

# SEVENTEENTH ANNUAL PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

OCTOBER 12 - 13, 1999

Oregon Convention Center  
Portland, Oregon

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

## TABLE OF CONTENTS

Preface.....	iii
Conference Officers/Committee Chairs.....	iv
Conference Planning Committee .....	v
Keynote Address – October 12	
The Stakeholders Win-Win Approach to Software Quality .....	1
Dr. Barry Boehm, University of Southern California	
Keynote Address – October 13	
Software Project Survival: A Tale of Two Projects.....	35
Steve McConnell, Construx Software	
Quality Assurance Track – October 12	
25 Things My Mother Never Told Me About Being A QA Manager.....	44
Mark Carver, Intel Corporation	
The Seven Habits of Highly Effective Inspection Teams.....	53
Steve Allott, Imago <sup>QA</sup> , Ltd	
Quality Engineering: In the Footstep of Goldie Locks.....	68
Albert Gallo, NASA	
What is Software QA, Anyway?.....	69
Sandy Raddue, Cypress Semiconductor	
Process Track – October 12	
Software Process Improvement – Avoiding the Pitfalls.....	77
Nicloe Bianco, Motorola	
Yes, But What Are We Measuring? .....	88
Cem Kaner, Consultant	
12 Steps to Useful Software Metrics.....	109
Linda Westfall, The Westfall Team	
Testing Track – October 12	
E-fective Testing for E-Commerce.....	119
Brian Hambling, Imago <sup>QA</sup> , Ltd	
The Trials & Tribulations of Testing A Web Application.....	127
Jennifer Brock-Smith, Software Quality Engineering	

Automate Your Tests – You Won't Regress It!.....	135
Steve Allott, Imago <sup>QA</sup> , Ltd	
Fact or Fiction: Developers Get All the Cool Stuff.....	158
Ingrid Canavan, Rational Software Corporation	
<b>Testing Track – October 12 continued</b>	
Communication Paradigm for Distributed Testing.....	159
Mahesh Balachandran, Micrografix, Inc. & Dao Hoang, Saltire Software, Inc.	
Testing for Relational Database Errors.....	173
Shirley A. Becker, Florida Institute of Technology	
<b>Software Excellence Award</b>	
Use of Metrics on the Jubilee Line Extentsion (JLE) Project.....	184
Ed Ko, Catherine Keane, Heather Duncan, Alcatel Canada	
<b>Process Track – October 13</b>	
What to Do After the Assessment Report.....	214
Curtis Cook, Oregon State University & Marcello Visconti, Universidad Technica Federico, Santa Maria	
Steamrolling the Organization with Process, or Is There a Better Way?.....	229
Neil Potter & Mary Sakry, The Process Group	
Process Improvement as a Capital Investment: Risks & Degerred Paybacks.....	241
Warren Harrison, David Raffo & John Settle, Portland State University	
Matching Process Change to Your Organization: A Case Study using ISO 9000.....	251
Mark Johnson, OrCAD	
<b>Requirements Track – October 13</b>	
Starting a Requirements Managment Initiative.....	258
Don Moreaux, Mike Young, & Kris Hartung, Hewlett Packard	
Requirements Traceability Improvement in Small Software Organizations.....	268
Timo Varkoi & Timo Mäkinen, Tampere University of Technology, Finland	
Taming Requirements Through Metrics.....	285
Dr. Linda Rosenberg, NASA, GSFC	
Capturing Requirements That Reflect Customer Intent.....	286
Markus K. Gröner & James D. Arthur, Virginia Polytechnic Institute & State University	
Requirements for Test Automation.....	303
Douglas Hoffman, Software Quality Methods	
<b>Management Track – October 13</b>	
Ensuring Outsourced Software Success.....	312
Daniel Clemens, MicroCrafts	

Remediation of a Y2K Problem in a CMM-1 Environment – A Case for Project Over Process .....	336
David Butt, Brian Hansen, Tiel Jackson & Sam Sanchez, Oregon Health Sciences University	
Planning for Project Surprises – Coping with Risk.....	344
Neil Potter & Mary Sakry, The Process Group	
Software Risk Management.....	360
Linda Westfall, The Westfall Team	
Solving the Software Quality Management Problem: The Next Step.....	367
James Mater, CEO Revision Labs, Inc.	
Index.....	368
Proceedings Order Form.....	last page

## Preface

Welcome to the 17<sup>th</sup> 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

Yesterday, during a discussion I was listening to, the question was raised are we making the wrong decisions when we manage our projects or is it a reasonable balancing of requirements? The question wasn't answered there. However, that is the very subject Dr. Barry Boehm will discuss in Tuesday's keynote presentation. He will tackle the issue of making tradeoffs between schedule requirements and quality requirements.

Many people in the software industry are thrust into positions of responsibility for software project outcomes, but few are given training in how to make them succeed. In Wednesday's keynote presentation, Steve McConnell describes two projects with very different outcomes, and analyzes the root causes of software success and failure.

I've seen several of our presenters write on software quality and I'm excited that I will get the chance to hear them. I also look forward to hearing our many paper presenters. I find they are solving many of the same problems I face. We can take a second message away from their presentations; you don't have to be a published author to have something worthwhile to say. Give serious consideration replying to the call for papers.

I was asked how we can put together the conference every year. It happens every year because a lot of very busy people volunteer their time to make it happen. I am amazed not only at the work they do, but in many cases at the fact they find the time to do that work. It's the time these volunteers devote that allows us to put on the conference for the low cost and the quality we have. If you want to give back to your profession while working with other excellent dedicated volunteers, there a place to volunteer on the conference feedback form.

Our spring workshops will be May 15 and 16, 2000 in Portland and Seattle. Our annual conference and fall workshops will be October 16, 17, 18, 2000 right here at the Oregon Convention Center. We look forward to seeing you again at these events.

Rick Clements, PNSQC President and Chair

## CONFERENCE OFFICERS/COMMITTEE CHAIRS

**Rick Clements – PNSQC President/Chair**

*FLIR Systems, Inc.*

**Ian Savage - PNSQC Vice President**

*Tiger Systems Inc.*

**Sue Bartlett - PNSQC Secretary,**

**Keynote 2000 Chair**

*Step Technology, Inc.*

**Doug Vorwaller – PNSQC Treasurer**

*Fred Meyer, Inc.*

**Dennis Ganoe – Board Member**

*Providence Health Plans*

**Ray Lischner - Board Member**

*Tempest Software*

**Paul Dittman – Program Chair**

*Intel Corporation*

**Dick Fairley – Keynote 1999 Chair**

*Oregon Graduate Institute*

**Sandy Raddue – Publicity Chair**

*Cypress Semiconductor*

**Piet van Weel - PNSQC Software**

**Excellence Award**

*BEST Consulting, Inc.*

**Shauna Gonzales - Birds of a Feather**

*ImageBuilder Software*

**Don White - Exhibits Chair**

*BidTek*

**Bhushan Gupta - Workshop Chair**

*Hewlett Packard*

## CONFERENCE PLANNING COMMITTEE

**Hilly Alexander**

**John Arce**

**Gerry Belt**  
*AMS*

**Pieter Botman**  
*True North Systems Consulting*

**Bob Brown**  
*Intel Corporation*

**Angel Cruz**  
*Hewlett Packard*

**Mark Carver**  
*Intel Corporation*

**John Elliott**  
*Mentor Graphics*

**Prabhala Ganesh**  
*Intel*

**Galina Golant**  
*Cypress Semiconductor*

**Kathi Harris**  
*Intel Corporation*

**Pat Hyland**  
*Wall Data*

**Mark Johnson**  
*OrCAD Corp.*

**Greg Jones**  
*Nations Bank*

**Mike Kress**  
*Boeing*

**Akiko McNair**  
*Timberline Software*

**Howard Mercier**  
*Step Technology, Inc.*

**Rick Murphy**  
*Wall Data*

**Tracy Samper**  
*Intel Corporation*

**Allen Sampson**  
*Intel Corporation*

**Erik Simmons**  
*Cotelligent*

**Vicki Shinneman**  
*Hewlett Packard*

**Bryan Stickel**  
*Flight Dynamics*

**Jim Teisher**  
*Credence Systems*

**Richard Vireday**  
*Intel Corporation*

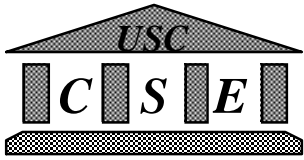
## **The Stakeholders Win-Win Approach to Software Quality**

Dr. Barry Boehm, University of Southern California

Resolving conflicts among software quality requirements and cost/schedule to implement those requirements is difficult because of incompatibilities among stakeholders' interests and priorities, complex cost-quality dependencies, and an increasingly large number of options for larger systems. This presentation describes an analysis/negotiation strategy and a tool (S-COST) that helps stakeholders to 1) identify appropriate resolution strategies for cost-quality conflicts, 2) visualize the options, and 3) negotiate mutually satisfactory balances among software quality requirements and the associated costs and schedules. Components of S-COST include a groupware support system for determining requirements as negotiated win conditions; a support system for identifying quality conflicts in software requirements; a quality attribute and risk conflict tool; and the COncstructive COst estimation MOdel (COCOMO).

Barry W. Boehm is a professor of Software Engineering and Director of the Center for Software Engineering at the University of Southern California. He is also Chair of the Board of Visitors for Carnegie Mellon University's Software Engineering Institute and Chair of the Air Force Scientific Advisory Board's Information Technology Panel. Dr. Boehm is an AIAA fellow, an ACM Fellow, an IEEE Fellow, and a member of the National Academy of Engineering. His contributions to software engineering include the COCOMO model, the Spiral model of software development, the theory W (win-win) approach to software requirements determination and management, and two advanced software engineering environments: the TRW software productivity system and the quantum leap environment. Dr. Boehm has a bachelor's degree from Harvard and doctorate degrees from UCLA.

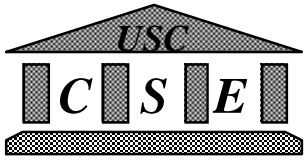




# **The Stakeholder Win-Win Approach to Software Quality**

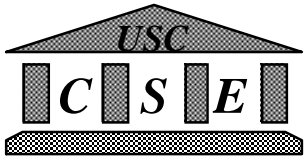
**Barry Boehm, and Hoh In**  
**(USC and Texas A&M U.)**  
**PNSQC Keynote Address**  
**October 12, 1999**

**Boehm@sunset.usc.edu**  
**<http://sunset.usc.edu>**



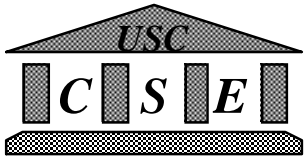
# Outline

- **What is Software Quality?**
- **A Conceptual Framework for Achieving Quality**
  - **The WinWin Spiral Model**
- **Negotiation Aids for Cost-Quality Requirements**
- **Experimental Results**
- **Conclusions**



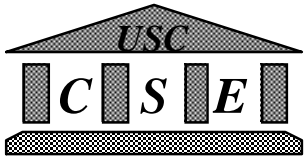
# What Is Software Quality?

- **Conformance to specifications?**
- **Accuracy, adaptability, affordability, availability, ...?**
- **Satisfying the customer?**
- **Something else?**



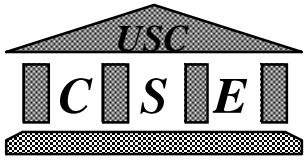
# Software Quality Is ...

- **Conformance to specifications?**
- **Some difficulties: What if ...**
  - **The specifications are wrong?**
  - **The budget and schedule are way overrun?**
  - **The code is inscrutable, scary to modify?**



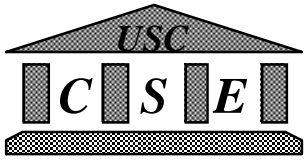
# Software Quality Is ...

- **Accuracy, adaptability, affordability, availability, ...?**
- **Some difficulties: What if ...**
  - **These attributes conflict with each other?**
  - **Customers can't specify how much of each they need?**
  - **Tradeoffs are impractical to analyze or prioritize?**
  - **Attribute levels can't be accurately measured?**



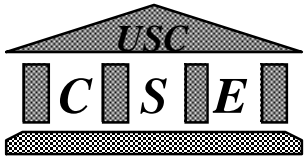
# Software Quality Is ...

- **Satisfying the customer?**
- **Some difficulties: What if ...**
  - **Customer is unhappy next month?**
  - **Users are unhappy?**
  - **Maintainers are unhappy, disempowered?**
  - **Developers are burned out?**



# Hypothesis: Software Quality Is

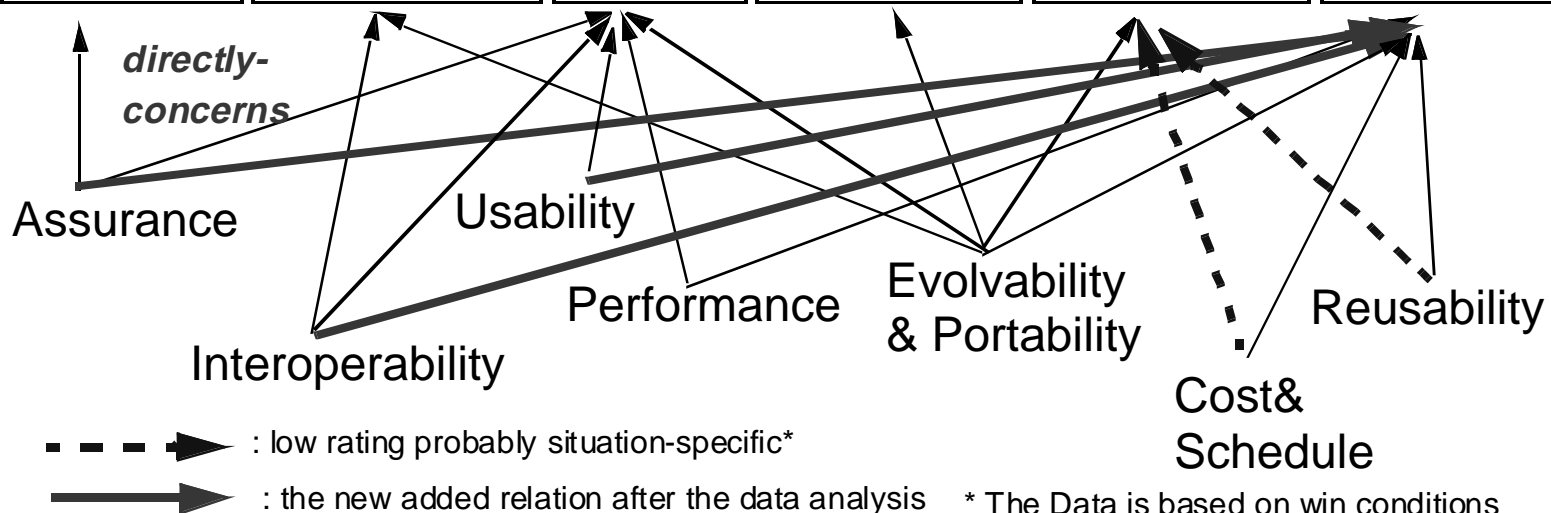
- **Making winners of all your system's stakeholders**
  - **Users, customers, developers, maintainers, interoperators, general public, other significant affected parties**



# Stakeholder/Attribute Relationship

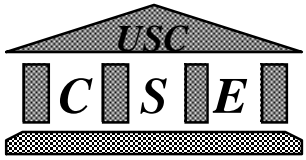
*Stakeholders and their roles & responsibilities:*

Avoid adverse system side-effects: safety, security/privacy	Avoid current and future interface problems	Execute cost-effective operational missions	Avoid low utility due to obsolescence; Cost-effective product support after development	Avoid non verifiable, expendable, flexible, reusable product; Avoid the delay of product delivery and cost overrun	Avoid overrun budget and schedule; Avoid low utilization of the system
General Public	Interoperator	User	Maintainer	Developer	Customer



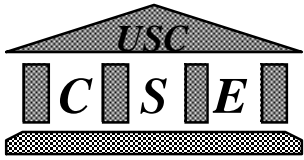
\* The Data is based on win conditions of 14 student digital library projects





# Outline

- **What is Software Quality?**
- • **A Conceptual Framework for Achieving Quality**
  - **The WinWin Spiral Model**
- **Negotiation Aids for Cost-Quality Requirements**
- **Experimental Results**
- **Conclusions**



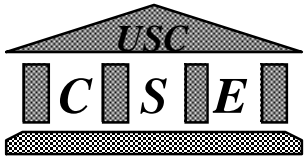
# The Fundamental Success Condition

**Your project will succeed**

**if and only if**

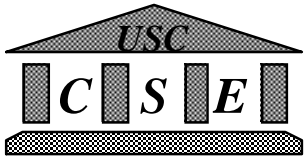
**You make winners of all the critical stakeholders**

- **Usually: Users, customers, developers, maintainers**
- **Sometimes: Interfacers, testers, reusers, general public**



# Win-Lose Evolves into Lose-Lose

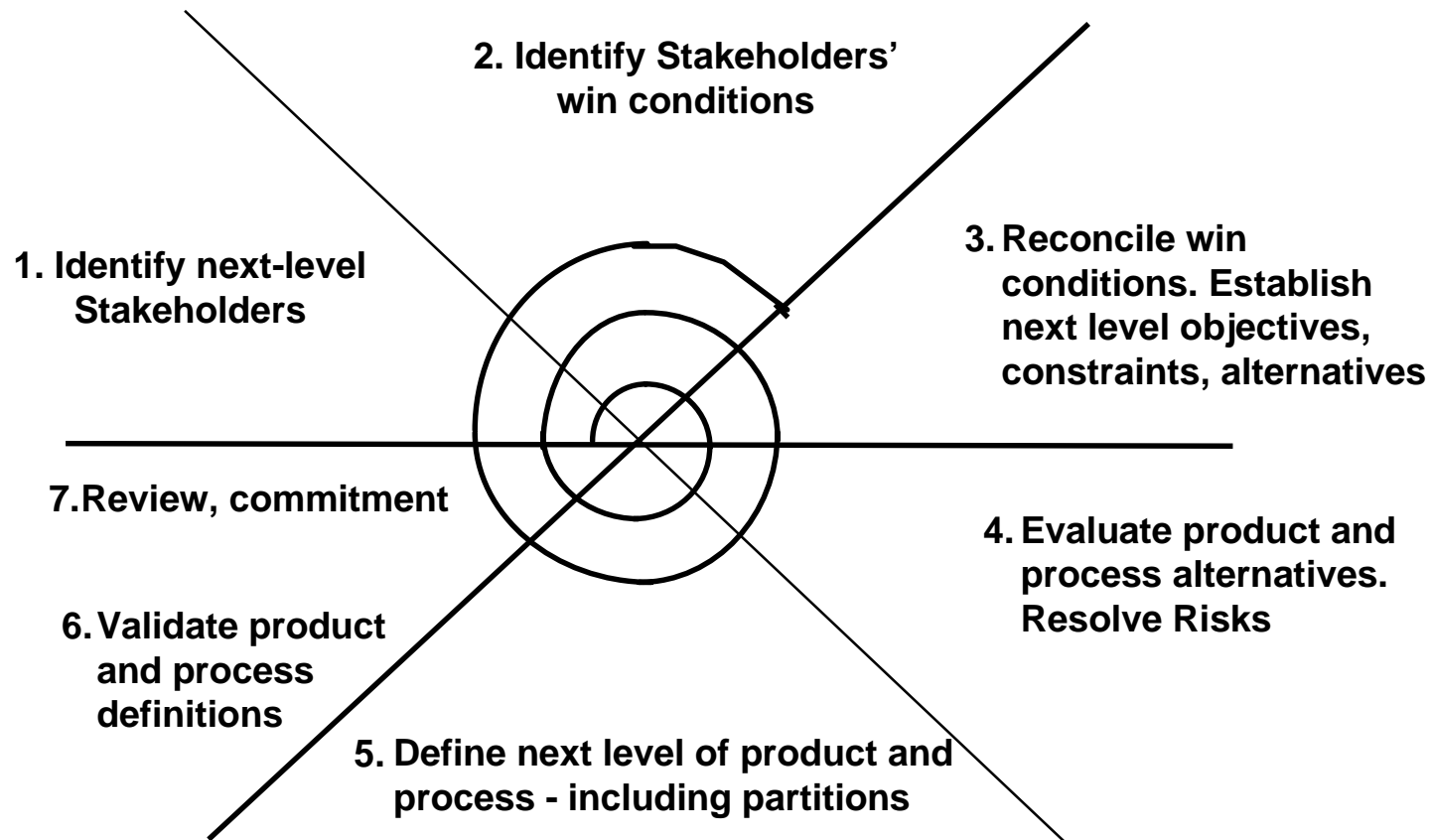
Proposed Solution	“Winner”	Loser
Cheap, Sloppy Product (“Buyer knows best”)	Developer & Customer	User
Lots of bells and whistles (“Cost-plus”)	Developer & User	Customer
Driving too hard a bargain (“Best and Final offers”)	Customer & User	Developer



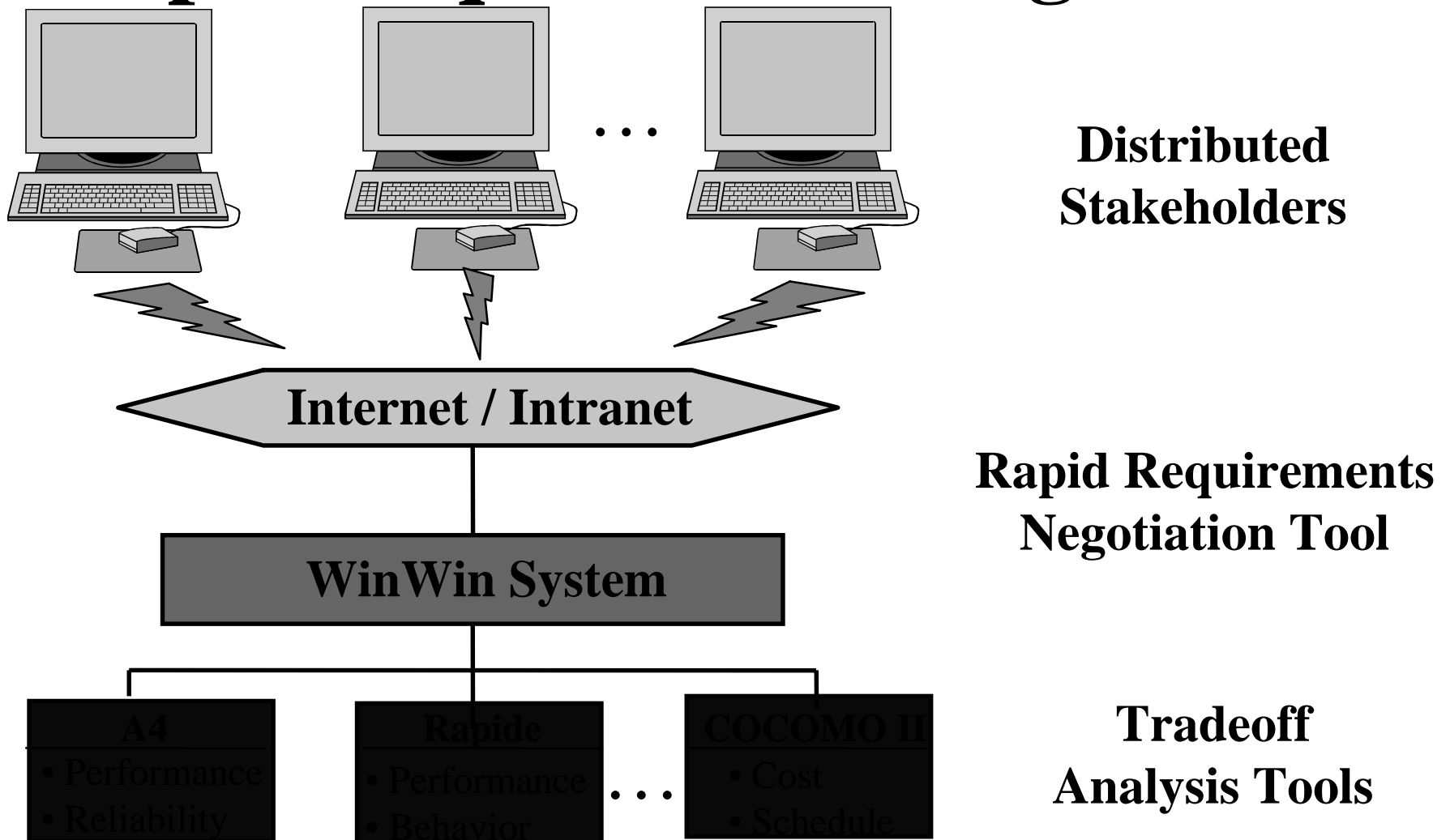
# Theory W Management Steps

- 1. Identify success-critical stakeholders**
- 2. Identify stakeholders' win conditions**
- 3. Identify win condition conflict issues**
- 4. Negotiate top-level win-win agreements**
  - Invent options for mutual gain**
  - Explore option tradeoffs**
  - Manage expectations**
- 5. Embody win-win agreements into specs and plans**
- 6. Elaborate steps 1-5 until product is fully developed**
  - Confront, resolve new win-lose, lose-lose risk items**

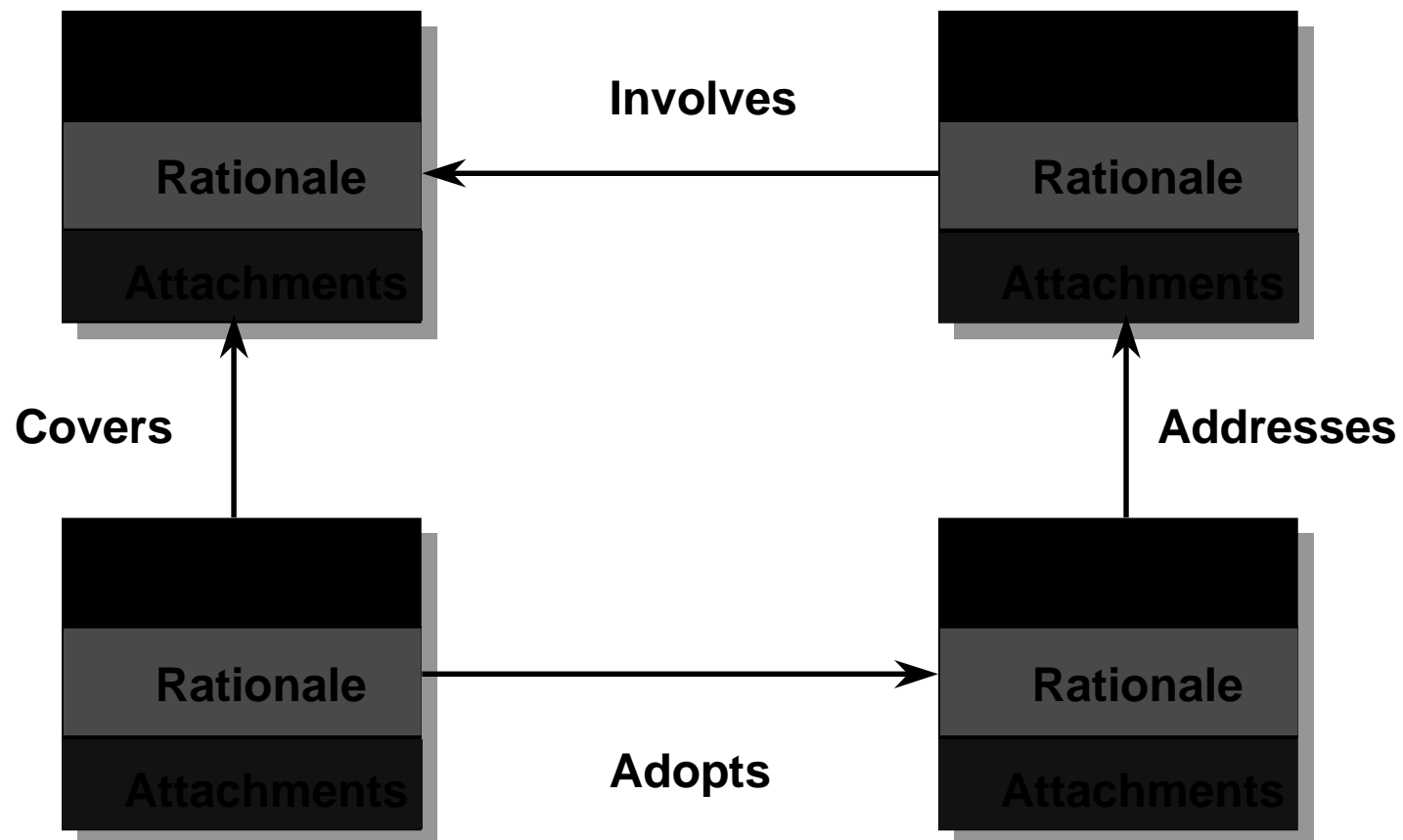
# Theory W Extension to Spiral Model



# Rapid Requirements Negotiation

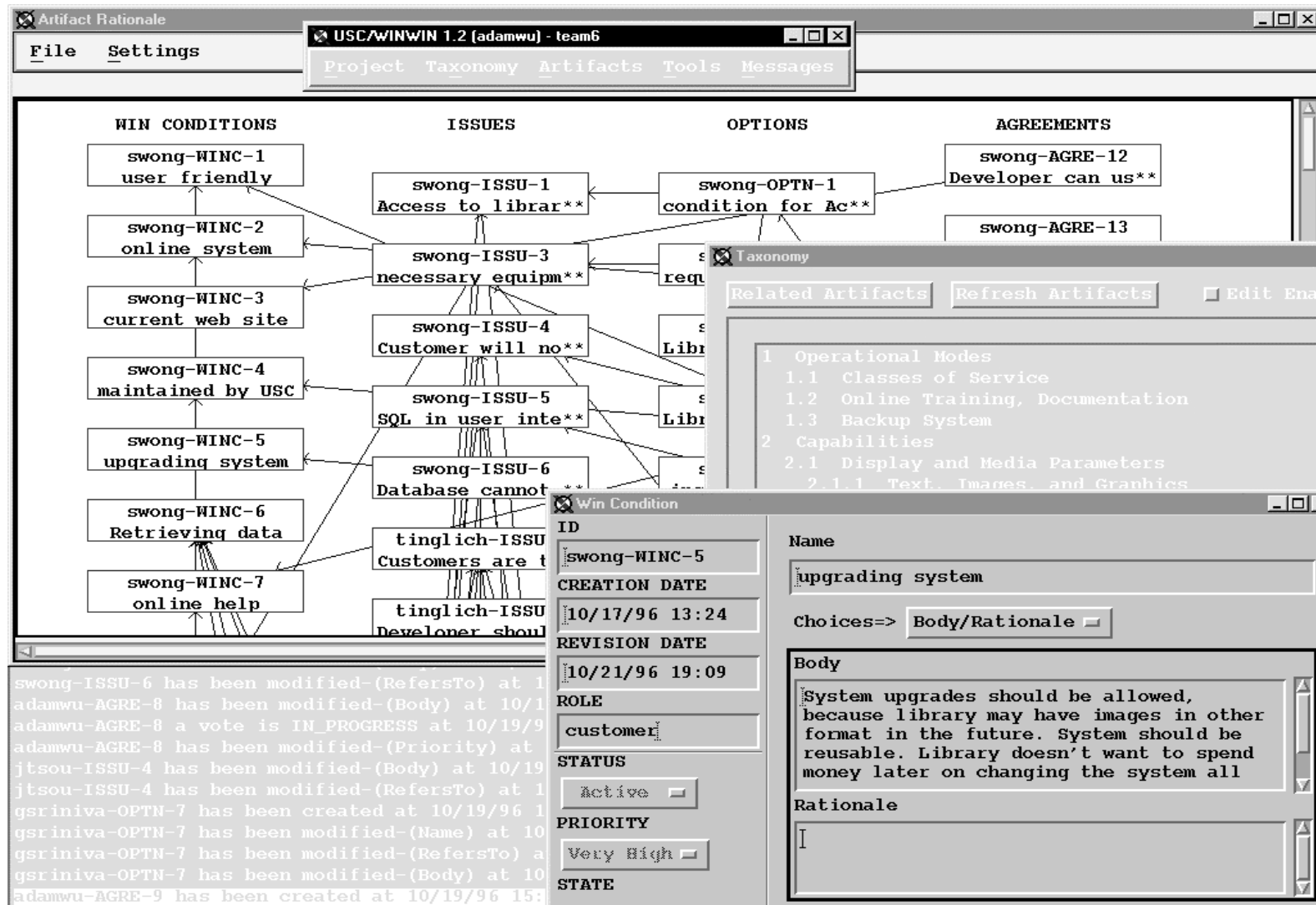


# WinWin Negotiation Model

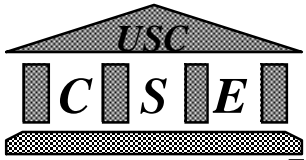




# WinWin Look and Feel



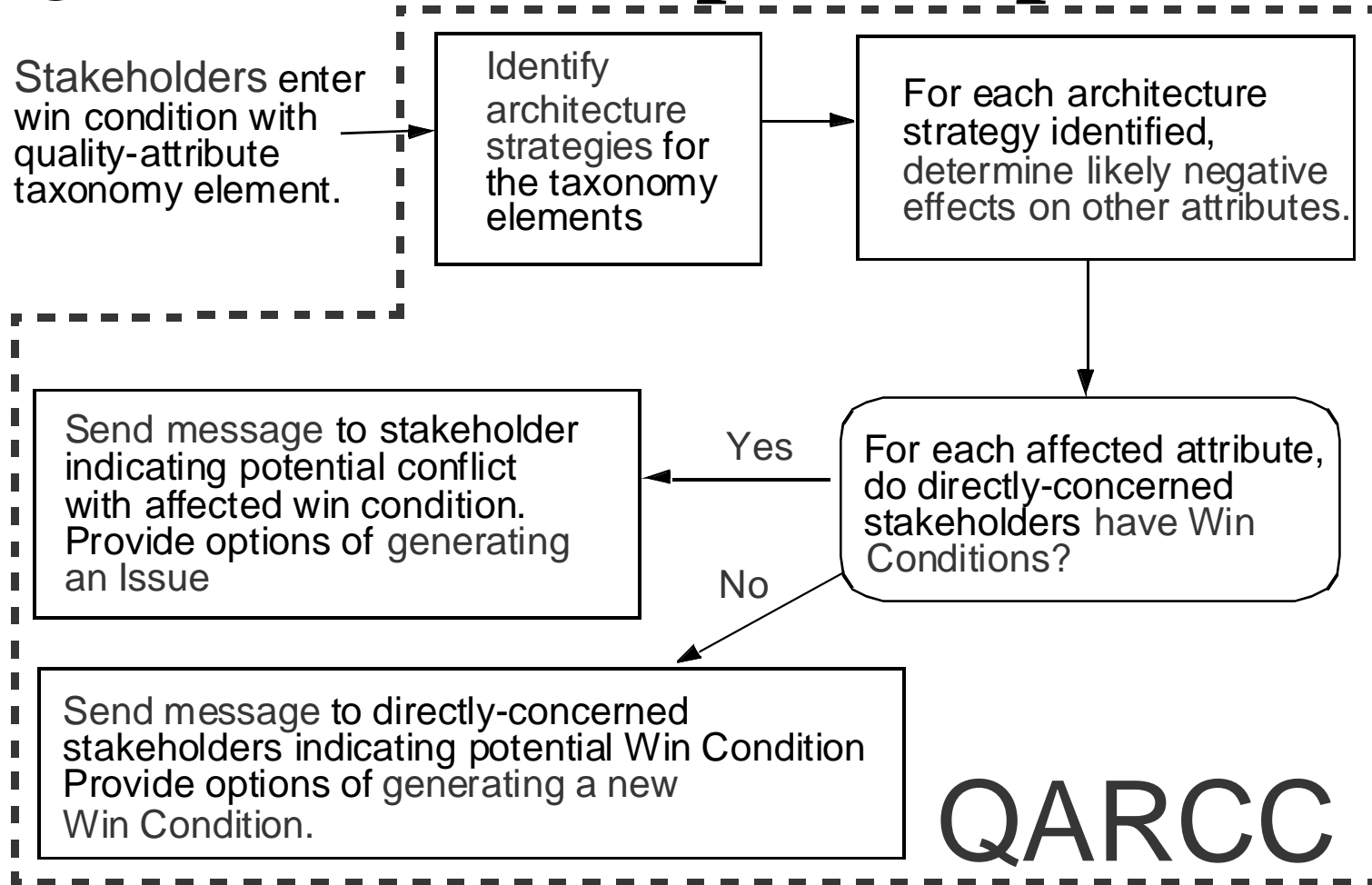




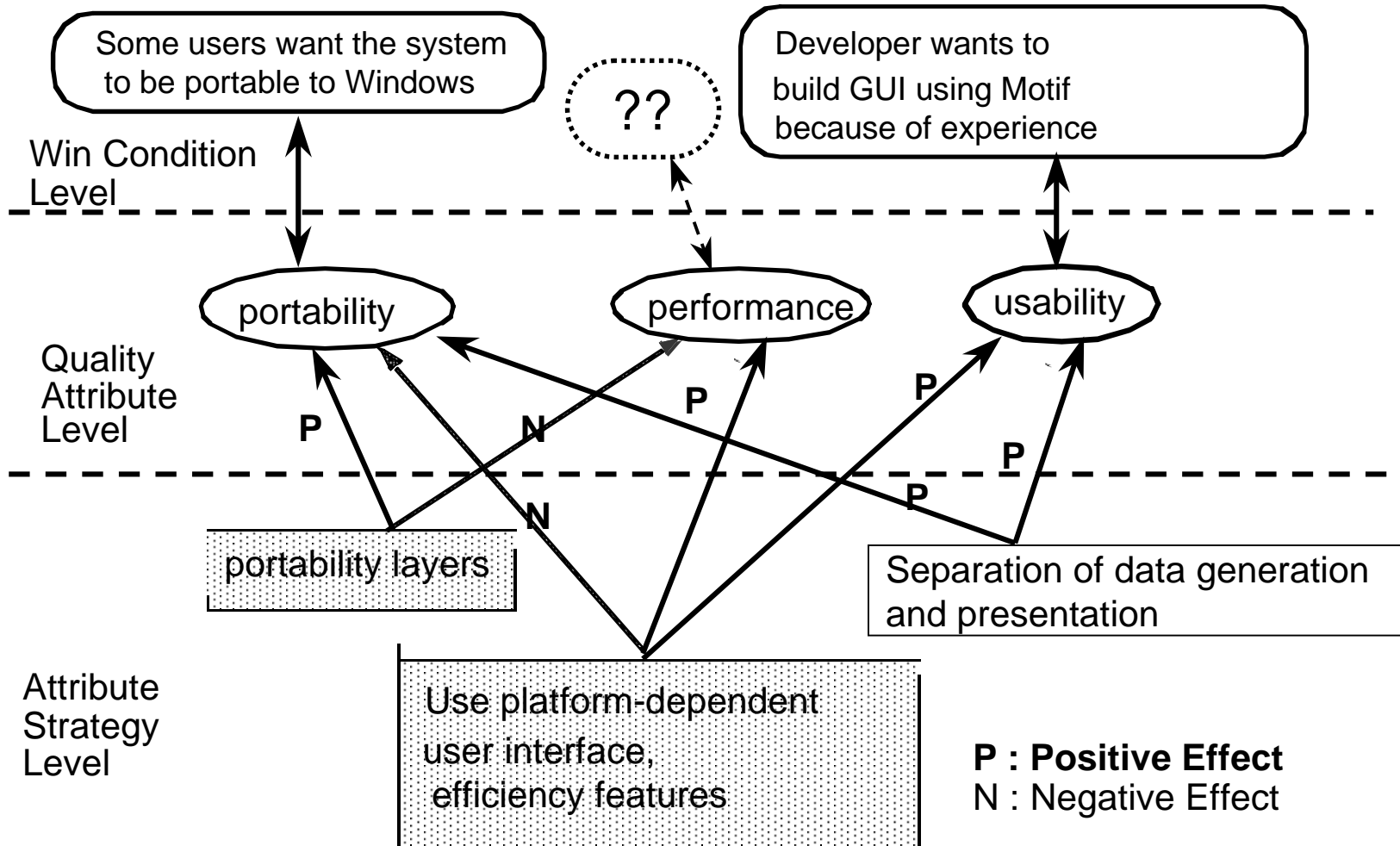
# WinWin System Challenge

- **With large number of stakeholder artifacts,**
  - Hard to identify all likely conflicts
  - Hard to determine good resolution strategies
- **Two tools developed to address challenge**
  - QARCC (Quality, Attribute and Risk Conflict Consultant)
  - S-COST (Software Cost Option Strategy Tool)

# QARCC Concept of Operation

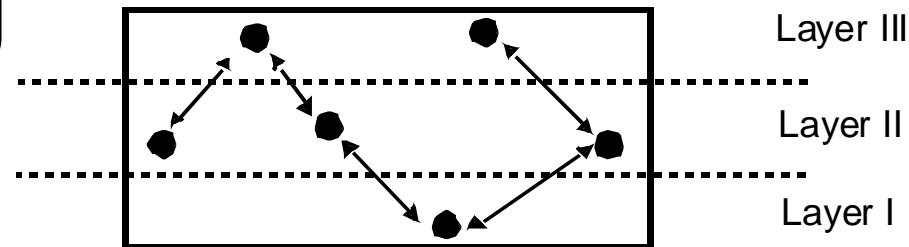


# Identifying Attribute Conflicts



# An Example of Architecture Strategy

## • Layering



- **Definition:** A hierarchical, architectural composition in which each layer can communicate only with the adjacent upwards or downwards layer
- **Preconditions:** Interface and protocol between a layer and an adjacent layer, request to pass data and/or control from layer to layer
- **Postconditions:** Data and/or control passed from layer to layer, or notification of interface/protocol violation
- **Effects on quality attributes:**
  - **Evolvability, Interoperability, Portability, Reusability:** (+, hide sources of variation inside interface layers)
  - **Performance** (-, need more interfaces, and data and/or control transfers, via protocol)
  - **Development Cost, Schedule:** (-, more to specify, develop, verify)



# QARCC: Potential Quality Conflicts

Quality Conflict Advice Note

The new Win Condition ( jkerner-WINC-6 ) on Development attribute results in potential conflicts with the following attributes:

