change day** we **BS** some with the next **shift** for a while, then **log out** and head home. If not, we **finish up the day's paper** and get some shut-eye."*

Based on this response our follow-on descriptive questions might be:

> "Can you give me an example of BS-ing?"
> "You've probably had some interesting experiences on three-alarms, would you tell me some of them?"
> "Could you describe logging out to me?"

**Performing a Domain Analysis**

Once we have identified some of the domains of meaning present in a culture, we need to understand how those meaning domains (also referred to in ethnography as "cover terms") and their contents (referred to as "included terms") are organized. The purpose of performing a domain analysis is to discover the semantic relationships that organize native terminology and domains. Keep in mind that cultural differences manifest themselves not only in the terminology or language of a culture, but also in the meaning systems that are derived from the terms. Ethnographic research has discovered over time that the following semantic relationships seem to be present in almost every culture:

| Universal Semantic Relationships | |
|---|---|
| Strict Inclusion | X is a kind of Y |
| Spatial | X is a place in, or part of Y |
| Cause-Effect | X is a cause or result of Y |
| Rationale | X is a reason for doing Y |
| Location for Action | X is a place for doing Y |
| Function | X is used for Y |
| Means-End | X is a way to do Y |
| Sequence | X is a step in Y |
| Attribution | X is an attribute of Y |

Cover and Included Terms

Using this model a domain is identified by what anthropologists refer to as a "cover term." Cover terms are used to identify a domain and through semantic relationships like the ones outlined above, they are related to the other terms that belong within the domain. These terms are referred to as "included terms."

Returning to the firehouse example, here is a sample of the results of a domain analysis:

| Included Terms | Semantic Relationship | Cover Term |
|---|---|---|
| ½ inch<br>1 inch<br>2 inch | is a kind of → | line |
| a quiet day | is a reason for doing → | deferred maintenance |

The process of domain mapping within ethnographic interviewing provides software requirements analysts a structured methodology for organizing the information gathered through interviewing and can dovetail well with requirements specification methods. For more on that connection see the Practical Applications section.

It is important to remember that the goal of conducting an ethnographic interview is to discover the language of the customer. The reason for discovering the language of the customer is to understand the customer's true needs and design a product that is aligned with those needs. An understanding of the semantic relationships between domain symbols that is based on customer input is essential to preserving the customer's point of view.

Semantic relationships are an unobservable component of the culture that underlies the symbols themselves. Domain analysis allows us to gain access to the meaning of those structures. Of course, once a domain analysis has been completed to determine the cover terms, included terms and semantic relationships, the results must be validated with the customer before the conclusions can be considered valid.

**Asking Structural Questions**

Domains are usually made up of other sub-domains. Structural questions seek to discover these additional levels of domains and any associated artifacts. They explore a domain entry point identified by the informant and discover where the semantic path leads. In other words, discovering a native term that has particular meaning to the culture is only the first step. Cultural meanings do not occur

in isolation; rather, they are organized into complex systems. Once a term has been discovered it will lead to the discovery of other related terms. Asking structural questions is a method to systematically look for those related terms.

Structural questions should be asked simultaneously with descriptive questions. By iterating through these two questions types, a requirements analyst can discover a domain and begin to systematically explore it.

Using our fire station example, the following would be relevant structural questions to ask:

> "Are there different types of lines?"
> "Three-alarm and four-alarm are two kinds of fires; how are they related to each other?"
> "What is the typical makeup of an engine crew?"

**Asking Contrast Questions**

Contrast questions seek to discover and verify the semantic relationships between the terms identified by the informant by contrasting them with one another. Whereas structural questions seek to determine meaning by discovering how one symbol is related to another, contrast questions seek to understand meaning by determining how one symbol is different fro another.

In the fire station example the following contrast questions might be of interest:

> "How is scheduled maintenance different from deferred maintenance?"
> "What does the Engineer do differently on a medical aid call versus a fire?"
> "How is a paramedic different from an EMT-Basic?"
> "When is a four-inch line used instead of a three- or two-inch line?"

## Practical Applications

Ethnography in its purely anthropological sense is a longitudinal research process that can take a year or more to complete. The end result of ethnographic research is almost always documented in book form. They key to success for the fast paced context of software engineering is to carefully select the techniques from the ethnographic toolbox that will yield the most benefit with the least overhead. The principles of the native's point of view, thick description, and minimizing cross-cultural translation must be protected in order to maintain the benefits of traditional ethnography. Beyond this, teams can create highly customized approaches to fit a variety of conditions.

Requirements elicitation efforts should be planned and organized based on factors including the domain expertise of the development team, the amount of access to customers and end users (time available, geography and culture), and similar factors. The spectrum of ethnographic techniques in this paper and others discussed in the references can be applied to generate elicitation efforts of varying depth, breadth, and length.

There are tradeoffs associated with modified approaches to ethnography. Ethnographic interviewing works best when augmented by participant observation. Participant Observation allows the researcher/analyst to gather information from direct observation of the culture under study. The level of participation can vary from non-participation, where one only observes the culture as an outsider, to complete participation where one is accepted as a full fledged member of the culture. Participant observation allows verification of information collected in previous interview and the discovery of new or overlooked material for future interviews. When combined with ethnographic interviewing methods it serves as a triangulation device for cultural information.

Speeding the process by using only one of the ethnographic methods (interviewing) results in "thinner" descriptions, less cultural detail and limited triangulation of the information gathered. In the traditional context of anthropological study, full ethnographies are conducted over a period of a year and their results are recorded in the form of a book or lengthy monogram. Obviously this model does not lend itself well to the time-critical world of software development and the Internet economy. A method of quickly and effectively documenting the results of ethnographic interviews for incorporation into the software design process is needed.

Use cases have become a very popular medium for documenting information related to requirements. In a practical sense, use cases can replace the book form results of traditional ethnography. The resulting information about user needs is the ideal input to product requirements.

Domain analysis leads to identification of the objects and classes needed to create system models using the Unified Modeling Language (UML) or other notations.

### *Benefits for Requirements Elicitation*

Spradley refers to the process of ethnography as "testing formal theory as opposed to testing grounded theory." If we substitute "requirements" for "theory" in Spradley's observation, the benefits for software design are easy to see. Viewing software

requirements as empirical data derived inductively from the customer rather than as a hypothesis about what the customer will want based on the expertise of the designers allows the SQA process to become data validation as opposed to hypothesis testing. Ethnographic interviewing is a tool for grounding the requirements elicitation process. Establishing and testing requirements grounded in empirical customer data reduces "techno-centrism" by allowing us to base software products on grounded requirements that have been inductively derived from the customer's own culture. It also enables the effective testing of the degree to which the software meets actual customer needs.

## Challenges of Internet-Time

In "internet time", everything happens at once from a systems engineering perspective. RAD and Parallel engineering are often used to speed development. This means requirements elicitation is ongoing throughout most of the project. The iterative nature of ethnographic interviewing nicely compliments other iterative tools such as the evolutionary software project management technique Evo promoted by Tom Gilb. Iterative process tools help enable the development team to keep up with Internet Time.

The result is adaptable software from a nimble process fed by a continuous stream of real customer requirements. It has never been more important to hear the voice of the customer than in this time-critical environment where the project team only gets one chance to deliver a successful product.

The software culture is still in its infancy. This fact is even more evident in the Internet culture. Cultures that are immature have not yet had the time to define themselves. This lack of definition results in incoherence and a high degree of variability in the perceptions of the members of the culture. This variability is very dangerous thing for product definition. If the members of a culture each have different perceptions of the culture, its artifacts, and its meanings, there is an increased risk of designing and developing a product that will not be successful. One way to minimize the risks introduced by this environment is to use multiple informants to gain a more aggregate picture of the culture. Other techniques from ethnography such as participant observation can also reduce the risk of capturing an incorrect picture of the culture.

In the world of eCommerce and other Internet-related companies, understaffing is rampant and there is usually one person who is tasked with defining a given software product. For ethnographic interviewing to succeed, it is critical that the information gained in interviews with that person be validated with other members of the culture. If a single resource is allowed to define a product, the product could meet its requirements very well, but still be a failure because its requirements were incorrect or incomplete.

## Conclusions

Ethnographic interviewing is a technique that when applied to the requirements elicitation process yields valuable information that is difficult to obtain through traditional techniques. There are six steps to the ethnographic interviewing process:

1. Selecting informants
2. Asking descriptive questions
3. Performing a domain analysis
4. Asking structural questions
5. Performing a taxonomic analysis
6. Asking contrast questions

Several things are important to remember when conducting ethnographic interviewing:

1. Select multiple informants to gain a broader picture of a culture.
2. Asking them descriptive and structural questions simultaneously to discover native language terms.
3. Analyzing the responses using domain analysis to determine the semantic relationships that create meaning for the culture.
4. Ask the informants contrast questions validate the semantic relationships discovered and to find additional ones.

Applying interviewing in isolation from other methods such as participant observation and creating a formal ethnographic record allows the tool to be used in the time-critical software development environment. However, this practice would be totally inadequate if the goal was to fully document and describe a large-scale and complex culture that had not been studied before.

Ethnographic interviewing is a valuable tool for the software requirements analyst. Through careful application using the methods outlines in this paper, the voice of the customer can be more accurately captured and represented in software products. Products based on the customer's culture will be more useful to the customer and a higher level of customer satisfaction can be achieved.

## Bibliography


### *Ethnography and Ethnographic Methods*

1. Fetterman, D. Ethnography Step by Step. Newbury Park, CA: Sage, 1989.
2. Saville-Troike, M. The Ethnography of Communication. New York, NY: Basil Blackwell, 1982.
3. Spradley, J. The Ethnographic Interview. Fort Worth TX: Harcourt Brace Jovanovich, 1979.
4. Spradley, J. Participant Observation. Fort Worth, TX: Harcourt Brace Jovanovich, 1980.

### *Ethnography & Design*

1. Beyer, H. & Holtzblatt, K. "Apprenticing with the Customer." Communications of the ACM, p. 45-52, May 1995.
2. Holtzblatt, K. & Beyer, H. "Requirements Gathering: The Human Factor." Communications of the ACM. p. 31-32, May 1995.
3. Holtzblatt, K. & Beyer, H. "Making Customer Centered Design Work for Teams." Communications of the ACM, p.93-103, October 1993.
4. Hughes, J. "Ethnography, Plans and Software Engineering." The Institution of Electrical Engineers, 1995.
5. Hughes, J. et al. "The Role of Ethnography in Interactive Systems Design." Interactions, p.57-65, April 1995.
6. Hughes, J. et al. "Presenting Ethnography in the Requirements Process." IEEE, p. 27-34, July 1995.
7. Hughes J. et al. "Designing with Ethnography: A Presentation Framework for Design." ACM 1997. 0-89791-863-0/97/0008.
8. Potts, C. & Newstetter, W. "Naturalistic Inquiry and Requirements Engineering: Reconciling Their Theoretical Foundations." IEEE, 1997. 1090-705X/97.
9. Rothstein, P. "The 'Re-emergence' of Ethnography in Industrial Design Today." 1999 IDSA Design and Education Conference Proceedings. www.idsa.org/whatsnew/99ed_proceed/paper019.htm.
10. Simonsen, J. & Kensing, F. "Using Ethnography in Contextual Design." Communications of the ACM, p.82-88, July 1997.
11. Sommerville, I. et al. "Integrating Ethnography into the Requirements Engineering Process" IEEE, p.165-173, 1992.
12. Viller, S. & Sommerville, I. "Social Analysis in the Requirements Engineering Process: From Ethnography to Method." IEEE, p.6-13, May 1999.

### *Requirements Engineering*

1. Gause, D. and Weinberg, G. "Exploring Requirements – Quality Before Design." New York, NY: Doset House, 1989
2. Weigers K. "Software Requirements." , Redmond: MS Press, 1999.
3. Lauesen, S. "Software Requirements – Styles and Techniques." Copenhagen Denmark: Samfundslitterature, 1999
4. Kulak, D. and Guiney, E. "Use Cases: Requirements in Context." New York, NY: ACM Press and Addison Wesley, 2000
5. Kensing, Simonsen & Bødker, 1996
6. Kotanya, G. and Sommerville, I. "Requirements Engineering – Processes and Techniques." West Sussex, England: Wiley, 1999
7. Leffingwell, D. and Widrig, D. "Managing Requirements – A Unified Approach." Reading, MA: Addison Wesley, 2000
8. Gilb, T. "Evo: The Evolutionary Project Manager's Handbook." Self-published at http://www.result-planning.com.


## Acknowledgements


The authors would like to acknowledge the support of Intel Corporation, the Platform Quality Methods and Software Quality Engineering Groups of the Intel Corporate Quality Network, and the Intel Architecture Labs in the preparation of this manuscript.

The authors also express their appreciation to the formal and informal reviewers who assisted in the preparation of this paper.

# Requirements-Based UML

Joseph D. Schulz
Technology Builders, Inc.
Technical Director, International Channels

## *The RBU Approach*

The purpose of this paper is to describe the "Requirements-Based UML" (RBU) development technique. RBU is a straightforward approach for integrating structured requirements analysis into a UML-based analysis and design effort. It involves a very high degree of customer participation and involves the creation of measurable requirement definitions before each stage of modeling and/or coding. In short, RBU is a methodology.

However, RBU is a *pragmatic* methodology. RBU includes only the essential tasks and is designed to be highly communicative and easily understood by both customers and professional development staff. Most often developed in direct cooperation with customers via a "Joint Application Design" (JAD) approach, the requirements are used to both design and validate the application functionality.

While RBU is a methodology, this paper only includes a brief description of the process. Accordingly, it is not meant to be used as a complete implementation guide for a professional development organization. RBU's major tasks and techniques are described here, but there has been no attempt to include all of the necessary components of a robust methodology (e.g., standards, procedures, forms, etc.). In addition, the examples contained within are merely illustrative of the overall approach.

## *Major Learning Points*

After reviewing this paper, the reader should gain additional insight into the following areas:

- The basic structure of the Unified Modeling Language (UML) development process
- The purpose and benefits of a structured Requirements Management (RM) process
- The RBU approach to incorporate structured RM into UML
- The appropriate "levels" of requirement information within the RBU context
- The importance of traceability across the development framework

# Presentation Summary

### *The Dreaded "M" Word*

Every project needs one and every developer follows one, whether formal or informal, prescribed or ad-hoc. Unfortunately, for most IS professionals the word "*Methodology*" invokes dreadful images of the worst kind. It often implies reams of unnecessary work, impossibly rigid standards, and lots of wasted time. When the average developer hears the dreaded word, they usually assume that the related project is doomed.

Of course, just the opposite is true. A development project that doesn't actively use some sort of methodology has relatively little chance of success. If such a project does succeed, it is merely through coincidence or sheer dumb luck. These are the kinds of projects that ramble about generating lots of paperwork but relatively few measurable results. They miss every major deadline because they change directions so frequently, and usually require large quantities of rework to "fix" previous mistakes. In a project without a methodology, there is usually no such thing as a *frozen* deliverable, so consequently there are *no* deliverables.

So, then what exactly is a methodology? In its simplest form, a methodology is a set of steps to accomplish a task. That's it. No fancy buzzwords or expensive terminology, just a set of steps. It is a plan that describes each task and its sequence relative to the others. After all, any job worth doing is worth planning for. As the old military adage goes, "If you fail to plan, you are planning to fail."

Of course, a robust methodology can also include many other components. Strictly speaking, a complete methodology includes not only task descriptions but also supporting components like task standards, technique guidelines, deliverable outlines, and quality metrics. These items, though, are merely present to supplement the basic purpose of the methodology, which is to identify and prioritize the work to be done. That is, to describe the set of steps needed to accomplish the goal.

### *Unified Modeling Language*

The latest emerging industry-standard in the object-oriented methodology arena is the *Unified Modeling Language*, commonly referred to as "UML". UML is a collaborative effort between the "Three Amigos" of the object-oriented analysis and design (OOAD) industry, i.e., Grady Booch, Ivar Jacobson, and Jim Rumbaugh. Each of these three had previously authored their own competing methodologies and realized the significant benefits of a truly global standard for (OOAD). By combining much of their previous work, the UML standard was born.

Of course, UML is not actually a methodology. Rather, it is a notational standard that can be used to implement the tasks within a methodology. By having a common notation, methodology and tool vendors can easily develop complementary solutions without requiring retraining of the

workforce. UML-based methodologies define differing sets of tasks, but the techniques all employ the standard graphical symbologies.

One such example is the "Rational Unified Process" (RUP) from Rational Software. RUP is a complete methodology developed by the "Three Amigos" which uses the UML notation to represent all of its deliverables. It includes suggested task plans and also defines guidelines and metrics that can be used to manage and measure the development process.

### *Use Case Models*

The UML specification includes graphical notations for many different diagram types, with most being optional steps based on the complexity of the application being developed. Relatively speaking, the "first" deliverable described in the UML notation is the Use Case Diagram. A Use Case Diagram graphically depicts the interaction between system users (i.e., "actors") and system functions (i.e., "use cases"). Subsequent UML diagrams build on this basic information to identify the system components, methods, and packages necessary.

Unfortunately, this focus on beginning with use cases creates a glaring deficiency in most UML-based methodologies, that is, they don't address the business-oriented application requirements. Instead of first defining the purpose and objectives for the development effort, UML methods begin by jumping directly to the software functions. This presupposes that the use case participants already know why they need a system and what the optimal solution should look like.

In reality, the most important part of any systems development effort is to first establish a firm understanding of the problem so that potential solutions can be effectively weighed. To do this, the key business requirements must be defined, including the return-on-investment justification for each. Once this *Objective B*aseline is established, proposed alternatives can then be measured to determine which best solves the stated problem. Without this requirements analysis, a UML-based approach may only help to deliver the *wrong* application faster and cheaper.

### *Requirements-Based UML*

One possible solution to this problem is the use of "*Requirements-Based UML*" (RBU). RBU is a structured approach for incorporating business-oriented requirements analysis into a UML-centric development method. It balances the need for non-technical business analysis against the need for the structured technical approach defined in UML. Furthermore, it identifies business requirements analysis as a precursor to software-centric use case modeling efforts.

RBU also relies on a more natural, textual format for requirements deliverables. Non-technical staff members are generally more comfortable with words than diagrams, so RBU business requirements are defined in sentences and paragraphs. These textual descriptions are then related to the graphical objects defined in the UML deliverables.

After the first level of UML diagrams is completed (use case models, collaboration diagrams, etc.), the requirements are refined into more detailed textual technical specifications. In turn,

these specifications are then related to the next round of UML diagram objects. This process of textual requirements leading UML modeling can continue to whatever level of detail is appropriate for the specific project.

By using this alternating approach with requirements and diagram objects, a more complete analysis and design model is produced. This provides a clearer picture of the application environment, including not only answering the "How?" questions for the application but also clarifying the "Why?" and "What?" as well. All too often development teams are eager to rush into coding and the latter two questions remained unvisited. It is these types of projects that are most often cancelled or rejected by the customers because they provide little business value.

### *About this paper*

That brings us to the stated purpose of this paper, which is to describe the "Requirements-Based UML" (RBU) development technique. As mentioned earlier, RBU is a straightforward approach for integrating structured requirements analysis into a UML-based analysis and design effort.

This paper will first describe the overall structure of the UML method, highlighting each of the major tasks and deliverables. Then, it will describe the need for a structured Requirements Management process. Finally, it will detail the RBU approach to integrating structured RM into a UML-centric development organization. This will include not only a description of the process, but also illustrative examples of each major deliverable and pragmatic guidance for implementing RM within this context.

As with most methodologies, the most common mistake made when using RBU is to "over-engineer" and to spend more time on deliverables than is cost-effective. Accordingly, this paper also includes a brief discussion about the appropriate leveling of requirement specifications and the value of maintaining traceability across development deliverables.

### *About the author*

Joseph Schulz is a seasoned developer with more than 18 years of professional information systems experience spanning a wide range of vertical industries. He has been actively involved in object-oriented development techniques since 1989. He has an extensive background in "non traditional" development that includes more than forty procedural, 4GL, CASE, and OOAD/OOP products on a wide variety of operating platforms.

Currently, Mr. Schulz is the Technical Director for International Channels at Technology Builders Inc. (TBI). In this position, Mr. Schulz works with organizations around the globe to help them improve their development processes. TBI is a leading software vendor in both Requirements Management and Quality Assurance and markets the "Caliber" family of products.

# UML Overview

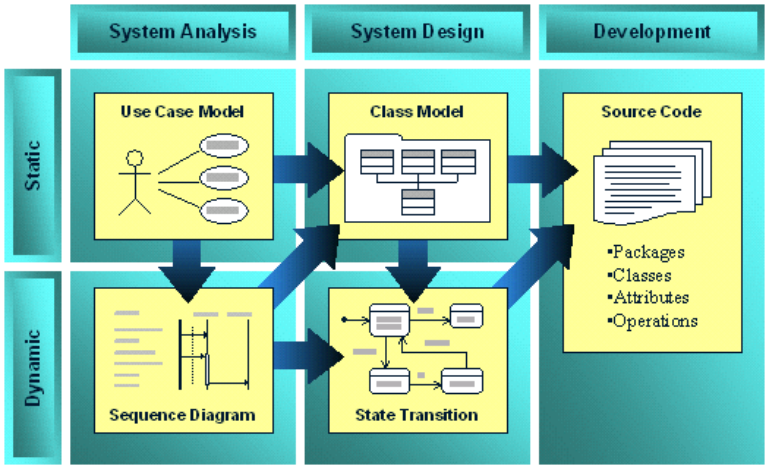
### *What is UML?*

The Unified Modeling Process (UML) is a common notation for structured modeling within an Object-Oriented Analysis and Design (OOAD) framework. It was originally developed by several of the leading OOAD methodologists as a means to help standardize the types and format of deliverables produced by the competing OOAD methods. While not strictly a methodology itself, UML describes the notation that methodology outputs employ.