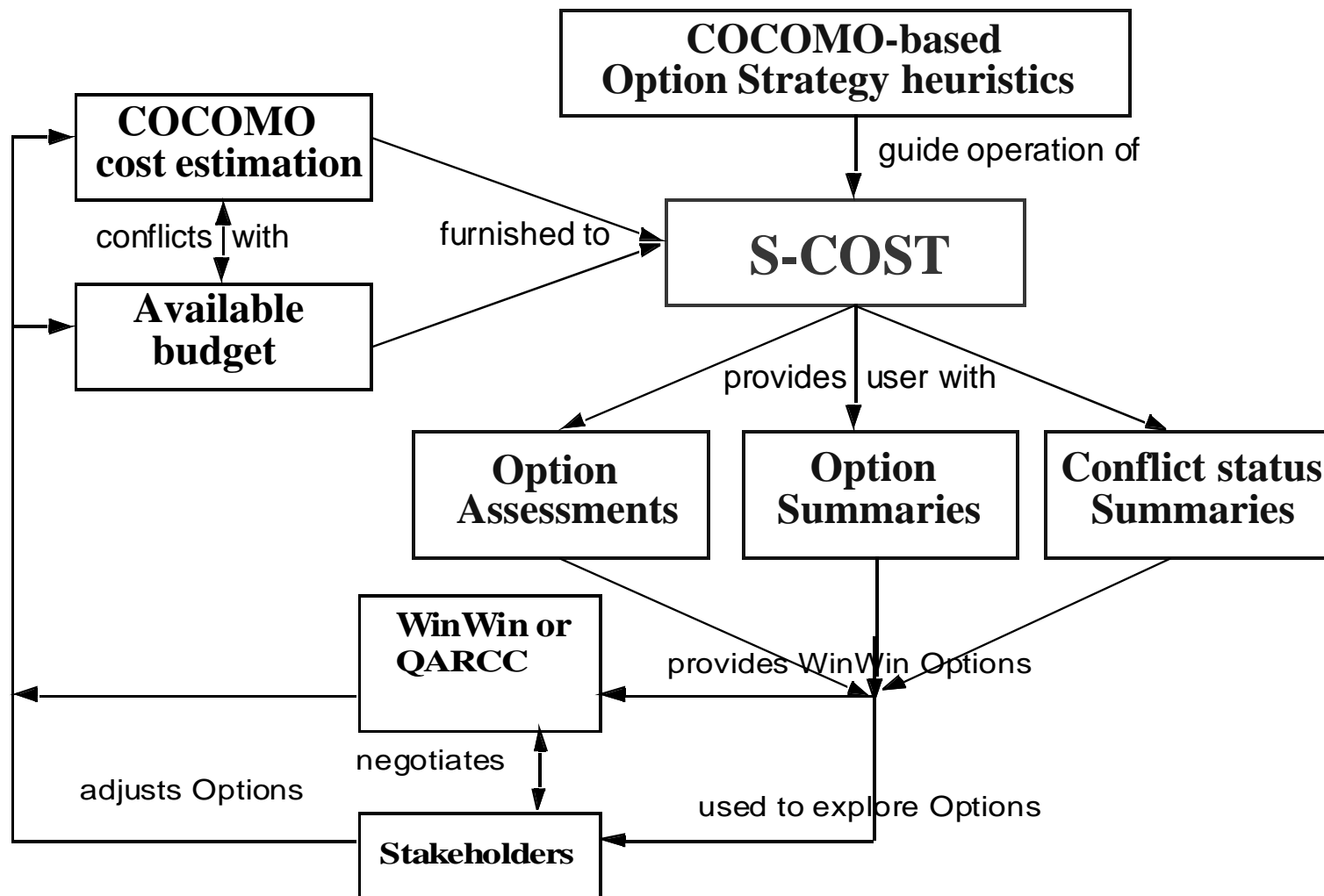
Quality Attributes:	Related Win Conditions
Performance	bose-WINC-6, Sensor data accuracy
<b>Assurance</b>	bose-WINC-18, Assurances on archiving capabil
Usability	bose-WINC-19, Assurances on archiving capabil
Evolvability	bose-WINC-21, Reilable
Interoperability	bose-WINC-23, Reilable
Portability	
Reusability	

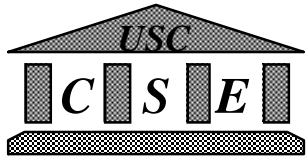
Potential Conflict between Win Condition on: Development and Win Condition on: Assurance Due to:

- verification, specification, Firewalling  
( The techniques such as verification, specification, and firewalling can improve assurance, but increase development cost and schedule. )
- Input checking, Integrity functions  
( The functions such as input checking and integrity can improve assurance, but increase development cost and schedule )
- Assertion type checking  
( The assertion type checking can reduce input errors and improve assurance, but increase development cost and schedule )

Create WinC      Create Issue      Help      Cancel

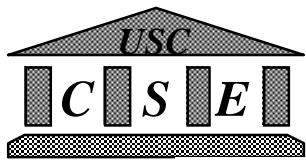
# S-COST Concept of Operation





# COCOMO-based Option Strategies

Option Strategies	COCOMO Parameter	Pros	Cons
Reduce/defer Functionality	KDSI, DATA	<ul style="list-style-type: none"> <li>- Reduce cost, IOC, and schedule</li> <li>- Smaller product to maintain</li> </ul>	<ul style="list-style-type: none"> <li>- Capabilities unavailable to stakeholders</li> <li>- Need to pay later if deferred</li> </ul>
Reduce/defer Quality	RELY, TIME, DOCU	<ul style="list-style-type: none"> <li>- Reduce cost, schedule, and complexity</li> </ul>	<ul style="list-style-type: none"> <li>- Stakeholders lose quality capabilities such as reliability and maintainability</li> </ul>
Improve tools, techniques, Platform Relax schedule constraint	TIME, STOR, PVOL, TOOL, SITE SCED	<ul style="list-style-type: none"> <li>- Reduce s/w cost and schedule</li> <li>- Improve maintainability and other qualities</li> <li>- Reduce cost if schedule was tight</li> </ul>	<ul style="list-style-type: none"> <li>- Increase tool, training, platform costs</li> <li>- Reducing tool, platform experience would</li> </ul>
Improve personnel capabilities	ACAP, PCAP, AEXP, PEXP, LTEX, \$K/PM	<ul style="list-style-type: none"> <li>- Reduce cost and schedule</li> <li>- Improve quality from personnel capability and/or application experience</li> </ul>	<ul style="list-style-type: none"> <li>- Projects losing better people will suffer</li> <li>- Potential staffing difficulties and delays</li> <li>- Increased cost/person-month unless low-cost outsourcing</li> </ul>
Reuse software assets	ADSI, DM, CM, IM	<ul style="list-style-type: none"> <li>- Reduce cost and schedule</li> <li>- May gain quality if the used assets have good quality</li> </ul>	<ul style="list-style-type: none"> <li>- Users may lose quality capabilities</li> <li>- Risk of overestimating reuse</li> </ul>
Improve coordination via teambuilding Architecture and risk resolution Improve process maturity level	TEAM  RESL  PMAT	<ul style="list-style-type: none"> <li>- Reduce cost and schedule by removing the inter-personnel overhead</li> <li>- Reduce cost and schedule by avoiding rework</li> <li>- Reduce cost and schedule by removing the efforts for fixing errors</li> <li>- Improve quality</li> </ul>	<ul style="list-style-type: none"> <li>- Uncontrollable for some situations</li> <li>- Additional overhead for risk management is necessary.</li> <li>- Additional overhead for applying CMM is necessary</li> </ul>
Improve development flexibility Increase budget	FLEX  Revised win condition	<ul style="list-style-type: none"> <li>- Reduce cost and schedule by familiarity and flexibility of software development</li> <li>- May enable product to reach competitive critical mass</li> <li>- May increase ROI</li> </ul>	<ul style="list-style-type: none"> <li>- Uncontrollable for some situations</li> <li>- Added funds may not be available</li> </ul>



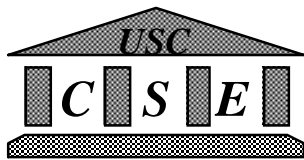
## COCOMO-based Option Strategies

Option Strategies	COCOMO Parameter	Pros	Cons
Reduce/defer Functionality	KDSI, DATA	- Reduce cost, IOC, and schedule - Smaller product to maintain	- Capabilities unavailable to stakeholders - Need to pay later if deferred
Reduce/defer Quality	RELY, TIME, DOCU	- Reduce cost, schedule, and complexity	- Stakeholders lose quality capabilities such as reliability and maintainability
Improve tools, techniques, platform	TIME, STOR, PVOL, TOOL, SITE	- Reduce s/w cost and schedule - Improve maintainability and other qualities	- Increase tool, training, platform costs - Reducing tool, platform experience would increase s/w cost
		- Increases tight	- Defer stakeholders use of product capabilities
			- People will suffer

### Improve tools, techniques or platform

- COCOMO cost drivers: TIME, STOR, PVOL, TOOL, SITE.
- Pros: Reduce s/w cost, schedule; Improve maintainability, other qualities
- Cons: Increase tool, training, platform costs; Reducing tool, platform experience would increase s/w cost





# S-COST: Option Assessments

ID

hohin-SCOST-1

ROLE

User

PRIORITY

Very High

Target Cost (\$K)

5000.00

Estimated Cost (\$K)

5379.02

Cost Difference (\$K)

379.02

Target Schedule (M)

16.00

Estimated Schedule (M)

16.42

Schedule Difference (M)

0.42

Option Generation Aids

Name

Deferring Functionality

<input type="checkbox"/>	DATA INTEGRATION	High	2895
<input type="checkbox"/>	OPS. MGMT (FROM MDIF	Medium	3516
<input type="checkbox"/>	WEATHER DATA PROC. <	Medium	3654
<input checked="" type="checkbox"/>	SS/WB QUERY/DISPLAY	Low	4206
<input checked="" type="checkbox"/>	MDIF PRIORITY UPGRAD	Low	5379

Total:

5379

Negotiation Aid

Operations:

Split

Resolution Strategies:

Reduce/defer Functionality

Resolution Strategies for the Option

Reduce/defer\_Functionality ( SS/WB QUERY/DISPLAY (NEW), KDSI, 260

Reduce/defer\_Functionality ( MDIF PRIORITY UPGRADES, KDSI, 260

Pros & Cons

Pros of this strategy are th

- Reduce cost,IOC,schedule

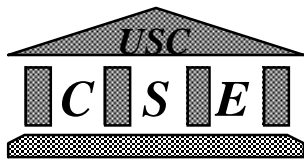
- Smaller produce to maintai

Cons of this strategy are th

Apply

Delete

Cancel



# S-COST: Option Summaries

Option Summary					
Option Strategies	ICOCOMO IDriver	Current Status	Improvement In Principle	More Realistic Improvement	Comments
Reduce/defer_Functionality	KDSI DATA	68 KDSI NM: 1.00	LO: 0.93 ( -7)	NM: 1.00 ( 0)	Reductions dependent on problem and system structure Data base size is not much controllable
Reduce/defer_Quality	RELY DOCU CPLX TIME	NM: 1.00 NM: 1.00 NM: 1.00 NM: 1.00	VL: 0.75 ( -25) VL: 0.89 ( -11) VL: 0.75 ( -25) NM: 1.00 ( 0)	NM: 1.00 ( 0) NM: 1.00 ( 0) NM: 1.00 ( 0) NM: 1.00 ( 0)	Reducing reliability is risky Reducing documentation can cause increased maintenance cost Complexity is not very controllable You have the lowest value. No improvement is available
Improve_tools_and_platform	TIME STOR PVOL TOOL SITE	NM: 1.00 NM: 1.00 NM: 1.00 NM: 1.00 NM: 1.00	NM: 1.00 ( 0) NM: 1.00 ( 0) LO: 0.87 ( -13) VH: 0.72 ( -28) XH: 0.78 ( -22)	NM: 1.00 ( 0) NM: 1.00 ( 0) LO: 0.87 ( -13) VH: 0.72 ( -28) HI: 0.92 ( -8)	You have the lowest value. No improvement is available You have the lowest value. No improvement is available Choose more stable platform; may decreased performance Buy and use more powerful software tools; Need to pay softwa Provide better inter-site communications
Improve_personnel_capability	\$K/PM ACAP AEXP PCAP PEXP LTEX PCON	AV:\$ 15K NM: 1.00 NM: 1.00 NM: 1.00 NM: 1.00 NM: 1.00 NM: 1.00	VH: 0.67 ( -33) VH: 0.81 ( -19) VH: 0.74 ( -26) VH: 0.81 ( -19) VH: 0.84 ( -16) VH: 0.84 ( -16)	HI: 0.83 ( -17) HI: 0.89 ( -11) HI: 0.87 ( -13) HI: 0.88 ( -12) HI: 0.91 ( -9) HI: 0.92 ( -8)	Risky; may get weaker people. Some outsourcing options. Dispatch or hire high-experienced analyst if available. Or c Buy application experience by consulting or hire high-experi Hire more productive programmers or increase program capabil Select popular platform or train developers to increase plat Train programmers to increase language and tool experience c Completion bonuses, other motivators
Relax_Schedule_constraints	SCED	NM: 1.00	NM: 1.00 ( 0)	NM: 1.00 ( 0)	You have the lowest value. No improvement is available

OK



# S-COST: Conflict Status Summaries

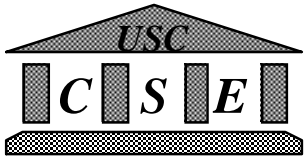
Option Strategy List

Double click to see artifact

hohin-SCOST-1.0	Very High	User	Reduce/defer_Functionality ( SS/WB QUERY/DISPLAY (NEW)
hohin-SCOST-1.1	Very High	User	Reduce/defer_Functionality ( MDIF PRIORITY UPGRADES, K
boehm-SCOST-1.0	High	Developer	Split (SS/WB Query, SS/WB Query:New, SS/WB Query:MDIF
boehm-SCOST-1.1	High	Developer	Reduce/defer_Functionality( SS/WB Query (MDIF), KDSI, :
boehm-SCOST-1.2	High	Developer	Reduce/defer_Functionality( SS/WB Query (NEW), KDSI, 9
boehm-SCOST-1.3	High	Developer	Reduce/defer_Functionality( SS/WB Query (MDIF), DATA, I
horowitz-SCOST-1.0	Very High	Customer	Relax Schedule constraints( SS/WB Query (MDIF), SCED, '

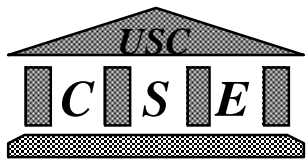
boehm-SCOST-1.0

Apply Delete Options Cancel



# Outline

- **What is Software Quality?**
- **A Conceptual Framework for Achieving Quality**
  - **The WinWin Spiral Model**
- **Negotiation Aids for Cost-Quality Requirements**
- • **Experimental Results**
- **Conclusions**



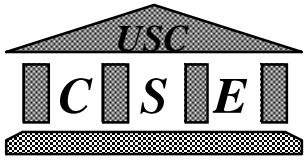
# PRELIMINARY EXPERIMENT RESULTS

- **Experimental Domain:** SEE (Software Engineering Environment) to support SGS (Satellite Ground Station); 21 win conditions were proposed
- **Hypotheses:**
  - H1: More quality conflicts are identified with QARCC than without QARCC
  - H2: QARCC will not falsely identify the quality conflicts

- **Experimental Results:**

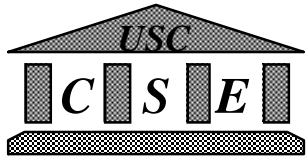
	Not Found by QARCC	Found, Significant	Found, Insignificant
<b>Experiment 1: WinWin user exercise WITHOUT QARCC</b>		2	0
<b>Experiment 2: WinWin user exercise WITH QARCC</b>	0	2 + 5	3

- **Assessment:**
  - H1 was supported: QARCC can help stakeholders to identify conflicts among software quality requirements
  - H2 was not supported: QARCC needs further refinement to avoid overloading users with insignificant quality conflict suggestions.



# CSCI577A DATA ANALYSIS RESULT I: CONFLICT RESOLUTION PROCESS

- 15 Multimedia Digital Library Systems (in CSCI577a Class during 1996-97) [Egyed-Boehm, 1997] -- Real Projects with USC library staffs
- Example: CNTV Moving Image Archives (School of Cinema/TV)
- Observations:
  - Most quality win conditions were noncontroversial
  - Most quality issues were straightforward to resolve
  - Most quality-conflict issues were resolved without comments
  - Most quality-conflict options were adapted without comments
- Negotiation Pattern Analysis Results:
  - Among the quality-conflict issues having only one option, the number of the issues resolved by ***directly accepting*** options were greater than the number of them resolved by ***adapting*** options.
  - Among the quality-conflict issues having multiple options, the number of the issues resolved by ***electing the best*** option(s) were greater than the number of them resolved by ***merging or accepting all*** options.



# Tables for Process Analysis

Table 1. Analysis Result of the Analysis of the WinWin Artifacts

Condition	Number of Artifacts
Win Conditions involved in issues	232/513 (45%)
<b>Quality</b> Win Conditions involved in issues	118/240 (49%)
The Number of issues having <b>NO</b> option	4/176 (2 %)
The Number of issues having <b>ONE</b> option	117/176 (66 %)
The Number of issues having <b>MULTIPLE</b> options	55/176 (31 %)
The Number of <b>Quality</b> issues having <b>NO</b> option	4/102 (4 %)
The Number of <b>Quality</b> issues having <b>ONE</b> option	67/102 (66 %)
The Number of <b>Quality</b> issues having <b>MULTIPLE</b> options	31/102 (30 %)
The Number of <b>Quality-Conflict</b> issues having <b>NO</b> option	3/61 (5 %)
The Number of <b>Quality-Conflict</b> issues having <b>ONE</b> option	39/61 (64 %)
The Number of <b>Quality-Conflict</b> issues having <b>MULTIPLE</b> options	19/61 (31 %)

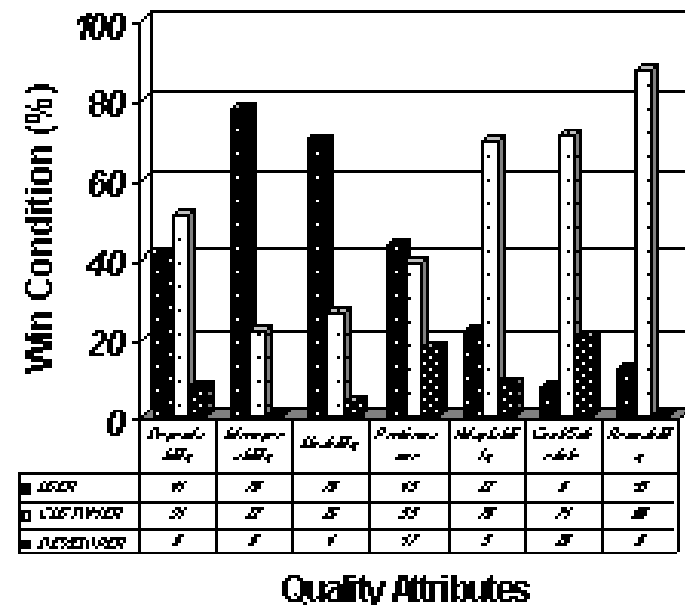
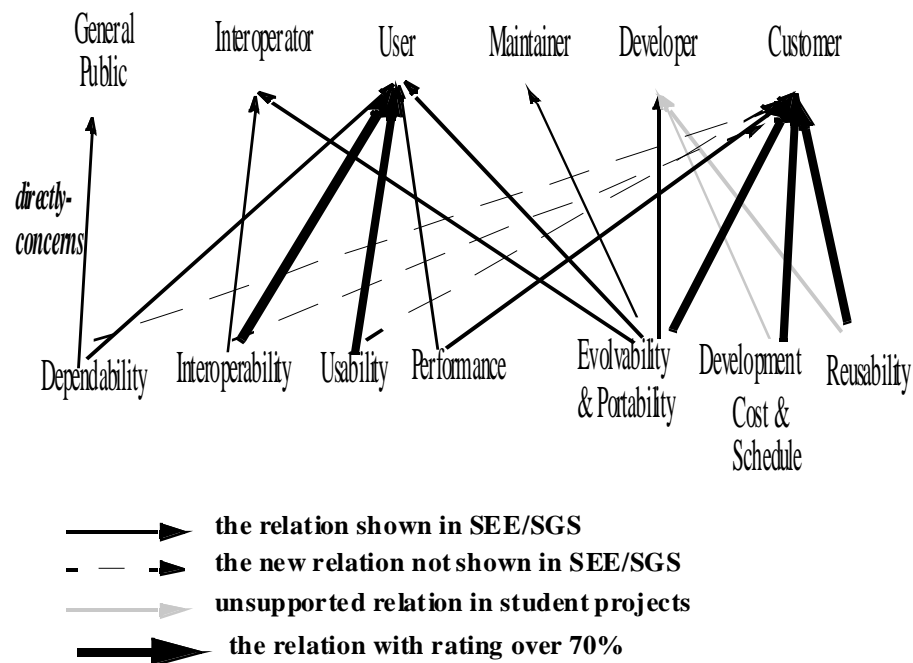
Negotiation Patterns for Quality-Conflict Issues Having One option	Number of Artifacts
<b>Accepting</b> an option <b>directly</b> to an agreement without changing the option semantically	23 / 39 (59 %)
<b>Adapting</b> an option to an agreement with changing the option semantically	14 / 39 (36 %)
<b>Decomposing</b> an option to multiple agreements	1 / 39 (3 %)
<b>Accepting</b> an option <b>contradictorily</b> to an agreement	1/ 39 (3 %)

Negotiation Patterns for Quality-Conflict Issues Having Multiple Options	Number of Artifacts
<b>Electing</b> the best option(s) among the proposed multiple options	8 / 19 (42 %)
<b>Merging</b> the multiple options into an agreement	5 / 19 (26 %)
<b>Accepting all</b> the multiple options into multiple agreements	6/ 19 (32 %)

Artifact Type	Total	With Comments	Without Comments
<b>All</b> issues	176	36 (20 %)	140 (80 %)
<b>Quality-Conflict</b> issues	61	10 (16 %)	51 (84 %)
<b>All</b> options	250	26 (10 %)	224 (90 %)
<b>Quality-Conflict</b> options	74	2(3%)	72 (97 %)

# CSCI577a Data Analysis Result II: Stakeholder-Attribute Relationship

- Comparison of the Relationship between Stakeholders' Roles and Quality Attributes (CSCI577a Class Project vs. SEE/SGS)



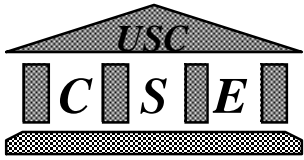




# CSCI577A DATA ANALYSIS RESULT III

## Frequency Analysis of Quality Attribute Strategies

	Cost-Resolution Option Strategy	Strategy Frequency in all 15 teams	Total
Identified Strategies by S-COST	Reduce/Defer functionality	22	30 (97 %)
	Reduce/Defer quality	1	
	Improve tools, techniques, and platform	2	
	Reuse software assets	3	
	Increase budget	1	
	Composition	1	
New Strategies (Not Identified by S-COST)	Buy Information	1	1 (3 %)
<b>Total</b>		31	31 (100 %)



# Conclusions

- **Stakeholder win-win approach provides**
  - Tailorable definition of software quality
  - Procedures for negotiating quality attribute tradeoffs
  - Process (WinWin Spiral Model) for specification, development, and risk-management of quality software products
- **WinWin groupware system provides**
  - Collaborative support for stakeholder win-win approach
  - Knowledge-based tools for software quality attribute conflict resolution

## **Software Project Survival:**

A Tale of Two Projects

Steve McConnell, Constux Software

Many people in the software industry are thrust into positions of responsibility for software project outcomes, but few are given training in how to make them succeed. In this presentation, Steve McConnell describes two projects with very different outcomes, and analyzes the root causes of software success and failure. McConnell draws from his direct experiences on shrinkwrap software projects and from his project consulting business. This talk is based on his best selling book, Software Project Survival Guide.

Steve McConnell is Chief Software Engineer at Construx Software where he divides his time between leading custom software projects, consulting on other companies' software projects, and writing books and articles. He is the author of Code Complete (1993), Rapid Development (1996), and Software Project Survival Guide (1998). His first two books won Software Development magazine's Jolt Excellence award for outstanding software development books of their respective years. McConnell has also written numerous technical articles and volunteers as Editor in Chief of "IEEE Software" magazine.

# *Software Survival Guide: A Tale of Two Projects*

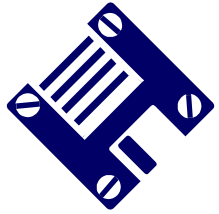
Steve McConnell

[stevemcc@construx.com](mailto:stevemcc@construx.com)

[www.construx.com/stevemcc/](http://www.construx.com/stevemcc/)

© 1998-1999 Steven C. McConnell.  
All Rights Reserved.

**Construx**  
SOFTWARE

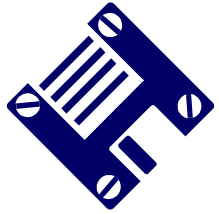


# *Project #1: Giga-Corp's Graph-It Project\**

- ❖ Bid:
  - ◆ \$400K
  - ◆ 5 months
  - ◆ 100% of required functionality
- ❖ Result:
  - ◆ \$1.5M
  - ◆ 14 months
  - ◆ ≈25% of required functionality
- ❖ Effective Cost Overrun: 1400%
- ❖ Effective Schedule Overrun: 1000%

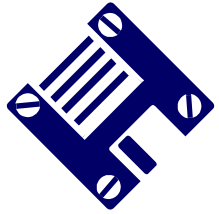


*\* Names have been changed*



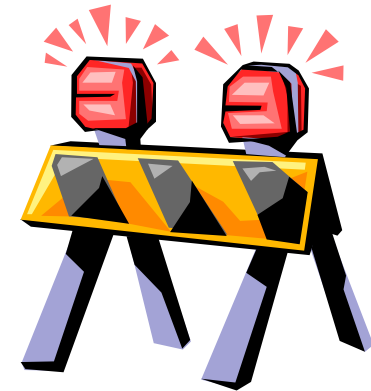
## *Why Did This Project Fail?*

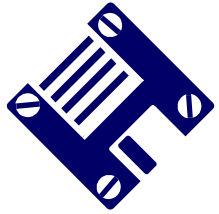
- ❖ Over-optimistic schedule
- ❖ Inexperienced development team
- ❖ Commitment-based planning
- ❖ No meaningful project tracking
- ❖ Inattentive management on the client side
- ❖ Better question: Why did anyone think this project would succeed?



# *Why Do We Need Software Project Survival Skills?*

- ❖ Schedule overruns
- ❖ Budget overruns
- ❖ Quality underruns
- ❖ Functionality underruns
- ❖ Canceled projects
- ❖ 90/90 problem:  
(lack of control and visibility throughout)

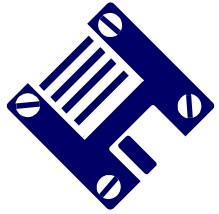




## *Projects Without Good Survival Skills® “Thrashing”*

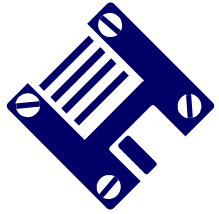
- ❖ Requirements expand 25-50%, often unnoticed
- ❖ Defects corrected late in the project, at great cost
- ❖ Software redesigned and rewritten during testing
- ❖ Known defects go uncorrected
- ❖ Integration nightmares
- ❖ Developers lose work without source code control
- ❖ Ship-mode thrashing: daily reestimation, bug triage, expectation management, etc.





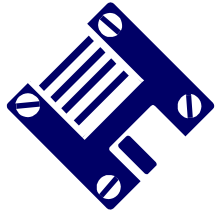
# *Causes of Software Failure: Standish Group CHAOS Report*

1. Incomplete Requirements (13%)
2. Lack of User Involvement (12%)
3. Lack of Resources (11%)
4. Unrealistic Expectations (10%)
5. Lack of Executive Support (9%)
6. Changing Requirements (9%)
7. Lack of Planning (8%)
8. Didn't Need Software Any Longer (8%)
9. Lack of Technical Management (6%)
10. Technology Illiteracy (4%)



## *Causes of Software Failure: KPMG Report*

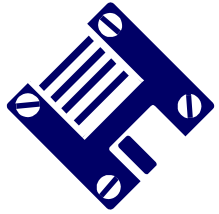
- ❖ Project Objectives Not Fully Specified (51%)
- ❖ Bad Planning and Estimating (48%)
- ❖ Technology New to the Organization (45%)
- ❖ Inadequate/No Project Management Methodology (42%)
- ❖ Insufficient Senior Staff on the Team (42%)
- ❖ Poor Performance by Suppliers of Hardware/Software (42%)



## *Future Plans to Prevent Failure (From KPMG Report)*

- ❖ Improved project management (86%)
- ❖ Feasibility study (84%)
- ❖ More user involvement (68%)
- ❖ More external advice (56%)
- ❖ None of the above (4%)

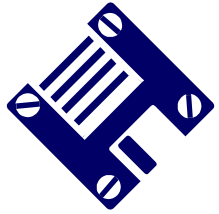




## *Project #2: ATAMS Project*

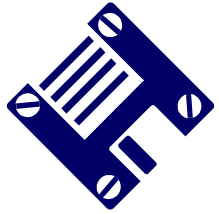
- ❖ Best Estimate:
  - ◆ \$10 million
  - ◆ 2 years
  - ◆ 100% of required functionality
- ❖ Result:
  - ◆ \$2 million
  - ◆ 1 year
  - ◆ “10 times as many functions as originally specified”
- ❖ *Key to Success*: “Techniques that were shown years ago to produce better software faster yet are still rarely used.”





## *Key Elements of ATAMS Success*

- ❖ Intact development team
- ❖ Active user involvement; UI prototyping
- ❖ Reuse of components wherever possible
- ❖ Extensive design reviews, code reviews, defect tracking
- ❖ “Native” risk management
- ❖ Active management to keep project team on track

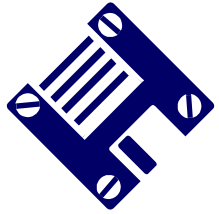


## *What are the General Keys to Success?*

Q: What are the most exciting/promising software engineering ideas or techniques on the horizon?

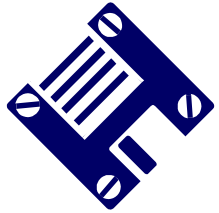
A: I don't think that the most promising ideas are on the horizon. They are already here and have been here for years but are not being used properly.

— David L. Parnas



## *What is Parnas Talking About?*

- ❖ Project planning and management practices
  - ◆ Automated estimation tools (1973)
  - ◆ Evolutionary delivery (1988)
  - ◆ Measurement (1977)
  - ◆ Productivity environments (1984)
  - ◆ Risk management planning (1981)
- ❖ Requirements engineering practices
  - ◆ Change board (1979)
  - ◆ Throwaway user interface prototyping (1975)
  - ◆ JAD sessions (1985)
  - ◆ Requirements scrubbing (1989)



## *The Point*

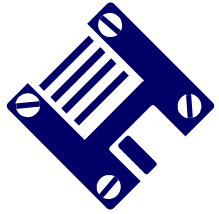
- ❖ Ineffective organizations often fail, even on low-risk projects
- ❖ Effective organizations often succeed, even on high-risk projects
- ❖ The fancy stuff is not what makes the difference--proven practices
- ❖ **Success = Planning \* Execution**

It depends on how carefully projects are planned and how deliberately they are executed--and that's just about the whole story!



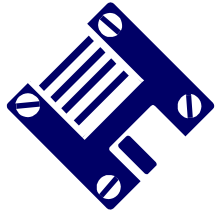


# *Survival Resources*



## *Project Survival Resources*

- ❖ Survival Guide Website:  
*<http://www.construx.com/survivalguide/>*
- ❖ NASA SEL Website:  
*<http://sel.gsfc.nasa.gov/>*
- ❖ IEEE Software Magazine  
*<http://www.computer.org/software/>*



## *Contact Information*

*Steve McConnell*

*stevemcc@construx.com*

*www.construx.com/stevemcc/*

*(425) 746-8390*

*Survival Guide Website*

*www.construx.com/survivalguide/*

**Construx**

S O F T W A R E

- ❖ Custom Software Development
- ❖ Best Practice Seminars
- ❖ Project Consulting

*info@construx.com*

*www.construx.com*

## "25 Things My Mother Never Told Me About Being An SQA Manager"

Mark Carver  
Connected Products Division  
Product Engineering Department (MS JF2-11)  
Intel Corporation  
2111 NE 25<sup>th</sup> Avenue  
Hillsboro, OR 97124-5961  
[Mark.Carver@Intel.Com](mailto:Mark.Carver@Intel.Com)

**Abstract:** Software Testing and Software Quality Assurance (SQA) are critical to the success of any company that develops or uses software. It is often one of the most mysterious, least understood, and challenging of all Software Engineering disciplines.

One of the most valuable individual processes for a test and SQA person is introspection - self-analysis and review. For an SQA or test lead, it is assessment of their team's culture, strengths, and weaknesses. This paper will provide the reader with 25 experience-based observations about success factors and traps that may apply to their SQA and test teams or members. Some, all, or none of the lessons or opinions in this paper may apply to a reader's particular circumstance. The key is to take the time to think about it. This paper will also identify inappropriate attitudes and behaviors as well as appropriate and successful ones. It is not intended to provide a convenient labeling mechanism to classify, humiliate, or isolate individuals. It is hoped that inappropriate attitudes and behaviors will be more easily identified for improving the situation and positive ones will be reinforced.

**Biography:** Mark Carver is a Product Engineering Manager for Intel Corporation responsible for the validation and delivery of the Intel® Create and Share™ Camera Pack and Intel® Video Phone in OEM and Retail Channels as well as other products from the Connected Products Division. He has over 15 years of experience in Software Quality Assurance, testing, and management in commercial and defense software. He has a BS/Computer Science from New Mexico State University and an MS/Systems Management from the Florida Institute of Technology.

# **"25 Things My Mother Never Told Me About Being An SQA Manager"**

## **Introduction**

Software Testing and Software Quality Assurance (SQA) are critical to the success of any company that develops or uses software. It is often one of the most mysterious, least understood, and challenging of all Software Engineering disciplines.

One of the most valuable individual processes for a test and SQA person is introspection - self-analysis and review. For an SQA or test lead, it is assessment of their team's culture, strengths, and weaknesses. The goal of this paper is to provide the reader with 25 experience-based observations about success factors and traps that may apply to SQA and test teams and members. Recognizing the presence of an unnoticed problem area is the first step to resolving or mitigating it. Recognizing the presence of unnoticed areas of success can be beneficial in magnifying and leveraging that success. A non-goal is to preach a particular dogma. Some, all, or none of the lessons or opinions in this paper may apply to a reader's particular circumstance. The key is to take the time to think about it. Another purpose of this paper is to identify inappropriate attitudes and behaviors as well as appropriate and successful ones. It is not intended to provide a convenient labeling mechanism to classify, humiliate, or isolate individuals. It is hoped that inappropriate attitudes and behaviors will be more easily identified for improving the situation and positive ones will be reinforced.

The primary source of these observations and experiences are from software development organizations. While centrality in a software organization is obvious, it is believed that these observations are beneficial to other forms of Test and SQA groups.

## **A Baseline Opinion: Testing Is Not Synonymous With Quality Assurance**

The titles "tester" and "software quality assurance" or "SQA" are frequently used interchangeably. In my experiences, this has created some unintended side-effects. To some this is a semantic issue. I believe quality assurance is a process that encompasses the entire life-cycle of a product or project and involves all members of the product or project team. Testing is a component or life-cycle phase, just as is design or requirements development. A tester is a specialist for this phase, just as a designer is for the design phase. I believe testing tends to show the absence of quality rather than its presence. The more subtle issue is that I've frequently observed that those titled as "SQA" are expected to ensure quality in the product, while not having the appropriate levels of influence, accessibility, and involvement in the project to do so.

## **The Lessons**

### **Culture and Environment**

**Know Your Company.** Every company has its own culture and attitudes about testing and quality assurance. It is important to understand this culture and these attitudes in order to not overestimate or underestimate the commitment to quality and associated support levels. This knowledge is key to a successful career in a particular company.

Know Your Team. Every team has its own strengths and weaknesses, personalities, legacy, and working environments. In some cases, the team is an integral part of the development process. In some cases, it is the handmaiden of the development team. In some cases, it is independent. The individuals who comprise the team are key to that team's success. The key is to know the people of the team, the working environment and culture and leverage them. For example, a Cowboy is often good for ad hoc testing but may not be for routine testing. A methodical person is often good for test planning or metrics generation but perhaps not for ad hoc testing. A detail-oriented person is often good for root-cause analysis.

### **Attitudes and Traps**

#### Your Job Is Not To Kill The Product...Usually.

*Characteristic:* In almost every team I've been associated with, there has been one individual that believed strongly that shipping with any bugs represents an unethical act by the corporation. Given the inevitability and even practicality of shipping with bugs, they often are disruptive to the team. However, there are times when a product should not be released. In this case, the team needs to present the situation in clear engineering terms rather than emotional and impassioned pleas.

*Approaches to change:* One approach to turn around this situation is to involve the team in an attitude adjustment where their job is to make sure that the best possible product is shipped, given the realities and circumstances of the project. By focusing on what problems really matter and which don't, they can help the entire project do the best possible effort.

#### The 'Chicken Little' Complex.

*Characteristic:* Some people appear to be in a constant state of crisis and every problem is a big problem. This may be limited to their work-life, or may be their way of life. If the person is perceived as being unable to separate what matters from what doesn't, and many, most, or all issues or problems found are presented or elevated as major or critical, with associated emotion or passion, their inputs will become increasingly discounted. They also run the risk of having the Bozo Bit flipped on them.

*Approaches to change:* The most common approach is to insulate them from the organization, and have their sponsor present the information for them after having duly filtered it in the context of other priorities and information. This tends to be time consuming for the sponsor and limits the growth of the individual with this problem. A different approach with longer term benefits is coaching and mentoring the individual to help them understand risks, trade-offs, situational assessment, and issue escalation and presentation in the context of project priorities and time-to-market.

#### The Arsonist.

*Characteristic:* There are some individuals who enjoy seeing other people, projects, or groups in conflict. They seem to find ways to trigger conflict and then step back and watch the fireworks with some sense of accomplishment or

satisfaction. This can occur in public forums/meetings or even individually, setting one person against another. This can be one of the most destructive behaviors in a team.

*Approaches to change:* This behavior has to be directly addressed with the individual and should include some ownership of resolving the conflict they are triggering. Peer pressure tactics generally have limited success because this person usually claims innocence.

#### Dilbert As A Role Model.

*Characteristic:* The Dilbert comic strip has brought humor to many daily situations and are commonly seen in offices and cubicles. 'Dilbert must work here' has been overheard in many companies. The subtleties of the environment of the comic series is one of incompetence, destructive behavior, selfishness, etc. on many levels. People who internalize any or all of these attitudes run the risk of becoming disenchanted with their surrounding organization.

*Approaches to change:* One approach is to directly discuss with the individual concerns and effects that their attitude has on themselves and those around them.

#### The 'Piranha Effect' or 'Frog-in-a-Pot'.

*Characteristic:* An individual task estimate may be challenged to bring schedules 'in.' A small new task may be added in the middle of a project which, in and of itself, is minor in accomplishing that task. A small feature may be added to a project late in the development process. Any of these, under different circumstances, could be taken in stride. However, in the aggregate of the situation they are overwhelming and increase risk to a project and/or morale. An individual piranha can inflict pain or injury, but in a pack can destroy their victim. A frog can be placed in a pot of water and will not detect a gradual increase to boiling or try to escape because the temperature change is so subtle. If you place a frog in hot water, it will do everything it can to escape. NOTE: Do not try this with a frog. It is a cruel thing to do.

*Approaches to change:* Additional tasks cannot be viewed in the narrow context of individual tasks, but must be assessed in the larger picture of all tasks. Once detected, it is reasonable to request a reassessment and reset on schedule or priorities.

#### The Bozo Bit.

*Characteristic:* There are many situations when an individual (or group) can lose some or all of the trust, confidence, or respect of other groups or individuals. When a negative label is applied to an individual under these circumstances, it is difficult to change the perceptions associated with that label. This label can persist and severely undermine an individual's or group's efforts. It can be severe enough to undermine or ruin careers.

*Approaches to change:* One effective approach has two elements: 1) sharing ownership of the problem with those who have applied the label by appealing to

them to assist in overcoming the perception (e.g. 'OK, we've identified a problem here, help me to overcome it'), and 2) to increase the diligence of the person or group and take a 'judge me by my work' approach. Management support and sponsorship is particularly important for overcoming this perception and they should be asked to take an active role in a 'get well' plan.

#### The 'Lightning Rod Effect'.

*Characteristic:* There are many reasons why a SQA or test group becomes the focus of unsolicited negative attention, whether deserved or not. Reasons include missing a major problem, inability to keep schedule, failure to keep other groups informed to the level they expect, etc. This negative attention tends to be increasing in nature and usually forces increased reporting, clarification, and explanation of activities and results. It also tends to draw increasing 'blame' on the group for the original issue plus 'every other thing that comes along.' If not quickly addressed, it will probably lead to flipping the Bozo Bit on the group.

*Approaches to change:* Actively seeking to deter or redirect attention is not generally effective. The need is to regain credibility and trust. One approach that may be effective is directly acknowledging the situation that caused the attention and overtly state steps that will be put in place to ensure the situation does not recur. This should be followed by updates on progress. Another approach is to involve the community that views this as a problem in the ownership of resolving it rather than allowing them to remain as snipers. This can also lead to creative solutions not considered by the individual group.

#### The Borg for Testers.

*Characteristic:* Testing can be considered to be the search for mistakes, errors, and faults. This is an important dimension of testing – to challenge the product or project and identify flaws. Some testers allow this 'negative testing' mentality to become part of their personality. While some of the most pessimistic testers I've known have been the most effective at testing, it takes its toll on the team and team interactions.

*Approaches to change:* If you identify a tester who is becoming increasingly cynical and pessimistic, one effective approach is to discuss this concern with them directly. Additionally, it may be useful to place them in a position that focuses on the positive aspects of the product and its delivery.

#### 'Your SQA Organization is Responsible for Quality'. (Make Sure There Has Not Been A Failure in the Empowerment Subsystem)

*Characteristic:* The quality of a product or project should be owned by all project team members. It is very difficult, and some would say impossible, for a particular group to ensure quality. For those who believe that this is possible, a great deal of empowerment of the SQA organization that permeates the entire project is necessary. It cannot be a 'tail-end' or intermittent effort in a life-cycle. This is one of the risks of being identified as a 'quality assurance' organization, unless the necessary empowerment is present. Empowerment, in this context, means not only the responsibility to accomplish something, but also the authority



and management support to ensure accomplishment. For example, if it had been determined that a requirements document is necessary, and management would not help enforce this necessity, and you were told to 'collect the e-mails,' 'interview the developers', or 'you should be able to figure out what it's supposed to do,' you're in an unempowered or powerless position.

*Approaches to change:* If a team has responsibility without authority or ability, then the missing component needs to be corrected. Ability can mean skills or resources (people or tools). Usually, correction of the circumstances requires working with the sponsor, and likely with the sponsor's sponsor.

### Technobabble.

*Characteristic:* Collecting information on status, issues, metrics, etc. is critical to any project. Just as important as the collection of data and information (information is processed data) is its presentation. Often a statement or assertion is presented with a detailed defense to head off any challenges to the statement or assertion. Many important messages get 'lost in the data' or 'lost in the details'. Presentations should: 1) state the problem, issue, status, etc. (e.g. 'We are on track for shipment'), 2) what this statement means (e.g. '... so we can proceed with the press releases'), and 3) caveats (e.g. '... the one issue that may impact the schedule is...'). Presentation of information can be likened to making a sale. Successful salesman always answer the question 'what's in it for me' or 'so what' before it's asked.

*Approaches to change:* Leave the details in the 'Backup Information' section of a presentation. Supporting data is always important. If an individual is presenting information effectively, this should be reinforced - even if they're from another team. Their presentation can also be used as a positive example. If someone is giving poor presentations, this should be addressed through coaching, training, or mentoring.

### Technology – Is It Real Or Memorex?

*Characteristic:* A typical problem is exaggeration of the benefits and value of new technology and tools. This is sometimes referred to as a 'silver bullet'. To effectively benefit from new technology, investment well beyond just the acquisition of the technology or tool is required. Just 'reading the manual' seldom works except for the most simple of technologies. Overestimation of the benefits in introducing new technology, underestimating the steps needed to adopt the new technology, and failure to gain adequate support can lead to the failure in adoption of new technology into an organization. This is a common problem with 'grass root efforts' that are not sufficiently sponsored or funded.