The current UML notational standard addresses the system analysis, design, and deployment steps in a development lifecycle. This version of the UML, v1.3, was approved in June 1999 by the Object Management Group (OMG). A new draft standard, v2.0, is currently in RFI review and will extend the current standard to include a range of other activities. The most notable addition expected in v2.0 is a common notation for business process redesign.

### *A Typical UML Process*

A UML-based development methodology usually involves a series of graphical models which are used to define the functional and technical aspects of an application system. Each model depicts a diagrammatic representation of one aspect of the application and is integrated with the other model objects. These models are then used as the component specifications for the construction phase of the project.



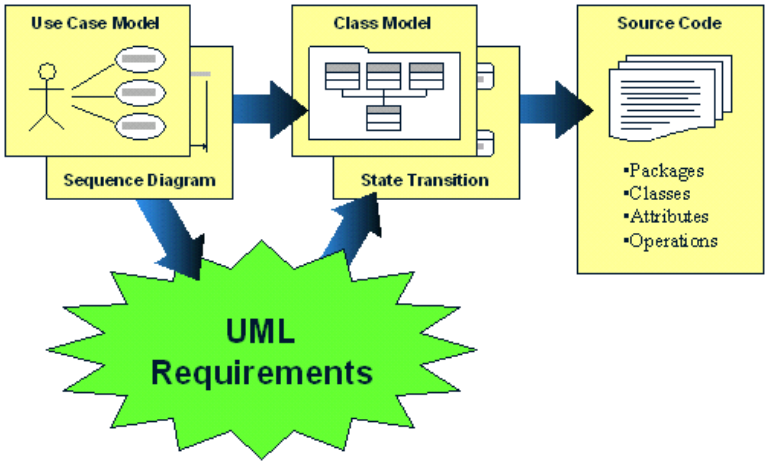

As shown in the graphic, the first and primary model developed in most UML-based methods is the Use Case diagram. Use Case diagrams are used to identify the external system boundary for an application by depicting the system functions ("use cases") that external entities ("actors") are able to interact with. Use Case diagrams are generally developed in very close collaboration with the application's ultimate customers or sponsors.

These Use Case diagrams are often then more fully described by the creation of either Object Sequence diagrams and/or Object Collaboration diagrams. Both of these diagram types serve to more fully describe the Use Case by including the nature and order of each of the major work steps within the Use Case. Together, some combination of these three diagrams provide the functional application requirements for a system.

At the beginning of the system design, then, the system analysis models are used as input to create the relevant Class diagrams and/or State Transition diagrams. These design models describe the technical structure of the application. Any of several deployment models (e.g., Package Deployment diagrams, etc.) may also be defined in order to complete the technical specification before code development begins.

### *UML Requirements*

In the context of a UML-based method as outlined above, the term "requirements" generally refers to a set of technical specifications that describe the software features in an application. These requirements are imperative statements of functionality that must exist in the developed code and are written as "Plain Language" textual sentences or paragraphs. This deliverable is often named the "System Requirement Specification" (SRS).




Most often, these "UML requirements" included in the SRS are developed as extensions of a Use Case diagram. For each Use Case defined, the complete set of mandatory characteristics is identified and documented in clear, concise language. Modelers will then use these requirement definitions to help complete and validate the systems design models, ensuring coverage of all required functionality.

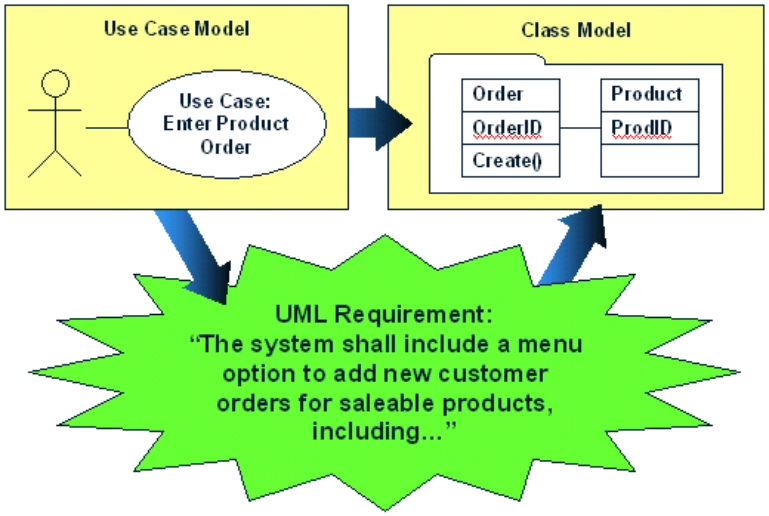
The SRS generally includes both functional and non-functional requirements. Functional requirements state a capability that invokes or performs an actor-oriented transaction. Non-functional requirements, on the other hand, state a characteristic of the application which limits or bound a designer's ability to develop a solution. Non-functional requirements usually include information about traits like performance and capacity limits, security rules and responsibilities, and/or technological considerations.

### *UML Requirements Example*

In the example pictured below, a Use Case has been defined named "Enter Product Order". This Use Case would exist on one or more Use Case diagrams and would be detailed with the inclusion of a Use Case narrative (e.g., pre-conditions, post-conditions, etc.). In the diagram, the appropriate actor(s) would also be associated to the Use Case.

As part of the transition from system analysis to system design, UML requirements would then be defined for this Use Case. These requirements would itemize the specific features necessary

in the software in order to fully accomplish the "Enter Product Order" Use Case. As mentioned above, this might include both functional requirements and non-functional requirements.




One such UML requirement for the "Enter Product Order" Use Case might be the statement that "The system shall include a menu option to add new customer orders for saleable products…". As a result of this requirement, when the user interface class is developed in the Class diagram the system designer will know to include a method to invoke this process. This may also be reflected in the appropriate State Transition diagram(s) as an event which triggers a change in state.

### *Benefits of UML Requirements*

The advantage of developing a structured SRS which includes both models and textual requirements is twofold. Firstly, the textual statements are often more communicative than UML notation to non-technical customers as they provide a written description of the functionality that anyone can read. The graphical notations can often be daunting for an uneducated customer, and so the textual descriptions are more comfortable for those without any previous UML training.

Secondly, the textual requirements provide a place to document software features that may not be readily apparent or do not exist in the graphical models. For example, non-functional characteristics like hardware constraints are difficult to include in UML models because they are typically global issues that cannot be incorporated into the description of just one model object.

So, UML requirements become the document-centric "bridge" between the graphical system analysis deliverables (Use Case diagrams, etc.) and the graphical system design deliverables (Class diagrams, etc.). Most often these requirements are managed with a word processing application and reviewed and approved in document format. This comfortable paradigm mimics traditional document-oriented analysis techniques.

### *Drawbacks to UML Requirements*

However, there are also disadvantages to limiting the requirements process to technical feature descriptions. First and foremost, by beginning the analysis process with Use Case definitions, the focus is immediately on the design of the software. Since Use Cases describe systemic solutions to problems, the derived UML requirements will address only the systemic characteristics as well.

With this approach, the only solution that can be developed will be one that can be automated with an application. This virtually ignores the relevant business issues that may be all or part of the problem as well. Often a minor business process redesign (like job function reorganization) can facilitate a more efficient application or even eliminate the need for an application at all.

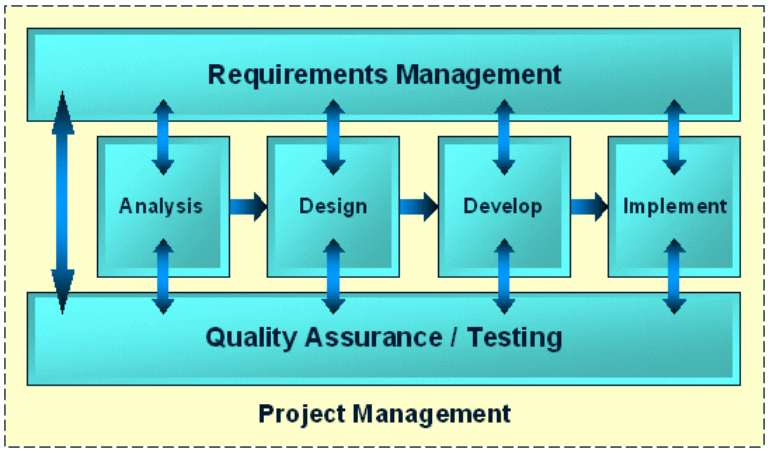
Also, UML requirements tend to include a lot of technical language since they are describing technical features. This is typically because they are written for the development organization to use as an input to the system design process. However, another goal of a structured requirements analysis is to validate the system analysis deliverables and the customers needed to do this are often non-technical. So, the personnel with the appropriate business knowledge may not be able to adequately understand the requirements definitions.

Finally, another problem with UML requirements is that they tend to focus on one business transaction at a time. Since they are most often derived from Use Cases, the requirements are documented with an eye toward that one transaction and often ignore the business workflow surrounding it. Without a highly structured reuse analysis, it is often possible to end up with highly efficient transactions that contain a lot of business redundancy between them.

# Requirements-Based UML (RBU) Overview

### *What is RBU?*

Requirement-Based UML (RBU) is a structured approach for integrating formal requirements analysis into a UML-based analysis and design effort. It balances the need for non-technical business analysis against the need for the system-oriented approach defined in UML by including a multi-level requirements definition. Instead of just the technical feature descriptions captured in traditional UML requirements (see previous chapter), RBU defines multiple requirements deliverables with a specific focus for each. Simply put, requirements management becomes a lifecycle task that runs in parallel with the OOAD tasks.




As with UML requirements, RBU requirements deliverables are defined in a natural, textual format. This allows non-technical customers to more comfortably review and understand the requirements information. These textual descriptions are then related to the relevant graphical objects defined in the UML-based deliverables.

Often, development teams prematurely rush into the development of the application solution and ignore the larger business issues. This can easily lead to inappropriate or expensive technological solutions. By using alternating "rounds" of modeling and textual specifications, the RBU approach helps to temper that tendency and delivers a richer, more complete picture of the business problem. In this manner, it helps to ensure a higher quality, more cost-effective solution.