*Approaches to change:* Four elements are usually required for successful adoption of new technology: 1) sufficient funds or time allocated for training, 2) acceptance by key players effected by the new technology, 3) sufficient funds for the maintenance of the technology, and 4) management sponsorship and support. Each one of these should be individually addressed and risk and contingency planning of each identified early in the decision process. If any of

these areas are high risk or unlikely, they should be addressed. In most organizations, canceling an effort that is unlikely to succeed is preferred over leading an effort that was 'doomed from the start.'

### If You're Not Going To Use It, Don't Measure It

*Characteristic:* People are usually uncomfortable with being measured, evaluated, monitored, or observed. Most data collected can be traced to the actions of an individual. It is not uncommon to see extraneous data collected, 'just in case.' If data is not going to be used, it should not be collected for two reasons: 1) the people-side of collecting data, and 2) the wasted efforts associated with the unnecessary data collection.

*Approaches to change:* The need, necessity, and purpose of data collected for metrics and others purposes needs to be periodically evaluated and collection efforts stopped where it is not needed. If the data is needed, in its transformation to information it should avoid being traceable back to an individual if at all possible.

### **Factors of Success**

Constructive Confrontation. To be successful, constructively confronting issues and problems is critical. Issues may include the need to reset schedules, a performance problem within the team or one that adversely impacts the team, etc. Constructive confrontation strikes fear into the hearts of many. However, effective constructive confrontation is not 'taking someone on', 'attacking' someone, a power struggle, or hostile. Intel® uses a Constructive Confrontation Problem Solving Technique that has proven to be effective and is a routine component of the culture. It addresses the problem in the following manner:

- Direct
  - ❖ Confront an issue with the appropriate person(s) - the person(s) who can own resolving it
  - ❖ Clearly define the problem and how it affects the work
- Objective
  - ❖ Base discussion on data, facts, observations or experiences
  - ❖ Seek the best solution for the situation and the company
- Positive
  - ❖ Confront the issue; **don't personally attack**
  - ❖ Direct any emotion towards the issue and its resolution
  - ❖ Work to resolve the issue while maintaining the working relationship
- Timely
  - ❖ Confront issues as they arise in meetings or when they are seen

- ❖ Issues should be confronted before they are overlooked, greatly hinder work situations, or negatively impacts corporate business.

Know your Sponsor. To be successful, every individual associated with quality assurance and testing has a sponsor – someone who will support them, back them up, and work through problems and issues with them. It is important to identify this sponsor and their support and sponsorship levels and encourage continuing or increased support. It may be that a sponsor may not even be aware that they are or need to be a sponsor.

Find a Way - Avoid the Victim Trap and Create 'No Whine' Zones. It is easy to fall into an attitude of being overwhelmed that results in routinely rejecting requests for additional support or investigations from development teams, marketing, etc. It should not be acceptable to say 'No', but acceptable to say 'When'. In many situations it is also useful to involve the party requesting the additional support in adjusting priorities and shifting tasks to accommodate the changes.

Be an Asset to the Development Team. Being in a quality assurance or test team provides some special insights into the product that can be lost by a development team because of their tendency for vertical involvement (focus on the area that they are responsible for). By being an advocate for the end-user and focusing on a holistic view of the product, quality assurance and test teams can provide valuable insights to the development team in terms of focus of what's 'really important' and identifying those areas that are 'good enough' in the context of project priorities and time-to-market.

Avoid Hypocrisy - Practice What You Preach. If your team expects something of another team, be sure that your team has the same expectations of itself. For example, if inspections are expected of the development team, be sure and inspect your own work products.

Things WILL Change - With or Without You. Things around the team will change with frequency for the better or worse either in process or practice based on normal team and corporate dynamics. Organizations are always having to adjust to budgetary dynamics (growth or contraction), schedule and market influences, new technology or adapting to competition, new leadership, changes in team membership, etc. Developing a personal work ethic of being proactive and an asset in stabilizing or guiding the team during change will make one an invaluable resource. This will lead to opportunities in planning and executing change, and personal influence in determining the change path.

Know Yourself. Every individual has strengths and weaknesses. Some are more perceptive than others. Some are better sleuths. Some are better writers. Some are better researchers. Some are better presenters. Some are better in confrontational situations. It is important to periodically assess and inventory personal skills and interests and use those that are strengths to the advantage of the team. It is also important to assess weaknesses and determine which need to be converted to strengths. Converting weaknesses to strengths can be accomplished through personal determination, training, coaching, and seeking progressive opportunities. For example, if a weakness is presenting in front of a group, a combination of training and seeking opportunities to present in front of groups can overcome this weakness.

Ownership – Ownership – Ownership. When a task or activity is accepted by an individual, they should follow it to closure or until ownership is transferred. Ownership is not transferred until the other person knows they now own the problem and agrees to complete it. Quality assurance and test teams lose credibility and support when issues ‘fall through the cracks’ because of lack of ownership or effective transference of ownership.

Most People Mean Well. It is easy to fall into the trap of believing that some other individual, team, group, or organization doesn’t like, respect, or support a quality assurance program. Every statement or action reinforces that belief, whether they were intended to or not. It is important to give other people the benefit of the doubt that they are well meaning and well intentioned. If there is a belief that someone is ill-meaning, constructive confrontation often dismisses the problem, mitigates it, or clarifies someone’s actions or intents.

Have A Vision. It is important to have a vision or plan for doing better next month than this month; better next year than this year. This is important for the morale of the team and critical to not sliding into and staying in a rut that is driven by focusing on tactical and day-to-day activities.

### **Final Comment**

At Some Point, It’s Only A Job. It is sometimes or often necessary to commit additional time to quality assurance and testing careers. Evening or weekend work is not unusual, particularly near the conclusion of projects. If projects are not properly managed, this extra effort can become permanent and have adverse effects on personal lives. It is important to remember that at some point, it’s only a job and not life.

### **Conclusion**

Software Quality Assurance and Testing are key to any organization that authors or uses software. It can be a very rewarding career field that is part art and part engineering. Understanding the work environment, the players, and ourselves will greatly increase our value to our organizations and resultant career satisfaction.



## **The Seven Habits of Highly Effective Inspection Teams**

Stephen K. Allott

Imago<sup>QA</sup> Limited  
4<sup>th</sup> Floor West  
High Holborn House  
52/54 High Holborn  
London  
WC1V 6RL

Tel: +44 (0) 20 7242 5100

Fax: +44 (0) 20 7421 8100

[sallott@imagoqa.co.uk](mailto:sallott@imagoqa.co.uk)

[www.imagoqa.com](http://www.imagoqa.com)

**Pacific North West Software Quality Conference**

**11<sup>th</sup> – 13<sup>th</sup> October 1999**

**Portland Convention Centre, Portland, Oregon**

## **Abstract**

We are what we repeatedly do.  
Excellence, then, is not an act, but a habit.  
Aristotle.

The Big One, in Blackpool, England is the biggest roller coaster in the UK. In our experience, implementing the Inspection process is like a riding a roller coaster. You start slowly, eventually reach a peak and feel on top of the world – for a while. Suddenly you're plunging downwards and almost reach the ground - a little boost of energy is required to send you up again.

This paper describes how we harnessed the energy of our Seven Habits to re-launch the Inspection process at ISS.

We had used an Inspection process very successfully since 1990 when we were less than a dozen people. However, as the company grew and more Inspections were run, it soon became evident that some improvements and re-training was required.

We knew that it might be a challenging job. It was certainly a big one.

We formed a Technical Working Group that was comprised of 6 cross-departmental representatives and trained them as Inspection Leaders. This group had the responsibility to lead Inspections as well as improving our Inspection process. We listened to everyone, re-designed educational material and used the Seven Habits of Highly Effective Inspection Teams to help get the message across. During the re-launch our focus switched from improving the process to concentrating on the excellence of our people. We argue that it is important to emphasise excellence throughout the Inspection process if it really is to become highly effective.

**Steve Allott, May 1998**

## Biography

### **Stephen K. Allott BSc.(Hons) MBCS**

Steve has over 20 years experience in IT at major organisations including ITT, ADP, The First National Bank of Chicago and Lloyds Bank. His background includes software development, project management and technical support.

He has many years experience in testing and is the programme secretary for the British Computer Society's (BCS) Specialist Interest Group in Software Testing (SIGIST). He is also a member of the Software Testing Certification Board administered by ISEB (Information Systems Examination Board).

Until recently Steve was the Test Manager at Integrated Sales Systems UK Limited (ISS). As well as managing the Testing Services team and improving the company's test process, Steve also took on the role of Inspection co-ordinator. Steve is a trained Inspection Leader and has 6 months experience in leading Inspections and improving the Inspection process at ISS. He has given several one day (internal) training courses on the basic concepts of Inspection.

Steve joined Imago<sup>QA</sup> in September 1998 as a Senior Training Consultant.

## **1. Introduction**

### ***1.1 Company Background***

Based in Twickenham, England, Integrated Sales Systems UK Limited (ISS) is a small to medium sized software company that builds and maintains large scale sales & marketing automation systems. The annual turnover is approaching £5 million and they were recently rated (by Ovum) as one of the top 10 suppliers in the world of CIS (customer interaction systems) software.

### ***1.2 A brief history of Inspection at ISS***

Since the company's inception, Inspection has been a part of the corporate culture. We were quite lucky in enjoying support at the Board level. Andrew Myers, the Technical Director, wrote the original Inspection process and was our first certified Inspection Leader. Our original process was very successful in finding important major and super major defects that would otherwise have seriously impacted the quality of the products.

We always emphasised the importance of planning Inspections and allowing sufficient time in project schedules to do them properly. The process was gradually improved by inviting authors to logging meetings.

## **2. Problems with Inspection at ISS**

### ***2.1 The Company's View***

A few key people had left, and there was a feeling that new starters had not had sufficient training in the process. Defects were found during the testing phase that management felt should have been picked up during Inspection. There was a general consensus that Inspection just "isn't as good as it used to be". Many of the staff felt that Inspections were of benefit but that they took too long.

The collection of Inspections data was still taking place but no analysis had been done. There was a huge backlog of data to review and analyse.

Management continued to support Inspection and knew that it would be of benefit if properly implemented.

### ***2.2 The Consultant's View***

In order to solve some of the problems we invited Dorothy Graham, Grove Consultants, to take a look at our Inspection process and offer some advice as to the best way forward. We summarised the key issues from her report:

- we did not know the effectiveness of our process
- we had too much discussion in logging meetings
- there were too many subjective comments
- rules, standards & checklists were out of date
- analysis of metrics hadn't been kept up
- we didn't make the benefits visible
- checking was limited to 2 hours
- there was no follow up or closure of the process



- we had more focus on code than upstream documents

### **2.3 What had to be done?**

It was clear to all of us that there was a great deal to be done both in the short term and the long term and there would be no ‘quick fixes’ to the problems. We needed to invest in an Inspection infrastructure, re-launch the process throughout the company, and analyse the metrics to demonstrate the effectiveness of our Inspection process.

## **3. Plans to re-launch Inspections @ ISS**

### **3.1 Formation of a Technical Working Group**

ISS Management during September 1997 considered Dorothy Graham’s report and recommendations. We recognised that although the process needed improvement, it did not require re-engineering. We also knew that modifying the process was not enough – we needed to modify behaviour as well.

Therefore we decided to set up a Technical Working Group [McFeeley96] to re-launch our Inspection process. This cross-functional group was made up of 6 representatives from both management and staff and was chaired by me, Steve Allott, the Inspection Co-Ordinator at ISS.

The group’s objectives were:

- To agree how best to implement Dorothy’s recommendations
- To ensure that everyone in the company received some training in Inspection
- To design new forms
- To update the processes / procedures
- To contribute new rules, checklists
- To monitor the Inspection process

### **3.2 Train a core group of Leaders**

The Technical Working Group soon discovered that improving the Inspection process was going to be more difficult than we initially thought. As members of the TWG we needed to truly understand the Inspection process – training was required. Dorothy Graham customised an Inspection Leader’s training course and delivered it at our offices in Twickenham. We utilised ISS documentation throughout the training to make it more relevant.

At the end of the course the group, chaired by Dorothy, conducted a process improvement session. From this session several suggestions for short-term (less than 3 months) improvements were identified. These gradually distilled into the Seven Habits that are now used to market the process internally.

### **3.3 Collect and analyse metrics**

One of our improvement suggestions was to utilise the Inspection data that we were collecting to produce relevant metrics. Therefore we designed a simple Access database to capture just the essential metrics that we thought would be useful. Once we had some experience of analysing this data we could always introduce new metrics later on. We decided to capture:

- Time spent on the Inspection (plan, kick-off, checking, logging, edit, follow up)
- Estimate of time saved
- Average checking rate
- Logging rate
- Number of major defects fixed
- Number of minor defects fixed

### ***3.4 Tracking Inspections***

We needed to make sure that people were using the new process. One way to accomplish this was to improve our tracking of Inspections. To facilitate this we introduced an Inspection Id process. Whenever anyone requested an Inspection they would fill out a simple e-mail form and send it to our Inspections mailbox. The Inspection Co-ordinator would assign a leader, enter the request into the database and assign the Inspection Id.

This process accomplished several aims:

- we got our Inspection id
- we sold Inspections by making it easier for people to find a leader
- we had the facility to chase up data

### ***3.5 Education for everyone***

A key element in our plans was to train everyone in some of the basic concepts of Inspection. This would especially help new employees but would also act as a refresher to existing staff. This was scheduled as a one-day overview course, which we ran every two weeks during January to April 1998.

Key to the success of this training was gathering strong support from the management team. Departmental managers encouraged their staff to attend training and Senior Managers attended the training to set an example. We also managed to persuade our Managing Director, Stuart Penny, to attend one of the courses.

### ***3.6 Internal marketing via the Intranet***

As Inspection was being re-launched at ISS, marketing was an intrinsic piece of our plan. We needed to give people a quick and easy way to access all the information they needed on Inspection without swamping them with paper.

One of our TWG members suggested utilising our company Intranet. We designed our own Inspection page for the Intranet. The page was structured with this goal in mind. Additionally, we structured the page around the process so that it would constantly be re-enforced in people's minds.

Every good marketing campaign deserves a slogan and ours was no exception. It is:

**Inspections @ ISS : Let's Work Together**

## 4. The Seven Habits of Highly Effective Inspection Teams

### 4.1 Introduction

As part of our marketing campaign we decided to borrow an idea from Stephen Covey's excellent book, *The Seven Habits of Highly Effective People* [Covey 89]. We decided to create our own set of Seven Habits, except ours were going to revolve around Inspection. They started off by being a list of process related items and evolved into people related issues that we want to emphasise within our company.

One of the key points that we wanted to stress in our Seven Habits was excellence. In his book, *Software Testing in the Real World*, Ed Kit said that "Critical thinking is required...that must transcend even the most well defined Inspection checklists". The best Inspectors must ask, "what's missing?" or, "what should have been written here that isn't". He concludes, "good inspection team people are worth their weight in gold in organisations" [Kit95].

As you read about our Seven Habits (see appendix 1) note that our 4<sup>th</sup> habit (we need excellent people) supports Ed's views on the composition of the Inspection team. Also, in developing our Seven Habits we found that we needed to add a new rule to our generic rule sets (NCP - non complete, see appendix 3), to enable our 'best Inspectors' to log these missing items.

### 4.2 Summary of our Seven Habits

A summary of our Seven Habits is shown below. Please see appendix 1 for a full description of our Seven Habits.

Habit 1	Request	Put the Inspection above your own work
Habit 2	Planning	Choose checkers carefully
Habit 3	Kick-off	Tell 'em and teach 'em but also set targets
Habit 4	Checking	Effort, endeavour & excellence is required from the checkers
Habit 5	Meeting	Control the discussion to log at the optimum rate
Habit 6	Edit	Trust the author, check the fixes
Habit 7	Exit	Keep the metrics confidential

## 5. The best laid schemes o' mice and men<sup>1</sup> . . .

Once we were into our re-launch of the process we started to encounter resistance from some of the staff. As we examined staff attitudes we realised that we had a few problems. We re-convened the Technical Working Group and examined each of the problems in turn.

### 5.1 *Too much emphasis on process*

By the first week in February we'd run 3 courses and trained about 18 people. There was a problem with code Inspections in that people reported more trivial defects and issues than previously. It was felt that the process had been over emphasised in the course and that people were now just 'going through the motions' in order to complete the Inspection.

The solution was two-fold:

- We could use our Seven Habits to emphasise the 'people issues' and in particular the effort required both in planning the Inspection properly and in individual checking.
- In addition we re-designed the training material to interleave the lectures with the practical elements of the course. The practical Inspection work was based around the company's staff handbook. This meant that the practical work was understandable to everyone in the group and we didn't have to spend time creating separate examples (e.g. code, SQL, functional specification) for different members of the group.

### 5.2 *Too many forms !*

At first we tried to use the master plan and data summary sheet from the book *Software Inspection* by Gilb & Graham [Gilb 93]. In principle the forms are good and have all of the information you would want. However, in practice when people tried to use the new forms they found that they were time consuming and difficult to fill out.

We took the approach of re-designing both forms and came up with a single master plan (see appendix 2) that flowed in the same way as our process, and allowed data to be entered directly into the database.

The new master plan doesn't show entry or exit criteria. We used the superset from the book and assigned a subset that applies to all documents for right now (see appendix 4). If there are any exceptions to these for specific documents we are sure that common sense will prevail.

### 5.3 *Generic rules sets too complex*

When we presented the 15 generic rules for Inspecting documents there was resistance and much discussion on every course. Some of the rules seemed contradictory and there was a fair bit of confusion.

We decided that a simpler approach was needed. The Technical Working Group re-designed the generic rule set (see appendix 3) by using the best of the 15 original rules, combined some together and also introduced two new rules that we felt were important. We also came up with the idea of a three-character mnemonic to 'tag' each rule. This was

---

<sup>1</sup> Robert Burns

easier to remember than a rule number and was also nice because the tag described the violation whereas the rule description specified the standard (e.g. for the rule ‘relevant’, the tag is IRL - irrelevant).

#### ***5.4 Not enough leaders***

We anticipated that this would be a problem but were surprised at how quickly a bottleneck formed. Once the new process got going there was a flood of requests for Inspections and the six of us in the Technical Working Group were quickly overtaxed. We were faced with a dilemma; turn people’s requests down and possibly damage our re-launch or relax the rules.

We chose the less damaging route, we agreed that anyone could lead an Inspection provided they had attended the overview course and had some mentoring from one of the current leaders. We scheduled another Inspection Leaders training course for the end of April for another nine people, all of whom had already attended a one-day overview course.

#### ***5.5 Inflexible application of the rules***

In the early stages some of our new leaders were over zealous in their application of the new rules. Specifically with regards to how much discussion should be allowed in logging meetings.

We solved this problem with one of our Seven Habits - let the leaders control the meeting and determine the level of allowable discussion. So long as you’re still able to log items at the optimum rate you are okay. Leaders are encouraged to move any long discussions to the end of the logging meeting. This was often called ‘the third hour’ and it was intended to rename this period ‘the discussion meeting’.

#### ***5.6 Staff Perceptions***

Although senior management has always supported Inspection at ISS there was a growing perception that perhaps Inspection was not always worth the effort involved.

In order to ‘nip this in the bud’ we invited Andrew Myers, our Technical Director, to join the Technical Working Group. He also volunteered to lead Inspections and use the new process.

### **6. Conclusions - “Let’s Work Together”**

#### ***6.1 Benefits achieved with our re-launch***

ISS has received many benefits from our re-launch of the Inspection process. Inspection has become an important part of the ISS culture again and people are responding favourably to the new process.

More documents than ever are being Inspected and we saw some definite wins from the new process. We detected several major defects early enough to realise true benefits in their correction, we have improved the overall quality of our product and we have encouraged our customers to join in the process as well.

We started to build our Inspection infrastructure and collect metrics from each Inspection so that the process can be monitored in future. By applying the entry criteria we prevented ‘loser Inspections’ from starting. As people have their documents Inspected they learn from their experiences and produce better documents the next time.

## **6.2 The next ride on the roller-coaster**

We realised that there is still much to accomplish and we had to apply constant energy to keep our roller coaster moving.

Here are a few of the issues that will be addressed in the near future:

1. Even with all the benefits, people still don’t enjoy participating in Inspections. We are looking into ways to motivate people but we have no magic solutions just yet. We are looking into prizes and awards for the best defect found. For now we will continue to encourage participation in Inspections through our appraisal and goal setting process.
2. We have style guides (rules) and checklists for some of our documents but still need to develop them for the rest. We do not think that re-engineering our process is necessary [ACM98], but we plan to redesign the issues log and our long-term goal is to have all forms in an electronic format on our Intranet.
3. Produce reports from our Inspections database to prove benefits, produce trend analysis and determine where we need to continue to improve as a company.

## **6.3 Recommendations - Let’s work together**

If you wish to apply the lessons discussed in this paper in your own company then we have five key recommendations:

1. Find a champion (and an Inspection co-ordinator) who is pragmatic and who is prepared to listen to the staff and make changes when things don’t work out at first.
2. Find a sponsor at the Executive management level who believes in your goals and will support your champion.
3. Remember that although process is important, it isn’t everything and you need real effort from real people to make Inspection work. So don’t forget the people issues.
4. Adapt what you read in the books and learn on the courses to your own company’s needs and expectations. Work together with your staff, management and the software testing community to build the right process for your company. Let’s all work together.
5. Know where you want to get to and have a realistic plan for how to get there. Don’t try to solve everything the first time out, recognise that incremental improvement can work.

## 7. References

- ACM 98. Johnson Philip M. Reengineering Inspection. Communications of the ACM. February 1998 / Vol. 41, No.2
- Covey 89. Stephen R.Covey. The Seven Habits of Highly Effective People. Simon&Schuster 1992 ISBN 0-671-71117-2
- Gilb 93. Tom Gilb & Dorothy Graham. Software Inspection, Addison-Wesley 1993 ISBN 0-201-63181-4
- Kit 95. Ed KIT. Software Testing in the Real World (page 73) Addison-Wesley 1995. ISBN 0-201-87756-2
- McFeeley 96. McFeeley, Bob. IDEAL(SM): A User's Guide for Software Process Improvement. CMU/SEI 96-HB-001

## 8. Acknowledgements

I would like to thank Andrew Myers for his continuing executive sponsorship of Inspections and also I wish to acknowledge the important contribution made by the Technical Working Group at ISS in re-launching our Inspection process. The Technical Working Group currently comprises:

Conrad Barton  
Dan Bennett  
Nigel Bobby  
Annette Giardina  
Mark Woodward

## Appendix 1 – The Seven Habits of Highly Effective Inspection Teams

### **1. Request**                      ***Put the Inspection above your own work***

Whether you're asked to lead an Inspection, or to be a checker, or simply to help with administration tasks you should give it the highest possible priority and consider it more important than your own work. Directors, Project Managers and Team Leaders should all recognise this and set priorities and schedules accordingly.

### **2. Planning**                      ***Choose checkers carefully***

Inspections will only work if the documents are understandable to everyone who's checking. You must choose checkers with the right skills (e.g. C++ experts) if you are going to have an effective Inspection. Especially with code, checkers must be able to understand the context in which is being written. Right now, producing rule sets and checklists that contain this 'expert' knowledge is not possible.

### **3. Kickoff**                      ***Tell 'em and teach 'em but also set targets***

Rather than a quick 5 minute kick-off meeting, hold a walkthrough (30 minutes maximum) to allow people to 'tune in' to the problem as well as explaining the Inspection issues. Where possible set targets and goals for number of defects expected, checking rates etc.

### **4. Checking**                      ***Effort, endeavour & excellence is required from the checkers***

Checkers must check the work product to the best of their ability. Checkers must find time or make time to do a proper and thorough job. This is not a picnic and no one said it would be easy. Use the rules and checklists to guide you but at the end of the day it's up to you to do a thoroughly excellent job.

### **5. Meeting**                      ***Control the discussion to log at the optimum rate***

Leaders must moderate the meeting and control the discussion whilst maintaining the optimum rate for logging issues. The leader has the option of deferring issues to the end of the logging meeting into the discussion meeting.

### **6. Edit**                      ***Trust the author, check the fixes***

We must trust the author/editor of the document to take responsible action on all of the logged issues, including reclassifying majors as minors (and potentially not fixing them). We also expect the author to tidy up the minor issues without making a song and dance about it. We will encourage our people to take professional pride in the work they have produced. However, we require leaders to follow up diligently and check that the fixes have been done. Whether or not this means reading every line of code again or just sampling some of the changes is left to the Inspection Leader.

### **7. Exit**                      ***Keep the metrics confidential***

Whatever method leaders use to run the Inspection, we expect them to keep appropriate metrics and to ensure that these are reported confidentially to the Inspection co-ordinator.



## Appendix 2 - Inspection Master Plan

### Planning

Inspection Leader		Date		Inspection ID	
Project Reference		Document Type			

	Document Name	Pages	Tag
Product			
Source(s)			
Rules			
Checklists			

How much time did the Leader spend planning the Inspection and checking the entry criteria ?  
 \_\_\_\_\_(hours)

### Kick Off Meeting

Date \_\_\_\_\_ Start Time \_\_\_\_\_ Finish Time \_\_\_\_\_ No of Attendees \_\_\_\_\_

### Individual Checking

Initials	Role	Plan Rate	Pages Checked	Time Taken	Majors Found	Minors Found	Improvement Suggestions.	Questions to Author

### Logging Meeting

Date \_\_\_\_\_ Start Time \_\_\_\_\_ Finish Time \_\_\_\_\_ No of Attendees \_\_\_\_\_

Majors Logged	Minors Logged	Improvement Suggestions	Questions to the Author	New Items Found in Meeting

### Leader Follow Up and Exit

Has the Author/Editor taken some action on each of the issues that were logged YES / NO ?

How much time did the Author/Editor spend fixing the defects? \_\_\_\_\_ (hours)

How much time did the Leader spend in follow up ? \_\_\_\_\_ (hours)

Major Defects Fixed	Minor Defects Fixed	Improvement Suggestions	Change Requests Raised

Please sign this form and give it to the Inspection Co-Ordinator who will update the Inspection database.

Leader Signature \_\_\_\_\_ Date \_\_\_\_\_

### Appendix 3 - ISS Generic Rule Set

Rule	Derived From	Name	Tag	Description
I1	G1	Consistent	INC	Statements must be consistent with other statements in the same or related documents.
I2	G2 + G5	Unambiguous	AMB	All documents must be unambiguous to the intended readership.
I3	New	Relevant	IRL	Similar ideas and concepts should be grouped together within the document and focus only on the main themes of the subject heading.
I4	G4	Detail	DET	Statements must be written at a level of detail that will make the document useable for all readers.
I5	New	Complete	NCP	Documents must be complete. This rule is used when it is fairly obvious that a major piece of important information has been omitted from the document.
I6	G6+G8+ G11+G12+ G14	Conformance	NON	All documents must conform to the ISS House Style defined in the appropriate template for each document type. Note: This includes but is not limited to: Headers, footers, fonts, version numbers, dates, change history, Inspection status.
I7	G7	Unique	UNQ	Ideas must be stated only once in documents.

Note: our rules are derived from a set of rules provided by Dorothy Graham on a training course to ISS in November 1997. These are in turn an updated version of the generic rules in the book ‘Software Inspection’ which she co-authored with Tom Gilb [Gilb93].

## Appendix 4 - ISS Generic Entry and Exit Criteria for Inspection

### Entry Criteria

Ref	Name	Description	Waived
GE1	AVETO	No veto by Author	
GE2	LVETO	No veto by Leader	
GE3	Source	Sources exited Inspection or marked NOT EXITED and mini Insp	
GE4	Rules	Applicable rules available in writing	
GE5	Rates	Master plan specifies optimum checking / logging rates	YES
GE6	Cert	Leader is trained and still certified	YES
GE7	Curs	Cursory examination of the product (5 minutes) looks OK)	
GE8	Auto	Automated checks have been run (e.g. spelling, grammar)	
GE9	Auth	Author is part of Inspection team	
GE10	Vol	Product document is volunteered by the author	
GE11	Und	Documents are understandable to each checker	

### Exit Criteria

Ref	Name	Description	Waived
XC1	Rate	Average checking rates within 20% of known optimum	YES
XC2	Rem	Computed majors remaining max of 0.3 per page (3 initially)	YES
XC3	Auth	The author's OK for exit	
XC4	Lead	The leader's OK for exit	
XC5	Data	Inspection data entered (form sent to Inspection Co-Ordinator)	
XC6	Fixed	All logged issues responsibly acted on including change requests	
XC7	All Ch	All chunks have exited before document exit	
XC8	Feedbk	Edited document made available to checkers	
XC9	Zero	Leader believes there are no customer observable defects left	YES
XC10	Samp	Exit on 1% sample if XC2 is not met (not XC6, 7 & 8)	YES

## **Quality Engineering: In the Footsteps of Goldie Locks**

Albert Gallo, NASA

Quality! Every supplier claims to have it and every consumer expects it. Yet, ask the kids what they want to be and “Quality Engineer” does not even make the list.

And yes, Virginia, there really are people who perform this often thankless task. Who are these strange beings? What does it take to be a good quality engineer?

This presentation profiles some individuals responsible for quality on NASA’s critical systems.

Albert Gallo is a Principal Staff Engineer with the Systems Quality Assurance Department at GSFC, NASA. Mr. Gallo has 15 years of Software Systems Engineering and Quality Assurance experience and has experience in all phases of Systems Development with an emphasis on database design. Mr. Gallo supports the Software Assurance Technology Center as a lead trainer for Continuous Risk Management (CRM), having provided training and consulting throughout the NASA agency. He also is responsible for reviewing and updating course materials as well as reviewing project risk management plans. Mr. Gallo holds Bachelors degrees in both Pure Mathematics and Computer Science as well as an M.S. in Technical Management from The Johns Hopkins University, Baltimore MD.

# **What is Software QA, Anyway?**

**By**

**Sandy Raddue**

This paper presents all dimensions and facets of the Software Quality Assurance Engineering profession. Software QA is defined using the American Society for Quality (ASQ) Body of Knowledge for the Certified Software Quality Engineer exam, and touches on the multiple areas of knowledge that make up the job also discusses working with project management to gain acceptance of the realm of Software Quality techniques.

The alternate title of this paper is "QA is NOT a Verb!"

## **Presenter Biography**

Sandy Raddue is presently a Staff Software QA Engineer at Cypress Semiconductor. She started in the wonderful world of Software Engineering in 1978, as her company's Punched Paper Tape expert. Since then, she has worked on a variety of projects and products, from video games to satellite telecommunications systems. Sandy has done development work as well, but has made Software Quality her mission.

She holds a Certified Computer Programmer certificate with Scientific Specialization from the Institute for Certification of Computer Professionals (ICCP), and also holds a Certified Software Quality Engineer certificate from American Society for Quality (ASQ). She is a member of the ASQ and IEEE, and a former member of the IEEE Software Engineering Standards Committees.

The author may be reached at the following:

C/o Cypress Semiconductor  
8196 SW Hall Blvd. Suite 100  
Beaverton, OR 97008  
Email: [sur@cypress.com](mailto:sur@cypress.com)

## **What is Software QA, Anyway? (Alternate title – QA is NOT a verb!)**

By

**Sandy Raddue**

**Cypress Semiconductor, Beaverton, OR**

Are you a Software QA Engineer? A Software QA Analyst? A QA Manager? What is your job description? Does it look like the following?

QA Manager - Great opportunity in software company. Creative, energetic environment. Compensation based on experience. The right person should have at least 2 yrs mgmt exp & must be able to demonstrate senior level exp in all phases of the QA process: Automated testing, test case identification, tracking management, test plan authoring. Looking for someone who understands how to take ownership of the QA process.

The management position described above is a test management position, not a Software Quality Assurance Management position.

Software QA Engineers - In this position, your role will be to develop test automation for firmware and network software. Additionally, you will also assist in development and implementation of test plans. Previous development or scripting experience needed. Ideal candidate will either have Perl combined with QA Partner, Visual Test, or PCL and Postscript in a testing environment.

The above position is a test position. The person filling that position is responsible for testing software and developing tests and test plans.

These roles are important. The general usage within the industry calls the above positions Software Quality Assurance positions. They are test positions. As such, they are only part of the discipline titled Software Quality Assurance. An important part, for sure, but only a part.

The manufacturing world, thanks to Deming, Juran, Crosby, and others, has a well-defined quality specialty. This specialty consists of engineering folks somewhere doing their magic to make sure that a product and a manufacturing process are designed to assure the quality of the finished product. Quality Assurance engineers are responsible for ensuring that when a product is designed, it is designed not only for functionality, but testability, reliability, maintainability, and all those other “-ilities” associated with quality. Quality Assurance engineers also design the test processes used by those who inspect the final goods, or work in progress. The individuals that execute these test processes are referred to as being in the field of Quality Control, or test, or final test, or inspection, or some such nomenclature. These descriptions are understood throughout the industry, and job descriptions are written to adequately differentiate between the positions. The field of manufacturing has had years to mature and develop specific job

descriptions for each “quality assurance” and “quality control” step of the design and manufacturing process.

The field of Software Engineering isn’t as mature as more “physical” engineering disciplines. The field of Software Quality Assurance is even less mature. This lack of maturity leads to a lack of control as evidenced by the fact that software projects, schedules, quality, and costs are out of control. Software management and software developers are sincerely trying to build the best product they can, but it obvious that there are serious quality issues with today’s software products – specifically, Software Quality Assurance issues.

In the arena of shrink-wrap software development, there is a notion of “good-enough” software. This somewhat ambiguous requirement might be good enough to meet market requirements and business needs, if the user requirements were understood.

In the regulated areas of software development – DOD, FAA, NASA, medical equipment, and other critical applications, there is no notion of “good-enough”. If the use of software risks the loss of life or property, there is no room for error. However, even if the development contract requires following rigid Software Quality Assurance methodologies, sometimes meeting the “spirit of the law” is all that is done.

### **Standardization**

There is a general misuse of the terms throughout the software industry. There are organizations working to formalize these definitions. The IEEE has worked for years to develop working standards for software documentation, and many project management guidelines. They are currently working to formalize the Software Engineer job description. The Software Engineering Institute (SEI) has defined the Capability Maturity Model (CMM) to help assess the maturity of a development process and organization. The International Standards Organization (ISO) has formalized the tracking of the software process.

The American Society for Quality (ASQ) is in the forefront of the effort of defining Software Quality Assurance. ASQ is recognized as the leading authority for quality in the United States. They have defined the Software Quality Engineer, and offer a certification in that field, as well as fields more closely related to manufacturing.

ASQ defines a Software QA Engineer as being proficient in the following areas:

- Software Quality Management
- Software Processes
- Software Project Management
- Software Metrics
- Software Testing & Verification
- Software Audits
- Configuration Management

The remainder of this paper defines the discipline of Software Quality Assurance within the above ASQ framework.

### **Software Quality Management**

Software Quality Management is the task of planning and overseeing the activities performed by the software quality team, as defined in the software QA plan for the project. This plan is written by the Software QA Engineer, and indicates what tools and methods are to be used by the Software Quality group for the particular project. Managing the software quality process can have unique challenges.

The software quality role may be perceived by developers as adversarial. The development part of the software team largely produces their product in private, and are generally uncomfortable with anyone “looking over their shoulder”, critiquing their work, their style, or their methods. Reducing the potential for conflict among team members is a particular challenge here.

### **Software Processes**

The Software QA Engineer is instrumental in defining and documenting the processes to be followed by the software engineering team. These processes tend to be departmental in nature and scope, but may be project specific. These processes include, but are not limited to, how to write requirements, how to document design, how to report defects, defect reporting system definition, requirements document template, third party relationship requirements, configuration management procedures, code inspection checklists, and coding guidelines. Making these processes understandable and easy to use reduces the possibility of conflict here.

### **Software Project Management**

Developers tend to estimate schedules by a best case estimate of how long it would take to write the code. They tend to not allow for meetings, tests, code reviews, design reviews, rework, requirements changes, documentation, sick days, or vacations.

The Software QA Engineer assists project management in defining the tasks, scheduling, and costs of a project. This includes determining when the preliminary project definition is complete enough to determine scheduling and costs (per processes defined above). The Software QA Engineer also reviews third party contracts to insure that vendor qualification processes are documented and followed. The Software QA Engineer also acts as traffic cop by making sure that the schedule includes milestone checkpoints and adequate time in the schedule for both development and validation. This is an opportunity to inform management (by use of reliable metrics gained from previous projects) that the developer’s proposed schedule is not realistic. The manager then has some of the information they need to make the right decisions up front, instead of having to always defend a late, over budget project.

Management often sees the Software QA group as the “bad guys” when the QA group is insisting that requirements are defined, or when they want adequate time for testing put into the schedule, or when they mandate that unit test cases be checked in to the



configuration management system when the source code is checked in. By striving to win active management support, and keeping rationale for decision making in the company's best interests, team contributions can be made that won't be perceived as hostile.

### **Software Metrics**

Software metrics is the collection of observations provided to the project team during and following completion of a project. Measurements are periodically taken of data associated with the project. Analyses are performed, and results are presented in meaningful ways. These measurements, and the associated analyses, are collectively known as metrics.

Examples of types of measurements are:

- KLOC (1000 lines of code)
- Number of function points
- Number of modules
- Number of user inputs
- Number of defects reported
- Number of defects resolved
- Number of defects found by customers
- Phase of project where defect is reported

The metrics derived from the above measurements might be:

- KLOC/month
- Function Points/module
- Comparative complexity levels of modules
- Number of defects found in pre alpha test phase
- Number of defects plotted with module complexity

Questions answered by some of the above metrics would be:

- Do the trends show that we're ready for release?
- Do we have large ratios of defects in certain modules?
- Do the trends show that we're ready for release yet?
- Is the rate of defect discovery decreasing?
- Don't the trends show that we're ready for release?
- Is the severity of defects decreasing?
- Do the trends show that we're ready for release? When will we be ready?
- Does the data show that we've found the majority of the defects?

The questions that should never be answered by metrics are:

- When does marketing want to ship?
- Have we found ALL the bugs?
- Is programmer X better than programmer W?
- Is tester Z better than tester Q?

## **Software Testing, Verification, and Validation**

Software testing, verification, and validation are the actions necessary to insure the performance of the software. Ideally, software test will find the defects present in the software, by using strategies and designing tests that optimize this process.

Verification is the process of ensuring that the correct functionality is implemented, or, “Are we building the product right?” Validation is the process of ensuring that the product meets specified requirements, or “Are we building the right product?”.

Testing is the “last line of defense” for finding the defects before they go to the customer. As the old saying goes, “You can’t test quality in.” Historically, management has spent the largest amount in the area of test. However, if the other areas of the software quality specialist are utilized (technical reviews, code inspections, requirements reviews, good process), testing can be more effective, with a result of better overall quality and fewer, shorter, test cycles.

Software testing encompasses three general areas: module (unit) testing, integration testing, and system testing.

Module testing is generally “white box”, where the engineer developing and executing the tests is familiar with the structure, design, and input and output requirements of the module under test. To perform economical module testing, it should be performed by the software developers, but it should be reviewed by a Software QA Engineer for completeness.

Integration testing (grey box) is conducted when two or more modules, each ideally fully tested as stand alone modules, are put together. This testing finds interface problems, such as data incompatibilities. The interactions between the modules, and therefore, inside the system proper, are available. Integration testing is done under the supervision of the Software QA Engineer.

System testing (black box) is performed following the integration of all the modules and subsystems that comprise the software under test. This testing evaluates how well the system meets its requirements. If this testing is performed by an independent entity, it is often called Acceptance testing. The testing is reviewed by Software QA.

All levels of testing contribute to the quality of the software. It is difficult to build a quality product without all types of testing. Imagine, if you will, an auto manufacturer that ONLY tests the parts (nuts, bolts), assemblies (starter, master cylinder), and subsystems (brakes, electrical, ignition) as final test, just as the car rolls off the assembly line. How much more difficult (and expensive) is it to isolate the problems and fix them? It is certainly easier to use accepted quality processes, and find the defects when they are cheapest to fix. Actions like vendor qualification (third party), incoming inspection (code review), lot sampling (module test), subassembly test (integration test) contribute to the economy of manufacturing, and directly affect the end quality result.

### **Software Reviews and Audits**

The Software QA Engineer participates in reviews, consisting of requirements reviews, design reviews, code inspections, test design reviews, documentation reviews, test readiness reviews, and others. Reviews are regularly conducted as a part of the development process, and records are kept so that data can be contributed to the metrics efforts.

The Software QA Engineer performs any audits, generally at the request of project management. Audits are intended to assess the levels of compliance to established processes and procedures. Types of audits include configuration management, configuration control, code inspection results gathering, defect data correctness, etc. Audits are information gathering activities. These activities should be used to evaluate the process, not the individuals.

### **Configuration Management**

Configuration management is the group of activities performed to ensure that the software can be reliably built the same way every build. This encompasses version control, build methodologies, and version numbering. The Software QA Engineer must be familiar with configuration management and configuration control procedures. Configuration control is the set of activities that manage change to the software. This may include a Change Control Board, which is responsible for prioritizing which defects, enhancements, or modifications are to be performed.

### **Software Quality Assurance and the Development Organization**

Software management is always in favor of good quality software. Often, the software development managers don't really know how to accomplish this. Managers may have the impression that more testing means better quality. It is up to the "proactive" Software QA professional to convince management that it is more cost effective to follow these methodologies during software development than to try to catch every defect in final system test. This may be difficult to do in some corporate environments, but rewarding when it happens.

When a development organization becomes large enough to employ one or more Software QA Engineers, it is the responsibility of those quality professionals to implement the above disciplines when possible. Developers do not intentionally write bad software, but they do need proof that new methodologies work at least as well as what they are used to using. If there is no direct proof available, management support is critical.

When working directly as part of the development team, the Software QA engineer will develop and monitor software processes, configuration management processes, and any standards and guidelines that have been accepted by the organization. In addition, the SQE will assist the team with reviews of requirements, design, code, and documentation. The Software QA Engineer will also write QA plans, test plans, test

design specs, and develop and execute tests and test programs. The measurements to be taken, and metrics to be derived, will also be defined and implemented.

When the Software Quality Engineering organization reports to a separate organization, the Software QA Engineer is often responsible for conducting audits on project teams to ensure that the above processes are being followed. In addition, trends and trouble spots may be identified by presentation of metrics and audit results to management.

### **Software Quality as a Career Choice**

Many people in today's job market look at Software Quality as a stepping stone to Software Development. It certainly can serve as a valuable training ground for software developers, because they learn the whys and hows of software engineering methodologies from the outside in.

As a distinct choice, however, Software Quality offers a challenging and rewarding career. Recognition from upper level management can be limited to negative attention when a defect reaches the field, which has led a colleague to state that "Software Quality is the unique field in which you are recognized by what should NOT be in the product". Software QA Engineering offers a different type of opportunity – one that is attractive to those of us who like the big picture, and are good at investigation and discovery, but are willing to leave the implementation details to others. The field of Software QA may not demand the intimate algorithmic knowledge that other areas of Software Engineering requires. However, the software quality discipline does require sound technical ability, knowledge of at least one programming language, familiarity with software implementation, excellent documentation skills, and thorough awareness of the software development process. In addition, the field requires excellent follow through, and the attitude of a true team player. There is interaction available with a larger number of team members, and generally a larger portion of the product. In addition, since so many software organizations have few true software quality assurance activities as part of the working process, there can be ample opportunity to introduce these methodologies, and somewhat define the role of the Software QA Engineer within the company.

The drawbacks are few, but they are strong. The first is that a Software QA Engineer may be perceived by a developer as someone "not good enough" to do development work, rather than an individual that has made a conscious career choice. The second is that unless the organization is willing to implement a true software quality program as I have described, it can be a dead end proposition.

If there is a strong desire to really make a difference, the field of Software Quality can certainly be rewarding.

# SPI – Avoiding the Pitfalls

By: Nicole Bianco

([NicoleBianco@Motorola.com](mailto:NicoleBianco@Motorola.com))

You have been selected to lead a Software Process Improvement (SPI) program for the software development organization. Your customers are Senior Management, 1<sup>st</sup> Line Management, and Practitioners. What barriers will you face? Who will become your allies and who will resist the changes about to occur? How will you reason with your foes while supporting your allies?

This paper will present reasons why people have different reactions to this type of program. Understanding the cause of these reactions will provide you with methods of how to dispel some of those reasons for your foes. Building on the allies' reasons will help you build support for your actions and activities. We will talk about Senior Management issues, the concerns of the practitioners who develop the software, and what the 1<sup>st</sup> Line Managers face for the future.

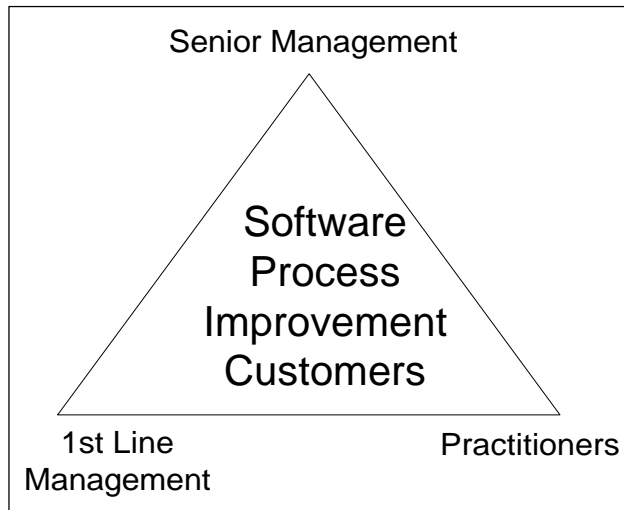
## About the Author

Nicole Bianco has worked in software development and management for more than 30 years in the areas of business, product software and embedded code. She is currently the Engineering Process and Methods Manager in the Internet and Networking Group at Motorola in Massachusetts. Publications include numerous articles in Software QA Magazine, Software Development, and presentations at fortune 500 companies nation-wide. Nicole is a member of the IEEE Computer Society, the Boston and Chicago Software Process Improvement Networks and the Boston and Chicago Patterns Discussion Group.

## SPI – Avoiding the Pitfalls

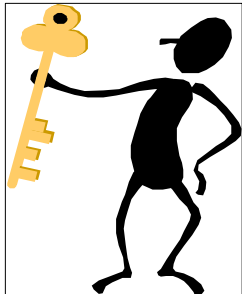
As anyone who has tried to implement change in an organization, or the way an organization works, will find there can be many barriers to success. An organization and each of its participants has developed a way of working that is comfortable, and it all seems to work well. However, there are figures that show by increasing the maturity of a software development organization through a Software Process Improvement (SPI) program can provide many benefits.

The benefits are to all of the SPI Leader's customers: Senior Management, 1<sup>st</sup> Line Managers, and the people who develop the software, or the practitioners. For those who see the benefits, there is urgent recognition of the need, and an eagerness to participate in the program. For those who are comfortable with the current working of the organization, there is apprehension and fear. In some severe cases, there is sabotage (intentional or not) or resignation. It is the goal of this paper to provide the person chartered to lead the SPI program with an understanding of the apprehension and fear, and to work with those to eliminate or minimize the barriers to success of the program.



### Causes of Apprehension

Senior Management is held to costs and production. Time and dollars are the enemy and the friend. Whether it is the production of a widget, or the creation of software to do a business function, it needs to be justified. If it is a business system, it needs to be created at a cost that does not exceed the benefit it provides. The product also needs to do what it was designed to do, and be provided within a time frame that makes business sense.

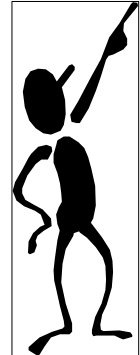


It makes no sense to deliver a Y2K solution in February 2000. It is just too late. This would indicate that being able to schedule work to be delivered in the time frame required is essential to satisfying the requirements of the product. Also, if it costs more in dollars to create than the solution is worth, it is equally worthless. An example here would be to spend 5 development resources creating and supporting a business system that could be manually managed by 3 clerks with the same level of quality.

Another issue Senior Management faces is a limited resource base from which to perform. It is not always possible to go out and hire more people to get the job done. Even if additional resources are made available through additional funding, sometimes the people are just not available to be hired.

The key to success of the Senior Manager is the ability of an organization to perform efficiently and effectively – creating the highest quality software that does what it needs to do with the smallest amount of resource expenditure.

1<sup>st</sup> Line Management is concerned with keeping his manager happy. Recognition comes from delivering that which was requested on time. They have been consistently recognized by the Senior Management for accomplishing this feat! Recognition has come in the form of financial and position rewards. Although it is possible that a person in that 1<sup>st</sup> Line Management position is satisfied with what he is allowed to accomplish, more often he is driven by the hope of higher positions in the organization with more responsibility. So, the recognition received fits their personal goals.



The 1<sup>st</sup> Line Managers gain success through use of the practitioners. The more effectively they use the resources they are given, which are primarily people, the more they can accomplish. If, at times, they need to over-utilize the resources they have to accomplish the goal, it will be done. Their managers may not be aware that the people are working around the clock for two weeks prior to the completion date committed. They will ignore some quality issues that can be managed by the resources over the first few weeks of the product's life to meet the date. The primary fear is failure to deliver.

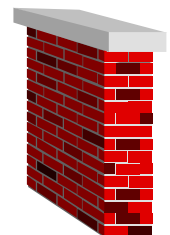
The Practitioners are more varied in what they want from their work. Some want the recognition, which they gain by receiving raises and promotions. Some are more driven by their own technical accomplishments. Others are excited about delivering something that does something good for the customer or the business. In other cases, it is the recognition they get from their peers based on their knowledge or their accomplishments. But in all cases, they need to please their manager. This is because, no matter what their personal goal, their manager provides them with the vehicle (or the job) to achieve it.



Their fears can also be varied. Failure to meet the commitment they made to their manager regarding what they would deliver is the fear of the one who wants recognition in a formal form. Failure to deliver a product that is reliable to the business customer is the fear of the person who wants to help the business. The one who thrives on technical accomplishments fails when their creativity is impaired. If the practitioner is motivated by peer recognition, failing to retain the position of 'resident expert' will be a fear.

## Walls

Walls can be built by any of these people because of fear. Fear of change is natural, especially when a person feels comfortable with how they work and the results they receive. Any change to their norm is a threat to that comfort, and can be resisted even though the person has good intentions.



Other times, people who do not understand the goals or potential results of the SPI activity can build walls. Some may fear that their jobs or security may be

threatened (this is common among 1<sup>st</sup> Line Managers). Some may fear that it will consume too many resources, and there is just too much to get done with the limited resources that are available. Some fear will come from a person's skepticism regarding the outcome of the investment (most often from Senior Managers). Some may fear that they will lose their ability to be creative in approaches to solving problems (which is common among Practitioners). One thing is true – you will not be successful if walls are in the way of your success.

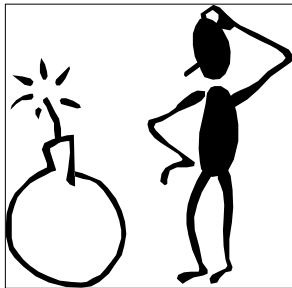
How do you know when you see a wall? First, you need to realize that the wall may not be apparent at first, but it will manifest itself at some point, subtle or not. Indications that there are walls can include:

- An individual's lack of interest about new methods of working
- Constant reasons why a person can not use a new process or work on writing or reviewing one
- Lack of resource commitment to determine or document processes
- Commitment to write or create processes with no follow-through
- A person who always finds something wrong with the processes, but does not commit to work on a solution

How do you start to break down the walls? Understanding the cause for the wall, and eliminating it or convincing the person that the fear is unfounded will help.

### Breaking Down Senior Management Walls

Remember that we talked about Senior Management's concern with resources and time, and that they are responsible to work within these bounds. Consuming human resources to develop on processes may put some previously committed-to deliverables at risk. Senior Management needs assurances that there will be a return on the investment (ROI) to commit to the program.



There are publications available that show significant cycle time, quality, and productivity improvements at SEI Level 3 (Software CMM V1.1), as well as decrease employee turnover and improve morale. While this is true, your goal should not be as arbitrary as that, but should be sold using a justification that will benefit your organization in one or more of these three attributes, and present that to the resistant manager.

This level of management deals with numbers. You need to show him that there will be an ROI. Showing the results from other companies may not prove the point, but it may be enough to intrigue him. Showing him how you could improve one area of the organization's operation may be enough to trigger a commitment to test it out. This will take some work.

Look at a painful problem for your organization's products. It can be missed schedules, quality, cost overruns, or any aspect of managing projects. Choose one that makes sense. Let's assume there is a quality problem in the first three weeks of the release of any new software product. Use one project as an example, and interview the people who were called on the problems to determine how much time they spent on fixing the problems after the release. Also ask them



how much effort would have been spent if these problems were found prior to release. Then, offer to create a process that will help decrease this impact and measure the results.

Single processes that can help decrease the number of defects in software that is released are:

- Improved testing techniques which maps test cases to the requirements of the software (Requirements Management),
- Peer Reviews,
- Business Customer involvement in the testing (Intergroup Coordination),
- Better planning and scheduling (Project Planning).

Select one of these, and commit to doing the process documenting (hopefully with the aid of some of your peers), and measure the results in the same way you took the baseline measurement. Proof in the results will be available when the process is written and the first cycle of use is done. It will be easy to get the commitment from Senior Management to progress after you show the results of such an activity.

### Breaking Down 1<sup>st</sup> Line Management Walls

This is the toughest wall because many of the walls from 1<sup>st</sup> Line Management come from their experience. They have learned that if they focus all of their efforts on delivering software, they will be rewarded. Now they are being asked to give up resources to develop processes, or at least to learn and use new, more formal practices.

Another fear is that if the 1<sup>st</sup> Line Manager is the one who is consulted regarding how to accomplish a feat by a Practitioner, they will feel threatened in their job, and resistance should be anticipated. They will be faced with learning a new set of skills. In this case, senior management has to understand this transition and provide the encouragement these people need to continue to meet the committed goals, while recognizing the change in the role these people play in the organization.



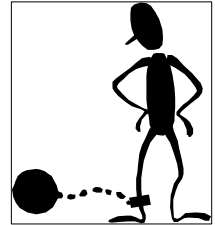
At risk is the loss of some resources that were supposed to be writing code because by using a more formal development process, they will be gathering requirements, designing systems, or (forbid it all) writing or reviewing processes. Coding may not even start until requirements are defined and design is complete. This will make a 1<sup>st</sup> Line Manager nervous, and resistant to the new processes. Working with data from a more mature organization can help ease the apprehension for these people, but when that is lacking, Senior Management commitment to help with the resource issues is essential.

Occasionally, 1<sup>st</sup> Line Managers fear loss of power when SPI programs are initiated. They have been directing how the work will be done. Once it is all proceduralized, there could be no need for this activity, and they are right. But this is not necessarily a job that 1<sup>st</sup> Line Managers should be doing. They should be managing resources, not the hands on activity involved with development. They should be working strategically, not tactically. This is not an easy wall to overcome, but in working with Senior Management, and preparing them for the inevitable, these people can learn that working for different goals (people and strategy instead of task oriented

activities), they can provide a higher impact on the business and organization. This is also a selling point for the Senior Managers.

### Breaking Down Practitioner Walls

As discussed previously, the practitioners will resist SPI for a number of reasons. In fact, an organization that goes through an SPI program tends to lose 8-10% of its practitioners, mostly due to their inability to conform. They either resist the changes and leave, or their peers start shunning them for not being a team player, and they lose the social aspects of work. Inability to be part of the organization is a wall that can not be broken down unless the person is willing to change.



Inherently, practitioners want to do a good job. For the most part, practitioners do not like rework. Showing how an SPI program can help eliminate rework can help win many of them over and they will be willing to commit to the program. An example is that if the requirements are well defined before construction begins, coding only needs to occur once. If coding only needs to occur once, then testing only needs to occur once.

With a well defined life cycle, containing appropriate documents at the end of phases, changes received to the requirements can be managed. Changes to the design and construction results can be handled methodically rather than in a 'slip it in' mode (many times, these are the cause of systems not functioning properly once implemented).



Caution comes in implementing a Peer Review process. This establishes a situation where a person who has developed something to the best of their ability has that deliverable reviewed by their peers. Proper training in the roles in a Peer Review, and how to conduct a peer review is critical. When executed properly, they are the best tools for decreasing defects, improving productivity, and building development teams that have ever been implemented. The information received from them is greatly valued by SPI proponents since it reveals process problems - those phases where defects are inserted.

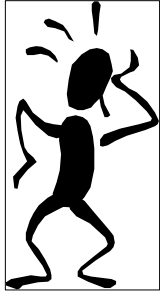
One of the best reasons for a practitioner to commit to the SPI program is one I heard in an assessment a few years back. One of the interviewees was asked, "What is the best part of your job?" The reply was, "I have processes that tell me how to work, and I no longer need to worry about that. Now I can concentrate on WHAT I am creating!"

### Reality of Walls

Walls do exist, and must be dealt with. Tearing them down one at a time, building on the strength of Senior Management, and the willingness of Practitioners to work better, the walls can crumble a little at a time in the beginning, with eventual blasting of the walls once some success is realized.

## Other Pitfalls of SPI

Other than the people and organizational side of walls and pitfalls, there are some that is often created by the SPI leader. That is the one of over-proceduralizing the organization. Too many processes that are too structured to allow tailoring can impair an organization's ability to create software. Processes should be logical, easy to understand, and can be tailored to fit the situation. Tailoring of processes should be encouraged, and records of the results should be kept. This will support activity to continuously improving methods of working.



The SPI leader has to avoid the stigma of being the process police. Processes are designed to help the development activities, not impair them. There is a balance that needs to be achieved between the cost of documenting development activities and the value received in the maintenance activities. Always look for the ROI when tailoring the processes for a specific project. When it makes no sense, or is not value added, abbreviate or abolish it. An example is the need to create a 'one time' job to fix a data file. Keeping some notes in a folder regarding what was done and why is appropriate, but to send the effort through the entire development cycle is overkill.

Recommended life cycle models for several types of projects would be appropriate, based on the work you do. At one organization, I had life cycle models for:

- Large system development
- Release management (new features in an existing product)
- Purchased Package implementation
- Emergency fix situations
- Small development activities (less than 80 hours or work)

## Table of KPA Benefits

Appendix A contains a list of benefits attained by implementing procedures that meet the goals of Key Process Areas (KPAs) of the Software CMM V1.1. If you are looking to sell a process development activity to one of the customers in the software development organization, review the benefits to see if it fits a need of that person. Then, when you are implementing that process, it will be easy to measure the results – the person who was convinced by your promise would love to prove you wrong or right!

## APPENDIX A – Table of Benefits of Key Process Areas

Level 2 Key Process Areas - Repeatable			
KPA	Major Activities	Benefits	Who Cares?
Requirements Management	Get the requirements defined properly, and develop and deliver that which was requested	Can help eliminate projects that are delivered late when requirements are used for planning activities	Practitioners Senior Managers 1st Line Managers
	When well defined, this provides a good baseline for estimating the work effort required		
Software Project Planning	Good task level planning provides a method of delivering on time and with a known budget due to estimating the efforts and costs of development	Will provide the ability to track progress and support the project manager's need to adapt the plans to meet the commitments made. Provides an awareness when a project is getting into overrun trouble or if it will be delivering earlier than anticipated	1st Line Managers
	Provides a method of knowing where the development activity is in relationship to time and effort expended / remaining		
	Provides ability to plan utilization of resources to accomplish high priority products in time		
Software Project Tracking and Oversight	Provides visibility of progress to plan, and awareness when plans need to be reviewed or modified	When a project is properly planned, it will provide awareness to the project manager regarding the ability to complete the project in the agreed to time with the agreed to content	Senior Managers 1st Line Managers
	Provides ability to plan future projects based on real life experiences of previous projects		
Software Quality Assurance	Verifies that the project management activities are occurring, and assures unbiased review of tracking to plans	Validates that the processes used by project management are appropriate for the success of the delivery.	Senior Managers
	Provides ability to predict accuracy of planning, quality, and delivery of products to be delivered		

## APPENDIX A – Table of Benefits of Key Process Areas (cont.)

Level 2 Key Process Areas - Repeatable			
KPA	Major Activities	Benefits	Who Cares?
Software Configuration Management	Developed deliverables will be controlled, retrievable, and secure	Provides a repository for software and supporting documents to be available for future maintenance activities, and supports a mechanism to change the deliverables in a controlled manner.	Practitioners
	Builds / Links of software products are controlled according to a plan, and everyone is aware of the contents		
	Changes to deliverables are controlled and documents, software, and test scenarios are linked		
Software Subcontractor Management	Costs for subcontracting software is controlled by	Provides an ability to outsource development of some products or parts of products with assurance that the results will be positive. Can make the development organization more responsive to customer requests.	Senior Managers 1st Line Managers
	Delivery schedules of subcontracted software is controlled and commitments are tracked		

## APPENDIX A – Table of Benefits of Key Process Areas (cont.)

Level 3 Key Process Areas - Defined			
KPA	Major Activities	Benefits	Who Cares?
Organization Process Focus	Results of use of processes is known, with strengths and opportunities for improvement communicated	Process development activities take a minimal effort from the total resource pool, while assuring it is scheduled with development related activities.	Practitioners Senior Managers 1st Line Managers
	Development activities for new procedures are coordinated in the resources pool		
Organization Process Development	Procedures are managed and controlled in versions	Organization-wide involvement in the process development activities, providing continuous improvement based on use and experience	Practitioners
	Information from the results of the processes is gathered and communicated		
Training Program	Results of training provided is assessed to assure it is value added	Training is value added, and employees feel there is an interest in their growth and ability to create new software.	Practitioners Senior Managers 1st Line Managers
	People receive training necessary to accomplish their jobs		
Integrated Software Management	A standard development life cycle model is defined, and tailored to each project for its specific needs	All projects are developed in a similar manner, language is common across the organization and management, and continuous improvement activities occur based on knowledge of results of prior projects.	Practitioners Senior Managers 1st Line Managers
	Standard project management provides predictable results		
	Experiments with the life cycle model, through tailoring, provides new methods of development		

## APPENDIX A – Table of Benefits of Key Process Areas (cont.)

<b>Level 3 Key Process Areas - Defined</b>			
<b>KPA</b>	<b>Major Activities</b>	<b>Benefits</b>	<b>Who Cares?</b>
Software Product Engineering	Tasks in the development cycle are well defined, producing predictable results	Documentation about the software is no longer lost or out of sync with the software product, which allows new employees to become productive quickly.	Practitioners 1st Line Managers
	Linkages between deliverables and versions of a product provide knowledge of product content		
	Testing occurs methodically by linkages between requirements, construction phases, and test scenarios		
	Continuous improvement loop for product enhancement and improvement allow for easily maintained software products		
Intergroup Coordination	All projects and resources impacted by other projects and resources are aware of their commitments	Test equipment, people, and all resources used in development are coordinated across the organization.	Senior Managers 1st Line Managers
Peer Reviews	Deliverables are reviewed prior to enter the next phase of development	Errors found in early phases of development decrease time required to fix the problems that are found at the end of development cycles.	Practitioners Senior Managers 1st Line Managers
	Review participants transfer knowledge of products developed, providing back ups for specific technologies and techniques used		
	Defects inserted into products can be reviewed to determine cause of insertion		

## **Yes, But What Are We Measuring?**

Cem Kaner, Consultant

Measurement is a Good Thing. But what relationship do common software quality “metrics” have to the things that we want to measure? Merely saying that some number measures efficiency, complexity, reliability, doesn’t make that number a valid useful measure of anything. In this talk, Cem Kaner reviews some of the principles of measurement theory and the questions (and suggestions) that they pose for us.

Cem Kaner consults on technical and software development management issues and teaches about software testing at UC Berkeley Extension, UC Santa Cruz Extension, and at several software companies. He founded and hosts the Los Altos Workshops on Software Testing. His book, *Testing Computer Software*, received an Award of Excellence. Mr. Kaner began working with computers in 1976, while a graduate student of Human Experimental Psychology.

Mr. Kaner came to Silicon Valley in 1983 and has worked as a programmer, human factors analyst, user interface designer, software salesperson, associate in an organization development consulting firm, technical writer, software testing technology team leader, manager of software testing, manager of technical publications, software development manager, and director of documentation and software testing. Mr. Kaner also practices law, usually representing individual developers, small development services companies, and customers. His focus is on the law of software quality.



# 12 Steps to Useful Software Metrics

Linda Westfall

The Westfall Team

westfall@idt.net

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

972-867-1172 (voice)  
972-943-1484 (fax)

**Abstract:** *12 Steps to Useful Software Metrics* introduces the reader to a practical process for establishing and tailoring a software metrics program that focuses on goals and information needs. The process provides a practical, systematic, start-to-finish method of selecting, designing and implementing software metrics. It outlines a cookbook method that the reader can use to simplify the journey from software metrics in concept to delivered information.

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

Very active professionally, Linda Westfall is Chair Elect of the American Society for Quality (ASQ) Software Division. She has also served as the Software Division's Program Chair and Certification Chair and on the ASQ National Certification Board. Linda wrote the Software Metrics and Software Project Management sections of the ASQ Software Quality Engineering course and co-authored the ASQ Software Metrics course.

**Key Words/Phrases:** Software Metrics, Software Measurement, Data Collection

# 12 Steps to Useful Software Metrics

## What Are Software Metrics?

Software metrics are an integral part of the state-of-the-practice in software engineering. Taking an activity based view, Goodman defines software metrics as "The continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products." [Goodman-93] Figure 1, illustrates an expansion of this definition to include software-related services such as installation and responding to customer issues. Software metrics can provide the information needed by engineers for technical decisions as well as information required by management. Taking a measurement based view, Schulmeyer defines a metric as "a quantitative measure of the degree to which a system, component or process possesses a given attribute. [Schulmeyer-98]

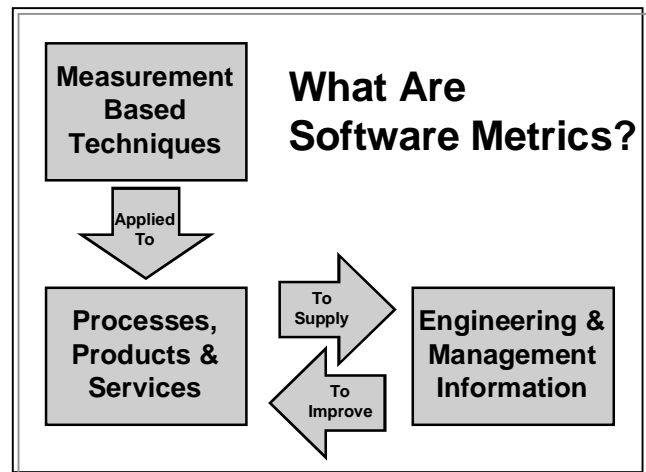


Figure 1: What Are Software Metrics?

If a metric is to provide useful information, everyone involved in selecting, designing, implementing, collecting, and utilizing it must understand its definition and purpose. This paper outlines twelve steps to selecting, designing and implementing software metrics in order to insure this understanding.

## Some Basic Measurement Theory

The use of measurement is common. We use measurements in everyday life to do such things as weigh ourselves in the morning or when we check to time of day or the distance we have traveled in our car. Measurements are used extensively in most areas of production and manufacturing to estimate costs, calibrate equipment, measure quality, and monitor inventories. Science and engineering disciplines depend on the rigor that measurements provide, but what does measurement really mean?

According to Fenton, "measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules" [Fenton-91]. An entity is a person, place, thing, event or time period. An attribute is a feature or property of the entity.

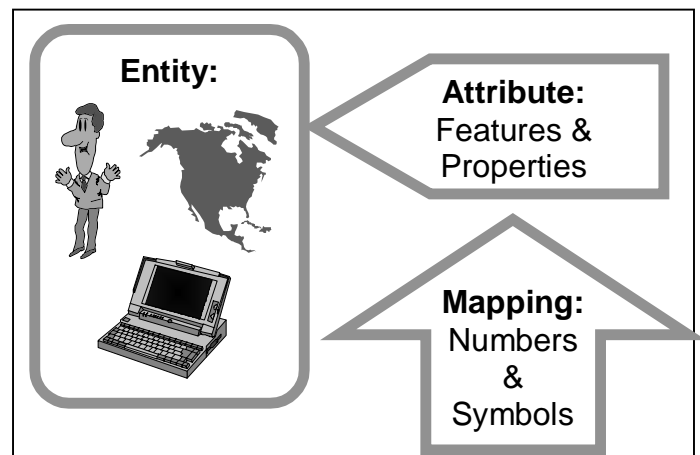


Figure 2: Measurement Defined

To measure, we must first determine the entity. For example, we could select a car as our entity. Once we select an entity, we must select the attribute of that entity that we want to describe. For example, the car's speed or the pressure in its tires would be two attributes of a car. Finally, we must have a defined and accepted mapping system. It is meaningless to say that the car's speed is 65 or its tire pressure is 75 unless we know that we are talking about miles per hour and pounds per square inch, respectively.

We will use the basic process model of input - process - output to discuss software entities. Software entities of the input type include all of the resources used for software research, development, and production. Examples of input entities include people, materials, tools, and methods. Software entities of the process type include software-related activities and events and are usually associated with a time

factor. Examples of process entities include defined activities such as developing a software system from requirements through delivery to the customer, the inspection of a piece of code, or the first 6 months of operations after delivery. Process entities also include time periods, which do not necessarily correspond to specific activities. An example would be the period between 1/1/93 and 2/1/93. Software entities of the output type are the products of the software process. These include all the artifacts, deliverables, and documents that are produced. Examples of software output entities include requirements documentation, design specifications, code (source, object & executable), test documentation (plans, scripts, specifications, cases, reports), project plans, status reports, budgets, problem reports, and software metrics.

Each of these software entities has many properties or features that we might want to measure. We might want to examine a computer's price, performance, or usability. We could look at the time or effort that it took to execute a process, the number of incidents that occurred during the process, its cost, controllability, stability, or effectiveness. We might want to measure the complexity, size, modularity, testability, usability, reliability, or maintainability of a piece of source code.

One of the challenges of software metrics is that few standardized mapping systems exist. Even for seemingly simple metrics like the number of lines of code, no standard counting method has been widely accepted. Do we count physical or logical lines of code? Do we count comments or data definition statements? Do we expand macros before counting and do we count the lines in those macros more than once? Another example is engineering hours for a project – besides the effort of software engineers, do we include the effort of testers, managers, secretaries, and other support personnel? A few metrics which do have standardized counting criteria include McCabe's Cyclomatic Complexity and the Function Point Counting Standard from the International Function Point User Group (IFPUG). However, the selection, definition, and consistent use of a mapping system within the organization for each selected metric are critical to a successful metrics program.

## **Introduction to the Twelve Steps**

Based on all of the possible software entities and all the possible attributes of each of those entities, there are multitudes of possible software metrics. So how do we pick the metrics that are right for our organizations? The first four steps defined in this paper will illustrate how to identify metrics customers and then utilize the goal/question/metric paradigm to select the software metrics that match the information needs of those customers. Steps 5-10 present the process of designing and tailoring the selected metrics, including definitions, models, counting criteria, benchmarks and objectives, reporting mechanisms, and additional qualifiers. The last two steps deal with implementation issues, including data collection and the minimization of the impact of human factors on metrics.

### **Step 1 – Identify Metrics Customers**

The customers for a metrics are the people who will be making decisions or taking action based upon the metrics– that is, the people who need the information supplied by the metrics.

There are many different types of customers for a metrics program. This adds complexity to the program because each customer type may have different information requirements. Customers may include:

- Functional Management - interested in the ability to apply greater control to the software development process, reducing risk and maximizing return on investment.
- Project Management - interested in controlling the projects they are in charge of, and communicating facts to their management.
- Software Practitioners - the people that actually do the software development. They are also responsible for collecting a significant amount of the data required for the metrics program.
- Specialists - including individuals performing specialized functions. (E.g., Marketing, Quality Assurance, Process Engineering, Configuration Management, Testing, Audits, and Assessments, Customer Technical Assistance).
- Users - interested in on time delivery of high quality software products.

If a metric does not have a customer, it should not be produced. Metrics are expensive to collect, report, and analyze so if no one is using a metric, producing it is a waste of time and money.

The customers' information requirements should always drive the metrics program. Otherwise, we may end up with a product without a market and with a program that wastes time and money. By recognizing potential customers and involving those customers early in the metric definition effort, the chances of success are greatly increased.

When talking to our customers, we may find many of their individual needs are related to the same goal or problem but expressed from their perspective or in the terminology of their specialty. Many times, what we hear is their frustrations.

For example, the Project Manager may need to improve the way project schedules are estimated. The Functional Manager is worried about late deliveries. The practitioners complain about overtime and not having enough time to do things correctly. The Test Manager states that by the time the test group gets the software it's too late to test it completely before shipment.

When selecting metrics, we need to listen to these customers and, where possible, consolidate their various goals or problems into statements that will help define the metrics that are needed by our organization or team.

In our example, all these individuals are asking for an improved and realistic schedule estimation process.

## Step 2 – Target Goals

Basili and Rombach [Basili-88] define a Goal/Question/Metric paradigm that provides an excellent mechanism for defining a goal-based measurement program. Figure 3 illustrates the Goal/Question/Metric paradigm.

The second step in setting up a metrics program is to select one or more measurable goals. These may be high-level strategic goals (e.g., minimizing cost or maximizing customer satisfaction). They may also be specific goals (e.g., evaluating a new design method or determining whether a product is ready to be shipped).

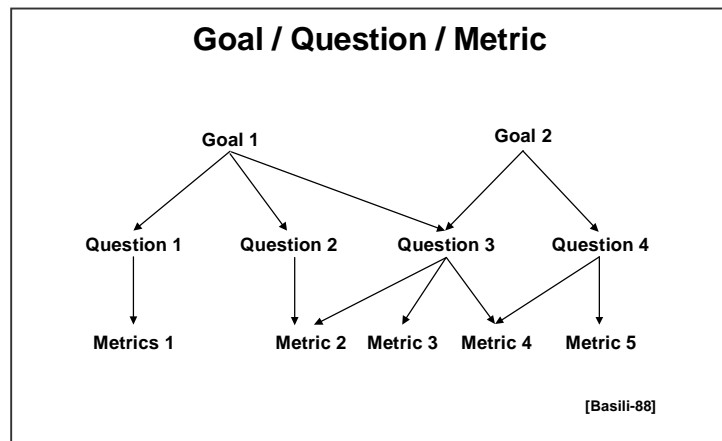


Figure 3 - Goal/Question/Metric Paradigm

Software metrics programs must be designed to provide the specific information necessary to manage software projects and improve software engineering processes and services. Organizational, project, and task goals are determined in advance and then metrics are selected based on those goals. The metrics are used to determine our effectiveness in meeting these goals.

The foundation of this approach is asking not so much "what should I measure?" but "why am I measuring?" or "what business needs does the organization wish its measurement initiative to address?" [Goodman-93]

Measuring software is a powerful way to track progress towards project goals. As Grady states, "Without such measures for managing software, it is difficult for any organization to understand whether it is successful, and it is difficult to resist frequent changes of strategy" [Grady-92]. Measurements help both management and engineers maintain their focus on their goals.

## Step 3 – Ask Questions

The third step is to define the questions that need to be answered in order to ensure that each goal is being obtained. For example, if our goal was to ship only defect-free software, we might select the questions:

- Is the software product adequately tested?
- How many defects are still undetected?
- Are all known defects corrected?

## Step 4 – Select Metrics

The fourth step is to select metrics that provide the information needed to answer these questions. Each selected metric now has a clear objective -- to answer one or more of the questions that need to be answered to determine if we are meeting our goals.

When we are selecting metrics, we must be practical, realistic, and pragmatic. Avoid the "ivory-tower" perspective that is completely removed from the existing software-engineering environment. Start with what is possible within the current process. Once we have a few successes, our customers will be open to more radical ideas -- and may even come up with a few of their own.

Also, remember software metrics don't solve problems. People solve problems. Software metrics act as indicators and provide information so people can make more informed decisions and intelligent choices.

An individual metric performs one of four functions. Metrics can help us **Understand** more about our software products, processes, and services. Metrics can be used to **Evaluate** our software products, processes, and services against established standards and goals. Metrics can provide the information we need to **Control** resources and processes used to produce our software. Metrics can be used to **Predict** attributes of software entities in the future. [Humphrey-89]. A comprehensive Software Metric program would include metrics that perform all of these functions.

The objective for each metric can be formally defined in terms of one of these functions, the attribute of the entity being measured and the goal for the measurement. This leads to the following metrics objective template:

To  $\left[ \begin{array}{l} \text{understand} \\ \text{evaluate} \\ \text{control} \\ \text{predict} \end{array} \right]$  the  $\left[ \begin{array}{l} \text{attribute} \\ \text{of the} \\ \text{entity} \end{array} \right]$  in order to  $\left[ \begin{array}{l} \text{goal(s)} \end{array} \right]$

An example of the use of this template for the "percentage of known defects corrected" metric would be:

To  $\left[ \begin{array}{l} \text{evaluate} \end{array} \right]$  the  $\left[ \begin{array}{l} \% \text{ known} \\ \text{defects} \\ \text{corrected} \\ \text{during} \\ \text{development} \end{array} \right]$  in order to  $\left[ \begin{array}{l} \text{all known} \\ \text{defects are} \\ \text{corrected} \\ \text{before} \\ \text{shipment} \end{array} \right]$

Having a clearly defined and documented objective statement for each metric has the following benefits:

- Provides a rigor and discipline that helps ensure a well-defined metric based on customer goals.
- Eliminates misunderstandings about how the metric is intended to be used.
- Communicates the need for the metric, which can help in obtaining resources to implement the data collection and reporting mechanisms.
- Provides the base requirements statement for the design of the metric.

## Step 5 – Standardize Definitions

The fifth step is to agree to a standard definition for the entities and their measured attributes. When we use terms like *defect*, *problem report*, *size*, and even *project*, other people will interpret these words in their own context with meanings that may differ from our intended definition. These interpretation differences increase when more ambiguous terms like *quality*, *maintainability*, and *user-friendliness* are used.

Additionally, individuals may use different terms to mean the same thing. For example, the terms *defect report*, *problem report*, *incident report*, *fault report*, or *customer call report* may be used by various organizations to mean the same thing, but unfortunately they may also refer to different entities. One external customer may use *customer call report* to refer to their complaint and *problem report* as the description of the defect in the software, while another customer may use *problem report* for the initial complaint. Differing interpretations of terminology may be one of the biggest barriers to understanding.

Unfortunately, there is little standardization in the industry of the definitions for most software attributes. Everyone has an opinion and the debate will probably continue for many years. Our metrics program cannot wait that long. The approach I suggest is to adopt standard definitions within your organization and then apply them consistently. You can use those definitions that exist within the industry as a foundation to get you started. For example, definitions from the IEEE Glossary [IEEE-83] or those found in software engineering and metrics literature. Pick the definitions that match with your organizational objectives or use them as a basis for creating your own definition.

## **Step 6 – Choose a Model**

The sixth step is to derive a model for the metric. In simple terms, the model defines how we are going to calculate the metric. Some metrics, called metric primitives, are measured directly and their model typically consists of a single variable. Other more complex metrics are modeled using mathematical combinations of metrics primitives or other complex metrics.

All modeling includes an element of simplification. This is both the strength and the weakness of modeling. When we create a software measurement model, we need to be pragmatic. If we try to include all of the elements that affect the attribute or characterize the entity, our model can become so complicated that it's useless. Being pragmatic means not trying to create the most comprehensive model. It means picking the aspects that are the most important. Remember that the model can always be modified to include additional levels of detail in the future. Ask yourself the questions:

- Does the model provide more information than we have now?
- Is the information of practical benefit?
- Does it tell us what we want to know?

There are two methods for selecting a model: use an existing model or create a new one. In many cases, there is no need to "re-invent the wheel." Many software metrics models exist that have been used successfully by other organizations. These are documented in the current literature and in proprietary products that can be purchased. With a little research, we can utilize these models with little or no adaptation to match our own environment.

The second method is to create our own model. The best advice here is to talk to the people who are actually responsible for the product or resource or who are involved in the process. They are the experts. They know what factors are important. If we create a new model for our metric, we must ensure the model is intelligible to our customers and we must prove it is a valid model for what we are trying to measure. Often, this validation can occur only through application of statistical techniques.

To illustrate the selection of a model, let's consider a metric for the duration of unplanned system outages. If we are evaluating a software system installed at a single site, a simple model such as minutes of outage per calendar month may be sufficient. If our objective is to compare different software releases installed on varying numbers of sites, we might select a model such as minutes of outage per 100 operation months. If we wanted to focus in on the impact to our customers, we might select minutes of outage per site per year.

## **Step 7 – Establish Counting Criteria**

The seventh step in designing a metric is to break the model down into its lowest level metric primitives and define the counting criteria used to measure each primitive. This defines the mapping system for the measurement of each metric primitive.

The importance of the need for defining counting criteria can be illustrated by considering the lines of code metric. Lines of code is one of the most used and most often misused of all of the software metrics. The problems, variations, and anomalies of using lines of code are well documented [Jones-86], and there is no industry-accepted standard for counting lines of code. Therefore, if you are going to use a metric based on lines of code, it is critical that specific counting criteria be defined. These criteria must also accompany the metric in all reports and analysis so that metrics customers can understand the definition of the metric. Without this, invalid comparisons with other data are almost inevitable.

The metric primitives and their counting criteria define the first level of data that needs to be collected in order to implement the metric. To illustrate this, let's use the model of minutes of system outage per site per year. One of the metrics primitives for this model is the number of sites. At first, counting this primitive seems simple. However, when we consider the dynamics of adding new sites or installing new software on existing sites, the counting criteria become more complex. Do we use the number of sites on the last day of the period or calculate some average number of sites for the period? Either way, we will need to collect data on the date the system was installed on the site. In addition, if we intend to compare different releases of the software we will need to collect data on what releases have been installed on each site and when each was installed.

## **Step 8 – Decide What's Good**

The eighth step in designing a metric is defining "what's good". Once you have decided what to measure and how to measure it, you have to decide what to do with the results. Is 10 too few or 100 too many? Should the trend be up or down? What do the metrics say about whether or not the product is ready to ship?

One of the first places to start looking for "what's good" is the customer. Many times, user requirements dictate certain values for some metrics. There may be product reliability levels that must be met. The customer may have a defined expectation of defect reduction from release to release or a required repair time for discovered defects. Another source of information is the metrics literature. Research and studies have helped establish industry-accepted norms for standard measures. For example, studies have established that modules of source code should have a McCabe's Cyclomatic Complexity of  $\leq 10$  or they may be error-prone and difficult to maintain.

The best source of information on "what's good" is your own data. Processes vary from group to group. Many metrics do not have industry accepted counting criteria (i.e., lines of code), and most documentation does not include how the metrics were calculated. Therefore, comparing your values to published standards may result in erroneous interpretations. Whenever possible, use your organization's own data to determine baselines and establish objectives for your metrics. If historical data is not available, wait until enough data is collected to reasonably establish current values.

Once you have decided where you are and where you want to be, you can determine whether or not action is needed. If no action is necessary, management can either turn its attention elsewhere or establish some monitoring actions to insure that the value stays at desired level. However, if improvements are needed, goals can be established to help drive and monitor improvement activities. When setting metrics goals remember:

- The goal must be reasonable. It's all right to establish a stretch goal, but if the goal is unrealistic, everyone will just ignore it.
- A goal must be relevant in the eyes of the people that must achieve it, or they will not understand (and not support) it.
- The goal must be associated with a time frame. Nothing happens overnight. To say a 50% backlog reduction without a "by when" is meaningless.
- The goal must be based on supporting actions. It may be reasonable to set a goal of 50% backlog reduction for defects if a special team is created to concentrate on fixing problems. If everything is "business as usual", do not expect setting a goal to have any effect.
- Unless it is achievable in a short time, the goal must be broken down into small incremental pieces. If the "by when" is a long way off, it is only human nature to procrastinate. If there is a year to

accomplish the improvement, it will be easy to put it on the back burner in preference to more pressing concerns. If we divide the same goal into 12 smaller end-of-the-month goals, actions are more likely to be taken now.

## **Step 9 – Define Reporting Mechanisms**

The ninth step is to decide how to report the metric. This includes defining the report format, data extraction and reporting cycle, reporting mechanisms, distribution, and availability.

The report format defines what the report looks like. Is the metric included in a table with other metrics values for the period? Is it added as the latest value in a trend chart that tracks values for the metric over multiple periods? Should that trend chart be a bar, line, or area graph? Is it better to compare values using stacked bars or a pie chart? Do the tables and graphs stand alone, or is there detailed analysis text included with the report? Are goals or control values included in the report?

The data extraction cycle defines how often the data snap-shot(s) are required for the metric and when they will be available for use for calculating the metric. The reporting cycle defines how often the report is generated and when it is due for distribution. For example, root cause analysis metrics may be triggered by some event, like the completion of a phase in the software development process. Other metrics like the defect arrival rate may be extracted and reported on a daily basis during system test and extracted on a monthly basis and reported quarterly after the product is released to the field.

The reporting mechanism outlines the way that the metric is delivered (i.e., hard copy report, email, on-line electronic data).

Defining the distribution involves determining who receives regular copies of the report or access to the metric. The availability of the metrics defines any restrictions on access to the metric (i.e., need to know, internal use only) and the approval mechanism for additions and deletions to access or standard distribution.

## **Step 10 – Determine Additional Qualifiers**

The tenth step in designing a metric is determining the additional metric qualifiers. A good metric is a generic metric. That means the metric is valid for an entire hierarchy of additional qualifiers. For example, we can talk about the duration of unplanned outages for an entire product line, an individual product, or a specific release of that product. We could look at outages by customer or business segment. Alternatively, we could look at them by type or cause.

The additional qualifiers provide the demographic information needed for these various views of the metric. The main reason additional qualifiers need to be defined as part of the metrics design is that they determine the second level of data collection requirements. Not only is the metric primitive data required, but data also has to exist to allow the distinction between these additional qualifiers.

## **Step 11 –Data Collection**

The question of "what data to collect?" was actually answered in steps 7 and 10 above. The answer is to collect all of the data required to provide the metrics primitives and the additional qualifiers.

In most cases, the "owner" of the data is the best answer to the question of "who should collect the data?" The data "owner" is the person with direct access to the source of the data and in many cases is actually responsible for generating the data. Table 1 illustrates the owners of various kinds of data.

Benefits of having the data owner collect the data include:

- Data is collected as it is being generated, which increases accuracy and completeness
- Data owners are more likely to be able to detect anomalies in the data as it is being collected, which increases accuracy
- Human error caused by duplicate recording (once by data recorder and again by data entry clerk) is eliminated, which increases accuracy



Once the people who gather the data are identified, they must agree to do the work. They must be convinced of the importance and usefulness of collecting the data. Management has to support the program by giving these people the time and resources required to perform data collection activities. A support staff must also be available to answer questions and to deal with data and data collection problems and issues.