In addition, the RBU approach addresses Quality Assurance as a lifecycle task as well. Requirements are thoroughly tested before any development is performed, detecting conflicts and omissions that would stall later development. At each stage, the QA/Testing plan is refined and more detail is added until specific test cases have been identified. By developing the test cases from the requirements rather than the code, a more complete test harness is be established.

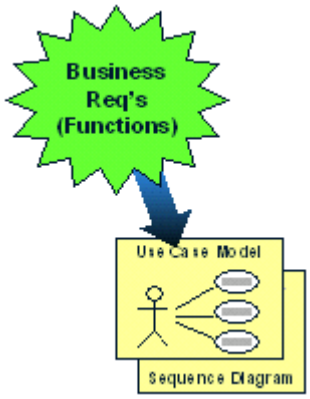
### *Typical RBU Process*

The RBU technique begins with a textual specification of all of the requirements for any solution to the business problem. These requirement statements define both the functionality required in the solution as well as the boundaries the solution must operate within. All of these requirement statements should be specified in a non-technical, "plain language" format. Ideally, the

customers will define these textual requirements themselves without restatement by the development staff.

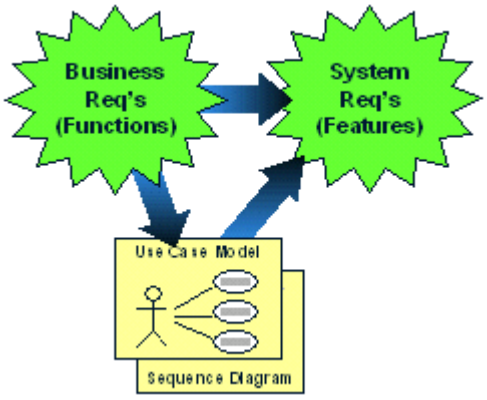
These requirements, usually referred to as "Business Requirements", should be defined without regard to how the application will look. Specifically, they should not reference menu options or screen formats. The intent is to capture a definition of the business process needed to completely solve the business problem. In essence, they answer the question of "What?" not "How?".

Once the Business Requirements are defined, they can then be used as the basis for developing the Use Case diagrams. Specifically, each functional requirement identified in the Business Requirements will initially correspond to one Use Case if it can be automated. If it can't be automated, then a manual transition plan will need to be developed.




While it may appear redundant to develop a 1:1 correspondence between Business Requirements and Use Cases, it isn't because further refinement will be performed on the Use Case diagram during system analysis. The mapping is only 1:1 at the beginning of this stage. Once the Use Case diagram is refined with "Extends" and "Uses" relationships, the mapping becomes a many-to-many relationship.

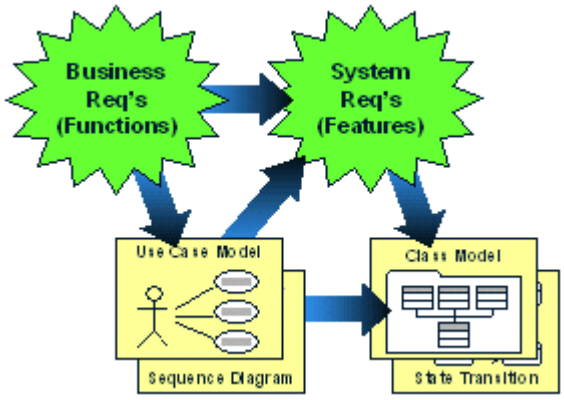
It is important, though, that this reuse analysis be performed on the Use Cases and not on the Business Requirements. This is to avoid corrupting the real business requirements with artificial technological constraints. All too often users are forced to redesign their business process in order to accommodate technology rather than the reverse. As the saying goes, "Just because you know how to use a hammer, not every problem is a nail". That is, define the business requirements for a solution before a specific technology is applied.




As part of the transition to system design, both the Business Requirements and the system analysis models are used to develop System Requirements. As with the Business Requirements, the System Requirements are stated in a textual format. However, unlike the Business Requirements, the System Requirements define the technical features of the application rather than the business needs. They are used to define the "How?" for the application.

At this point, relationships should be established between the System Requirements and the previous deliverables. For example, each System Requirement should be an implementation of at least one Business Requirement and at least one Use Case. These relationships are then analyzed to look for inconsistencies in the model. For example, if any Business Requirement does not have at least one "child" System Requirement, there is a gap

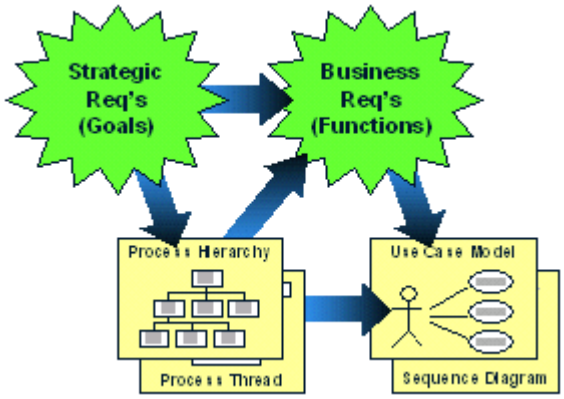
somewhere in the Use Case model. On the other hand, if there are System Requirements without at least one "parent" Business Requirement, then the scope of the original project has been increased, either intentionally or unintentionally.




Once the System Requirements have been fully defined and quality checked, they are used in conjunction with the Use Case model to develop the Class model. As with the System Requirements, relationships should be established to the "parent" objects in the previous deliverables and used to look for inconsistencies and omissions in the Class model.

Finally, the system design and implementation models are then used to develop the application code itself. By using this "matrix" approach to building the UML deliverables, the resulting application is more complete and of higher quality.

If a formal business process model is desired, the RBU process can be extended to support this work as well. Although the UML specification does not include notation for business process models, there are many popular methodologies which do. For example, the CSC Lynx method is used by many modeling tools to implement this work.

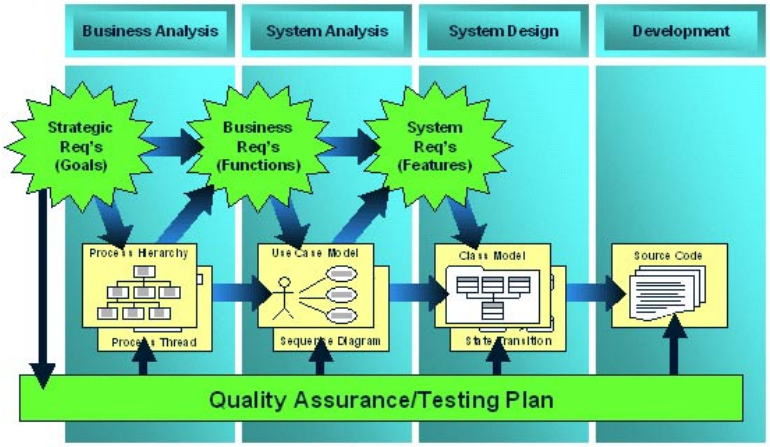



In these cases, additional diagrams are developed before the UML deliverables listed above are created. These might include models like a Process Hierarchy diagram and/or a Process Thread diagram. Both of these diagrams show the flow of work through an organization without regard for job titles and application boundaries. Their purpose is to optimize the organizational process before an application system is designed.

Using the RBU approach, the Process Hierarchy and Process Thread diagrams would be preceded by a set of requirement definitions. These requirements, called "Strategic Requirements" define the goals of the organization, including the objective metrics used to measure success. These Strategic Requirements become the guiding principles used to govern which possible business process is the most desirable.

As with previous requirements, Strategic Requirements are related to other requirements and the modeling objects. Specifically, Strategic Requirements should be related to the Business Requirements necessary to accomplish the goals and to the processes in the business models that implement them. Again, these relationships can be inspected for inconsistencies before moving forward in the development lifecycle.

Finally, as mentioned earlier, the RBU method includes a third parallel activity for Quality Assurance and Testing. This set of work is performed by the QA organization and is used to detect flaws in the requirements and models deliverables and to develop the test harness used for verification of the application code. By deriving the test cases from the requirements in a progressive manner, the resulting test plan will be more complete and should validate both functional and operational performance.

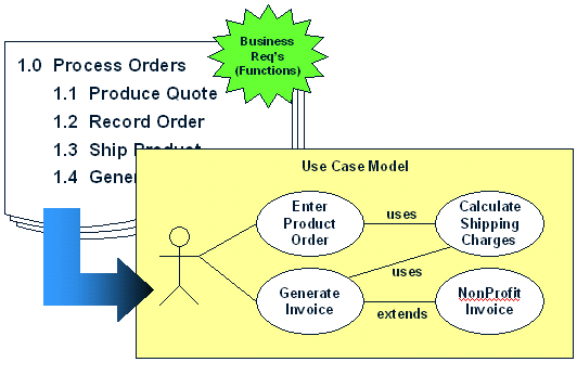



At first glance, the RBU approach may seem to introduce additional work on the project team because there are more steps than in a traditional UML-based method. However, in the opinion of this author, the work that RBU dictates is not additional work, but rather it is a matter of formalizing work that is already being done with informal methods. By purposefully addressing these steps, the quality of the work will increase and productivity may actually improve. At the very least, the quality of the software product itself will be measurably higher.

### *RBU Example*

To demonstrate the RBU technique, consider the example of the "Enter Product Order" Use Case shown in the previous chapter. How did the user and/or development staff conclude that this Use Case was necessary? How did they know which system functions would be appropriate to solve the business problem?

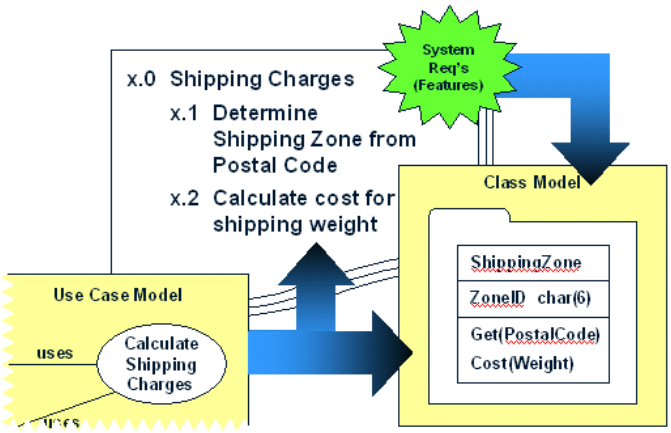
Most often, the answer to these questions is that the Information Systems staff asked the customer what the solution should be. This assumes, however, that the customer has the information and expertise necessary to make this decision. Very often that is not the case and dangerous assumptions are introduced into the development effort before a single line of code is written.