Owner	Examples of Data Owned
Management	<ul style="list-style-type: none"> <li>• Schedules</li> <li>• Budgets</li> </ul>
Engineers	<ul style="list-style-type: none"> <li>• Time spent per task</li> <li>• Inspection data including defects</li> <li>• Root cause of defects</li> </ul>
Testers	<ul style="list-style-type: none"> <li>• Test cases planned/executed/passed</li> <li>• Problem reports from testing</li> <li>• Test coverage</li> </ul>
Configuration Management Specialists	<ul style="list-style-type: none"> <li>• Lines of code</li> <li>• Modules changed</li> </ul>
Users	<ul style="list-style-type: none"> <li>• Problem reports from operations</li> <li>• Operational hours</li> </ul>

Table 1: Examples of Data Ownership

A training program should be provided to help insure that the people collecting the data understand what to do and when to do it. As part of the preparation for the training program, suitable procedures must be established and documented. For simple collection mechanisms, these courses can be as short as one hour. I have found that hands-on, interactive training, where the group works actual data collection examples, provides the best results.

Without this training, hours of support staff time can be wasted answering the same questions repeatedly. An additional benefit of training is that it promotes a common understanding about when and how to collect the data. This reduces the risk of collecting invalid and inconsistent data.

If the right data is not collected accurately, then the objectives of the measurement program cannot be accomplished. Data analysis is pointless without good data. Therefore, establishing a good data collection plan is the cornerstone of any successful metrics program. Data collection must be:

- Objective: The same person will collect the data the same way each time.
- Unambiguous: Two different people, collecting the same measure for the same item will collect the same data.
- Convenient: Data collection must be simple enough not to disrupt the working patterns of the individual collecting the data. Therefore, data collection must become part of the process and not an extra step performed outside of the workflow.
- Accessible: In order for data to be useful and used, easy access to the data is required. This means that even if the data is collected manually on forms, it must ultimately be included in a metrics database.

There is widespread agreement that as much of the data gathering process as possible should be automated. At a minimum, standardized forms should be used for data collection, but at some point the data from these forms must be entered into a metrics database if it is to have any longer term usefulness. I have found that information that stays on forms quickly becomes buried in file drawers never to see the light of day again.

Dumping raw data and hand tallying or calculating metrics is another way to introduce human error into the metrics values. Even if the data is recorded in a simple spreadsheet, automatic sorting, data extraction, and calculation are available and should be used. Using a spreadsheet or database also increases the speed of producing the metrics over hand tallies.

Automating metrics reporting and delivery eliminates hours spent standing in front of copy machines. It also increases usability because the metrics are available on the computer instead of buried in a pile of papers of the desk. Remember, metrics are expensive. Automation can reduce the expense, while making the metrics available in a timelier manner.

## Step 12 – The People Side of the Metrics Equation

No discussion on selecting, designing and implementing software metrics would be complete without a look at how measurements affect people and people affect measurements. Whether a metric is ultimately useful to an organization depends upon the attitudes of the people involved in collecting the data, calculating, reporting, and using the metric. The simple act of measuring will affect the behavior of the individuals being measured. When something is being measured, it is automatically assumed to have importance. People want to look good; therefore, they want the measures to look good. When creating a metric, always decide what behaviors you want to encourage. Then take a long look at what other behaviors might result from the use or misuse of the metric. The best way I have found to avoid human factors problems in working with metrics is to follow some basic rules:

**Don't measure individuals:** The state-of-the-art in software metrics is just not up to this yet. Individual productivity measures are the classic example of this mistake. Remember that we often give our best people the hardest work and then expect them to mentor others in the group. If we measure productivity in lines of code per hour, these people may concentrate on their own work to the detriment of the team and the project. Even worse, they may come up with unique ways of programming the same function in many extra lines of code. Focus on processes and products, not people.

**Never use metrics as a "stick":** The first time we use a metric against an individual or a group is the last time we get valid data.

**Don't ignore the data:** A sure way to kill a metrics program is to ignore the data when making decisions. "Support your people when their reports are backed by data useful to the organization" [Grady-92]. If the goals we establish and communicate don't agree with our actions, then the people in our organization will perform based in our behavior, not our goals.

**Never use only one metric:** Software is complex and multifaceted. A metrics program must reflect that complexity. A balance must be maintained between cost, quality and schedule attributes to meet all of the customer's needs. Focusing on any one single metric can cause the attribute being measured to improve at the expense of other attributes.

**Select metrics based on goals:** Metrics act as a big spotlight focusing attention on the area being measured. By aligning our metrics with our goals, we are focusing people's attention on the things that are important to us.

**Provide feedback:** Providing regular feedback to the team about the data they help collect has several benefits:

- It helps maintain focus on the need to collect the data. When the team sees the data actually being used, they are more likely to consider data collection important.
- If team members are kept informed about the specifics of how the data is used, they are less likely to become suspicious about its use.
- By involving team members in data analysis and process improvement efforts, we benefit from their unique knowledge and experience.
- Feedback on data collection problems and data integrity issues helps educate team members responsible for data collection. The benefit can be more accurate, consistent, and timely data.

**Obtain "buy-in":** To have 'buy-in' to both the goals and the metrics in a measurement program, team members need to have a feeling of ownership. Participating in the definition of the metrics will enhance this feeling of ownership. In addition, the people who work with a process on a daily basis will have intimate knowledge of that process. This gives them a valuable perspective on how the process can best be measured to ensure accuracy and validity, and how to best interpret the measured result to maximize usefulness.

## Conclusion

A metrics program that is based on the goals of an organization will help communicate, measure progress towards, and eventually attain those goals. People will work to accomplish what they believe to be important. Well-designed metrics with documented objectives can help an organization obtain the information it needs to continue to improve its software products, processes, and services while maintaining a focus on what is important. A practical, systematic, start-to-finish method of selecting, designing, and implementing software metrics is a valuable aid.

## References

- [Basili-88] V. R. Basili, H. D. Rombach., 1988, The TAME Project: Towards Improvement-Oriented Software Environments. In *IEEE Transactions in Software Engineering* 14(6) (November).
- [Fenton-91] Norman E. Fenton, 1991, *Software Metrics, A Rigorous Approach*, Chapman & Hall, London.
- [Goodman-93] Paul Goodman, 1993, *Practical Implementation of Software Metrics*, McGraw Hill, London.
- [Grady-92] Robert B. Grady, 1992, *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Englewood Cliffs.
- [IEEE-83] *IEEE Standard Glossary of Software Engineering Terminology*, ANSI/IEEE Std 729-1983, The Institute of Electrical and Electronics Engineers, New York, NY, 1983.
- [Jones-86] Capers Jones, 1986, *Programming Productivity*, McGraw Hill, New York.
- [Humphrey-89] Watts Humphrey, 1989, *Managing the Software Process*, Addison-Wesley, Reading.
- [Schulmeyer-98] G. Gordon Schulmeyer, James I. McManus, *Handbook of Software Quality Assurance, 3<sup>rd</sup> Edition*, Prentice Hall PTR, Upper Saddle River, NJ, 1998.

# **E-ffective Testing for E-commerce**

*By Angelina Samaroo, Steve Allott and Brian Hambling, Imago<sup>QA</sup>, UK*

## **Introduction**

What is e-commerce? For the purposes of this paper, e-commerce (also known as e-business) is defined as the software and business processes required to allow businesses to operate solely or primarily using digital data flows. E-commerce is often associated with web technology and is commonly transacted via web portals, but e-commerce is much more than the provision of a web page as the customer interface. The creation of integrated business processes (Enterprise Resource Planning), the integration of collections of disparate software applications, each designed to facilitate a different aspect of the business (Enterprise Application Integration), the extension of software and business processes to embrace transactions with suppliers' systems (Supply Chain Management), the need for increased security for transactions over public networks, and the potential volume demand at e-commerce sites all provide new and unique challenges to the e-commerce development community – challenges which will require novel and innovative solutions and which will need thorough testing before they are allowed to go live.

Why is testing important in the e-commerce environment? The first and primary reason is because e-commerce is, by its very nature, business critical. In the third quarter of 1998, Dell's e-commerce site exceeded \$10 million in daily sales; the E\*Trade site currently exceeds 52,000 transactions per day, giving a cost of one-day failure of around \$800,000; and the travel industry in Europe will be worth \$2 billion by 2002, according to Datamonitor. The immediacy of the customer, with its implied promise of rapid delivery at competitive prices, and the sheer accessibility of the web, all combine to create potentially massive demand on web sites and portals.

The second reason is that e-commerce is a massive and growing market place but one which requires large up-front investment to enter successfully. There are already 5.8 million web sites worldwide, 2.5 million of which have been created this year (1999). The International Data Corporation (IDC) estimates that the e-commerce market will grow from over \$5 billion in 1998 to \$1 trillion in 2003. The average cost of development of an e-commerce site is \$1 million, says the Gartner Group, and will increase by 25% annually over the next 2 years.

The third reason is because the history of e-commerce development has been littered with expensive failures, at least some of which could have been avoided by better testing before the site was opened to the general public. (In e-commerce terms, 'the site' means the entire architecture from suppliers through back-end systems and front-end systems to

the customers; it typically includes Intranet, Internet and extranet applications as well as legacy systems and third party middleware).

## **The Testing Challenges**

### ***Business Issues***

A successful e-commerce application is:

- Usable. Problems with user interfaces lose clients.
- Secure. Privacy, access control, authentication, integrity and non-repudiation are big issues.
- Scaleable. Success will bring increasing demand.
- Reliable. Failure is unthinkable for a business critical system.
- Maintainable. High rates of change are fundamental to e-commerce.
- Highly available. Downtime is too expensive to tolerate.

These characteristics relate in part to the web technology that usually underlies e-commerce applications, but they are also dependent on effective integration and effective back-end applications. E-commerce integrates high value, high risk, high performance business critical systems, and it is these characteristics that must dominate the approach to testing because it is these characteristics that determine the success of e-commerce at the business level.

### ***Technical Issues***

The development process for e-commerce has unique characteristics and some associated risks. It is generally recognised that a 'web year' is about 2 months long. In other words, a credible update strategy would need to generate e-commerce site updates roughly monthly. For this reason, Rapid Application Development (RAD) techniques predominate in the e-commerce environment, and in some cases development is even done directly in a production environment rather than in a separate development environment. RAD techniques are not new, and it is generally agreed that they work best where functionality is visible to the user – so web site development would seem to be an ideal application area. Unfortunately, though, other aspects of e-commerce are at least as important as the front-end. The end-to-end integration of business processes and the consequent severe constraints placed on intermediate processes make them less than ideal application areas for RAD.

These changes increase risk and create new challenges for testers, because time pressures militate against spending a longer time testing sites before they are released. At the same time, the technical environment of front-end systems is changing very rapidly, so change is imposed on e-commerce sites even when the site itself is not changing. This requires more regression testing than would be expected in a conventional application to ensure that the site continues to function acceptably after changes to browsers, search engines

and portals. New issues have also come to the fore for testers, notably security of transactions and the performance of web sites under heavy load conditions.

If we consider an e-commerce site as made up of a front end (the human-computer interface), a back end (the software applications underlying the key business processes) and some middleware (the integrating software to link all the relevant software applications), we can consider each component in isolation.

### ***Front End Systems***

*Static Testing.* The front end of an e-commerce site is usually a web site that needs testing in its own right. The site must be syntactically correct, which is a fairly straightforward issue, but it must also offer an acceptable level of service on one or more platforms, and have portability between chosen platforms. It should be tested against a variety of browsers, to ensure that images seen across browsers are of the same quality. Usability is a key issue and testing must adopt a user perspective. For example, the functionality of buttons on a screen may be acceptable in isolation, but can a user navigate around the site easily and does information printed from the site look good on the page when printed? It is also important to gain confidence in the security of the site. Many of these tests can be automated by creating and running a file of typical user interactions – useful for regression testing and to save time in checking basic functionality.

*Dynamic Testing.* Applications attached to an e-commerce site, either by CGI programming or server extensions, will need to be tested by creating scenarios that generate calls to these attached applications, for example by requiring database searches. The services offered to customers must be systematically explored, including the turnaround time for each service and the overall server response. This, too, must be exercised across alternative platforms, browsers and network connections. E-commerce applications are essentially transaction-oriented, based on key business processes, and will require effective interfacing between intranet-based and extranet-based applications.

### ***Back End Systems***

The back end of e-commerce systems will typically include ERP and database applications. Back end testing, therefore, is about business application testing and does not pose any new or poorly understood problems from a business perspective, but there are potential new technical problems, such as server load balancing. Fortunately, client-server system testing has taught the testing community many valuable lessons that can be applied in this situation. What is essential, however, is to apply the key front end testing scenarios to the back end systems. In other words, the back end systems should be driven by the same real transactions and data that will be used in front end testing. The back end may well prove to be a bottleneck for user services, so performance under load and scalability are key issues to be addressed. Security is an issue in its own right, but also has potential to impact on performance.

## ***Middleware and Integration***

Integration is the key to e-commerce. In order to build an e-commerce application, one or more of the following components are usually integrated:

- Database Server
- Server-side application scripts/programs
- Application server
- HTML forms for user interface
- Application scripts on the client
- Payment server
- Scripts/programs to integrate with legacy back-end systems

The process of developing an e-commerce site is significantly different from developing a web site – commerce adds extra levels of complexity. One highly complex feature is that of integration.

If an application is being built that uses a database server, web server and payment server from different vendors, there is considerable effort involved in networking these components, understanding connectivity-related issues and integrating them into a single development (executable) environment. If legacy code is involved, this adds a new dimension to the problem, since time will need to be invested in understanding the interfaces to the legacy code, and the likely impact of any changes.

It is also crucial to keep in mind the steep learning curve associated with cutting-edge technologies. Keeping pace with the latest versions of the development tools and products to be integrated, their compatibility with the previous versions, and investigating all the new features for building optimal solutions for performance can be a daunting task. Also, since e-commerce applications on the web are a relatively new phenomenon, there are unlikely to be any metrics on similar projects to help with project planning and development.

The maintenance tasks of installing and upgrading applications can also become very involved, since they demand expertise in:

- Database administration.
- Web server administration.
- Payment server administration.
- Administration of any other special tools that have been integrated into the site.

Technical support should also be borne in mind.

Correctly functioning back-end and front-end systems offer no guarantees of reliable overall functionality or performance. End-to-end testing of complete integrated architectures, using realistic transactions, is an essential component.

## Ten Key Principles of Effective E-Commerce Testing

Over the decades since Information Technology (IT) became a major factor in business life, problems and challenges such as those now faced by the e-commerce community have been met and solved. Key testing principles have emerged and these can be successfully applied to the e-commerce situation.

**Principle 1. Testing is a risk management process.** The most important lesson we have learned about software testing is that it is one of the best mechanisms we have for managing the risk to businesses of unsuccessful IT applications. Effective testing adopts a strategy that is tailored to the type of application or service being tested, the business value of the application or service, and the risks that would accompany its failure. The detailed planning of the testing and the design of the tests can then be conformed by the strategy into a business-focused activity that adds real business value and provides some objective assessment of risk at each stage of the development process. Plans should include measures of risk and value and incorporate testing and other quality-related activities that ensure development is properly focused on achieving maximum value with minimum risk. Real projects may not achieve everything that is planned, but the metrics will at least enable us to decide whether it would be wise to release an application for live use.

**Principle 2. Know the value of the applications being tested.** To manage risk effectively, we must know the business value of success as well as the cost of failure. The business community must be involved in setting values on which the risk assessment can be based and committed to delivering an agreed level of quality.

**Principle 3. Set clear testing objectives and criteria for successful completion (including test coverage measures).** When testing an e-commerce site, it would be very easy for the testing to degenerate into surfing, due to the ease of searching related sites or another totally unrelated site. This is why the test programme must be properly planned, with test scripts giving precise instructions and expected results. There will also need to be some cross-referencing back to the requirements and objectives, so that some assessment can be made of how many of the requirements have been tested at any given time. Criteria for successful completion are based on delivering enough business value, testing enough of the requirements to be confident of the most important behaviour of the site, and minimising the risk of a significant failure. These criteria – which should be agreed with the business community – give us the critical evidence that we need in deciding readiness to make the site accessible to customers.

**Principle 4. Create an effective test environment.** It would be very expensive to create a completely representative test environment for e-commerce, given the variety of platforms and the use of the Internet as a communications medium. Cross-platform testing is, naturally, an important part of testing any multi-platform software application. In the case of e-commerce, the term ‘cross-platform’ must also extend to include ‘cross-browser’. In order to ensure that a site loads and functions properly from all supported platforms, as much stress and load testing as possible should be performed. As an



absolute minimum, several people should be able to log into the site and access it concurrently, from a mixture of the browsers and platforms supported. The goal of stress and load testing, however, is to subject the site to representative usage levels. It would, therefore, be beneficial to use automated tools, such as Segue's SilkPerformer or Mercury Interactive's LoadRunner, for performance/load testing.

**Principle 5. Test as early as possible in the development cycle.** It is already well understood and accepted in the software engineering community that the earlier faults are detected, the cheaper the cost of rectification. In the case of an e-commerce site, a fault found after shipping will have been detected as a failure of the site by the marketplace, which is potentially as large as the number of Internet users. This has the added complication of loss of interest and possibly the loss of customer loyalty, as well as the immediate cost of fixing the fault. The fact that e-commerce development is rapid and often based on changing requirements makes early testing difficult, but testing strategies have been developed by the RAD community, and these can be mobilised for support. Perhaps the most important idea in RAD is the joint development team, allowing users to interact with the developers and validate product behaviour continuously from the beginning of the development process. RAD utilises product prototypes, developed in a series of strictly controlled 'timeboxes' – fixed periods of time during which the prototype can be developed and tested – to ensure that product development does not drift from its original objectives. This style of web development makes testing an integral part of the development process and enhances risk management throughout the development cycle.

**Principle 6. User Acceptance Testing (UAT).** The client or ultimate owner of the e-commerce site should perform field testing and acceptance testing, with involvement from the provider where needed, at the end of the development process. Even if RAD is used with its continuous user testing approach, there are some attributes of an e-commerce site that will not be easy (or even possible, in some cases) to validate in this way. Some form of final testing that can address issues such as performance and security needs to be included as a final confirmation that the site will perform well with typical user interactions. Where RAD is not used, the scope of the provider's internal testing coverage and user acceptance testing coverage should be defined early in the project development lifecycle (in the Test Plan) and revisited as the project nears completion, to assure continued alignment of goals and responsibilities. UAT, however, should not be seen as a beta-testing activity, delegated to users in the field before formal release. E-commerce users are becoming increasingly intolerant of poor sites, and technical issues related to functionality, performance or reliability have been cited as primary reasons why customers have abandoned sites. Early exposure of users to sites with problems increases the probability that they will find the site unacceptable, even if developers continue to improve the site during beta testing.

**Principle 7. Regression testing.** Applications that change need regression testing to confirm that changes did not have unintended effects, so this must be a major feature of any e-commerce testing strategy. Web-based applications that reference external links need regular regression testing, even if their functionality does not change, because the

environment is changing continuously. Wherever possible, regression testing should be automated, in order to minimise the impact on the test schedule.

**Principle 8. Automate as much as possible.** This is a risky principle because test automation is fraught with difficulties. It has been said that a fool with a tool is still a fool, and that the outcome of automating an unstable process is faster chaos, and both of these are true. Nevertheless, the chances of getting adequate testing done in the tight time scales for an e-commerce project and without automation are extremely slim. The key is to take testing processes sufficiently seriously that you document them and control them so that automation becomes a feasible option – then you select, purchase and install the tools. It will not be quick or cheap – but it might just avoid a very expensive failure.

**Principle 9. Capture test incidents and use them to manage risk at release time.** A test incident is any discrepancy between the expected and actual results of a test. Only some test incidents will relate to actual faults; some will be caused by incorrect test scripts, misunderstandings or deliberate changes to system functionality. All incidents found must be recorded via an incident management system (IMS), which can then be used to ascertain what faults are outstanding in the system and what the risks of release might be. Outstanding incidents can be one of the completion criteria that we apply, so the ability to track and evaluate the importance of incidents is crucial to the management of testing.

**Principle 10. Manage change properly to avoid undoing all the testing effort.** Things change quickly and often in an e-commerce development and management of change can be a bottleneck, but there is little point in testing one version of a software application and then shipping a different version; not only is the testing effort wasted, but the risk is not reduced either. Configuration Management tools, such as PVCS and ClearCase, can help to minimise the overheads of change management, but the discipline is the most important thing.

## **Conclusions**

E-commerce is both familiar and novel. Some of the technology is relatively novel, and the application of that technology to a complete business is certainly novel, but the problems of creating business processes to operate a business in a wholly new environment overshadow all of that novelty with some familiar and intractable problems. Paradoxically, it is in the more familiar areas of the technology that the most serious problems arise, because the emergence of e-commerce has placed new and challenging requirements on this relatively old technology that was designed for a quite different purpose.

Testing is crucial to e-commerce because e-commerce sites are both business critical and highly visible to their users; any failure can be immediately expensive in terms of lost revenue and even more expensive in the longer term if disaffected users seek alternative sites. Yet the time pressures in the e-commerce world militate against the thorough testing usually associated with business criticality, so a new approach is needed to enable

testing to be integrated into the development process and to ensure that testing does not present a significant time burden.

The very familiarity of much of the technology means that tried and true mechanisms will either be suitable or can be modified to fit. Rapid Applications Development (RAD), in particular, suggests some promising approaches. Like most new ventures, though, e-commerce must find its own way and establish its own methods. In this paper we have suggested some testing principles that have stood the test of time and intermingled them with some lessons learned from similarly challenging development environments to give e-commerce testers a starting point for their journey of discovery.

# The Trials and Tribulations of Testing a Web Application

By Jennifer Smith-Brock

June 1, 1999

This paper is based on work experience. Prior to this project, the experience was focused on testing large, high-risk, financial applications utilizing structured testing techniques. These observations are based on this previous experience and exposure. *The Trials and Tribulations of Testing a Web Application* comes from the experience of testing a Web application with prior testing experience, but in an environment with extensive structure and disciplines. The learning curve was extraordinarily steep during this project and this is an opportunity to pass along some of the knowledge gained in order to assist others as they encounter the same or similar issues. There are three points that need to be made regarding this effort. The first is the necessity of test plan reviews. Second is configuration (change) control, management and plans, and third, the value of test automation in a Web environment.

First, a little about my background. I have been in Software Development for over 15 years. As stated above, I am accustomed to structured methodologies. Prior to this project test, my experience was writing test cases from design documents, knowing the expected results and documenting them before ever running a test. The results of the tests were verified against the test cases, and a defect was written against either the application or the test cases for any differences. Sounds pretty “text book” right? In order to have this much structure, the environment, the application, and the processes must be predictable. The fact that a Web environment is unpredictable was quite a discovery. This is due to the magnitude of variations possible on one application. There are three points which need to be made: 1) the need for automation on a Web application is much greater than traditional development environments; 2) the need for change control; and 3) the importance of the test plan and the review of it, especially when system documentation is unavailable. If these three points are missed, please email me @ [Jbrock@sqe.com](mailto:Jbrock@sqe.com).

Now for a little about the project. Comparably, it is a much smaller application, with less risk and a smaller development team from traditional projects. The complexity is by no means lower therefore. Logically one would think that the complexity of a project would change based on factors such as lines of code, risk associated with it, and the number of developers working on it. This turned out not to be the case with Web development. Just the fact that it is a Web application increases the complexity of it. Such factors as the infinite number of users (you may be able to argue this point if you want to), the large number of variations, and the number and various versions of browsers, contribute to the complexity of the project. Additionally, each one of these combinations must be multiplied by the number of operating systems and their versions, making the project even more complex.

## Why is Web development different?

There are many factors that go into making Web development different. One of the most obvious is the Netscape and Internet Explorer issues. There is a lack of consistency causing many differences in the behavior to one command. This is probably the most interesting aspect, too. I will discuss this at length later. Another factor is there are 13 main Web servers worldwide, most of which are in the United States. When developing for the Web, your users can be accessing from any type of operating system, browser, version, or modem. You name it. The list goes on. Coming from a defined test methodology background, and being thrown into Web development and testing, it was quite a shock to find all these variables with a reasonably small project. There is no way possible to test everything, just as we always knew was true for more defined development projects. In a traditional development project, you know the operating systems, the versions, and to a point, you know the users. If you don't, you can certainly become acquainted with them reasonably easily. It isn't nearly as easy with Web development. You don't know who your users are until they start using the site.

These factors exist because we are now a global community and we reach out to others throughout the world. It is very exciting and is what we have been working toward. Now we have to learn how to play in this new arena of all operating systems being used for one application, all browsers being accessed, all versions being used, etc. One of the factors eluded to previously is the competition between Netscape and Internet Explorer. It is obvious when developing in ASP (Microsoft product) that they weren't considering Netscape compatibility. The only solution to this is standards for the browser community and Web-based languages. Until that happens we all must be aware of the differences and be ready to respond to them. Before I move on to the next thought, one large factor is the fact that the technology is changing so rapidly you may be developing using a technology which didn't even exist when the browser or the operating system was introduced. I hope most of your users are keeping up with the current technology, but you will find that some are not. Be ready to address the issues when they come up.

Identifying the differences between Web development and traditional development is the first step in being able to address the issues. As taught in structured testing courses, the first thing to do is identify the risks for a project. Knowing all the variables in the project is the first step in identifying the risk. The next step would be to assess the risk. What is the probability? What would be the worst case scenario, and what percentage of our users could have issues, etc.? Keep in mind that there will always be risks associated, and there are many factors which go into this. For example, if you are using frames in your project, the earlier versions of the browsers don't support them because they didn't exist when the browsers were developed.

## How is the development process different?

Speed. The one answer that comes to mind is the time in which things are changing. The technology exists to make these changes quickly, so we are all doing it. It used to be that we would write specifications for what we wanted to do, then would design and code it. We would then look at it to see if it really was what we were looking for. Now, it is much easier and cost effective to just write the code, look at it and see if it is what you really wanted. The code is user-friendly. We aren't working with Assembler, FORTRAN or even COBOL any more. We are using Java script, ASP, and Visual basic. There is a large difference in using these languages versus the "older" languages. Have we given up anything in this evolution? Absolutely, we have given up our structure, our documentation, and our baseline. Is there something we can do about it? Absolutely. We can define our structure, our documentation, and our baseline. It isn't what it used to be, but we can define what it needs to be.

The development process is different because the development tools are different, the users are different, and the scope is different. As mentioned above, the programming languages are very different than they were a decade ago. The users have become computer savvy. Many users who never used to have a computer now have one and can talk bits and bytes with you. The overall scope is different. A decade ago, if you said your user audience for software project was infinite, people would have laughed at you because it was finite. Now it is infinite. All of these factors contribute to the challenge of testing a Web-based software application.

What traditional methods can be used?

There are many traditional methods, which are still applicable, and in many cases more meaningful than before. Risk assessment and identifying the success criteria is very critical with rapid development. By knowing what your risks are, you can manage the project to reduce the risks. By knowing what the success criteria is, you can also manage the project to a successful end. Without knowing either, a saying from Alice in Wonderland comes to mind, "If you don't know where you are going, any road will take you there".

Identify what traditional methods you may be trying to fit into a project that may not work. For example, we began our project writing use cases, We thought this was the answer for all issues. What we found was a lot of time was spent in developing the use cases and our executive sponsor wanted a Web application, not a bunch of paper telling him how it was going to be used. It was a real revelation. We then jumped into developing a prototype and an incremental development methodology. This was an issue we could have avoided had we identified the success criteria up front. Obviously development of use cases was not considered successful. Development of a Web application was within a defined timeframe. After that painful experience, my recommendation is to define upfront the risks associated with the project, and continue to update

because you will discover more as you progress. The importance of defining the success criteria up front can't be emphasized enough. This definition can always be brought back in order to keep the project team (and executive sponsors) focused. Be flexible however, because things will change.....

Prepare for change.

There will be change on the project unless you are working in a vacuum. For example, Internet Explorer 5.0 was in Beta when we started and was released when we went live. This raised some very interesting development issues for us. Since it was in Beta for the majority of the time, we were developing and testing it wasn't static.????? How do you verify the results when the browser is changing also? It probably will be an issue with most Web development. Again, define the success criteria and identify the functionality that must work for your application, and verify that it does. This will require constant verification because you don't have control over the variations of usage.

We changed our methodology as stated above, we changed our players, and we changed our Internet provider. The key is to have a structure flexible enough to accommodate the change without losing too much . You will not have the time to start over.

Adaptability without sacrificing quality is crucial to the success. For instance, you identified that the project must be completed by a certain date. Don't let that change. Another essential factor is to reduce risk. Don't let that be compromised. How you go about accomplishing these goals may change and probably will.

If you find the method you are using is not working, then modify the approach. The methods must be adapted easily, again, without sacrificing the quality of the project. When we changed our strategy from Use Case to an Incremental Development approach, we kept the use cases for reference, built the prototype, and wrote detailed requirements based on the prototype. This may not work on every project, but it worked very well for ours. We were able to see and feel the application, review the requirements, and stay on target. Of course, the requirements changed many times and at times arrived at the original destination. Again, always keep track of the success criteria for the project, and make sure you are anchoring back to it. It is imperative to clearly define what it is and anchor back to it on a regular basis.

This is a great segue into the next thought of identifying what the definition of quality is on the project. It will depend on the Users, the investors, culture, needs, and many more factors. It is imperative to define the quality standard as early as possible in order to work toward that goal. In some applications the viability of links will help define the quality, accuracy will more likely be a key quality standard on financial applications, and the cosmetics will help define the



quality of a graphics site. To summarize, it is critical to the success of the project to define as early as possible, what a successful project is and to keep targeted toward that goal. Not all applications or audiences define quality in the same manner.

Evaluate and communicate the need for Change Control.

Have a process for managing change. Change happens rapidly and has vast effects, without a process and hopefully some tools to support, it will quickly become unmanageable. The risk factors may not be directly financial in nature but they are indirectly from a credibility, reliability and usability standpoint. A defined process for moving code and verifying changes is crucial in this environment. Even items you have previously verified, such as links you may have in your application, can change. Therefore it is critical to the success to have a process for handling changes you are aware of and responsible for, but also for changes you are not aware of and have no control over.

We set up a test server to verify the changes before moving to the host site. We developed an automated smoke test to verify the code upon entrance to this test site. This helped us identify any obvious differences. We then conducted a formal manual test of the application and the changes we were experiencing. We prioritized the change requests resulting from the test efforts, and decided to move or modify the application based on risk, success criteria, etc.

The risk of not having a structured process is extremely high. By making modifications to the site without a validation process, you take the risk of modifying code that will be compatible with one browser version and not others, effecting x-number of users. You take the risk of effecting code in other areas, which you have not covered.

The necessity for control is not limited to the changes going onto the host site. The changes going onto the test site must also be controlled. This step is as important, if not more important, than what is going to the live site. If these aren't under control, it is difficult to really know what is undergoing test. The tests are executing, but what software those tests are running under is not a known factor.

Change control is crucial to the success of a Web project. There are far too many opportunities for failure if not implemented as part of the development strategy. The process doesn't need to be complex or automated. It must be consistent and predictable though. The developers must know and understand the process, then follow it. The organization (or person) responsible for authorizing the changes must understand the variables with Web development. As stated earlier, the application may behave differently based on the browser; the links may no longer work. If the definition of quality for the application includes the viability of the links, then it is important to validate them periodically. This is a process, which under traditional development methods and applications

would not be necessary because all changes were under your control. That is not at all the case in this environment. It is important that the changes, which are under your control, are controlled well.

The need for Automation on this type of project is immense. The fact is that there are many items which need to be verified with each change or on a regular basis, which could easily be automated. The links could be verified automatically, as well as the clicks and the security. These are all time consuming tests to run manually, and can affect the quality of the site if not run. These are also items which need to be tested on a regular basis. The recommendation is to review the application while under development and determine the items which need to be verified regularly. Build a smoke test to accommodate the items, which can be run as the application is turned over for test. This smoke test will then serve two purposes. One, it will determine if the code is ready for testing. If it isn't, it will fail. It will also conduct some necessary verifications which would be extremely time consuming if completed manually. It is necessary to continue the test process even when the code isn't changing due to all of the above factors change around the code and effect how the application responds. For example, the code hasn't changed at all, but there are key factors on the site which have changed. One factor would be the release of a new version of the browser and the links which are on the site changing. By having an automated smoke test to run, you can verify the site is still stable without spending vast amounts of time and energy. In other words, it is money well spent up front to automate this type of test. It will pay for itself many times over the life of the smoke test and application.

Based on the necessity of the automation as stated above, the need for a review of the test plan is critical. The above mentioned smoke test is a test, which will be run many times over. It is crucial that the test plan this is based on is accurate. Imagine that you have an automated test which you, your business, and your customers rely on for accuracy, then you realize there is a flaw in the test plan which all the verification is based on, both manual and automated. As an experienced tester, you know that the test isn't always right. Yes, it is true that sometimes even we can be wrong. Therefore, it is extremely important to review the test plans for accuracy. The project which this paper is based on was light on system documentation. The best way to document how the code was to behave was through the test plans. The plans ended up being turned over to the developers to make the application match the plan. Imagine if those plans had not been reviewed before doing so. A few hours spent upfront are worth saving hundreds of hours on the back end. One of the challenges encountered while undergoing this process was participation. Being the author of the test plan, I was asking for feedback. Many people didn't have time to review it in the detail I was requiring. I had to educate the reviewers on the importance of their feedback. I wasn't looking for spelling or grammar corrections. I was looking for content accuracy. If the expected result was not stated correctly or there was a misunderstanding of how the behavior should be, I wanted to hear about it then.

Once we completed this process, the plans (scripts) were turned over to the developers and we requested them to conduct the tests and make the application work like the plans (scripts). The developers loved it. They had a much better understanding of what the expectations were and how to “get to” the various places on the site.

Here are some additional things to consider when developing and testing Web applications:

- Cookies
- Various Browsers
- Connection speeds
- Identify the users
- Volume
- Bookmarks
- Search Engines
- Operating Systems
- Meta Tags
- ISP's
- Links

Summarizing the above points, Web development is very different from traditional development methods. I hope that some of the information above can be used by some of you in the audience. It was a painful experience the first time through for myself and I would like to help alleviate some pain for future first timers in Web testing. As rapidly as Web development is changing, some of the points may no longer be valid. I doubt that the need for automation, reviewed test plans, and change control will change unless to become even more important for the success of these projects.



## **Automate Your Tests - You Won't Regress It!**

**Stephen K. Allott**

**Imago<sup>QA</sup> Limited  
4<sup>th</sup> Floor West  
High Holborn House  
52-54 High Holborn  
London  
WC1V 6RL**

**Tel: +44 (0) 20 7242 5100**

**Fax: +44 (0) 20 7421 8100**

**e-mail: [sallott@imagoqa.co.uk](mailto:sallott@imagoqa.co.uk)**

**web: <http://www.imagoqa.com>**

### **Abstract**

Test automation is both desirable and possible but there are many pitfalls along the road to success. Rather than learn the hard way, by trial and error, we hope that our experience and history will help you in automating testing in your organisation. This paper describes my experiences as a Test Manager at a software house (Integrated Sales Systems UK Limited – ISS) over three generations of test automation development. The case study described in this paper will be included in a new book by Mark Fewster and Dorothy Graham, “Automating Software Testing”, to be published by Addison-Wesley in 1999.

### **Key Words:**

Test automation, software testing, data driven, case study, test tools.

## 1 Introduction

I was inspired by this quote from George Santayna - “he who does not study history is doomed to repeat it”. As a rookie programmer for ITT back in 1976 I remember quite clearly setting out the tests for my own PL/1 programs painstakingly on paper tape! Little did I realise that, more than twenty years later, the disciplined training I had as a programmer would be invaluable as I wrestled with the very latest CAST (Computer aided software testing) tools.

Thanks to Annette Giardina for reviewing this paper, and to Dan Reid, Iain Ollerenshaw, Richard Hind, and Anthony McAlister for their significant contribution to test automation at ISS.

## 2 The Software Under Test

Integrated Sales Systems (ISS) designs, builds and maintains large-scale sales and marketing automation systems. Our flagship product is called Oxygen™ which is a sophisticated client-server application that provides essential management information for the sales and marketing organisations of large financial and pharmaceutical companies.

Oxygen™ runs on several different flavours of Windows operating systems (e.g. W95, W98, NT4) and supports a variety of relational databases (e.g. Oracle, Sybase) depending upon the customer's requirements. Oxygen has a component-based architecture and is delivered with many additional components, some configured in-house, and others built by third party companies. The "core product" contains the common functionality for all variants of our products; different customers have individualised or standard front ends to the core product. The product is positioned to implement the latest technology as soon as it is available, and often serves as a beta test site for new versions of operating system and development software.

## 3 First Generation

### 3.1 *What was our initial reason to automate?*

The decision to try test automation was made for a number of reasons:

- because we thought it would save time and speed up the testing
- because repetitive testing of our application was dull - in the early days there was no independent test team therefore the project team had to do all of the testing.
- because it is there – i.e. the tools existed and looked a lot of fun; “The time for testing tools is now” [Kit 95-1].

### **3.2 Tool Selection**

A decision was taken at the end of 1995 to purchase five licenses of SQA TeamTest®. This comprised the automated test execution tool SQA Robot® and the defect tracking and test management tool SQA Manager®. We chose these tools primarily for their integration with each other, and their support for testing Graphical User Interfaces (GUIs) on Windows. SQA TeamTest® was also reasonably priced and the supplier (Systems FX, later Cyrano) offered good training and consultancy to help us get started. We did not feel at the time that it was worth doing a formal evaluation of the tool.

### **3.3 From Capture/Replay to Scripts**

As with most companies the first tentative attempts at automating tests relied on using the capture/replay component of the automated testing tool. This was unsuccessful and eventually abandoned. It has been our experience that companies have to go through this stage of capture/replay to see “if it will work for them”. [Allott96 - discussion of capture/replay experiences at a UK bank]. There is a valuable learning experience from this exercise, but it can take a long time. In our first generation, we went through this stage but also moved on enough to see where we could go next.

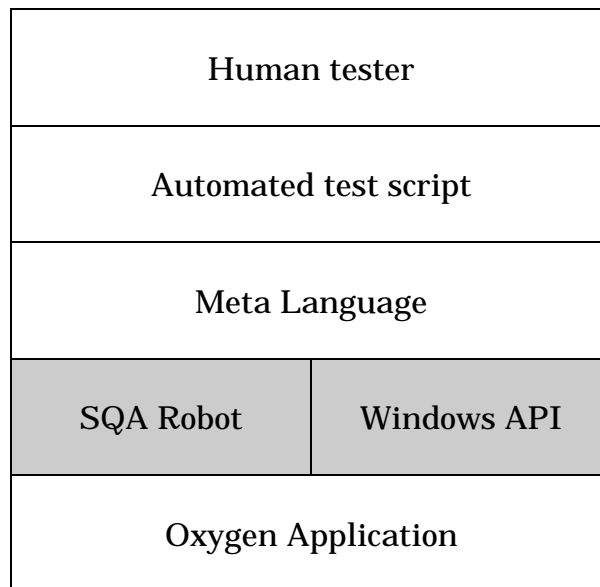
Within our customer delivery area, developers use tools to design and configure the GUI portion of the Oxygen™ application for individual customers. Common functionality often exists in many parts of the system, and needs to be tested in each part and for each customer. We started by capturing these repetitive tests and also the regression tests for updates to common functionality.

The basic commands captured in the test script were then modified to make the scripts more generic, using SQA Basic (the scripting language supplied with SQA Robot). Dan Reid designed and wrote a series of high level functions that enabled complex test activities to be done in a single command. For example a function with two parameters (screen name and wait time) replaced several test tool script commands (navigate to a particular screen, check if it is there, if not wait for it to appear and give it the focus etc.)

These high level functions are much easier to understand, as the complexity of the scripting language is hidden from the developer of the test. Dan also used Excel spreadsheets to hold screen names, wait times and other parameters to drive the tests. The spreadsheets also held the specific data input for each test case.

Dan’s work inadvertently started us towards the development of a meta language i.e. our own test automation language that simply called low level SQA Basic functions to perform the tasks required.

As we built more functions the 'meta language' effectively grew into another architectural layer between the tester and the application as shown in the diagram below. The tester would write the script, which contained meta language statements, which either accessed the tool, or went directly to the application program interface.



We realised that if the meta language were fully developed we would have a powerful automated testing infrastructure. Other products could replace the tool specific layer, if necessary, and testers did not need to know the specific details of each individual automated tool.

We stopped short of actually developing this meta language due to time constraints - we wanted to get something working for the current release and also we wanted to focus on testing our software rather than building test tools. However, we had made some progress and had automated some tests.

### **3.4 Costs & benefits of our First Generation**

License costs for our tool were approx. £2800 per seat and we had five seats. After the initial experimentation Dan probably spent around 3 to 4 man-weeks developing his ideas. So the tool costs were £14,000 (around \$22,500), and the development time roughly £5000 (\$8000).

The main benefit was in taking a tool that no one liked and was difficult to use, and showing how it could be made usable. Our application is very flexible and configurable; every screen, profile and field can be determined by the underlying database. Each customer has a totally different version of the product. Manual testing therefore could be very tedious as similar tests (e.g. on date fields) would potentially have to be repeated on many different systems.

Automating tests by using simple capture replay at the GUI level was obviously a step backwards in technology with our computer generated approach to application development. Add to this the need to test on multiple environments and you see that manual testing is virtually impossible if complete coverage is required.

It was debatable whether or not Dan's time would have been better spent just writing more test scripts rather than trying to get the tool to work in our environment. However, he did demonstrate that it would be worthwhile pursuing these ideas further.

### **3.5 *Problems with our First Generation***

#### **3.5.1 *Automated test tool issues***

1. The automated test tool was difficult to apply to test the core product (common functions).
2. There were a number of problems with the automated test tool, many of which were overcome by work-arounds in SQA Basic.
3. The tool vendor cannot be expected to know your application in detail and so specific problems that we found were often difficult for them to track down. We found that we needed to strip out the application-specific parts of the test to create a script that fails at a simpler and more basic level. Then the vendor has a better chance of fixing the problem.

#### **3.5.2 *General issue***

1. We found that gaining acceptance of the automated testing technology was an uphill struggle. Developers liked the idea of automated testing, but tended to shy away from the tool for two reasons; the learning curve on SQA Robot and the seemingly unsophisticated user interface.

### **3.6 *Lessons learned from our First Generation***

1. You need programming expertise to develop the test automation infrastructure.
2. You need a champion (as with any kind of change) who can manage the increasingly high expectations that project managers and developers have of the test automation technology.
3. The automated test tool technology always lags the development capabilities, at least for our leading-edge applications. However, this should not be used as an excuse to blame the tool vendor and suspend the test automation effort.



## **4 Second Generation**

### **4.1 *Why did we continue with test automation?***

In spite of the problems encountered in our first generation, management decided (July 1997) that it was important to continue on the path of automated testing for two main reasons:

1. Our product had become more sophisticated which led to a higher test resource requirement.
2. With the transition to Windows 95 we were in a legacy situation and now had 12 environments to support (4 different databases and 3 flavours of operating systems).

The challenge for us was to build an automated test tool infrastructure based on the knowledge we had accumulated in the previous two years.

### **4.2 *What we did this time***

A small test team (3 people) was formed, and a new release of TeamTest® was acquired. Iain Ollerenshaw, a Computer Science student from the University of St. Andrews, joined the team as part of his work experience. Iain adopted a data-driven approach to scripting. We chose a Microsoft Excel® spreadsheet as the mechanism for storing the test input data, but we could have easily used parameter or variable assignments within the header files of the scripts. The spreadsheet approach provides more flexibility in that non-programmers can read, sort and otherwise modify the test data. The scripts were coded directly in SQA Basic without using the capture/replay facility to generate the scripts.

To put this approach into practice we started all over again but this time did not try to automate everything [Fewster 97]. We limited the scope to simple navigation and basic record creation tests. Often this is referred to as a breadth test, but we call it a 'smoke test'. This means that we are trying to prove that the application and database build was good on a variety of test environments. Be warned that sometimes a smoke test has to go quite deep within the code to prove that the build was ok. Until the smoke test ran successfully, we knew that it was not worthwhile to commence manual testing on that build.

The tool supplier (Cyrano) provided some initial on-site training in July 1997. We followed this up with a full day workshop where the team experimented with the tool and immediately put into practice what they had learned. During the initial training Cyrano made some very useful suggestions which helped us with our design. One of these was that it is important to have a good structure to the tests and a sensible naming convention for all of the automated test procedures. We were fortunate in that we were able to align our testware naming conventions with that used by application developers.

### 4.3 Our new regime

Each major application component or functional area can be described by a two character code. The application is too functionally rich to show everything but some examples are shown in the table below.

Code	Component	Description
DY	Diary	Standard calendar functionality to record appointments
MT	Meetings	Allows schedule and planning of meetings, approvals etc.
EV	Events	Records events such as sales calls, visits, letters sent, telephone calls etc.
QY	Query	Allows query on many different database fields
CP	Campaign management	Campaign management module

Each component has a main GUI screen (called a profile) and several other profiles depending upon the complexity of the particular component. Our test procedures were created in a three tier hierarchy as follows:

- Main test procedure
- Test Activity
- Test functionality

We created one main test procedure for each component. Each of these test procedures could be run on its own and would eventually fully test the component by calling one or more test activity procedures. We identified four possible test activities namely:

- Navigation
- Creation
- Editing
- Deletion

Each test activity procedure would in turn call one or more individual test procedures that ran a test for a specific area of the component's functionality.

We have used the example of the diary functionality to illustrate our naming convention in detail.

Test Procedure Name	Type	Description
DY	Main	Main test procedure for the Diary component
DYN	Activity	Navigation test procedure for the diary
DYC	Activity	Creation test procedure for the diary
DYE	Activity	Edit test procedure for the diary
DYD	Activity	Deletion test procedure for the diary
DYCWK	Test	Tests that create diary items in 'week to a page' mode – note specific tests were called DYCWK01, DYCWK02 etc.
DYCID	Test	Tests that create diary items in 'day to a page' mode

The tool allows you to create new shell scripts in order to create a particular mix of the individual test procedures. This works fine but was very tedious to use when several rapid changes were required. We built a very simple front end using Microsoft Visual Basic® that allowed us to pick and choose specific tests that we required in a particular run.

A common test procedure read in the test input data from the spreadsheet into a global array that could be accessed by all subsequent test scripts. This was done for performance and programming efficiency reasons.

When, for example, the diary create test procedure was executed, it read its input data values from the diary create test input spreadsheet. There would be one row in the spreadsheet for each diary item we wished to create. Each column in the spreadsheet would represent a particular data field on the appropriate 'create diary item' profile. For example, date, time, item type, summary text and so on. We also had a column in the spreadsheet to indicate whether or not the data was valid (i.e. success would mean that the application should correctly create the diary item record in the database) or invalid (i.e. we expect an application error dialog box to pop up). The tests were sensitive to the valid/invalid column in the spreadsheet and would take the appropriate action.

Note that we also had another 'application configuration' spreadsheet that held the 'meta data' to describe various aspects of the configuration to the tests. This would include, for example, path names, locations of results directories and log files, locations of specific test scripts.

In our opinion there are three types of test data:

1. The meta data to configure and run the automated test suite
2. The test data itself; inputs and expected outputs for each test case
3. The test database; a populated baseline on top of which the tests will be run

One thing that we soon learned was the importance of being able to get back to a baseline situation. For example, having created several diary items with one test procedure, we would then run a test that deleted them all. This allowed the diary create test to be run again under its original circumstances. We knew that ideally we should have rolled back the whole database but this was not practical at the time as several testers were sharing this test environment.

The test suite that Iain built was run daily for about 4 weeks on a number of different test environments. The primary purpose was not, as you might expect, to find bugs, but to demonstrate confidence in the daily builds of the Oxygen application.

#### ***4.4 Costs & benefits of our Second Generation***

Iain spent almost 6 man weeks developing and running the automated smoke tests. The tool cost was the maintenance contract with the vendor or around £2000 (\$3200). Iain's time would have been around £7500 (\$12,000).

This new approach proved to be effective in both achieving and proving the stability of our application. This in turn allowed us to concentrate our main effort on manually testing the new functionality. The automated test suite did find some new bugs, although this was not its primary benefit.

There was a psychological effect that we started to see here. If developers knew that their software was going to be run through automated test software by another developer, they might put in an extra effort to make sure it would not fall over. Without Iain's smoke tests we would have potentially wasted several man-weeks of effort in just proving the stability of the builds.

As a result of developing our automated smoke tests on this latest release of software and demonstrating their value to others within the company we changed people's attitudes towards this technology. We anticipated that we would be able to do more next time as a result of this success.

## **4.5 Problems with our Second Generation**

### **4.5.1 Automated test tool issues**

1. It was difficult to get object properties into script variables.
2. Unexpected window handling could be improved.
3. Our application toolbar was treated as a single object by the tool and so we could not click on any toolbar buttons and we had to rely on menu selections and accelerator keys.
4. Identifying certain objects on the screen was difficult with the version of the tool that we had.
5. Using mouse clicks to drive tests was unreliable and so we tended to favour access using menu options, accelerator keys and tabbing to the appropriate fields. This was troublesome at times, for example when there were accelerator key clashes or when there was no alternative to the mouse operation.
6. Sometimes but not always the automated test suite stopped at a 'menu select' operation. We avoided the problem by using a short delay prior to the menu select command in the test script.

### **4.5.2 General Issues**

1. Differences in performance between Oracle & Sybase versions of our application on different platforms meant that when the tests were moved from one database to the other, slight modifications had to be made to any delays or wait state events in the tests.
2. Our application sometimes set the tabbing order to tab to protected fields i.e. read only fields. This meant that to enter data in a field following the read only field we had to tab twice to access it; if this feature is changed then the test will require modification.
3. Application changes that the test team were not aware of caused some confusion, for example when the main menu item called "system" was changed to the more widely used and recognised "file" and also when the main menu caption name was changed. Both examples caused the automated test suite to fail. This was not a technical problem, it was a communication problem. The developers were unaware at the time that their seemingly minor changes would affect the automated test suites.
4. If an automated test found a minor bug, we could never guarantee that it would be fixed before the next automated test run. We need to keep the test in the suite so that when these minor bugs are fixed we have a complete set of tests. Until the bug is fixed, however, we will get a failure report on our test log when we run this test suite.

5. During development of the automated test suite we used a version control system to keep track of different versions of the test scripts. The test tool did not support a similar feature for its test procedures.
6. We had bugs in some of the automated test scripts. We soon realised that we needed to set up a test environment to test new versions of the test scripts and associated Excel spreadsheets.
7. We had still not gained acceptance by our developers of the automated test technology. This is unsurprising given the number of problems we had with the tool.

#### **4.6 *Lessons learned from our Second Generation***

1. We found a few bugs with the tool which, although minor from our perspective, the fact that they were found automatically impressed the other managers.
2. We realised that it is a lot more difficult than we thought at first to build an automated testing infrastructure.
3. We also learned to be careful not to oversell the technology. When our managers see the application starting up on its own and running through a sequence of 'tests' they can form an impression that it is doing a lot more than it really is. This has to be carefully managed so that they continue to support the project but give adequate time to do a proper job.
4. Once we had developed the tests, they had to be 'released' along with the application software and installed on the target test environments. When we found bugs in our test scripts we had a dilemma; where do we re-test the fixes to the test scripts, how and when do we apply version control?

## **5 Third Generation**

### **5.1 *Designing a new Infrastructure***

By now the company had grown to a size where it was appropriate to establish a larger independent test team (6 – 8 people). A new recruit to the test team, Anthony McAlister, worked almost full time on test automation from January through March 1998. He was able to build on the work done by Dan and Iain and successfully ran the original automated tests as a regression suite on our 3.3.1 release. Also, a bug fix release (3.3.2) was tested using these tests. It is notable that Anthony had no formal training as a tester until he joined ISS but was experienced with Windows 95 and was a recent Computer Science graduate.

However, Anthony experienced several problems in getting to grips with automated test technology, and it turned out that the tests were still not robust enough for the new 32 bit version of Oxygen™ (known as release 4.0). Therefore we decided that a fresh look at the problem was needed and so the third generation approach was born. We enlisted the help of Grove Consultants (Mark Fewster) in a design review of our proposed 'Third Generation' test automation infrastructure.

Mark suggested that the directory structure for the test infrastructure should map that of the development environment. He also participated in a design review of the new automated testing infrastructure and test suite. The basic approach was that we would design and program each of the automated test scripts that we required using SQA Basic or Visual Basic as appropriate. The code would be as generic as possible.

Richard Hind, our Senior Test Engineer on the team, proposed that we break down the application functionality into just three types of test activity. Furthermore he suggested that each of these test activities be further subdivided into test automation levels according to the degree of complexity involved in developing the test script.

### **Characteristics of Level 1**

A Level 1 test would exercise the simplest aspect of the functionality of a particular module. It has the following characteristics:

- It is usually straightforward to test manually
- It is easy to automate
- The automated test is likely to work
- It is unlikely to find a new bug

### **Characteristics of Level 2**

Level 2 tests explore all aspects of a particular module except those that require interfaces to other components. Level 2 tests have the following characteristics:

- It is possible but time consuming to test manually
- It looks easy to automate, but doesn't always turn out so
- The automated test is likely to have bugs
- It sometimes finds a bug

### **Characteristics of Level 3**

These tests will exercise the deepest level of functionality in a module including those which interface to other components. They have the following characteristics:

- Difficult if not impossible to test manually
- Hard to automate
- Unlikely to run successfully, repeatedly
- Very likely to find a bug

### **5.2 Estimating the effort involved**

For each major component or functional area of the application we identified which levels were appropriate for each of the three test activities. This matrix then provided us with the basis of an estimating tool for the test automation project. Level 1 tests we estimated would take approximately a half day of effort. Level 2 tests would take two to three days and Level 3 tests would take a week.

<b>Functionality</b>	<b>Level 1</b>	<b>Level2</b>	<b>Level 3</b>
Navigation			
Edit			
Query			

Simply selecting a menu and clicking OK was the easiest form of navigation and could be considered Level 1. Navigating several profiles deep and then clicking on the right mouse button to pull up another menu might be Level 2. Trying to navigate to areas of the system provided by dynamic context sensitive menus would be Level 3. A similar exercise was carried out for edit and query test activities. Note that edit includes creating database records, modifying them and deleting them.

During the estimating phase, we realised that it was not practical to automate everything as we would have to spend too much time and effort to realise a good payback. However, if we settled for regression capability and 'proof of build' then we could quickly automate using Level 1 tests. A compromise was reached and we built all of the Level 1 tests, some Level 2 tests and a few Level 3 tests.

### **5.3 Building the Infrastructure**

During implementation of the third generation approach, Anthony converted many of the old test procedures into SQA Basic scripts. However, he still experienced several problems with the scripting language and discovered some more limitations of SQA Robot. Therefore,



since SQA Robot will allow more than one scripting language, we eventually decided to use Microsoft Visual Basic® (VB) to continue development of our automated test infrastructure.

#### **5.4 Scope – Included**

We built VB code to perform the following general functions:

- Send keys (e.g. ALT F, but not CTRL ALT DEL yet)
- Interrogate menu structures (using resource files)
- Send messages to W95 or NT (click button, etc.)
- ODBC (open database connectivity)
- Log results to a window/file

These functions were kept together in a single VB module (GENERAL.BAS) We then constructed additional VB modules for navigation (NAVUTIL.BAS), data creation (CREATE.BAS) and editing/query (EDIT.BAS) as well extending our simple VB front end which allows selection/execution of the tests.

#### **5.5 Scope – Excluded**

When running an automated test procedure, SQA Robot automatically logs bugs into the defect tracking system, SQA Manager. This was a useful feature of the tool. However, we chose not to develop this interface in our tool. Experience had shown that investigation and analysis was required prior to determining if the issue was truly a defect. Therefore we saw limited benefit from this feature. For example, the problem might be with the automated test script or the test data or some combination of the two rather than with the application. If we logged a bug every time an automated test script failed we would have alienated developers from the technology.

#### **5.6 The Tests**

The design of our automated test infrastructure allowed us to build the following tests of our application:

- Simple navigation within the application
- Exercise all combinations of menus
- Creation and subsequent modification of basic records
- Ability to query the database

The software uses CSV (comma separated values) files as input for each test case and initiated SQL queries to determine that the records have been successfully added to the database. Note that SQA Robot could have performed an equivalent function. However, we could not implement this because of ODBC clashes between the SQA repository and our application database (both used Sybase SQL Anywhere).

We focused on tests directed at our primary application functionality and have excluded (for the moment) testing interfaces to external and third party software (e.g. mail, mail merge, communications, reporting, OLE interfaces).

### ***5.7 Costs & benefits of our Third Generation***

We estimate that we've spent about three man months on our third generation efforts, around £15,000 (\$24,000). Benefits have been substantial in terms of proving the daily software builds and proving the test automation infrastructure. A few new bugs were found by the tool.

There is often debate in the testing industry as to whether or not all of the development effort for automated testing is worthwhile. Some experts suggest that we would be better off spending the time building more manual tests. However, with modern GUI based client/server applications running on several operating system platforms on different databases we believe there is no real alternative to test automation. The payback is not a true financial saving, rather it is a payback in terms of improved quality, quicker time to market, more satisfied customers.

### ***5.8 Problems with our Third Generation***

#### ***5.8.1 Automated test tool issues***

1. The automated test tool did not keep up with the application development technology. We feel that this would have happened whatever tool we had used; for obvious reasons the tool technology is always likely to lag behind.

Specific examples are:

- It could not properly handle 32 bit application menu structures especially those which are dynamically generated such as context sensitive menus.
- There were various new Windows controls (treeview, toolbar) that it did not recognise and for which it provided no workaround.
- It found some of our application modal dialog boxes difficult to identify especially when several (without captions) were presented in sequence.

2. Sometimes we could not trust the data written back to the logfile. We had situations where tests had failed but they were logged as having passed. We were never able to properly explain this.
3. Sometimes tests would stop and the tool would hang for no apparent reason. The tool appeared to have a personality of its own and was somewhat temperamental. You really needed to get a feel for how it worked and coax it into action where necessary. Anyone who has used an automated test tool will understand exactly what we mean!
4. There was no 'fuzzy logic' at all in the tool; everything had to be programmed down to the last upper case letter. Although there was a feature of wildcarding in the script language, if you wanted to select the File menu you had to set the case correctly; compare this to a human tester who'd have no problem if the manual script asked him/her to choose the file menu (in lower case) and not the File menu.
5. The tool was not good at conditional processing. For example, a database query may present you with an additional modal dialog (to enable you to restrict the search) if you attempt to select too many records. A manual tester will obviously react correctly even if they have never seen this additional dialog before. The automated test script did have conditional logic to process this dialog if it appeared. The problem seemed to be that when it did appear, the window appeared too quickly for the tool to be able to identify it correctly and the test subsequently failed. As a workaround we configured our database to ensure that the window always appeared or never appeared so that the test could be written without the conditional logic.
6. The tool used an SQL database and accessed this via ODBC. So does our application. Therefore we could not use SQL utilities to interrogate or modify the database whilst we were running the tool.
7. We used PVCS to control the application code versions. The way we had structured our test repositories, everyone had to run the same automated tests. The tool did not support version control. We could have structured our tests as two projects, one for the new version and one for the old, but we would have needed manual version control.
8. Unexpected events were not handled very well. When testing, you expect error windows to appear – they're unexpected yet anticipated as generally one of the reasons for running the tests is to make the system fail and generate the error window. Your test tool logic should be able to cope with this. However, an unanticipated unexpected event sometimes happens such as a debug window appearing or a message from the system administrator. The tool did not handle this concept at all well. All that the tool could do was to send an escape key or other

single character to any unexpected window or dialog box that it did not expect.

### **5.8.2 General Issues**

1. Again managing the expectations of others was important as we were not making as much progress as we would have liked.
2. We should have allocated more time to this test automation effort and run it as a project on its own, rather than try to fit it in as part of the test effort for the 4.0 release.

### **5.9 Lessons learned from our Third Generation**

1. Test automation needs to be funded and managed as a proper internal development project.
2. Failing to properly define requirements and skipping the design stage can lead to abandonment of the technology.
3. There will be a maintenance cost of the infrastructure we have built.
4. Automated test technology is not yet mature and the champion needs to be aware of this and plan accordingly. Quick wins and incremental improvements, demonstrations to management are all ways to gradually gain acceptance of test automation.
5. Our new automated testware is very portable. No installation is required – we just ship a VB.EXE wherever it is required. There are no significant training requirements (other than in test design of course).
6. We now have much tighter integration between our Visual Basic® automated toolset and our Oxygen™ application.
7. Developing our own test suite using VB has made us more aware of timing issues and we have to try and make our testware behave more as a human tester would. With the tool we'd often just fire off a command to the application and expect it to cope.

8. We were able to abandon test hooks in the application. The tool needed them but since we develop tests and code together we're always aware of what the application is doing or about to do and so there seems to be little point in building test hooks as the testers are much closer to the developer.

## 6 Summary and Recommendations

### 6.1 Characteristics of our three generations

The table below shows the characteristics of our three generations from 1995 to mid 1998. Note that it is not only the development environment that changes; the testware has its own life cycle too!

	<b>1<sup>st</sup> generation</b>	<b>2<sup>nd</sup> generation</b>	<b>3<sup>rd</sup> generation</b>
Application code	16 bit – pre-3.0	16 bit 3.2 / 3.3	32 bit 4.0
Controls			Treeview Toolbar
Operating Systems	Windows 3.11 Windows NT	Windows NT4 Windows 95	Windows 95 Windows OSR2 Windows NT4 Windows 98
Web Browser	N/A	IE3	IE4
Databases	Sybase SQL Anywhere	Sybase SQL Anywhere Oracle Personal Oracle	Sybase SQL Anywhere Oracle Personal Oracle
SQA Robot	4.0	5.1	6.1
Defect database	Paradox	SQL Anywhere	SQL Anywhere
Automation approach	Capture replay	SQA Basic Data driven Spreadsheets	Visual Basic Data driven Resource Files

### 6.2 Long Term Strategy

We would like to be able to design our tests so that after a successful single environment manual run, all other platforms / environments will be tested automatically. To accomplish this we need to enhance our test automation infrastructure so that it can run on different operating systems and support several database variants without any fundamental changes.

We will have three layers to our test infrastructure:

- 1 Infrastructure Layer – General functions (must be platform independent)
- 2 Application Utilities - Navutil, Create, Edit – application specific
- 3 Test Suite - scripts and spreadsheets to implement the test cases

We had realised that as well as the automated test scripts themselves, there is an awful lot of code needed behind the scenes to make this all run in an efficient manner. This includes:

- A front end to enable pick and mix selection of tests
- A mechanism for logging errors
- Version control
- A scheduling mechanism to enable unattended testing

The test suite will be structured around application functionality which will call various utility scripts in the application layer. For example, Meetings, Campaigns, Diary will call the appropriate lower level utilities (navutil, create etc) that will themselves call the general functions to push buttons, click on windows etc.).

In our opinion the technology of test automation must be integrated with the methods used for test planning & design of the manual tests. These have primarily been paper based and include, for example, test plans, test specifications, and checklists. Automated tests will never totally replace manual tests. However, the management of these test assets should be via a database solution rather than based on paper documents.

We envisage a strategy whereby bug system, coverage tool, manual and automated tests and the results will all be linked together by either home grown or bought in third party test asset management software.

We continue to move forward with test automation. Our products are expected to run on an increasing diversity of operating systems and databases. Furthermore as we incorporate third party components into our product we will have to test more communications links and interfaces.

### **6.3 Recommendations**

1. The champion must keep going at all times when introducing automated testing. The champion has to accept that change management does not happen only once but at every stage of the tool's use. The champion also has to be prepared to handle setbacks and overcome doubts caused by them. The champion must continue to sell the ideas and benefits of automated testing at all times and manage other people's expectations carefully.
2. Make sure the rest of your testing process is reasonably mature before you start to automate.
3. Before you buy an automated test tool first consider your requirements. [KIT 95-2].
4. Evaluate a test tool in your own environment. Do not rely on the sales pitch or vendor demonstrations. Use an evaluation copy on a real project.
5. Consider carefully where in the project life cycle and test process that you will use test automation. At unit test, integration test or system test stages? For regression or performance testing? We have found that the best use of test automation is where we want to prove a new application build will run on a variety of test environments. This actually saves loads of manual testing effort and allows the team to find the deeper, more serious bugs. Although this is not yet true regression testing, it is headed in the right direction. More importantly it builds confidence in the concepts of automated testing.
6. Design your tests before you automate them. Remember that bugs are found by the tests, not by the tool, and that sometimes the act of attempting to build the test finds a bug.

### **6.4 So is this a success story?**

We have certainly gained much by automating testing. We are able to test basic fundamental things very easily, but we certainly have not automated all of our testing. At least now we realise the limitations of test automation and can use it where it will be of most benefit to us.

Every advance we have made seems to have generated more and deeper problems. We seem to have done "the right things", yet we still encountered many technical problems, perhaps due to being overambitious when trying to automate all components of our leading-edge application software.

Although we started off intending to use a commercial tool, we have moved almost completely away from that to our own hand-crafted automation using Visual Basic as our test tool scripting language.

Test automation has had a chequered history at ISS. It is still not fully accepted by the developers and management has yet to see tangible benefits. However, our level of test process maturity has improved such that we are ready to face new challenges. We are now working more closely with our developers to define requirements for future test automation projects.

Firstly we will attempt to port the current technology and test scripts to version 4.1 of Oxygen. Secondly, if this is successful, we will design and build a complete regression test pack for the Oxygen application. Rather than treat automation as part of a software release activity, both these projects will have a separate project manager and be funded and managed exactly as any other internal project.



## 7 IF . . .

Another source of my inspiration for continuing as test automation champion can be found in Rudyard Kipling's famous poem 'IF' :

If you can keep you head when all about you  
Are losing theirs and blaming it on you,  
If you can trust yourself when all men doubt you,  
But make allowance for their doubting too;  
If you can wait and not be tired by waiting,  
Or being lied about, don't deal in lies,  
Or being hated, don't give way to hating,  
And yet don't look too good, nor talk too wise.

If you can dream - and not make dreams your master,  
If you can think - and not make thoughts your aim,  
If you can meet with Triumph and Disaster,  
And treat those two impostors just the same;  
If you can bear to hear the truth you've spoken  
Twisted by knaves to make a trap for fools,  
Or watch the things you gave your life to, broken,  
And stoop and build 'em up with worn out tools.

If you can make one heap of all your winnings  
And risk it on one turn of pitch and toss,  
And lose, and start again at your beginnings  
And never breathe a word about your loss;  
If you can force your heart and nerve and sinew  
To serve your turn long after they are gone,  
And so hold on when there is nothing in you,  
Except the Will which says to them: "Hold on!"

If you can talk with crowds and keep your virtue,  
Or walk with Kings - nor lose the common touch,  
If neither foes nor loving friends can hurt you,  
If all men count with you, but none too much;  
If you can fill the unforgiving minute,  
With sixty seconds worth of distance run,  
Yours is the earth and everything that's in it,  
And - which is more - you'll be a Man, my son!

By Rudyard Kipling.

## 8 References

Allott S. 1996. Automated Testing - Is this a kind of magic? Presentation to the EuroSTAR 96 conference, Amsterdam, 2<sup>nd</sup> - 6<sup>th</sup> December 1996.

Fewster, M. 1997. Common mistakes in automated testing, presentation to the British Computer Society Special Interest Group in Software Testing, June 1997.

Kit E. 1995-1. Software Testing in the Real World pp4-5 & pp152-153, Addison-Wesley, ISBN 0-201-87756-2

Kit E. 1995-2. Software Testing in the Real World pp118-125, Addison-Wesley, ISBN 0-201-87756-2

Steve Allott, September 1998.

## **1999 Pacific Northwest Software Quality Conference**

### **Fact or Fiction: Developers Get all the Cool Stuff**

**Presented by  
Ingrid Canavan  
Rational Software  
(503) 708-2097  
[Icanavan@Rational.com](mailto:icanavan@Rational.com)  
[WWW.Rational.com](http://WWW.Rational.com)**

It seems that developers have all the cool tools to make developing applications easier and quicker. What used to take weeks to write or rewrite may now take only minutes. QA must keep up with development and manual testing is not enough. Today QA has access to a wide range of tools to make testing faster and easier and thereby improving the overall quality of the application. But which ones will have immediately impact? What about the long term? Where are the alligators? How about that Internet? Where do you start?

#### **Biography:**

Ingrid Canavan has a solid industry background in QA, test and test management. She is currently a SW Engineering Specialist with Rational Software Corporation specializing in tools for the QA professional. From 1996-98 she provided expert QA consulting for Client Server Group. Prior to 1996, she was QA and Technical Support manager for First DataBank, responsible for 36 shrink-wrap products, seven toolkits and three knowledge acquisition tools on seven operating systems.

# Communication Paradigm For Distributed Testing

## Abstract

Communication of bugs and test cases among distributed testers has always been a confusing, time consuming affair. To improve such communication among distributed testers, a language framework was utilized providing two tools; one for expressing and capturing test actions at the transmitting end, and the second to interpret and play them back at the receiving end.

On the transmitting side, a test sequence generator presents the tester with a set of permissible actions as he is led from program invocation to exit. His responses are constrained to specific choices, which are codified to form a test sequence. The user can then transmit the test sequence to the respective distributed testers. At the receiving side, the test sequence is played back by the distributed tester with the aid of a test sequence processor; reducing the level of human interpretation.

This communication paradigm could facilitate precise, understandable, repeatable, editable and archivable communication of bugs and test cases.

### **MAHESH BALACHANDRAN**

**Quality Assurance Specialist**

**Micrografx, Inc. — 7585 SW Mohawk, Tualatin, OR 97062**

My major responsibility is to develop test automation for our process visualization and simulation product. This involves extending the reach of existing testing practices by creation and implementation of automated test plans, test utilities and metrics. Worked as Software Testing Engineer with Saltire Software, Inc. from December 1996 to March 1999.

#### **Education:**

B.Tech. Mechanical Engineering from Indian Institute of Technology — Madras, India (1985)

MS Environmental Science and Engineering from Oregon Graduate Institute — Portland, USA (1996)

### **DAO HOANG**

**Software Testing Engineer**

**Saltire Software, Inc. — 12700 SW Hall Blvd., Tigard OR 97223**

My main responsibility is testing the algebra system and other products which consists of generating automatic test sequences and running test sequences, writing test scripts for testing the UI of a product, documenting test sequence coverage as well as the whole testing process, writing and modifying verification functions, reporting bugs to the developers and verifying fixed bugs in the bug database. I am also heavily involved with the larger quality assurance process, of which, the software testing is an important part.

#### **Education:**

AA degree from Clark College — Vancouver, WA (June, 1993)

BS degree from Portland State University — Portland, OR (June, 1997)

# Communication Paradigm For Distributed Testing

**Mahesh Balachandran,**  
**Dao Hoang,**

**Micografx, Inc.**  
**Saltire Software, Inc.**

## 1. Introduction

### 1.1. Problem

In today's software testing environment, it is common to pool testing personnel from distributed locations. These remote resource locations or nodes communicate with each other and with a central office which acts as a clearing house or hub. Problems arise in the following areas:

#### 1.1.1. Understandability

Communicating details of bugs, application enhancements or clarifications about appropriate program behavior cannot be completely satisfied by traditional written and oral communication. While natural language is rich enough to succinctly convey a wealth of descriptive information, we users of the language seem to fall short in the conveyance of precise, unambiguous facts. The interpretation of language, even within members speaking the same native tongue, is fraught with lapses in understanding. The problem is compounded when communicating with people whose native tongues are different.

#### 1.1.2. Precision

Some test cases require a detailed set of steps with extensive data input. The test sequence writer is prone to making errors while writing the test script. Moreover, it is time consuming for the person interpreting the test script to replicate the steps while re-creating the test case. The chances of error would increase the more detailed the script gets.

#### 1.1.3. Retention

Unless the test script is automated, the manually generated test script would have to be recreated by hand and re-created every time the test needs to be run. Modifications to the script are very laborious to effect.

## 1.2. Solution

To address the problems above, we recognized the need to devise a language framework that would be understandable to the transmitter and the receiver of bug and test cases communication. This language would consist of tokens representing permissible user actions and data objects in the application. By restricting communication to these tokens, and with implicitly enforced sequencing of these tokens, understandability would be enhanced. Tokenization or codification of the responses lends itself to machine interpretation and consequently to auto playback and archiving. The two components of this language framework would be capture user actions and playback user actions.

### a. Capture User Actions

A GUI driven test sequence generator was designed. The user is presented with a dialog box listing a set of possible user actions available at program invocation(based on the specific resource and time constraints at the company in which such a system is implemented, one may have to judiciously restrict the set of permissible actions at each step). The user is requested to choose which of these actions he would like performed. Based on his response, a token would be

generated and stored in a text file. Based on the specific user response, the next set of possible user actions would be presented to the user requesting a response. If the particular user action warrants input data, the user can specify a data file or write in a string in a prescribed format. The user would thus be guided through a tour of the application, allowing him to choose responses along the way. At the end of the tour, he would be given the option of entering another test sequence or to quit the test sequence generator. The set of all choices made by the user would thus be captured in the text file and would constitute a unique test sequence.

#### **b. Playback User Actions**

A test sequence processor was designed for this purpose. It consists of all the functions needed to interpret the test sequence, accept data where entered, execute user actions, perform verification where ordered and log test results to a log file. It would prompt the tester for the location of the test sequence file. It would then step through each of the text codes in the string, call the appropriate functions and data and perform the requisite actions.

Created within a programming environment like Visual C++, editing, debugging and archiving of these test sequences could be accomplished.

### **1.3. Benefits**

An unambiguous communication tool is thus developed enhancing understandability, repeatability, precision, efficiency, playback, recall, and archiving.

## **2. FloorRight™ as a Case Study**

The distributed testing communication paradigm was prototyped and tested at Saltire Software, Inc. by the authors. The key components of the system, the test case processor and the test case generator were the brain-child of our supervisor, Robin.J.Y.McLeod. These components form the heart of the automation efforts at Saltire Software and form the engine of a software-usability testing tool. However, this paper will focus on the integration of these components into the prototyping of the distributed testing communication system. The application chosen for the prototype was Saltire Software's product, FloorRight2.0™.

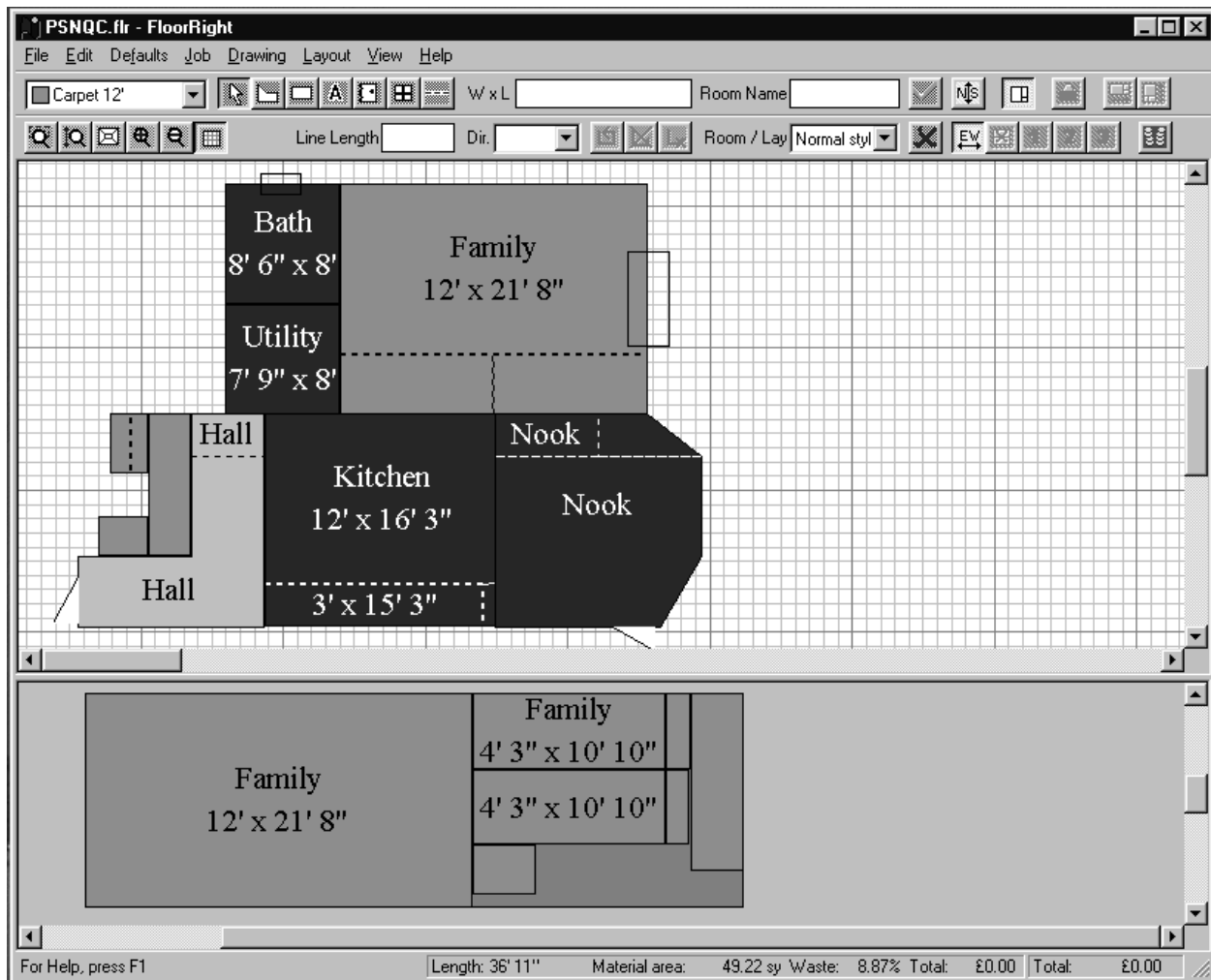


Figure 1. Sample of FloorRight™ user interface.

## 2.1. Description of the Application

FloorRight™ 2.0 is a Windows-based application to estimate floor-covering material like carpet, vinyl, wood/laminates and ceramic. Information on room dimensions or room specifications is entered, the carpet ply orientation is specified, the maximum allowable seams per room, other layout properties and the type of material in each room are assigned. The application then displays each of the rooms with the locations where the seams should appear. The way the stock should be cut is also displayed. A bill of materials with pricing information is also computed. The layout computation process can be roughly divided into five stages.

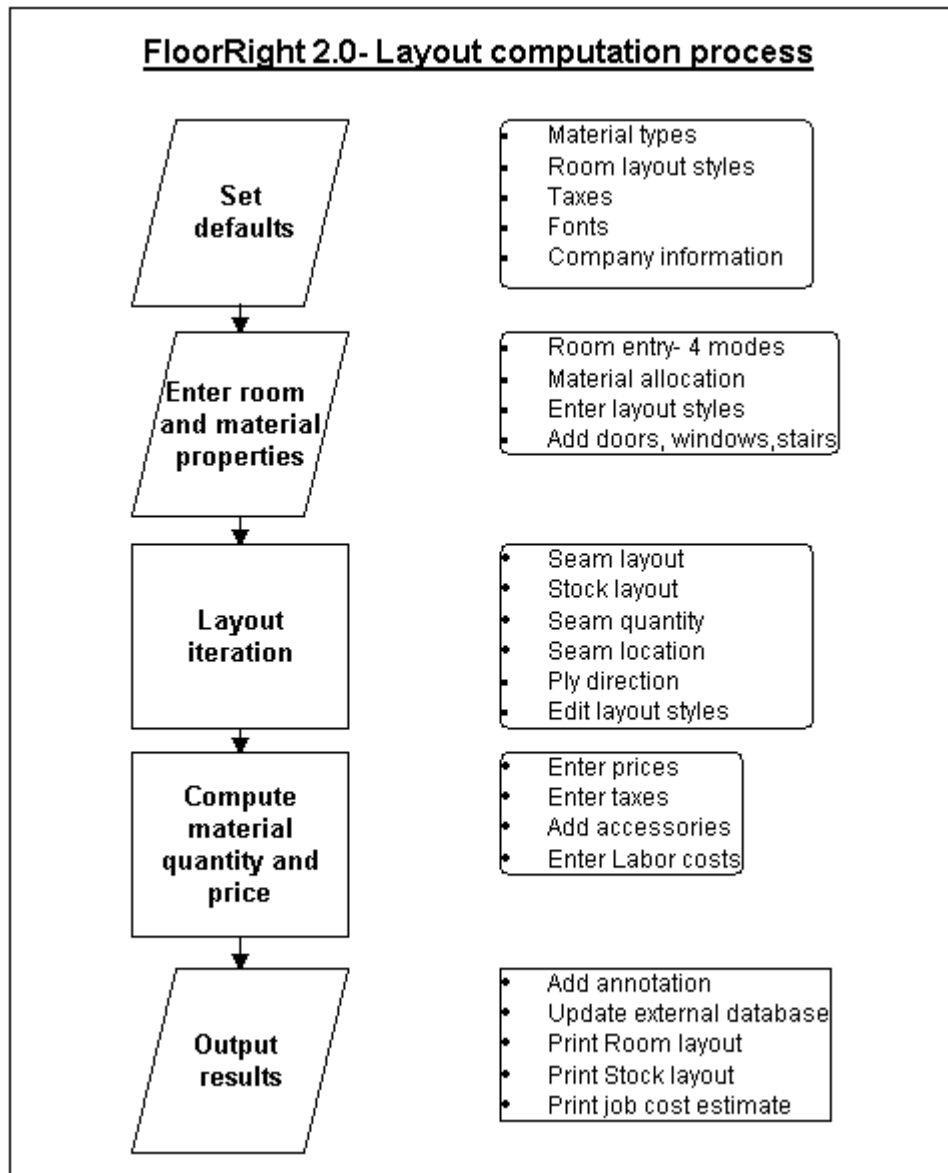


Figure 2. FloorRight™ Layout Computation Process

### 2.1.1. Set Defaults

At this stage, the user defines templates for materials and the preferred method of laying them out. He enters data most commonly used in his day to day operation. These include:

#### a. Materials:

The materials most commonly sold along with their properties like roll width, material name, units-of-measurement, and pattern dimensions if applicable

#### b. Layout Styles:

Create templates for rooms based on the layout style. Layout styles pertain to the number of allowable seams in a room, the permissibility of seams in hallways, or whether T-seams within T-seams are to be allowed.

#### c. Taxes:



Pertinent taxes applicable to materials and labor for the different regions

**d. Font:**

The preferred setting for text appearing in the application and output

**e. Company information:**

Name and address of the company as the user would like it to appear on the printouts

**f. Options:**

The units-of-measurement of room dimensions and area of carpet required, , and setting of global parameters for the layout algorithm and the screen appearance

### **2.1.2. Entering Room Dimensions, Materials and Layout Styles**

At this stage, the user is ready to enter data applicable to the job at hand. These would include:

**a. Drawing and Dimensioning Rooms**

There are four modes of drawing rooms: Using the rectangle tool, the rectangle edit box, the line draw edit box and the line draw tool.

**b. Assigning Materials**

There are two modes for assigning materials- a “Quick entry” mode and a “Detailed mode”. Materials can be newly created or a previously defined material can be chosen.

**c. Assigning Layout Styles**

Similarly layout styles for a particular room can be created new, or a previously defined laying style can be chosen.

**d. Stairs, doors and windows**

Tools to facilitate entry of door, windows and stairs are provided.

### **2.1.3. Layout Iteration**

At this stage, the layout algorithm is invoked, which gives an initial estimate of the seam layout and the way the stock will be cut. The user has the option of eliminating specific seams, or to flip their positions. He could also have the algorithm suggest two other layout configurations. If there are specific rooms whose layouts he wishes not to have changed while the algorithm offers the other two option layout options, he can have them locked. The user at this stage may want to redefine the layout options for the rooms and continue with editing the layout until he is satisfied with the seam orientation and the quantity of material required.

### **2.1.4. Pricing**

The user opens the “Job cost estimate” dialog to enter prices for each of the materials. Associated items like pad, adhesive, accessories, pre and post laying work and labor are entered with their process. Taxes as applicable are entered here.

The user can access an external carpet inventory database to import specific materials and their prices. Once the pricing is complete, he can export the computed material quantities and invoicing information to the external database for further processing.

### 2.1.5. Printing

The user can print out the following:

- a. Rooms with seams showing
- b. Rooms without seams showing
- c. Stock layout
- d. Job cost estimate

Notes to go along with the printouts can be added.

## 2.2. Algorithm

The problem of fitting rooms with pieces cut from a roll of finite width and length is a classic optimization problem. To date, there is no analytical solution to optimally complete the process. Nor does it lend itself to any of the numerical optimization methods. The cost function is discontinuous and there are both non-linear and discrete constraints. The methods used for bin packing and roll-cutting do not apply here. The method of attack for problems in this class is to apply heuristic solutions based on the body of knowledge already available with carpet laying experts. Other mathematical considerations, quite outside the scope of this paper, are applied to seek “good” solutions

## 3. Components of the Distributed Communications System

There are two main components of the communication system at a single node. These are:

- a. Test Sequence Generator
- b. Test Sequence Processor

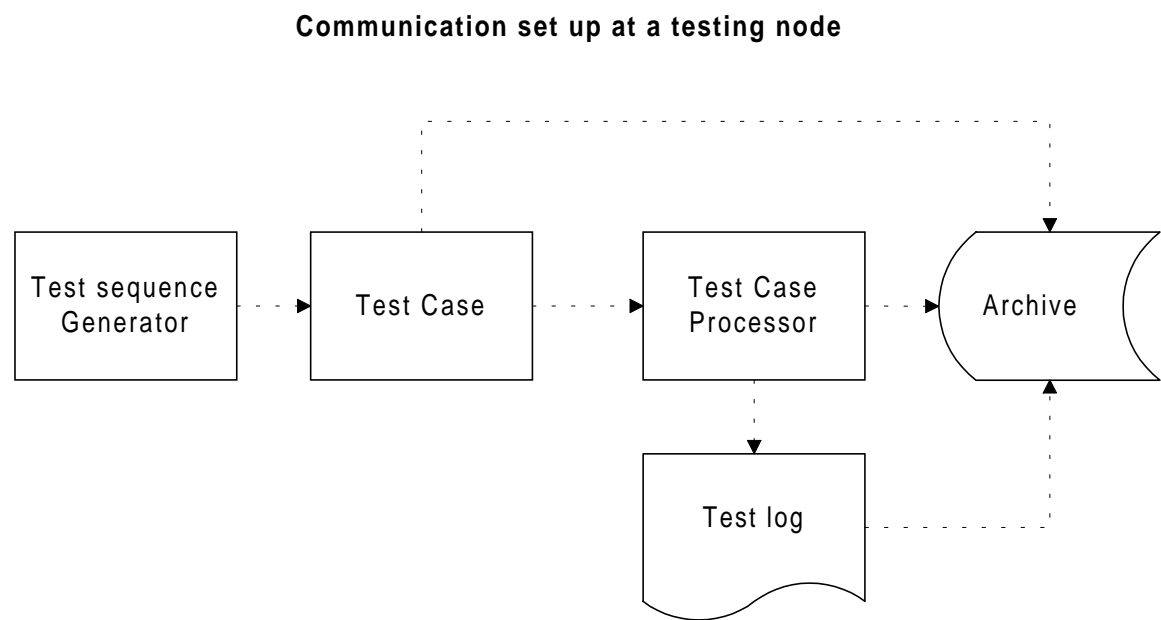


Figure 3. Communication setup at a testing node

### 3.1. Test Case Generator

A program was written to present the user with a stepwise list of possible user options. A tree of options had to be devised. The following is a list of the steps taken in the design of the test sequence generator program.