In an RBU-based project, the Use Case model would be preceded by a set of Business Requirements which document the business solution to the problem. In our example, the Business Requirements would include information about not only the two automated tasks ("Record Order" and "Generate Invoice") but also the two manual tasks that surround them

("Produce Quote" and "Ship Product"). Together, these four tasks identified in the Business Requirements describe the complete business solution necessary to solve the stated problem. Now the development team has sufficient information to make an informed decision about which tasks can be automated and how best to implement them.

Initially, only two Use Cases are defined. These would be named "Enter Product Order" and "Generate Invoice" and would be related to the two Business Requirements that are being automated. During the course of the system analysis, however, additional Use Cases might be identified in order to encapsulate reusable logic (e.g., "Calculate Shipping Charges") or to extend the model for alternate courses (e.g., "Non-Profit Invoices"). These additional Use Cases would not be directly related to Business Requirements but would instead derive their relationships through other Use Cases.




Once the Use Case model was completed, the System Requirements would then identify the functional and non-functional software features needed to automate the Use Case definitions. These System Requirements would then, in turn, be used to help define the objects in the system design models (e.g., Class diagram, State Transition diagram, etc.).

### *Benefits of the RBU Approach*

There are many benefits to using the RBU approach instead of traditional UML-based methods that treat requirement as "features". The most important is that a structured requirements-based approach to development will dramatically improve the level of communication between end-users and the development staff. By providing a non-threatening textual format for deliverables, customers without training in UML notation are able to participate in the application specification. The requirements documents produced will be easier to read and more likely to be reviewed by the appropriate customers. All of this means more feedback which will lead to higher quality deliverables.

Another major benefit is that RBU provides a facility to document the entire business solution, not just the automated subset of it. Where UML modeling techniques make the assumption that a systemic solution is available, RBU requirements do not. This, in turn, provides a much richer picture of the solution and allows the project team to make more informed decisions about what automated functionality can and should be included in the application.

In fact, business improvements are often suggested as a result of the RBU requirements analysis that have nothing to do with application development. These solutions would typically not even be discussed during a UML-based project. Just as often, applications cannot efficiently solve the

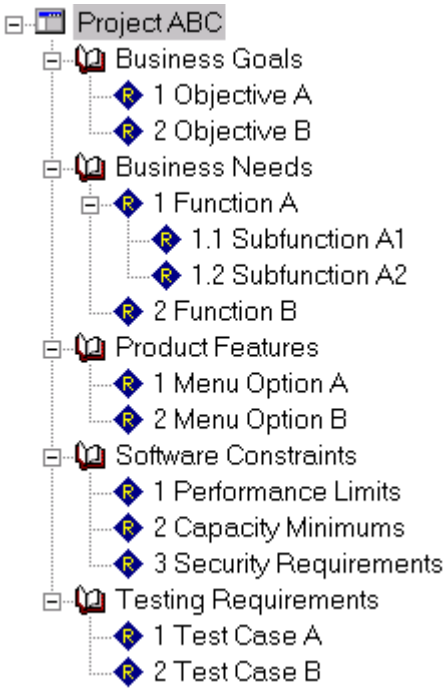
root cause of the business problem being solved because it is not an automated problem. It is important to understand this before an expensive development project is launched and then later cancelled due to lack of substantive results.

Finally, a third major benefit of RBU is that the scope of the application can be managed based on customer needs rather than on software features. By using the requirements as an integral part of the change control process, change requests can be evaluated on the basis of the business improvement in the Strategic Requirements and/or Business Requirements. Then, using the relationships established between the various RBU deliverables, the complete impact of a change can be determined before the change is approved.

# Advanced Topics

### *Requirement Hierarchies*

The requirements deliverables described in the preceding chapters are all intended to be document-oriented artifacts with textual statements of requirements. While this format is easier for end-users to review and approve, it can make it difficult to find specific sections when changing or searching the requirement information. To help resolve this problem, most requirement deliverables are organized in requirement hierarchies.




A requirement hierarchy is simply a "tree-like" structure of requirements with similar requirements grouped into common "branches". The major branches of the tree correspond to the requirements deliverables listed above (Strategic Requirements, Business Requirements, etc.). The subordinate branches are defined based on the business area or organizational structure most appropriate to the project under development.

For example, in the Strategic Requirements analysis, the goals are most often subdivided by the organizational unit(s) being analyzed. Within each unit, then, the specific objectives might be identified and listed in priority order. However, during Business Requirements analysis, it may be more convenient to organize the sub-branches by business area or logical transaction.

No matter how the requirements hierarchy is organized, the most important aspect of the hierarchy is to understand and manage each requirement as an individual object within the set. Rather than treating a requirements document as a single block of text, each requirement should be treated as a separate entity and uniquely identified. In this manner, attribute identification, historical tracking, and object traceability can all be managed at the requirement level.

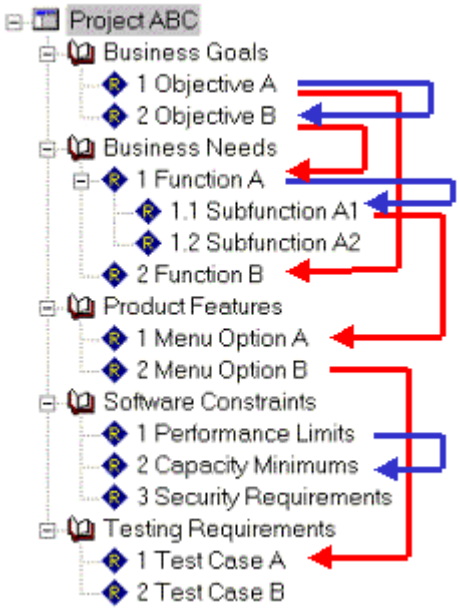
### *Requirement Traceability*

Once the requirement hierarchy has been established, the rules for relationships between the requirements and the UML objects can be defined. These relationships, usually referred to as "traces", identify the dependencies between the various development objects. Typically, these traces are used to understand the impact of a request during the change control process as well as ensuring that changes are completely propagated throughout the development model.

For traces between requirements, relationships can either be established between two requirements on the same branch of the hierarchy or across branches. Most often, traces within one branch of the hierarchy are established to show a logical precedence between the two requirements.

For example, in the picture on the right there are two primary business goals that have been identified for *Project ABC*. These are named *Objective A* and *Objective B*. However, in addition to the textual definition of each, there is business rule that must be defined in the requirements document. Specifically, *Objective A* must be met before *Objective B* can be attempted. This precedence relationship is modeled as with traceability link between the two requirements, shown as a blue arrow in the graphic. It indicates that *Objective A* is the "logical parent" of *Objective B* and that any changes to *Objective A* must be reviewed to determine their impact on *Objective B* as well.

Traces between major branches in the requirements hierarchy, however, generally indicate a developmental dependency. That is, requirements in earlier stages of development should have traces to requirements in later stages of development in order to show the progression of the development effort.

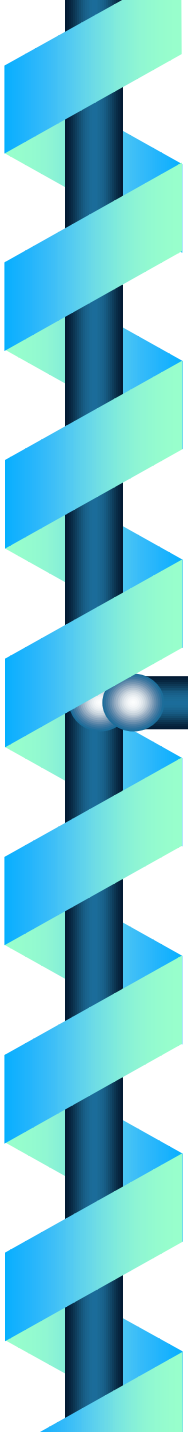



In our example, *Objective A* is one of the business goals that must be met by the project. In order to show the development dependency to the business functions, a traceability link is established between *Objective A* and *Function B*. This indicates that *Function B* is necessary in order to accomplish *Objective A*. This relationship is shown on the graphic with a red arrow. Any changes to *Function B* must be reviewed to determine their effect on the project's ability to accomplish *Objective A*.

Generally, only "direct" traces are modeled between requirements. "Indirect" traces can then be implied by following the chain of the parent-child relationships. For example, in the *Project ABC* tree, *Objective B* has an indirect relationship to *Menu Option A* by following the chain through the intermediate nodes of *Function A* and *Subfunction A1*.

Traces are also established between the requirements and UML objects in much the same way. For example, if *Function A* was implemented by *Use Case 1*, a dependency trace would be defined between the two to show this relationship. Subsequent impact analysis could then be performed by following the trace from the requirement to the UML model or vice versa.

Without these traces, proposing changes during the development lifecycle becomes a subjective effort depending entirely on the memory of the requirements analyst(s). While this may occasionally be effective, most often it leads to understated estimates and inconsistencies in the software design. Requirements traceability makes change control an objective, rational process.



# Requirements-Based UML

**Joseph D. Schulz**

*Technology Builders, Inc.*

*Technical Director, International Channels*

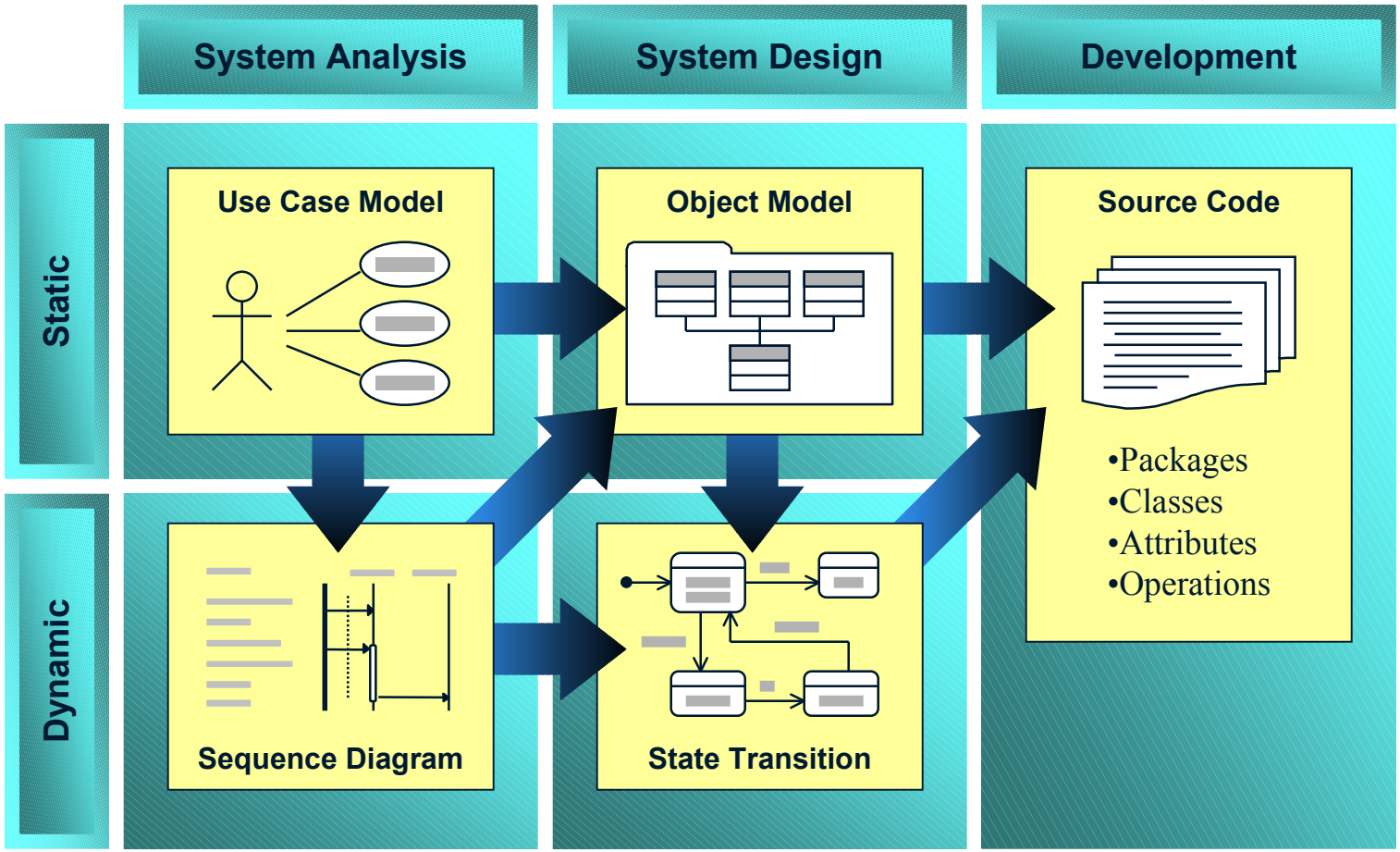
# Agenda

- Introduction
- What is UML?
- How do Requirements fit into UML?
- What is Requirements-Based UML?
- Why use requirement hierarchies?
- Why manage requirements traceability?
- Questions?

# UML in a Nutshell

- Unified Modeling Language
  - Common notation for structured modeling within an Object-Oriented Analysis/Design (OOAD) framework
- UML methods focus on deriving component specifications from Use Case definitions
  - Consists of a series of integrated models
  - Component specifications are then used as the basis for coding efforts

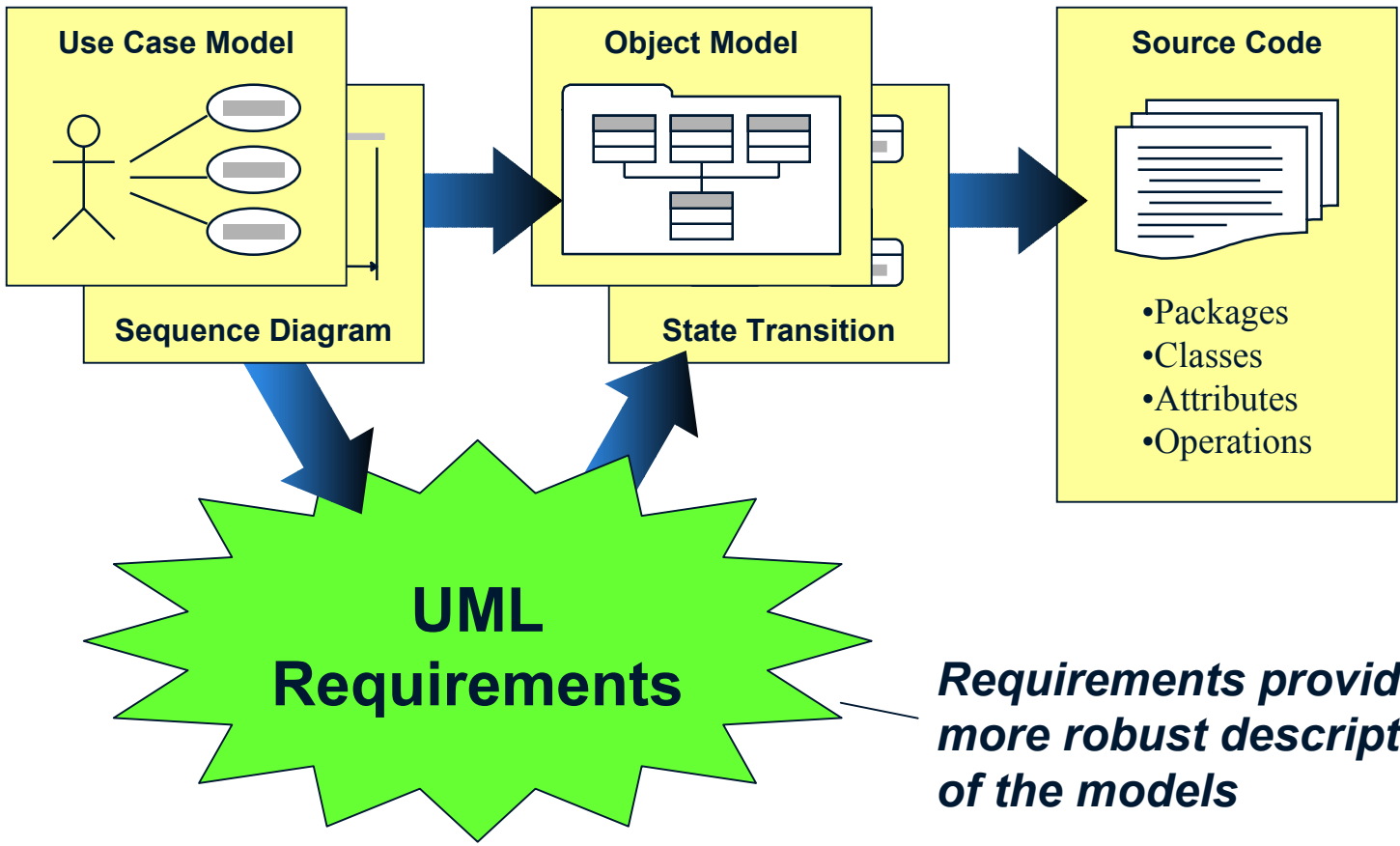
# A Typical UML Process



# Typical UML Requirements

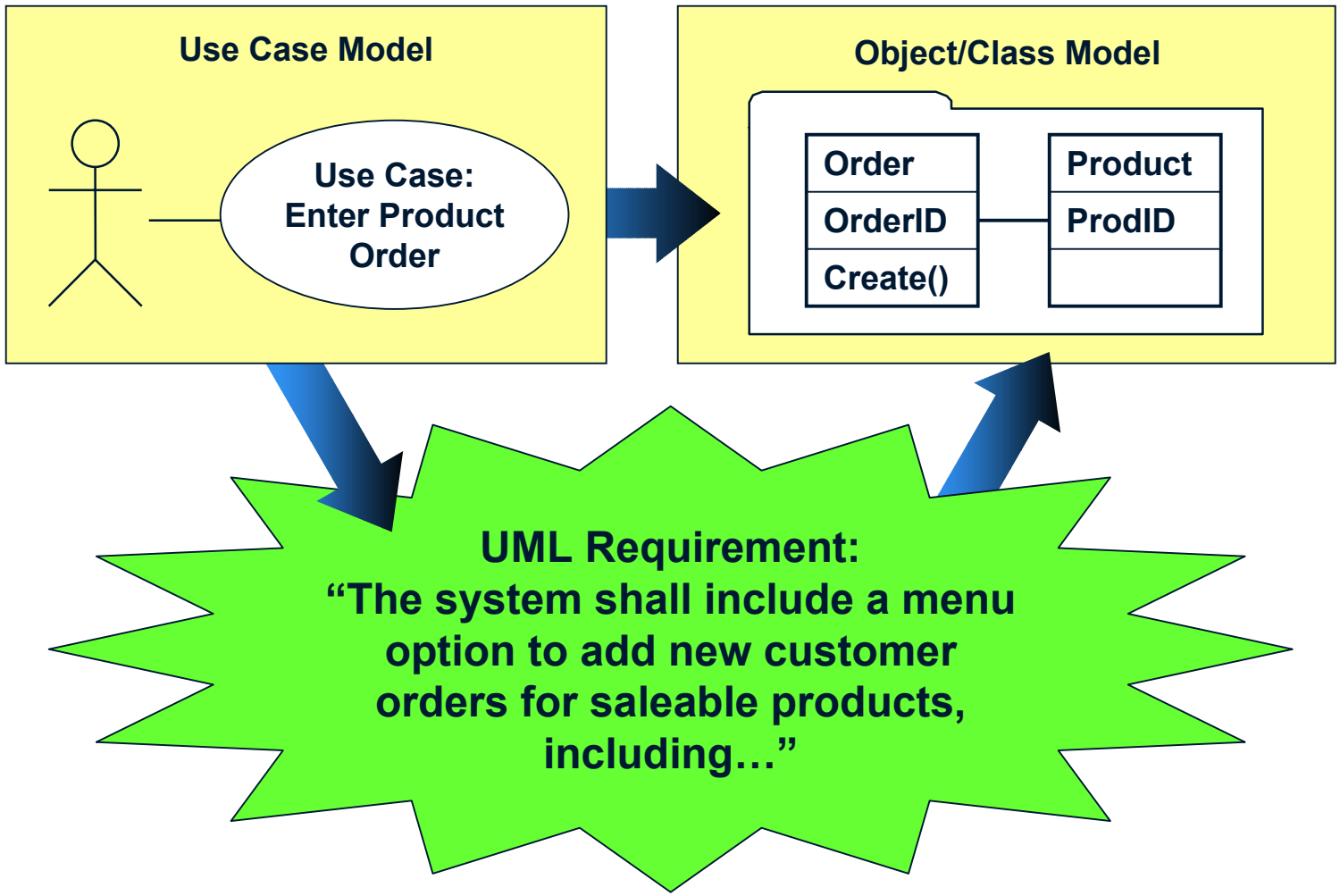
- UML "Requirements" are ...
  - Textual specifications of required software functionality
  - Derived from the Use Case models
  - Describe the primary technical features of a software component or package
  - Written by analysts for developers
  - Validated by customers/end-users

# Requirements as Features




***Requirements provide a more robust description of the models***

# For Example...


Use Case Model
Use Case:
Enter Product
Order
Object/Class Model
Order
OrderID
Create()
Product
ProdID
UML Requirement:
"The system shall include a menu option to add new customer orders for saleable products, including…"