#### 3.1.1. Design of the Test Case Generator

- **STEP1 - Listing the Options (Gross Encapsulation)**

The first step in the process was to list out the main functional areas of the application. The set of most probable sequences that these functional areas would be called upon to perform was then ascertained. Examining the program flow in such a manner as described above, gave us a second chance to correct the specification document.

- **STEP2 - Program Flow Possibilities**

The general sequence of traversing these functional modules was ascertained based on our assumptions of what the user might think most logical and simple. Further analysis uncovered more complex routes the user might want to take. Using our judgment on the probability of a certain sequence being executed and whether certain sequences would be logically needed, we limited the number of routes the user could take. (Please refer to the Program Flow Chart.)

- **STEP3 - Listing the Options (Converging to Simplest Constituents)**

Each gross functional block was decomposed into the next level of detail. The sequential flow was determined using the same process as Step 2. This process was repeated until the simplest constituent user action units were reached.

- **STEP4 - Assigning Tokens**

Each of the various user action options is assigned an unique alphanumeric token. It is advisable to assign meaningful codes to improve readability and maintainability of the code and test sequences.

### 3.1.2. Using the Test Sequence Generator Program

The Test sequence generator program was written within the Visual C++ 4.0 environment using C++ and Microsoft foundation classes (MFC's). When the program first start, a dialog box, see figure 4 below, pops up with allow the user to enter a path and a file name so that the sequence of tokens can be written to.

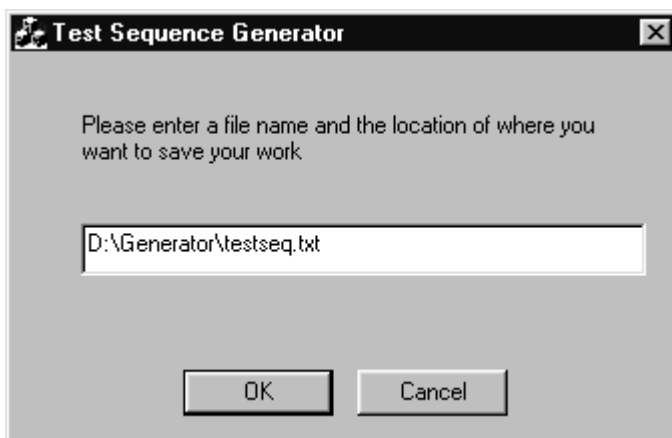


Figure 4. Test Sequence Generator Dialog Box



Figure 5. File Open Dialog Box

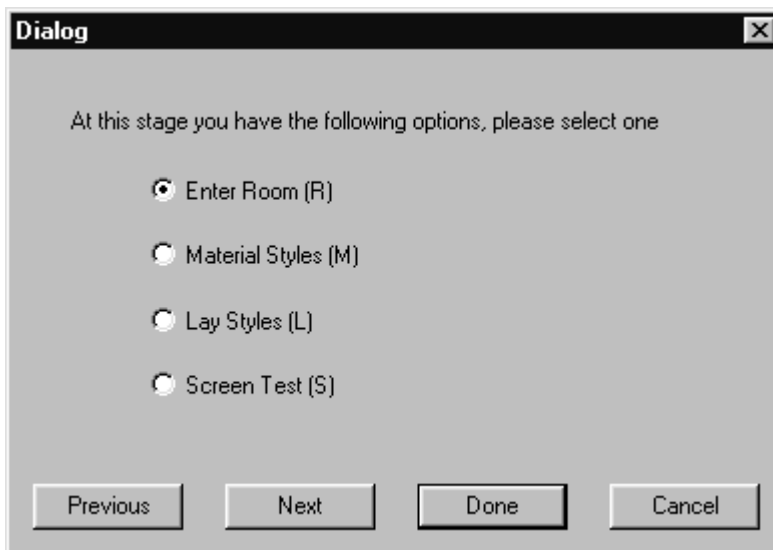


Figure 6. Rooms and Material Styles Dialog

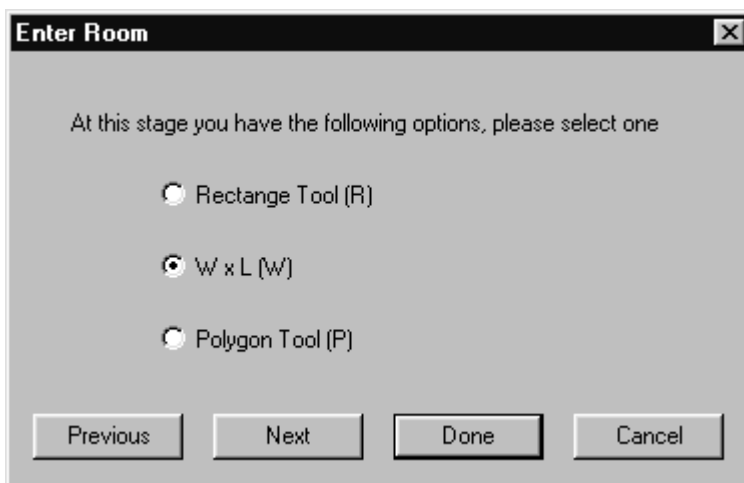


Figure 7. Tool Options for Drawing Room Geometry

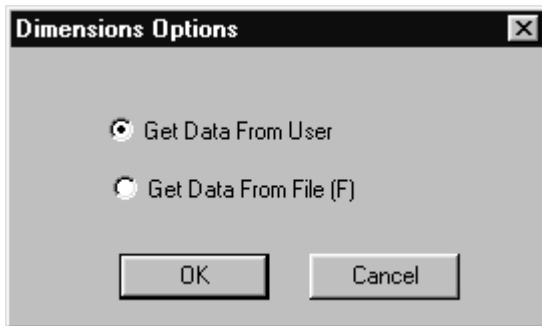


Figure 8. Getting Room Dimensions Options

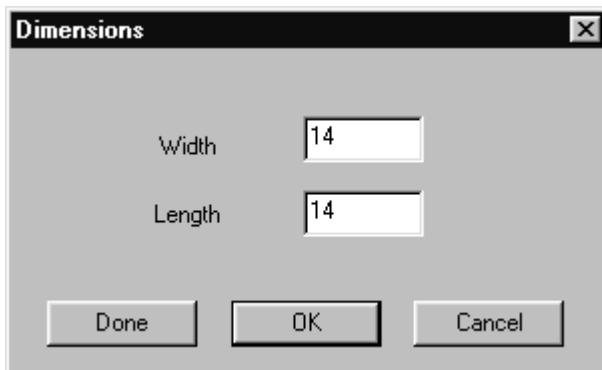


Figure 9. Getting Rooms Dimensions from the User

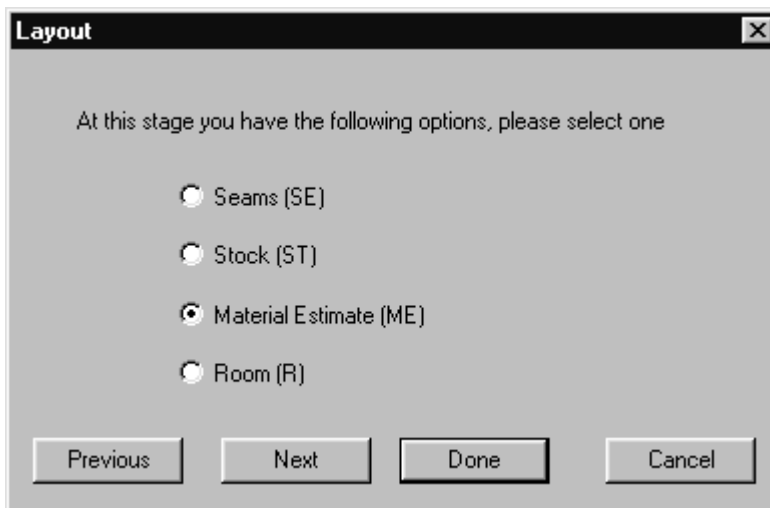


Figure 10. Layout Computation Options

Each of the dialogs allowed the user to move back and forth through the sequence of dialogs. In case the user wanted to generate a few random rooms, he could access a room generator we designed.

Each of the options selected by the user as he traversed a complete test sequence was converted to a token and written to a test file.

A sample test sequence wherein the user wanted to open a new file(*N*), enter rooms(*R*) of dimensions 14x14, 15x15 and 16x16, and then see the material estimate(*ME*) would in its simplest form look as below:

*N,R,14x14,15x15,16x16,ME*

## **3.2. Test Case Processor**

### **3.2.1 Design of the Test Sequence Processor**

This program was written using the scripting language provided with Rational Software Corporation's Rational Visual Test 4.0 <sup>TM</sup>test automation tool. This language is derived from Visual Basic and has a host of predefined functions, which enable one to access and manipulate MS Window's GUI controls and custom controls.

#### **3.2.1.1. Isolating Application Variables, Sub-routines and Functions**

To enable easy maintainability of the data, all GUI controls of the application were tokenized in a header file so that changes to the controls during development could be incorporated at a single location, i.e., the header file containing the tokens. Most functions to access and manipulate the application controls were placed in an include file. Each function was documented with the purpose of the function, the parameters it would take, and the expected values it would return or the results that the function was supposed to accomplish.

#### **3.2.1.2. Concurrent Development and Implementation**

The program structure is modular. We could code up functions to deal with just one possible path and get testers to start sending feedback. Concurrently, the code could be expanded to cover more functionality. The test sequence generator could likewise be updated so that the users have more options to choose from. So, in this manner the test sequence processor can become functional before it is fully completed. As far as maintainability and future enhancements of the logic, the code allows for the addition of more functions to handle new functionality as and when it is incorporated without re-writing code.

### **3.2.2. Using the Test Sequence Processor**

The program will open the test sequence file and read the first token. This token will be paired with a function, which will be called to execute the appropriate action, which the token represents. The test sequence processor then reads in the next token and performs the requisite action, be it:

- a. execute a function,
- b. generate a test sequence,
- c. read data from an existing file,
- d. activate verification code, or
- e. write to file, print or display output

In this way the user actions as tokenized in the test sequence are articulated into playback without human intervention. The program will continue processing the test sequence string until EOL is reached. Suitable error checking was incorporated into the line parsing component of the test sequence processor.

## Communication within the test network

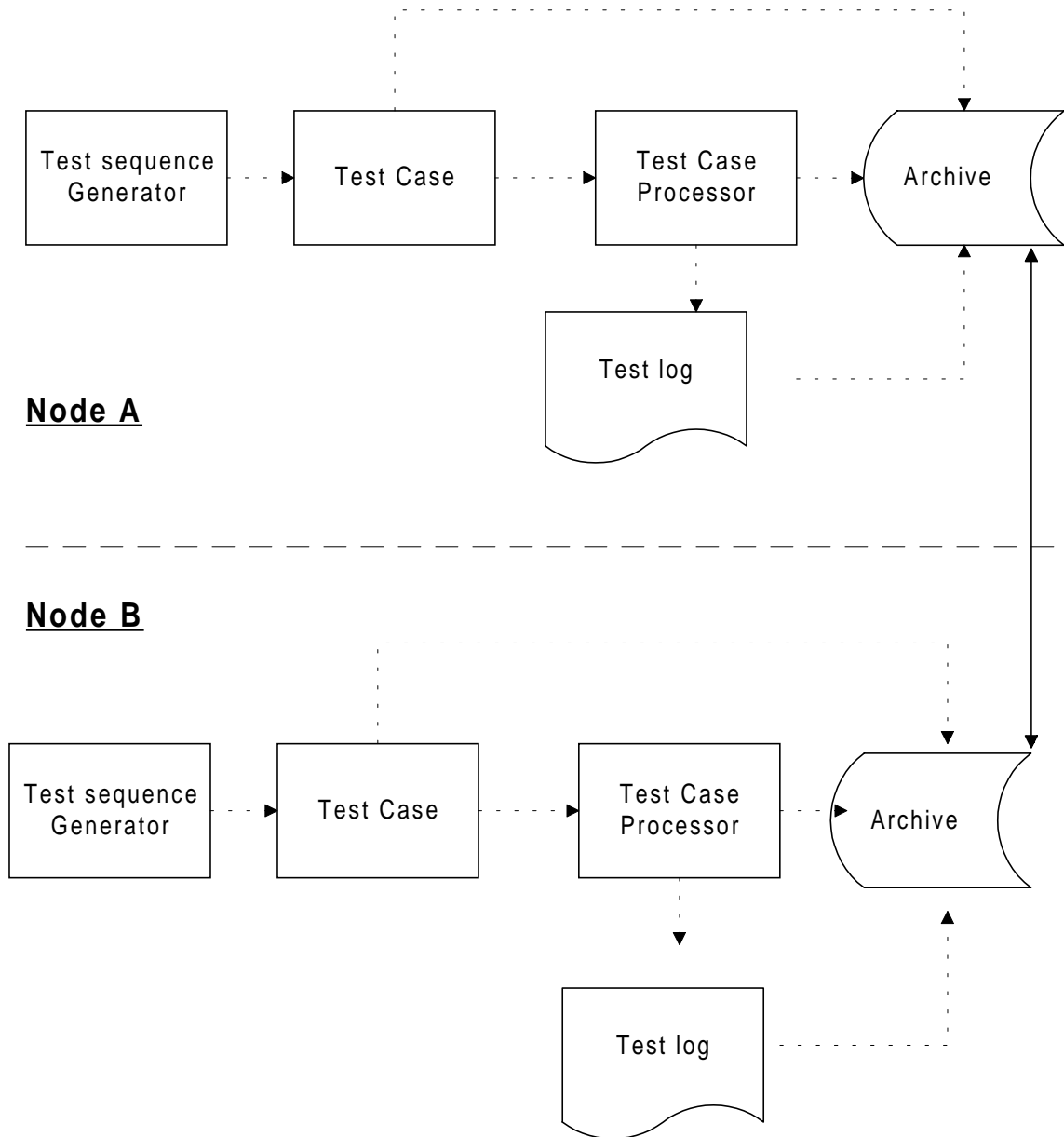


Figure 11. Communication setup for distributed testers

The flowchart above schematically represents communication within a distributed network of two testers. The tester at Node A would open up the test sequence generator program and enter the responses pertaining to his specific bug/test sequence. The test sequence would be archived by him and forwarded to the tester at Node B. It could be that the tester at Node B supervises the tester at Node A and will screen the bug for legitimacy. The tester at Node B will open up his test sequence processor program and play back the test sequence sent to him. Based on his findings, he may request the tester at Node A to create more test sequences and/or update the archive.

This concept can be extended to test networks with multiple testers and locations. Details of managing the archive would be an important feature of the network but a detailed description is outside the scope of

this paper. A version control database would suffice in local archive synchronization, retention and recall of test cases and bug information. Other database administration tasks would be creating backup routines and upgrading the database when necessary

## **4. Benefits**

### **4.1. Understandability**

By using this model to capture feedback from remote associates, all parties in the chain of communication are implicitly required to respond within the confines of the test language. This increases the probability that communication of bugs and test cases within the distributed test environment will be unambiguous.

### **4.2. Efficiency**

Since feedback is articulated in machine-interpretable code, the test case can be played back with or without modifications to the script eliminating the time and labor overhead of re-creating the script. The test case generator is a quick way to code test cases since the tester is essentially not writing a single line of code. All he does is navigate a series of dialogs and respond with the choices therein. Another element which contributes to efficiency is that these test cases can readily be added to a regression test suite.

### **4.3. Precision**

Accurate playback can be maintained since errors in re-creating the script are pre-empted.

### **4.4. Archiving**

Script files thus generated can be archived using any revision control system.

### **4.5. Benefits to the programmer**

#### **4.5.1. Analysis of non-reproducible bugs**

In some instances, bugs communicated by testers are not reproducible. If the test script is available, it is possible to modify the test sequence by manipulating variables in the test scenario to force the error to surface. The additional benefit is that this process of changing the values in the test scenario will add to the number of test cases. While the same process could be used in manual testing, the availability of test cases that can be regressed in automation does count as a benefit.

#### **4.5.2. Simplifying test cases**

Many testers write bug reports with a lot of extraneous detail, which do not really contribute to program malfunction. Making the bug report/test case an easily editable script, allows the tester to quickly pare the test down to the smallest number of code statements possible to still reproduce the fault. This off loads some of the fault isolation the developer would normally do, or the tester might be required to prove.

## **5. Conclusion**

The communication model for distributed testing model described above is a prototype implementation of the key concepts of a test case processor and a test sequence generator to solve many problems associated with communication of bugs and test cases among distributed testers. The potential benefits of such a system warrant moving from the prototyping stage to a full scale implementation. An implementation of the same is planned in the near future.



## **Bibliography**

1. **Cem Kaner, Jack Falk, and Hung Quoc Nguyen;** Testing Computer Software, Second Edition, International Thomson Computer Press, **1998**
2. **Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman;** Data Structures and Algorithms, Addison-Wesley Publishing Company, 1987
3. **Thomas R. Arnold II;** Visual Test 6 Bible, IDG Books Worldwide, Inc., 1999
4. **Edward Kit,** Integrated Test Design and Automation, Software Development, February 1999

# **The Application of a Software Testing Technique to Uncover Data Errors in a Database System**

Shirley A. Becker  
&  
Anthony Berkemeyer  
Software Engineering & Computer Science  
Florida Institute of Technology  
150 West University Blvd.  
Melbourne, Florida 32901  
Ph: (407)674-8149 Fax: (407)674-8192  
[becker@cs.fit.edu](mailto:becker@cs.fit.edu)

## **ABSTRACT**

It is proposed that database systems be assessed for quality using an approach based on stochastic software testing. This approach requires an understanding of potential data errors ranging from syntactic mistakes to undetectable dirty data. A data error classification schema has been developed to categorize the type of data errors that could be found during testing. This classification schema provides a basis for developing test cases inclusive of data, constraints, rules, and supporting queries. A process is described whereby a database system is tested using the test cases and data validation requirements to generate feedback on data quality.

**Key Words:** Database Systems, Data Quality, Software Testing

## **Biography:**

Dr. Shirley A. Becker is an associate professor of computer science at Florida Tech and co-director of its software engineering research center. Dr. Becker has published numerous articles in software engineering, team processes, and database technology. Dr. Becker is also the editor-in-charge of the Journal of Database Management.

# **The Application of a Software Testing Technique to Uncover Data Errors in a Database System**

## **1. Introduction**

For many of today's operational software systems, the quality of the data is of great concern. Much of the data "mess" is hidden from end users by the filtering that occurs when a report is generated (Gordon, 1996). Unfortunately, the errors are obscured in the report that then is erroneously assumed to be correct. This false assumption means that inconsistent, missing, dirty, and other data problems virtually go undetected.

To compound the problem, many legacy systems are used without any mechanisms for identifying problems associated with data representation, standards, normalization, and redundancy, among others. Most practitioners would agree that these systems provide a wealth of historical information if the data could be relied upon as having a relatively high quality. The proliferation of data warehousing and related research to find efficient ways of relying upon historical data supports this matter (Inmon & Hackathorn, 1994). Though manual techniques have been developed to "clean" the data, there is still the issue of the prohibitive cost of using such techniques.

The underlying issue is an organization's ability to assess data quality given the size and complexity of its operational and legacy software systems (Hoxmeier, 1997). The multiplicity of data correctness problems associated with the use of software systems makes it virtually impossible to identify and fix each problem. What is needed is an effective means of obtaining feedback on the quality of the data in order to make decisions on quality improvements.

It is proposed that data be assessed for quality much the same way that software applications are assessed. There is much to be gained by applying the techniques used in software development in dealing with system size and complexity issues. Traditional testing techniques, for example, are seldom used to test all possible components of a system as was done a decade ago. It has been found that this type of coverage-based testing is infeasible given today's systems; and as a result, more sophisticated testing techniques have been developed.

One such technique, developed by Whittaker (1997), relies on stochastic testing to provide valuable feedback on a software system's defects. This testing technique is based on sampling as a means of quantifying the validity of the software system. This paper describes initial work on the application of this approach to obtain feedback on the quality of data.

The paper is organized as follows. Section 2 describes the various data quality issues that may be addressed in a testing environment. Section 3 presents the background on the proposed testing environment. Section 4 describes the testing process that would be used to uncover data errors. Section 5 concludes the paper and addresses future research opportunities.

## 2. Data Quality

The phrase “data quality” can be interpreted in many different ways depending on the organization’s data requirements (Orr, 1998). Data quality may range from obvious syntactic mistakes to undetectable dirty data. It is important to understand the types of data errors that could occur in order to develop effective ways of identifying them during the testing process.

A data quality classification schema has been developed that is comprised of *correctness*, *completeness*, *comprehension*, and *consistency* classes (Greenfield, 1996; Fox et al., 1994). Table 1, organized by the classification schema, lists the data errors that potentially could cause major problems during the use of a software system. Of course not all of these data issues may be relevant during system use. However, it is important to realize that there are many types of data errors that could occur with or without actually causing a system failure. Each of the data quality classes is briefly described below.

- *Data consistency* is concerned with ensuring that data is represented “semantically the same way” within and across tables in a database system. Data consistency is not maintained when data with the same meaning physically have varying data types and lengths. Data inconsistencies may occur when data elements are hidden in aggregated data. Data consistency is also impacted when integrity constraints are not used properly. Inconsistencies may occur when child records are not associated with a parent record or when key values don’t match due to nulls or inaccuracies (e.g., spelling mistakes) in data elements.
- *Data completeness* is associated with missing or inaccessible data as a result of inadvertently storing nulls, blanks, abbreviations, truncations, or partial data. Completeness problems become an issue when integrity constraints are missing or inappropriately enforced as in the example of cascading updates or deletions across tables.
- *Data correctness* is concerned with corrupt or wrong data being stored; as well as, dirty data being shown to the user under the guise of valid data. Data corruption occurs, for example, when data is incorrectly manipulated through the use of views. Nonexistent data is shown to the user when incorrect joins are performed over nonkey attributes (refer to Date(1995) for a discussion of loss projection). Though this doesn’t impact data storage, it can cause major problems for the user when the data is assumed to be valid. Data correctness is also impacted when integrity constraints are incorrectly used or inadvertently disabled. Check, unique, or null constraints may be too restrictive causing problems with data insertion, updates, and deletions. Finally, data entry errors cause problems when misspelled, missing, partial, or wrong data is stored.
- *Data comprehension* problems may be inherited from legacy systems that contain cryptic, obsolete or unknown data due to changes in the real world applications (e.g., zip code change from 5 to 9 digits). When data is aggregated or concatenated instead of being stored as a set of attributes, vital details may be lost.

**Table 1: Data Quality Factors**

Factor:	Description:	Example:
<i>Data Consistency Issues:</i>		
Varying Data Definitions	The data type and length for a particular attribute may vary in tables though the semantic definition is the same.	Social security number may be defined as: Number (9) in one table and Varchar2(11) in another table.
Varying Data Codes & Values	The data representation of the same attribute may vary within and across tables.	A flag representing yes or no may be defined in many ways. The value 'no' may be represented as: 'NO', 'N', 'n', '0', '1', or blank.
Misuse of Integrity Constraints	When referential integrity constraints are misused, foreign key values may be left "dangling" or inadvertently deleted.	An employee record is deleted but his/her dependent records are not deleted.
Nulls	Nulls may be ignored when joining tables or doing searches on the column.	The supervisor (Smith) has been entered as a null value for an employee (Jones). A report of all employees supervised by Smith would not list Jones.
<i>Data Completeness Issues:</i>		
Missing data	Data elements are missing because of a lack of integrity constraints or nulls are inadvertently not updated.	A shipment's date of estimated arrival is null thus impacting an assessment of variances in estimated/actual arrival data.
Inaccessible Data	Inaccessible record due to missing or redundant unique identifier value.	Customer numbers are used to identify a customer record. Because uniqueness was not enforced, the customer ID (45656) identifies more than one customer.
Missing Integrity Constraints	Missing constraints can cause data errors due to nulls, nonuniqueness, or missing relationships.	Part records with a supplier identifier exist in the database but cannot be matched to an existing supplier.
<i>Data Correctness Issues:</i>		
Loss Projection	Tables that are joined over nonkey attributes will produce nonexistent data that is shown to the user.	Lisa Evans works in the LA office in the Accounting department. When a report is generated, it shows her working in Marketing and Accounting.
Incorrect Data Values	Data that is misspelled or inaccurately recorded.	123 Maple Street is recorded with a spelling mistake and a street abbreviation (123 Mapel St)
Inappropriate Use of Views	Data is updated incorrectly through views.	There is a view that contains nonkey attributes from base tables. When it is used to update the database, null values are entered into the key columns of the base tables.
Disabled Integrity Constraints	Null, nonunique, or out of range data may be stored when the integrity constraints are disabled.	The primary key constraint is disabled during an import function. Data is entered into the existing data with null unique identifiers.
Misuse of Integrity Constraints	Check, not null, or foreign key constraints are inappropriate or too restrictive.	Check constraint only allows hardcoded values of "C", "A", "X", and "Z". But a new code "B" cannot be entered.
<i>Data Comprehension Issues:</i>		
Data Aggregation	Aggregated data is used to represent a set of data elements.	One name field is used to store surname, firstname, middle initial, and last name (e.g., John, Hanson, Mr.).
Cryptic Object Definitions	Database object (e.g., column) has a cryptic, unidentifiable name.	Customer table with a column labeled, "c_avd". There is no documentation as to what the column might contain.
Unknown or Cryptic Data	Cryptic data stored as codes, abbreviations, truncated, or with no apparent meaning.	Shipping codes used to represent various parts of the customer base ('01', '02', '03'). No supporting document to explain the meaning of the codes.

What is needed is a means of identifying these data errors in order improve and maintain the quality of a software system. We propose the use of a software testing technique to assist in uncovering data errors while validating system behavior.

### 3. Testing

The sampling used for stochastic testing of a software system is an important aspect of this work because of its representation of the real world population. With a sufficiently large and unbiased sample, feedback can be obtained not only on the software reliability but also on the quality of the data. Expensive, exhaustive testing techniques can be replaced with sampling to identify problem areas in metadata (database objects) and data.

There are several advantages to using this approach in assessing data quality. There is a certain statistical confidence in the test results based on sample size. As the sample size increases so does the confidence in the test results as being representative of the whole population (refer to Whittaker & Poore (1993) for an in-depth discussion). In addition, those components of the system that are used most often or are of greater interest than other components would appear in a significant number of test cases as testing is based on usage.

The testing process that provides a basis for this work is presented in Table 2. This approach requires a profile of the quality factors to be included as part of the testing activity. The profile of the quality factors includes validation criteria and data requirements necessary to determine the success or failure of the system once a test case has been executed. The table identifies the artifacts that would be produced after each process step. The final step produces a set of logs that would include the number of system defects, an audit log of data changes, and a data error log that would contain data errors associated with each quality factor under assessment.

**Table 2: General Description of the Testing Process**

Process Step	Explanation	Artifacts
Identify quality factor(s) to be assessed.	The quality factors drive the data validation criteria necessary for effective testing.	Quality factors.
Model the system component.	The set of states representing the component's behavior is constructed in a directed graph. Each arc is assigned a probability between 0 and 1 (inclusive). The sum of the exit arcs probabilities from each state = 1.	Behavior model.
Determine sample size.	Sample size is based on statistical confidence and reliability.	Sample size.
Develop test sequences.	Test sequences are generated based on usage probabilities.	Test sequences.
Develop test data & data validation criteria.	Test data is generated for each sequence inclusive of quality factors & data validation criteria.	Test cases with validation criteria.
Perform testing and capture results.	The test cases are executed & defects/data errors logged. Data may be gathered before and after test case execution based on validation requirements.	Audit log, data error report, & defect log.

Figure 1 depicts the testing environment whereby the inputs, process, and outputs provide the basis for obtaining feedback on the quality of the data. The inputs of this testing environment would include a sample of test cases with database transactions of reads, updates, deletions, and insertions. Inputs also include the selected data quality factors with associated validation criteria, which become the basis for analyzing the test results.

The behavior model, shown in the middle of the figure, represents the system's states and usage probabilities. The usage probability associated with each arc is based on past experience, extracted from audit logs, or randomly generated.

The outputs produced by the testing activities would include the number of defects (as defined in the traditional sense of testing), database audit logs that would contain old and new data values, and a data error report. The output requirements are determined by the validation criteria as specified as part of the inputs of the testing environment. Based on the validation criteria, the audit log may be updated and the data error report may contain the output of SQL queries that were executed to provide additional information about the quality of the data.

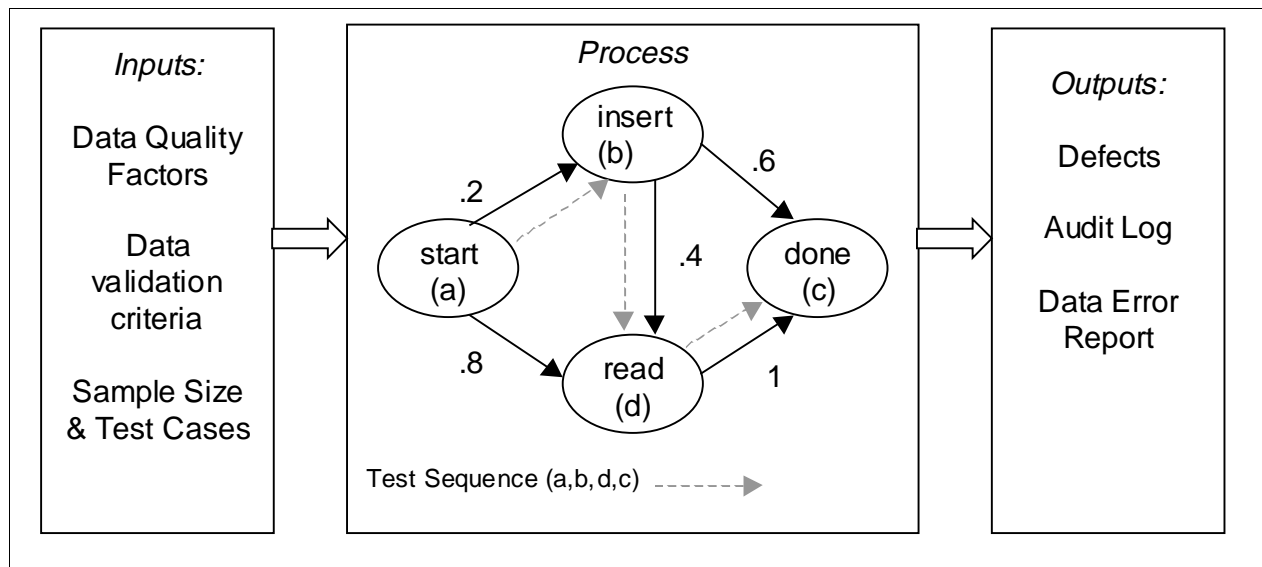


Figure 1: The Testing Environment

The next section will provide an overview of the testing environment in terms of a simple application for updating projects.

#### 4. The Project Update Application

A behavior model (Figure 2a) represents a project update application (Figure 2b), which is part of a web-based project management tool (Becker & Ladino, 1999). The arcs in the model are labeled with probabilities of use to represent the real-world activities associated with project updates. The model shows that ninety percent of all project updates are successful in that project records were found and data modified. This is reflected as the commit state that ensures changes are made permanently to the database system. The rollback state, which has a ten-percent

probability of occurrence, is executed when the update fails thus ensuring that any temporary changes are not permanently recorded in the database system.

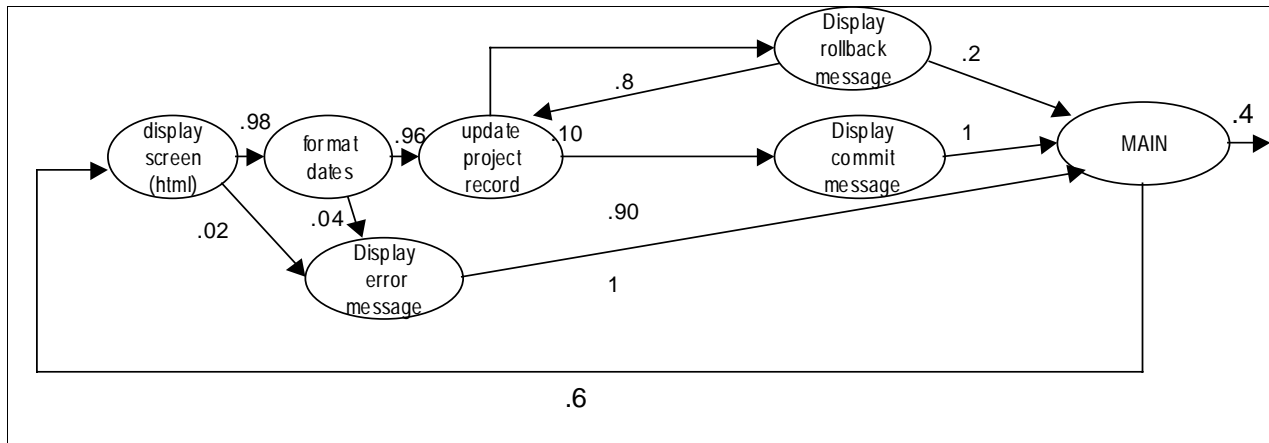


Figure 2a: Graph Containing Components of the Update Project Application

```

/*****
* This procedure updates the table PROJECTS *
*****/
PROCEDURE u_projects
(p_id      IN projects.id%TYPE,
p_name     IN projects.name%TYPE,
p_manager  IN projects.manager%TYPE,
p_sdate    IN owa_util.dateType,
p_edate    IN owa_util.dateType)
IS
pr_edate DATE;
pr_sdate DATE;

/*web-based application */
BEGIN
  http.htmlOpen;
  http.bodyOpen;
  pr_sdate := owa_util.todate(p_sdate);
  pr_edate := owa_util.todate(p_edate);

/* SQL component */
  UPDATE projects
  SET name      = p_name,
      mgr_name = p_manager,
      start_date = pr_sdate,
      end_date  = pr_edate
  WHERE id = p_id;
  COMMIT;

/* Error Component for the Rollback */
...
END u_projects;

```

Figure 2b: Part of the Code Design for Update Project Application (Becker & Ladino, 1999)



This example exhibits such simple behavior that one would think there was little opportunity for data problems to occur as the unique identifier either matches a record or is not found in the database. Even in this simple example, however, there is the potential for a range of data errors. These data errors would not typically be found during traditional means of software testing, as a defect would constitute a system failure. System failure, for example, could occur during the execution of the html statements, date function calls, or the SQL. The testing results would uncover defects in the execution of the code but not necessarily data corruption or integrity problems. To illustrate this point, Table 3 identifies potential data problems associated with the update project application that would not have resulted in a system defect.

**Table 3: Potential Transaction and Data Errors That May Go Undetected.**

<i>Transaction Problem</i>
The SQL syntax doesn't deal with upper or lower case data resulting in "no record found" when the record should have been found (e.g., 'P123' <> 'p123').
<i>Data Problems</i>
The record identifier is not unique. This could be the result of a missing integrity constraint (unique or primary key) or syntax (both 'p123' and 'P123' are in the database as separate records).
The manager's name is a long text field. This may cause data problems in terms of the ordering of the surname and first and last names ('Jones, Sally, Ms.' or 'Ms. Sally Jones'), missing data elements (e.g., no surname is included), and syntactic (spelling) mistakes.
The manager's id must match an existing employee social security number to maintain referential integrity. A missing integrity constraint would not ensure this matching occurs thus allowing a manager to exist without a matching employee record.

In order to follow the testing process that has been proposed, each test case must include both data and data validation criteria in order to identify system defects and data errors. Test cases are selected based on the probabilities of use as represented in the behavior model.

We start by identifying test sequences that reflect how the system is used. For our example, test sequences are illustrated by the following:

Sequence 1: *display screen(A), get today's date(B), update project(C), commit(D), quit(F).*

Sequence 2: *display screen(A), get today's date(B), update project(C), rollback(E), update project(C), commit(D), quit(F).*

Sequence 3: *display screen(A), get today's date(B), update project(C), rollback(E), quit(F).*

Next, the data validation criteria are added based on selected data quality factor(s). The quality factor under investigation is *missing integrity constraints* as part of a completeness check. The data validation criteria will include both primary key and foreign key checks to ensure that as projects are updated data integrity remains intact. In this case, the foreign key constraint is on manager's id (social security number), which means that a manager cannot be assigned to a project unless he or she exists as an employee in the Employee table.

The data validation criteria are expressed as a set of rules that would be used to determine whether there are data errors resulting from the execution of the test case. Table 4 shows several

data validation criteria associated with the first test sequence (A,B,C,D,F). These rules and their links to data quality factors would reside as physical data in the database.

**Table 4: Illustration of Data Validation Criteria for Primary Key Integrity Constraint**

Rule:	Validation
1. One and only one record found by project_id	One record returned. If more than one record, then update error report. A query is executed to check database objects for disabled primary key constraint.
2. When no records found, execute “like” query.	No records were found. A query is executed to search for similar text patterns matched against the p_id. Error report is updated with similar p_ids.
3. When no records found, execute query to search for missing values (nulls).	No records were found. A query is executed to determine if there are missing records identifiers. Error report is updated with existence or nonexistence of nulls.

Depending on the rule, we may need to track old and new data values in order to determine whether a data error has occurred. This requirement is also stored with the rule and becomes part of the information associated with each test sequence. When the rule specifies that old and new data values need to be stored, an audit log entry is made during testing activities. In our example, the primary key integrity validation would not require an audit log entry because we are only concerned with finding one record matching on the primary key. For other quality factors, however, it would be imperative to have old and new data values to ensure that data errors have not occurred.

Table 5 shows a test case whereby data and validation criteria data have been added to test sequence (A,B,C,D,F). As part of the data validation criteria, the test case identifies potential queries that would be used to study data errors after test case execution. In this case, SQL queries would produce additional information used to assess whether the integrity constraints are enabled, and to determine whether the project identifier exists in the database. For manager’s social security number, the Employee table is searched for data type and data values similar to the test case data.

Once the testing activity is completed, the data errors need to be analyzed in terms of the overall impact on the database system. The type and number of data errors discovered may require further study to determine the best approach to resolving them. In the test case presented above, integrity constraints may be enabled to ensure data integrity for future use. More importantly, would be the need for data cleaning or conversion depending on the severity of the problems found. Though this discussion is beyond the scope of this paper, future research efforts need to address automated tools for data maintenance.

## 5. Conclusion and Future Research

It is proposed in this paper that stochastic testing techniques be applied to the testing of data quality in software systems. This type of testing allows for a sampling of software behavior that would represent real world use of the system. In this respect, the software applications that are used most often would have a higher representation in the test cases generated.

**Table 5: Test Case (A,B,C,D,F) and Potential Data Errors**

Test Data	Potential Data Errors due to lack of Integrity Constraints
<b>DATA:</b> 'P123' 'Audit Features' '344-78-9856' '02-10-1999' '09-01-1999'	<b>PRIMARY KEY:</b> <ul style="list-style-type: none"> <li>'P123' does not uniquely identify one project. (Note: <i>this would not result in a test defect if the primary key integrity constraint were disabled</i>).</li> <li>'P123' does uniquely identify one project but is stored as 'p123'. (Note: <i>this would not result in a test defect, as the record would not be found.</i>)</li> </ul>
<b>QUALITY FACTOR:</b> integrity constraints	
<b>RULES:</b> pk: one record with correct id fk: manager identifier is in the Employee table	
<b>SUPPORTING QUERIES:</b> <b>Primary Key:</b> select * from Project where p_id like '&123'; select count(*) from Project where p_id is null; <b>Foreign Key:</b> select * from Employee where emp_id like '344&'; Both: /*check for enabled constraints*/ select * from user_constraints;	<b>FOREIGN KEY:</b> <ul style="list-style-type: none"> <li>The manager (SS#344-78-9856) does not exist in the Employee table. (Note: <i>this would not result in a test defect but would cause a data error</i>).</li> <li>The manager (SS#344-78-9856) is in the Employee table but the data is stored as a number data type (344789856). (Note: <i>this would not result in a test defect but would cause a data error</i>).</li> </ul>

Traditional testing of software systems identifies defects in terms of the successful execution of a particular test case. The problem with this approach when searching for data errors is that the execution path may be completed successfully while data errors go undetected. What is needed is a means of testing software applications in terms of defects and data errors.

A testing environment has been described that would include the selection of data quality factors as part of the testing activities. Each data quality factor would have a set of rules that would identify validation criteria and expected outcomes. The validation criteria would provide the means for assessing data quality associated with each test case. As a result, testing would include an evaluation of data appropriate to the quality factor that has been selected.

It is important to develop automated means (e.g., SQL queries) that would supplement each test case execution. An automated environment is currently being studied to support the rule-based validation criteria. Several operational database systems have been analyzed using this approach in order to initiate the development of a common set of validation criteria rules associated with quality factors.

There is a significant amount of learning potential as the result of data analysis and error reporting. Further study is needed to ensure that the validation criteria rules and supporting queries are updated based on testing experience. These findings need to be incorporated into the data and data objects that represent the quality factors and supporting rule set.

Finally, a severity rating system needs to be developed that would provide feedback on the impact of the data errors encountered. It may be the case that the data error is trivial and doesn't impact the use of the system. But, in most cases, the data errors will point to a much larger problem in that data is incorrect, inaccessible, missing, or corrupted.

## References

Becker, S. and Ladino, D. (1999). "A Technical Infrastructure for Process Support," **Software Process Improvement: Concepts and Practices**, IDEA Group Publishing, Hershey, PA.

Date, C.J. (1995). **Introduction to Database Systems**, Addison\_Wesley Publishing, Reading, MA.

Gordon, K.I. (1996). "The Why of Data Standards – Do You Really Know Your Data," <http://www.island.net/~gordon/whystds.htm>.

Greenfield, L. (1997). "An (informal) Taxonomy of Data Warehouse Data Errors," <http://pwp.starnetinc.com/larryg/errors.html>.

Fox, C., Levitin, A. & Redman, T. (1994). "The Notion of Data and Its Quality Dimensions," **Information Processing and Management**, Vol. 30, No. 1.

Hoxmeier, J. (1997). "A Framework for Assessing Database Management," **Proceedings of the ER'97 Workshop on Behavioral Models and Design Transformations: Issues and Opportunities in Conceptual Modeling**, Los Angeles, CA.

Inmon, W. H. and Hackathorn, R.D. (1994). **Using the Data Warehouse**. Wiley & Sons NY, NY.

Orr, K. (1998). "Data Quality and Systems Theory," **Communications of the ACM**, Vol. 41, No. 2.

Whittaker, J. (1996). "Certification Practices," **Cleanroom Software Engineering Practices**, IDEA Group Publishing, Hershey, PA.

Whittaker, J. (1997). "Stochastic Software Testing," **Annals of Software Engineering**, Vol. 4.

Whittaker, J. & Poore, J. (1993). "Markov Analysis of Software Specifications," **ACM Transactions on Software Engineering and Methodology**, Vol.2, No.1.