# Benefits of UML Req's

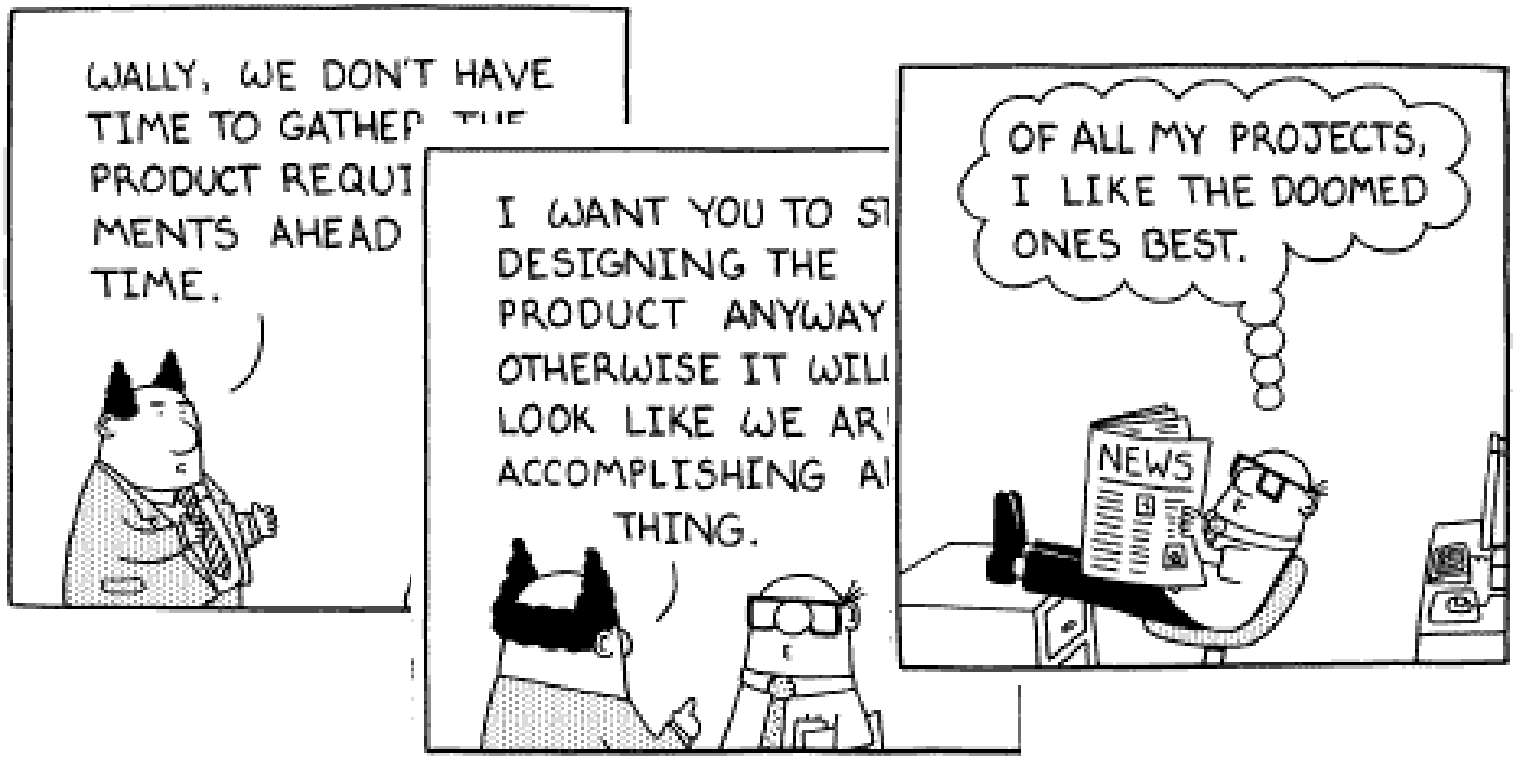
- UML Requirements ...
  - Provide a "plain english" textual description of software functionality
    - Describe the Use Case models for non-technical stakeholders
    - Validate the functional characteristics of the application
  - May include "non-functional" characteristics
    - Performance, Capacity, Security, etc.

# Problems with UML Req's

- UML Requirements ...
  - Assume too much
    - No focus on the "business" of the problem
    - Since they are derived from Use Cases, they assume an application is needed
  - Include too much technical detail
    - Usually written for developers but are validated by non-technical customers
  - Exclude the surrounding workflow

# The Requirements Challenge

## … According to Dilbert



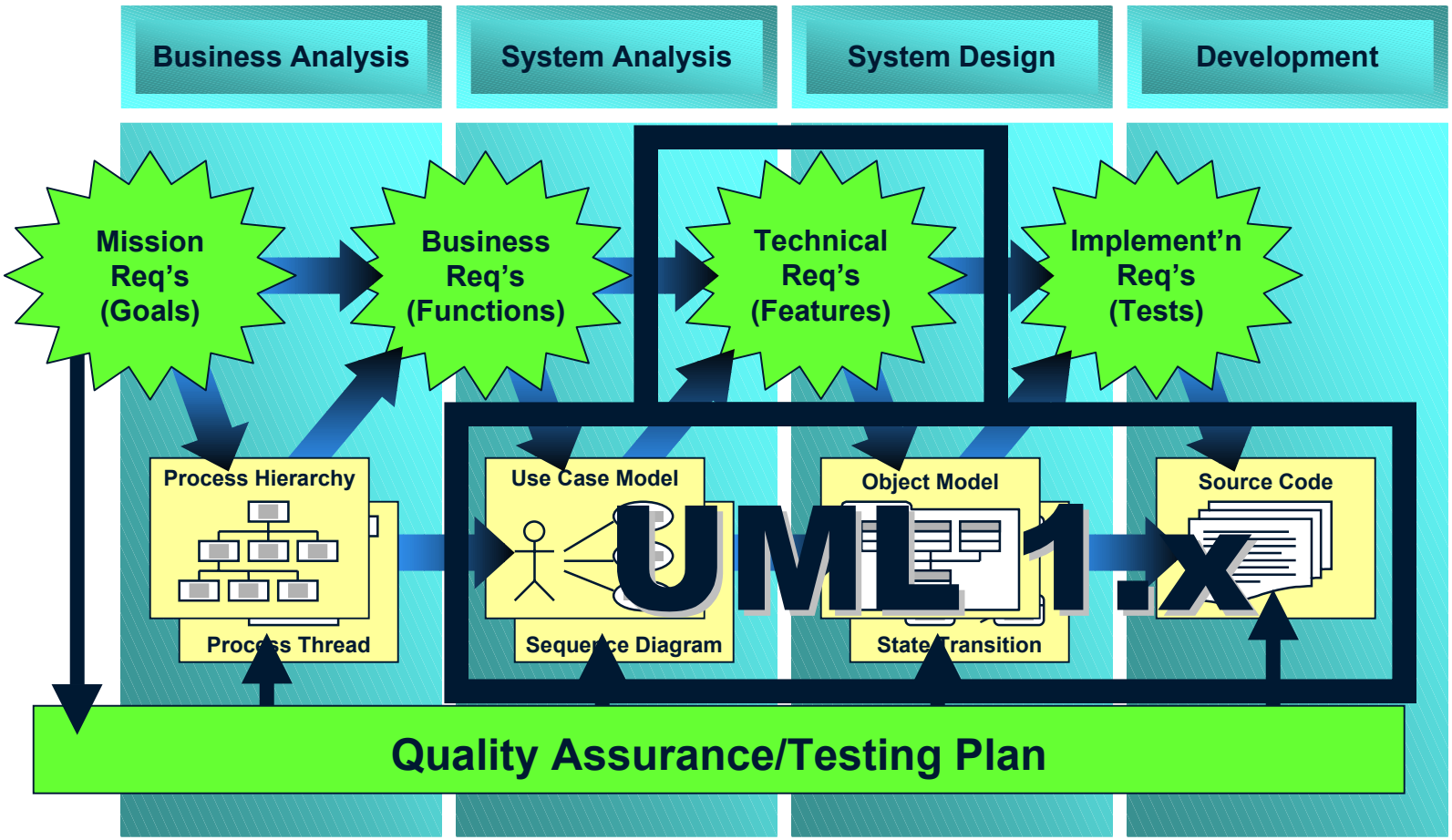
# Requirements-Based UML

- RBU is …

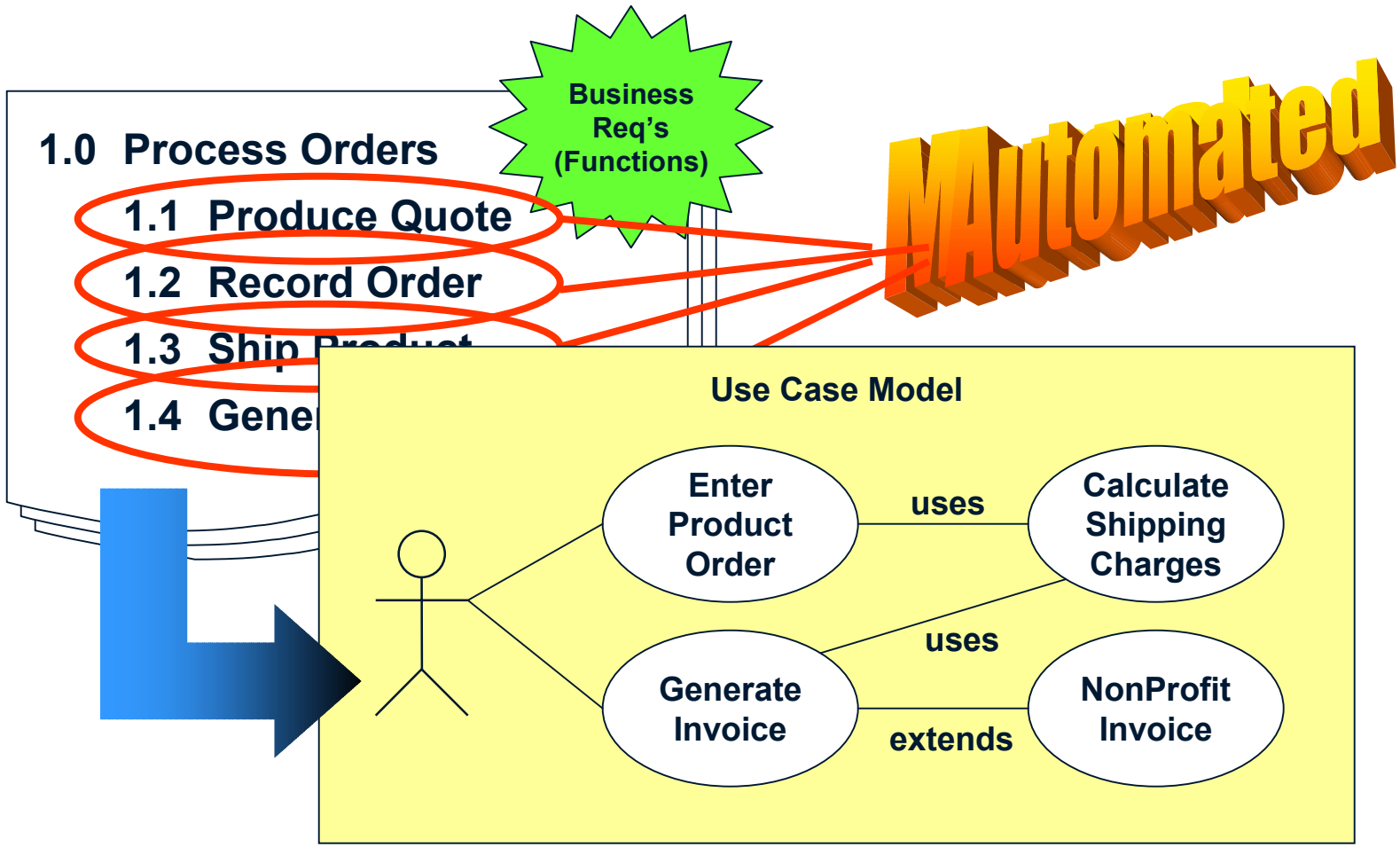
  *"A structured approach for integrating formal requirements analysis into a UML-based analysis and design effort"*

  - A *pragmatic* methodology
  - Based on textual requirement descriptions designed for non-technical stakeholders
  - Driven by *customer-defined* requirements for business and/or application functionality

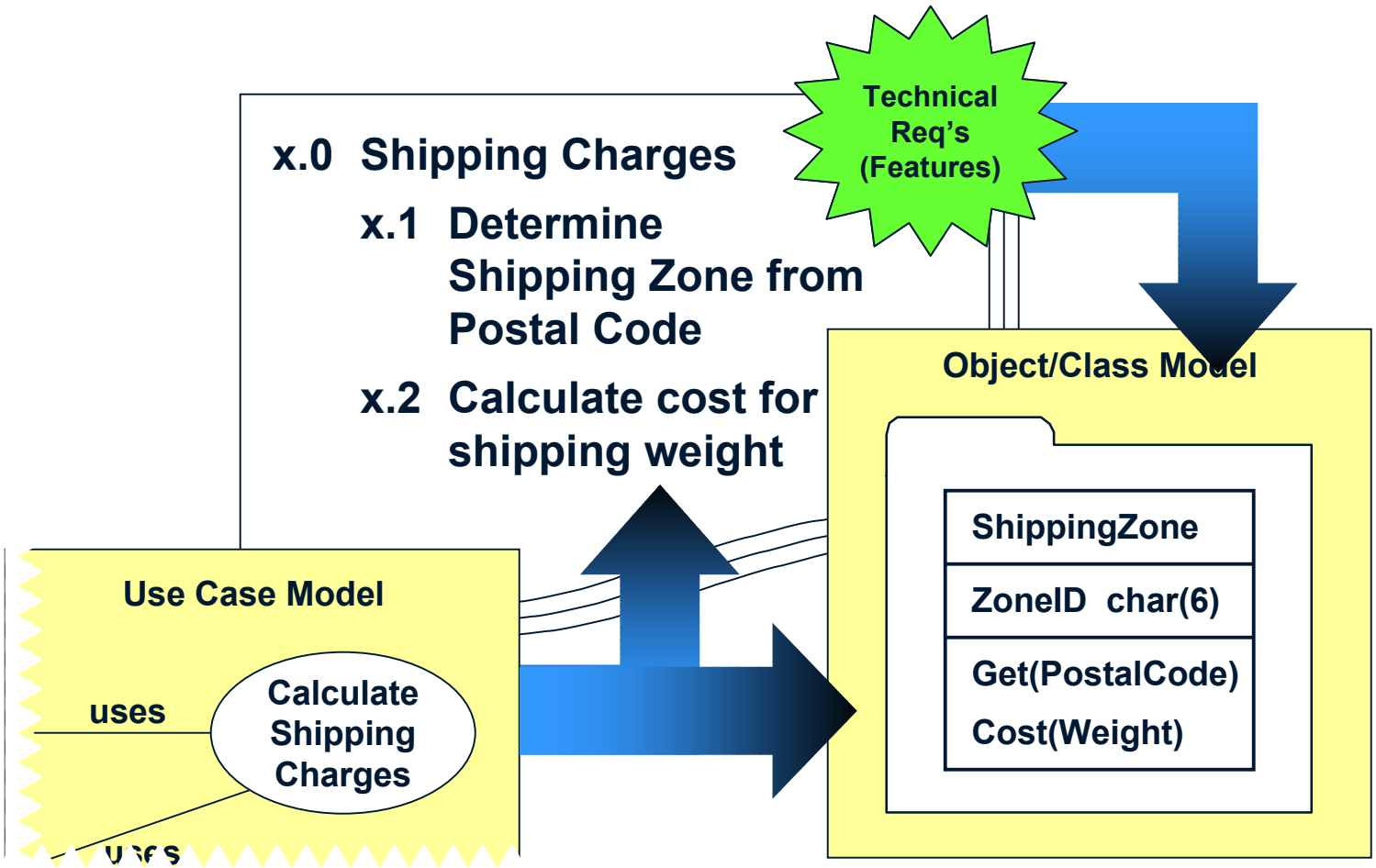
# Requirements-Based UML



# For Example...


Business Req's (Functions)
1.0 Process Orders
1.1 Produce Quote
1.2 Record Order
1.3 Ship Product
1.4 Gener
MAutomated
Use Case Model
Enter Product Order
uses
Calculate Shipping Charges
uses
Generate Invoice
extends
NonProfit Invoice

# For Example (cont.)...

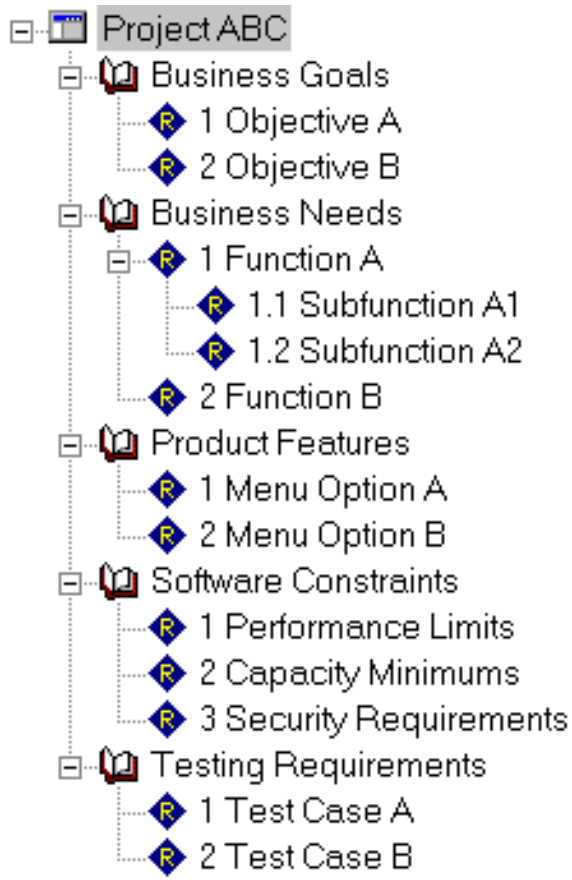


# Benefits of RBU

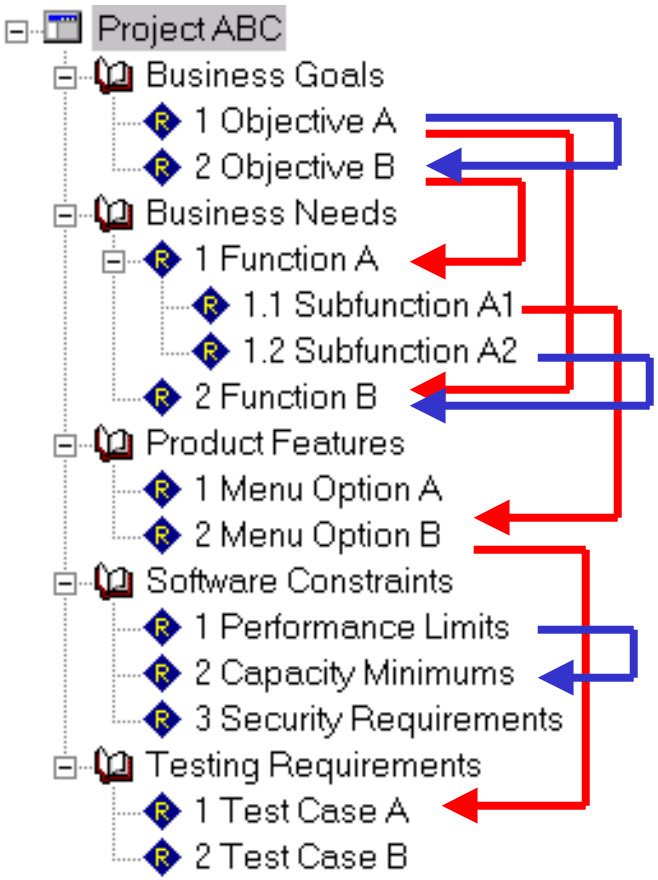
- Improves communication with customers
  - Provides a non-technical, non-threatening, textual report format
  - Produces a richer picture of the application
  - Describes the surrounding workflow context
- Facilitates scope management based on customer business needs
- Provides opportunity to redesign the process
  - "Just Say No!"

# Requirement Hierarchies

- Requirements are typically organized into tree-like hierarchies
  - Different types of requirements are identified at each stage of the process
  - Different audiences participate in each type of requirement analysis



# Requirements Traceability




- Traceability is used to identify relationships between requirements
  - Composition
  - Dependencies
- When changes are proposed, traceability links are examined for impact analysis
  - More accurate time and cost estimates

# In Summary...

Traditional UML-centric methods ...

- Focus on technical modeling tasks
- Treat requirements as "features"
- Quickly plunge into detailed specifications

Requirements-Based UML ...

- Focuses on continual customer involvement
- Delivers non-threatening content
- Facilitates transition between phases

# 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

To order a copy of the Proceedings, please send a check in the amount of $35.00 each to:

PNSQC/Pacific Agenda
PO Box 10142
Portland, OR 97296-0142

Name _____

Affiliate _____

Mailing
Address _____

City _____

State _____

Zip _____ Phone _____