**Use of Metrics on the  
Jubilee Line Extension (JLE) Project**

Presented by:

Ed Ko

Alcatel Canada, Deputy Project Manager

5172 Kingsway, Ste. # 270

Burnaby, British Columbia Canada, V5H 2E8

Tel: (604) 434-2455

Voice Mail: (604) 434-2375 ext. 111

Email: [ko@vansel.alcatel.com](mailto:ko@vansel.alcatel.com)

Catherine Keane

Alcatel Canada, Sr. Quality Assurance Specialist

5172 Kingsway, Ste. # 270

Burnaby, British Columbia Canada, V5H 2E8

Tel: (604) 434-2455

Voice Mail: (604) 434-2375 ext. 113

Email: [ckeane@vansel.alcatel.com](mailto:ckeane@vansel.alcatel.com)

Heather Duncan

Alcatel Canada, Deputy Project Support Manager

5172 Kingsway, Ste. # 270

Burnaby, British Columbia Canada, V5H 2E8

Tel: (604)434-2455

Voice Mail: (604) 434-2375 ext. 181

Email: [hduncan@vansel.alcatel.com](mailto:hduncan@vansel.alcatel.com)

## **ABSTRACT**

Use of Metrics in project delivery of a  
“Mass Transit Mainline Control System (MCS)”

A mass transit mainline control system is, by software industry standards, a very large software system. It can in fact be larger than some of the largest airport control systems currently in operation around the world.

Like any very large-scale software development effort, an MCS is subject to the effects of estimate inaccuracy, challenging progress and cost control, as well as the delivery process methodology.

This paper covers how the metrics were established to support an incremental software development and an evolutionary delivery process. The paper will also show how the effective use of metrics was instrumental in the day to day decision making process to steer the overall project effort, from the perspectives of organizational, technical and risk management.

The paper explains how effective metrics collection can be implemented using automated means, and how metrics can become a key tool to report on project progress. The cost of metrics collection is also covered.

Finally, the paper provides a series of lessons learned which should be taken into consideration in organizing and directing large software development effort, especially for Mass Transit Systems.

## **INTRODUCTION**

Alcatel Transport Automation is one the world leaders in the development, installation and commissioning of transport automation systems. A worldwide office structure lends itself to a global market. In Canada,

Alcatel headquarters is in Toronto, Ontario. Alcatel, an ISO 9001 certified company, has successfully designed and delivered world class, driverless, light rail rapid transit systems world wide.

Alcatel Canada is developing a large scale MCS for the Jubilee Line Extension (JLE) project. This London Underground Limited (LUL) project is the largest, most complex software development project currently in Europe. An MCS makes extensive use of software, which is becoming more and more complicated. This is a result of the increased level of complexity in modern implemented systems, making use of moving block technology, central control of communication devices and functions such as power, tunnel ventilation, and other supportive elements of transit systems.

It would not be uncommon to have an MCS with the following measures:

- More than 500,000 lines of “C” code;
- Covering over 2,000 specific requirements;
- With over 500 data tables;
- With approximately 500,000 data elements;
- Dealing with more than 130 different dialogs , or distinct Graphical User Interfaces screens;
- Approximately 100 software modules; and
- Having a project schedule containing over 8,000 activities.

The project contract was signed in December 1993, with an original completion date of March, 1996. The initial plan was to perform the re-signalling upgrade to the existing line in parallel with the construction and commissioning of the extension. In mid-1995 the approach was changed by the Customer, resulting in a new baseline schedule (programme) to commission the upgrade prior to the extension. Delays to civil works and technical difficulties encountered by other contractors brought about further changes to the commissioning approach, resulting in a further 6 baseline schedules. The approach now being followed involves commissioning of the extension prior to the upgrade of the existing line, with incremental releases of software being made at 2 to 4-month intervals.

The project organization underwent significant changes over the course of the project. In the past two years, the project organization has stabilized and is now working efficiently. The following key groups were established:

- Systems Engineering, responsible for requirements analysis and system design
- Software Engineering, responsible for detailed design, coding and debugging
- System Integration and Test, responsible for integration and testing of the system
- Test and Commissioning (London), responsible for installation and testing in the “real world”
- Project Support, responsible for schedule management and configuration management
- Contract Administration, responsible for handling contractual aspects of contract variations
- Quality Assurance, responsible for ensuring that the product and processes are in accordance with the customer’s expectations.

Large-scale software project development and implementation such as the MCS is a risky business. Such an undertaking demands extensive tools, processes, as well as a very focused and skillful project team in order to achieve successful project delivery. Elements which contribute to a successful project completion include the timely monitoring of the project health, an effective assessment of risks and quick implementation of mitigation measures. It is, therefore, essential that the project team know at all times where they are in the development process, and have indicators of potential problems.

Metrics are the tools necessary to measure the progress of a software project, especially a large scale project. They aid project staff in quantifying various aspects. They are the numbers that help guide schedule decisions and sometimes help justify project direction. They offer a view across the project from different angles. Identification of metrics required, fine-tuning and analysis are necessary to establish, assess and improve the information collected. This requires effort, attention to detail, and an ability to understand and use the information wisely. Metrics, therefore, become a roadmap, or a compass which can be considered a lifeline between project initiation and project conclusion. Metrics provide a leading indication of project health and product quality by measuring status against plans or targets. The data must be used to support decisions on all matters of the project.



The identification of metrics is of vital importance, as the metrics will represent the monitoring elements or zones, which will, in turn, be used to assess the health of the project. Once the metrics are identified, they must be established in order to meet the intent of use. A procedure must then be set up to collect the metrics in a consistent and efficient manner. Most important of all, the metrics must be used effectively to monitor and guide the project to a successful delivery.

During the development and implementation of the metrics program, several key decisions and actions were taken to ensure its success:

Previous attempts at implementing a metrics program on the Jubilee project had failed, mainly due to a lack of dedicated resources and a lack of direction. Therefore the first step taken in implementing the metrics process was to appoint a dedicated metrics team. The team consisted of two people, one allocated (at 50%) for definition and review of the metrics to be collected, and one for detailed implementation of the metrics collection.

Once a basic set of metrics had been defined, the team focused on automating the metrics collection process. Scripts were written to extract the required information on a weekly basis from the Defect Tracking tool (DDTS), the source code and the project schedule.

An important step in the development of the metrics was the decision to publicize them on a “Metrics Wall”, located in a high-traffic area of the office. This helped to promote the metrics in several ways;

- By publicizing the metrics, employees are given a message that there was nothing “secretive” about the metrics collection process.
- Employees provide additional feedback on the clarity and presentation of the metrics
- The Wall serves as a visible measure of the health of the project, and is available for staff, customers and senior management to view at any time.
- The Wall shows that the metrics are being used and are not just being collected as a “paper exercise”

The metrics are assessed on a regular (weekly) basis to identify problem areas and to discuss process improvements. They are reported to Senior Management on a monthly basis. In addition, formal metrics assessment meetings are held to discuss more general trends. These meetings involve members of Systems Engineering, Quality Assurance, the Defect review Board, Configuration Management and Project Management.

## **IDENTIFICATION OF METRICS**

In a perfect world or in an ideal organization, there is an impressive array of measures, which will serve in the estimation of a proposal, a portion of work, or a complex project. These measures come from actual values and observations made as a result of typical project delivery. Once a very accurate set of metrics is available, the next projects will inevitably become “Perfect Projects” where all work will be totally predictable, exactly estimated, and with absolutely no risks.

The theory sounds good but why is it that project development does not seem to even tend towards this “perfect world”? The unavailability of exact metrics from previous projects implies inability to estimate with a high degree of precision for the current project. Without a solid metrics system in place, the project becomes unsuccessful. Metrics do not have to be collected from the beginning of a project in order for the project to succeed. Metrics identification and collection can begin at any time within a project lifecycle. It is the dedication to their use that will help make the project successful. Start immediately. Identification and generation of a set of metrics, will provide the intelligence needed to improve, and tend towards perfection. It is, therefore, necessary to identify a set of metrics that will capture the information, which will in turn indicate the health of the work in progress.

Due to the size and complexity of the JLE MCS, the challenges associated with scope creep and a lack of co-ordination between interfacing systems (provided by other contractors), the project was constantly subject to customer requirements changes and close scrutiny by the Senior Management of Alcatel Canada.

The following metrics were identified as a means to monitor progress of the MCS development, identify its risk areas, and provide a basis for change management.

- Project Size Metrics
- Productivity Metrics
- Scope Change Metrics
- Schedule Management Metrics
- Cost Management Metrics
- Defects Management Metrics

## **THE ESTABLISHMENT OF METRICS**

On the JLE project, specific metrics were established to provide measurements for the project management aspects identified above. We must keep in mind that a metric is a quantitative measure of the degree to which a system, component or process possesses a given attribute.<sup>1</sup> This section provides a definition of such metrics, and describes how they were established.

### **Project Size Metrics**

- Source Line Of Code (SLOC)

The SLOC count is the number of lines in a source code file, excluding comments and blanks. The MCS software is delivered using an incremental approach, meaning that each software release adds functionality to the previous one. Given the approach, an in-house tool was used to compare source code files against a baseline, and produced a SLOC count of additional, changed, and total size of the MCS.

- Number of Requirements

At the start of the project, the customer prepared a set of contractual requirements called the Particular Specification (PS). Since then there have been significant clarifications, modifications and additions to the requirements and the PS was not kept up to date. The Alcatel project team undertook an exercise to partition and update the requirements, and document them in a database. These requirements have since been accepted by the customer and are maintained to provide a basis for test validation and to track further scope changes.

### **Productivity Metrics**

Like any disciplined project, a comprehensive Work Breakdown Structure was the basis of the work packages on the JLE project. These work packages, or cost accounts, were employed to generate metrics to measure productivity in various project areas.

- **Software Implementation Productivity**

This metric measures the cost of producing integrated software in units of SLOC per hour. It is generated by dividing the SLOC produced by the effort expended in coding and integration activities.

- **Test Preparation Productivity**

This metric measures the efficiency of test procedure preparation in units of test cases per hour. It is derived by dividing the number of test cases produced by the total hours spent on the test preparation.

- **Defect Closure Rate**

This metric indicates the average number of hours spent on closing defects. It includes effort expended on the analysis, coding, testing, and verification of the defect but excludes any management effort.

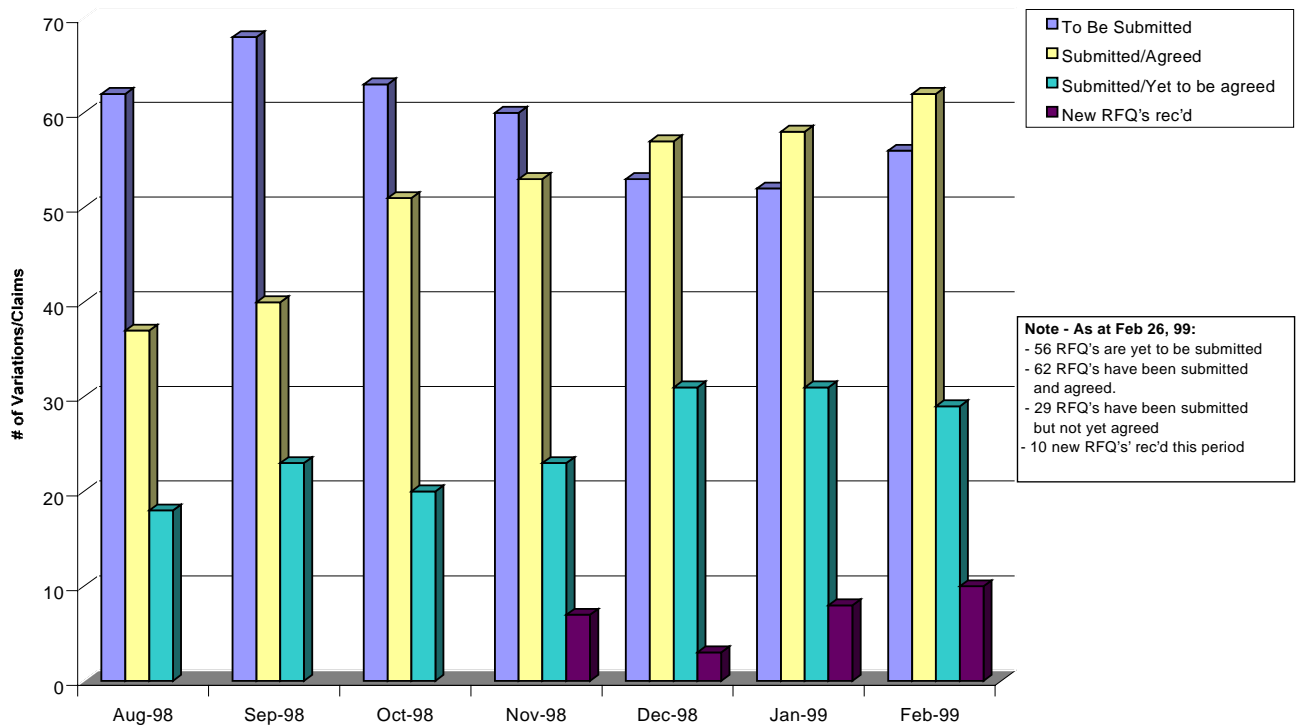
Since not all defects are of the same level of difficulty, the effort to resolve one defect may be quite different from another. A fair sample size is required to provide a meaningful indicator of the defect closure rate. On the JLE project, defect closure rate was calculated using cost data, which was associated with closing over 1,000 defects.

### **Scope Change Metrics**

- Number of Requests For Quote (RFQ)

On the JLE project, the MCS is used to control and monitor trains, and to provide communication between the operators and various information systems. The MCS in effect interfaces to systems that were provided by other contractors.

Issues at the interfaces and progress of other contractors have prompted numerous scope changes which impacted the MCS development, in both the commercial and technical sense. Each scope change request, or contractual variation, from the customer was tracked by the RFQ procedure. The procedure enforces a proper change control process, which includes estimation, schedule impact, change approval, and a commercial follow-up. The RFQ metrics track number of RFQs raised, submitted, and agreed with the customer. Figure 1 below displays the various pieces of RFQ information tracked on the JLE Project.

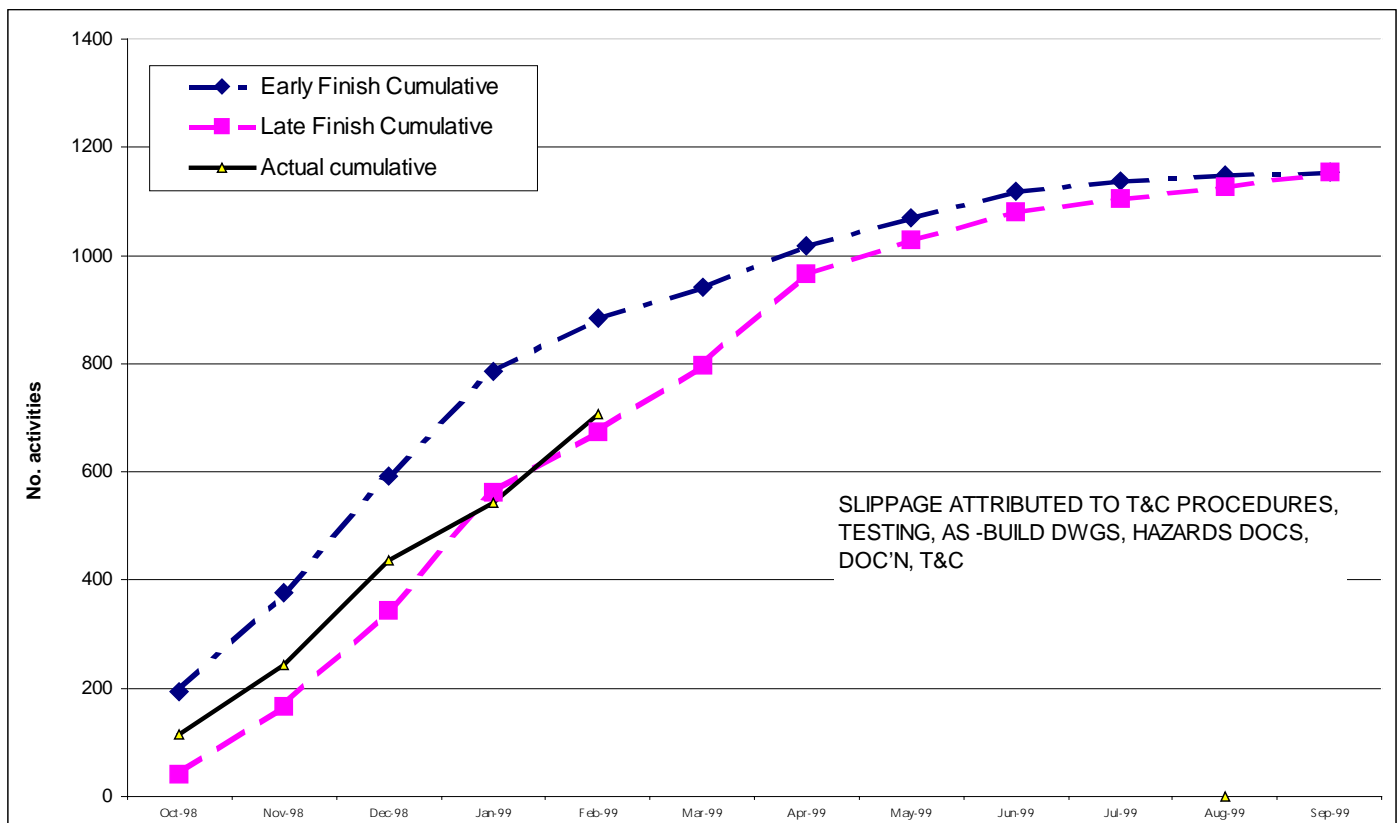


## Schedule Management Metrics

**Figure 1 RFQ Metrics Chart**

- Activity Completion Curve

This curve compares the total number of completed schedule activities against targeted early start and late start activities. On the JLE project, status to the schedule was collected on a weekly basis. The schedule update generated the necessary data for the activity completion curve. Figure 2 below shows the JLE project Activity Completion Curve.



- Test Metrics

On the JLE project, the integration and commissioning test progress are measured by comparing the actual number of tests prepared and executed against a set of targets. The test metrics comprised of both the targeted and actual number of test steps prepared, executed, passed, and failed.

- Number of Contract Document Deliverables in the Requirements List (CDRL)

This metric tracks the actual number of document submissions against a monthly target. The target is derived from the project schedule which tracks document submission milestones, and the submission milestone is generated from the regular schedule update.

### Cost Management Metrics

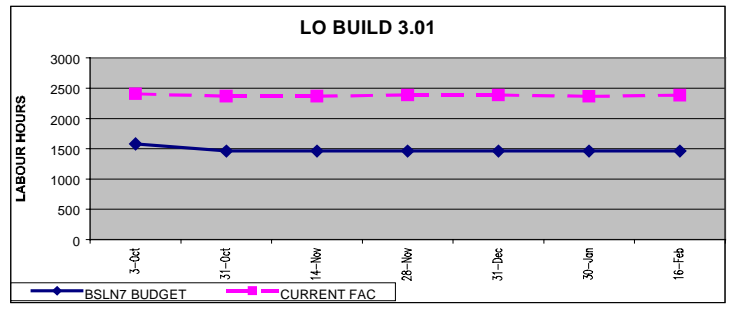
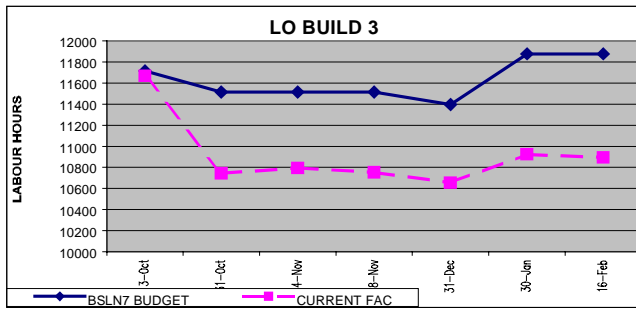
- Forecast At Completion (FAC) Trends

FAC, calculated at the work package level, provides a projection of the project cost at completion. It comprises of actual hours spent to date and the estimated hours to go. This is a key management metric

employed on all Alcatel Canada's projects. The data required for the FAC calculation are derived from the regular cost accounting report and the schedule update.

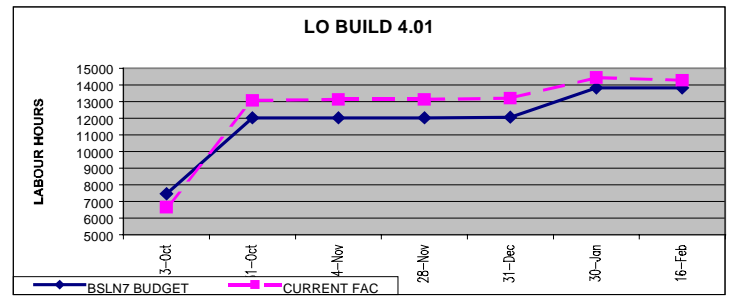
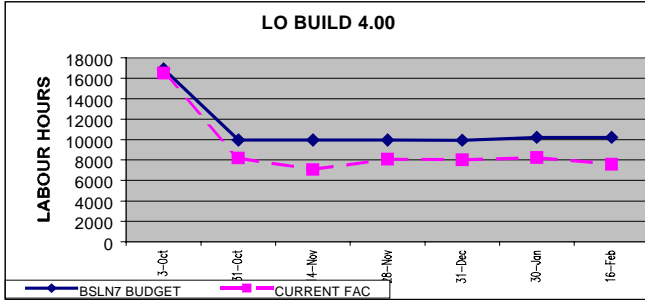
On the JLE project, FAC metrics were collected at a work package level, and rolled up to the level of work package managers, functional departments, software releases, and the project. Figures 3 and 4 illustrate FAC trends by s/w releases and functional departments, respectively.





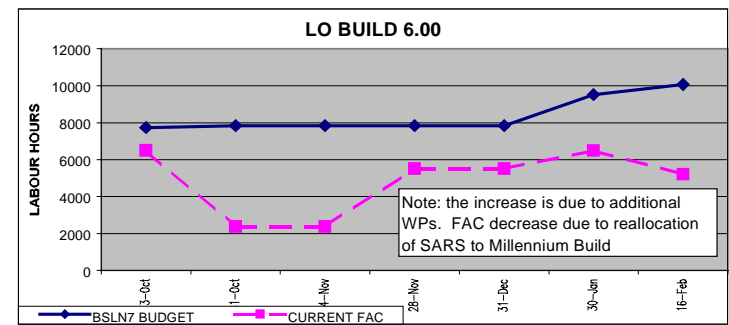
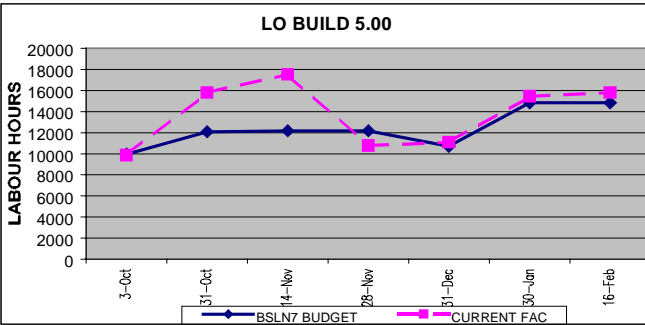
	3-Oct	31-Oct	14-Nov	28-Nov	31-Dec	30-Jan-99	16-Feb-99
BSLN7 BUDGET	11717	11515	11515	11515	11395	11877	11877
CURRENT FAC	11668	10743	10793	10753	10655	10924	10894

	3-Oct	31-Oct	14-Nov	28-Nov	31-Dec	30-Jan-99	16-Feb-99
BSLN7 BUDGET	1580	1460	1460	1460	1460	1460	1460
CURRENT FAC	2406	2368	2368	2386	2386	2364	2384



	3-Oct	31-Oct	14-Nov	28-Nov	31-Dec	30-Jan-99	16-Feb-99
BSLN7 BUDGET	16917	9947	9947	9947	9931	10216	10216
CURRENT FAC	16509	8206	7063	8077	8029	8236	7592

	3-Oct	31-Oct	14-Nov	28-Nov	31-Dec	30-Jan-99	16-Feb-99
BSLN7 BUDGET	7467	12029	12029	12029	12061	13827	13827
CURRENT FAC	6650	13074	13130	13129	13201	14452	14283

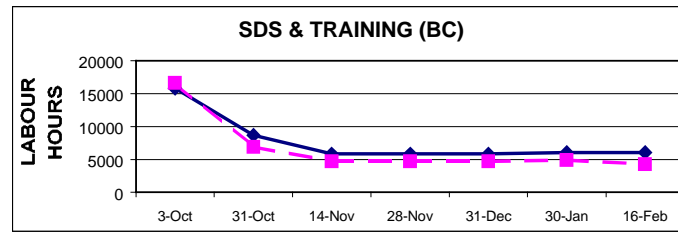


	3-Oct	31-Oct	14-Nov	28-Nov	31-Dec	30-Jan-99	16-Feb-99
BSLN7 BUDGET	9960	12093	12173	12173	10700	14836	14836
CURRENT FAC	9868	15794	17534	10760.5	11082	15439	15798

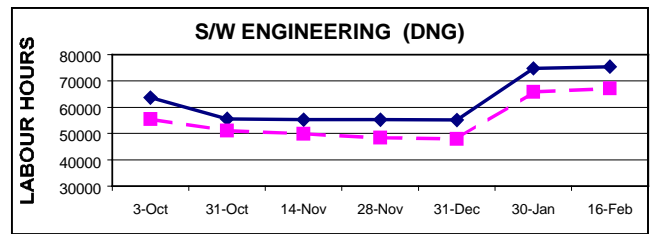
	3-Oct	31-Oct	14-Nov	28-Nov	31-Dec	30-Jan-99	16-Feb-99
BSLN7 BUDGET	7736	7836	7836	7836	7836	9518	10064
CURRENT FAC	6496	2376	2376	5516	5516	6488	5210.5

Figure 3 – FAC by S/W Releases

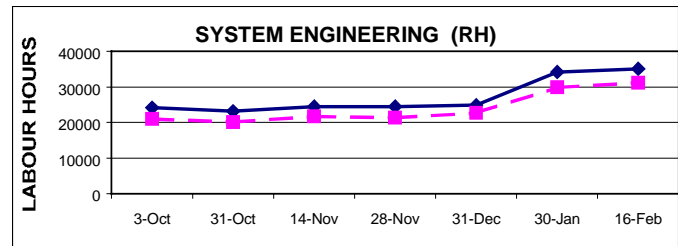
Note - Includes Baseline 7 WPs only



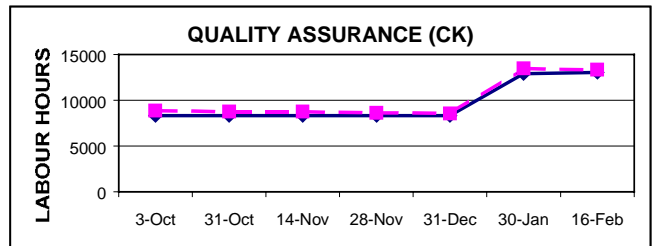
	3-Oct	31-Oct	14-Nov	28-Nov	31-Dec	30-Jan-99	16-Feb-99
BSLN7 BUDGET	15795	8680	5837	5837	5837	6061	6061
CURRENT FAC	16618	6914	4697	4697	4697	4884	4276



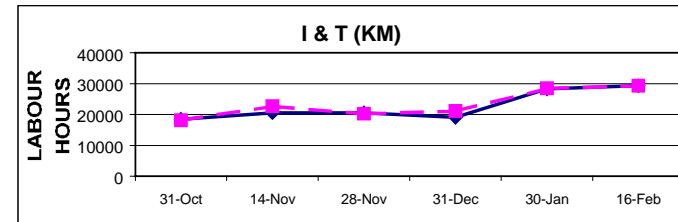
	3-Oct	31-Oct	14-Nov	28-Nov	31-Dec	30-Jan-99	16-Feb-99
BSLN7 BUDGET	63654	55539	55211	55211	55203	74767	75397
CURRENT FAC	55461	51102	49974	48357	47935	65806	67161



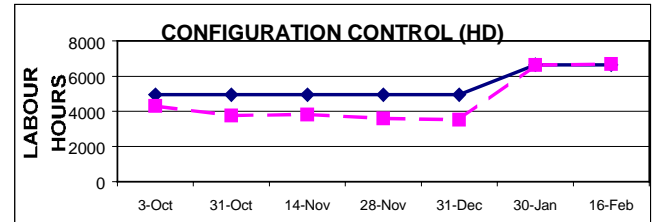
	3-Oct	31-Oct	14-Nov	28-Nov	31-Dec	30-Jan-99	16-Feb-99
BSLN7 BUDGET	24189	23149	24464	24464	24,874	34,212	35,078
CURRENT FAC	20962	20134	21680	21358	22669	29874	31152



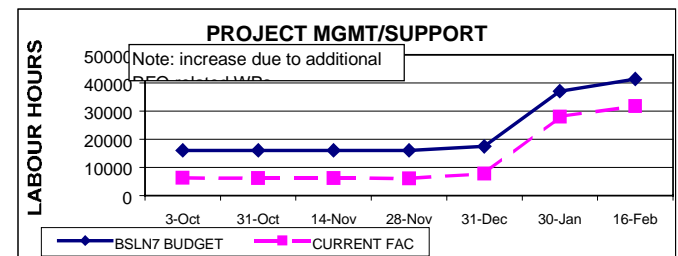
	3-Oct	31-Oct	14-Nov	28-Nov	31-Dec	30-Jan-99	16-Feb-99
BSLN7 BUDGET	8313	8313	8313	8314	8314	12899	13035
CURRENT FAC	8855	8723	8732	8604	8550	13443	13311.5



	31-Oct	14-Nov	28-Nov	31-Dec	30-Jan-99	16-Feb-99
BSLN7 BUDGET	18471	20527	20527	19053	28301	29311
CURRENT FAC	18087	22677	20217	21060	28425	29271



	3-Oct	31-Oct	14-Nov	28-Nov	31-Dec	30-Jan-99	16-Feb-99
BSLN7 BUDGET	4938	4938	4938	4938	4938	6638	6638
CURRENT FAC	4296	3751	3812	3597	3518	6627	6689



	3-Oct	31-Oct	14-Nov	28-Nov	31-Dec	30-Jan-99	16-Feb-99
BSLN7 BUDGET	16114	16144	16144	16144	17462	37062	41442.5
CURRENT FAC	6311	6177	6177	6082	7792.5	28110	31835.5

Figure 4 – FAC by Functional Departments

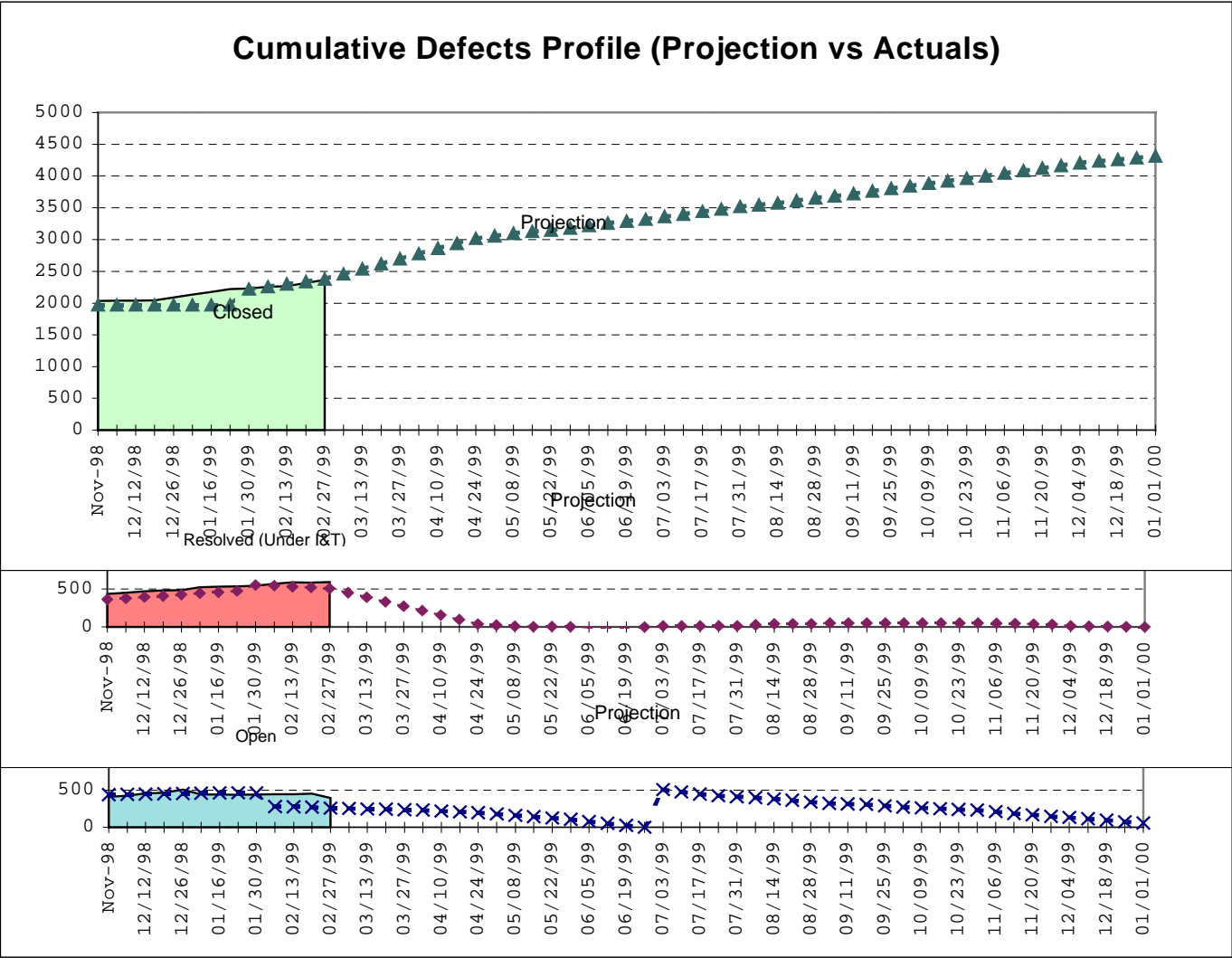
- Earned Value

This metric, tracked at the work package level, is calculated by multiplying the budget by the percentage completion. This metric provides a measurement of the cost performance of the work package managers.

### **Defects Management Metrics**

- Defect Projection Model

The defect projection model (refer to Figure 5) was derived on the JLE project to provide a weekly target for defect closure. Parameters for the projection model include number of resources, defect closure rate, number of existing defects, and number of new defects expected. Based on these parameters, the target for defect closure is generated. The actual number of new and closed defects is tracked against the projection model, and is reflected in the projection model as appropriate.



**Figure 5 – Defects Projection Model**

- Defect Trends

Tracking of the MCS defects followed a state transition process, whereby each defect normally goes through a lifecycle of problem and impact analysis, implementation, and testing. At each state transition within the lifecycle, data fields were populated to record crucial information such as the origin, type, point of detection, and severity of the defects. These data, or defects trends, were very useful for the root-cause analysis and helped identify improvements within the development process.

Figure 6 provides a breakdown of the severity, type, and origin of the MCS defects.

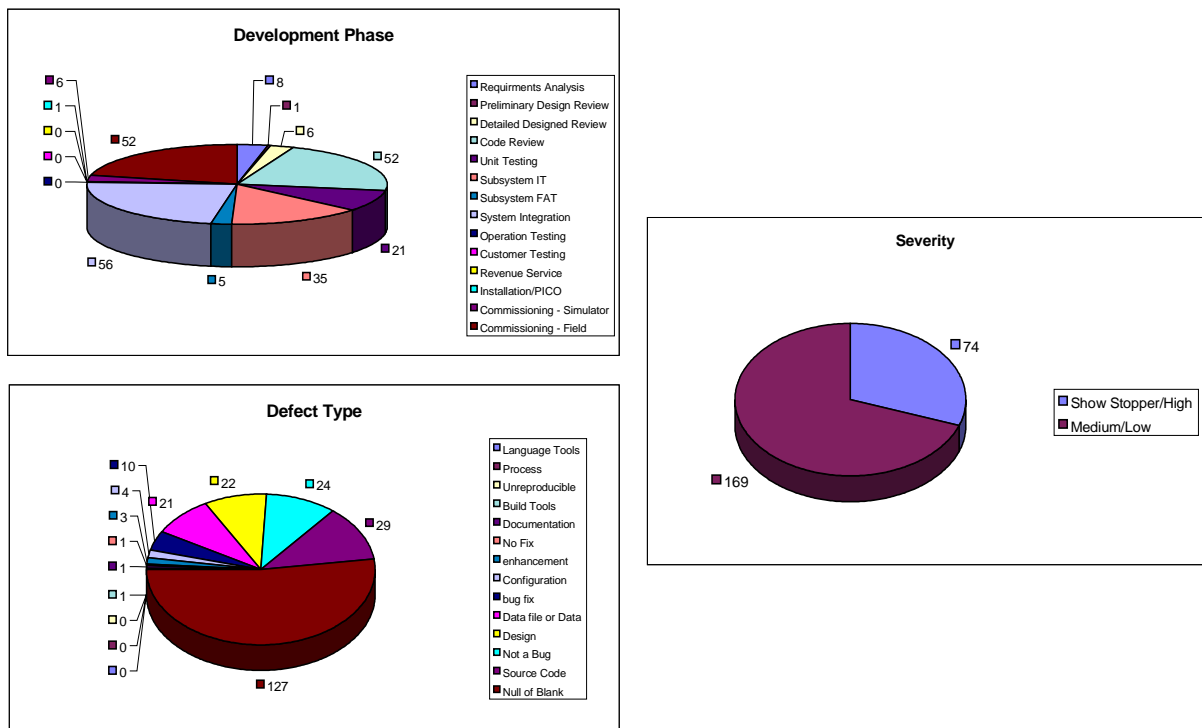


Figure 6 – MCS Defects Severity

## METRICS COLLECTION PROCESS

Metrics on the Jubilee Project are collected through the Configuration Management (CM) group as part of CM status accounting. A metrics engineer is dedicated to the task of collecting and presenting the information. Depending on the type of information, it is collected weekly or monthly. To maintain visibility of the metrics and to emphasize their importance and use, the data are posted on the Metrics Wall – a highly visible board located at a central area within the project office. The metrics are presented in a well organized manner for the ease of comparing one type of information against another. Colors are used extensively to provide interest and to highlight significant pieces of information.

As identified earlier in the paper, a variety of metrics are tracked on the JLE project. The sources for the metrics are just as wide. They include the project planning tool, Primavera, MS Excel, MS Access, a home grown tool for counting source lines of code, DOORS requirements tool and the defects tracking tool, Distributed Defect Tracking System (DDTS). The departments directly involved are Project Controls, Configuration Management, Accounting, and Project Administration. Other departments are indirectly involved as they update information and thus use the tools that are used for metrics collection. Metrics collection is automated wherever possible. This helps cut down on the collection process time. Some of the project tools already have the reporting capabilities available. For others, information is entered into an MS Access database or a spreadsheet which, in turn, generates a metrics report. However, it is not always appropriate to rely on automated tools. A small change in process can affect the information collected. Monitoring the metrics for validity or consistency is necessary prior to monitoring the metrics for project health.

It is important to note that metrics are not free. There are the obvious costs, such as the cost of establishment, collection, and production. The cost of assessment by senior staff is necessary if the metrics are to be used. There are also intangible costs. Metrics provide visibility to both the project organization and the Senior Management. They may expose the project team and even individuals to the scrutiny of Senior Management.

Likewise, the cost of not collecting and using metrics can be significant. Project indicators in one form or another are available within any project. The decision has to be made to identify what information to examine, to collect it accurately and to use that information wisely. There also has to be a willingness to change direction if necessary. Metrics may point to the need for a project to focus elsewhere, or to change established processes. If there is such an indicator, and it is not acted upon, then the metrics are not being used effectively. There is little value in collecting information, assessing it, then not acting on requirements identified in the assessment. It is only through use and action can the true value of metrics be recuperated.

## **EFFECTIVE USE OF METRICS**

There are numerous ways to use metrics within the Project Management discipline. If we accept the fact that Project Management is the art of delivering projects within the time, cost and schedule allocated, then a project manager must exercise control on all elements of the project. As such, you can't control what you can't measure.<sup>2</sup> Therefore it is of utmost importance to plan what will be measured and pay close attention to the interpretation of the measurements. Useful metrics have to be independent of the conscious influence of project personnel<sup>3</sup>.

The metrics can be grouped into four categories:

- To monitor the health of the project, including the level of quality;
- To support decision making;
- To perform root-cause analysis for focusing improvements; and
- To collect historical data for future comparison and planning.

## **SLOC**

The size measures of software is important in software engineering because the amount of effort required to do most tasks is directly related to the size of the software<sup>4</sup>. In addition tracking software size is important to determine the delivery status of a project. On the Jubilee project, the SLOC metric provided a solid measure which helped determine the work accomplished and the work yet to be done. The work was

partitioned based on an incremental development approach. As such, the total software was subdivided into many software builds. Each software build was evaluated according to the functional architecture where the functions were assembled into software threads or modules. These were estimated in terms of SLOC. The principle of software estimation used is outside the scope of this paper but it is interesting to note that the software estimations were better than 83% accurate. Figure 7 below shows the software estimate graph used. It is important to note that monitoring the SLOC metric may be dangerous if related to measure performance of individuals. Therefore the SLOC metric is principally used as guide and not for evaluation of staff. In any case they must be used carefully and with judgement.<sup>5</sup>



## New & Changed SLOCS

	Estimated	Cumulative Estimated	Coded	Cumulative Coded	Total Size
<b>Buils 0 - 4</b>	230,157	230,157	235,117	235,117	
<b>Line Opening</b>					
LO 1	20,000	250,157	21,646	256,763	202,702
LO 2	5,200	255,357	7,849	264,612	210,035
LO 3	10,010	265,367	10,907	275,519	211,204
LO 3.01	2,500	267,867	6,546	282,065	216,807
LO 4.00	19,460	287,327	17,707	299,772	232,013
LO 4.01	3,200	290,527	12,531	312,303	242,045
LO 5.00	5,850	296,377	15,014	327,317	250,177
LO 5.01	20	296,397	7,297	334,614	253,397
LO 5.02	3,230	299,627	7,040	341,654	256,825
LO 6	27,430	327,057	5,059	346,713	258,260
LO 7	2,000	329,057	0	346,713	0
<b>Millenium Build</b>					
MB1	39,200	368,257	0	346,713	0
MB2	14,000	382,257	0	346,713	0
MB2.01	0	382,257	0	346,713	0
MB2.02	1,550	383,807	0	346,713	0
<b>Migration</b>					
CI-1 SMD/Depot Control System	incl. MB2	383,707	0	346,713	0
CI-2 Fixed Block Auto-Routing	incl. MB1	383,707	0	346,713	0
CI-3 Full Radio Functionality	4,095	387,802	0	346,713	0
CI-4 Automatic Failover of a Failed Processor	incl. MB1	387,802	0	346,713	0
CI-5 Full Alarm Functionality	2,395	390,197	0	346,713	0
CI-6 Software Technician	27,261	417,458	195	346,908	0
CI-7 Full Communication Functionality	14,040	431,498	1,391	348,299	0
CI-8 Full Train Management Functionality	9,945	441,443	1,228	349,527	0
CI-9 Full Train Monitoring Functionality	7,079	448,521	3,648	353,175	0
CI-10 Full Interlocking Control	7,781	456,302	1,955	355,130	0
CI-11 Ventilation	27,805	484,107	5,703	360,833	0
CI-12 Full Line Control	11,382	495,488	1,985	362,818	0
CI-13 Power	15,093	510,581	3,227	366,045	0
CI-14 ATO Control	12,881	523,462	6,259	372,304	0
CI-15 Full Regulation Facilities	11,723	535,185	3,327	375,631	0
CI-16 ATP Control	4,750	539,936	1,899	377,530	0
CI-17 Where Info	24,825	564,761	14,342	391,872	0
<b>Totals</b>	<b>564,861</b>	<b>564,861</b>	<b>391,872</b>	<b>391,872</b>	

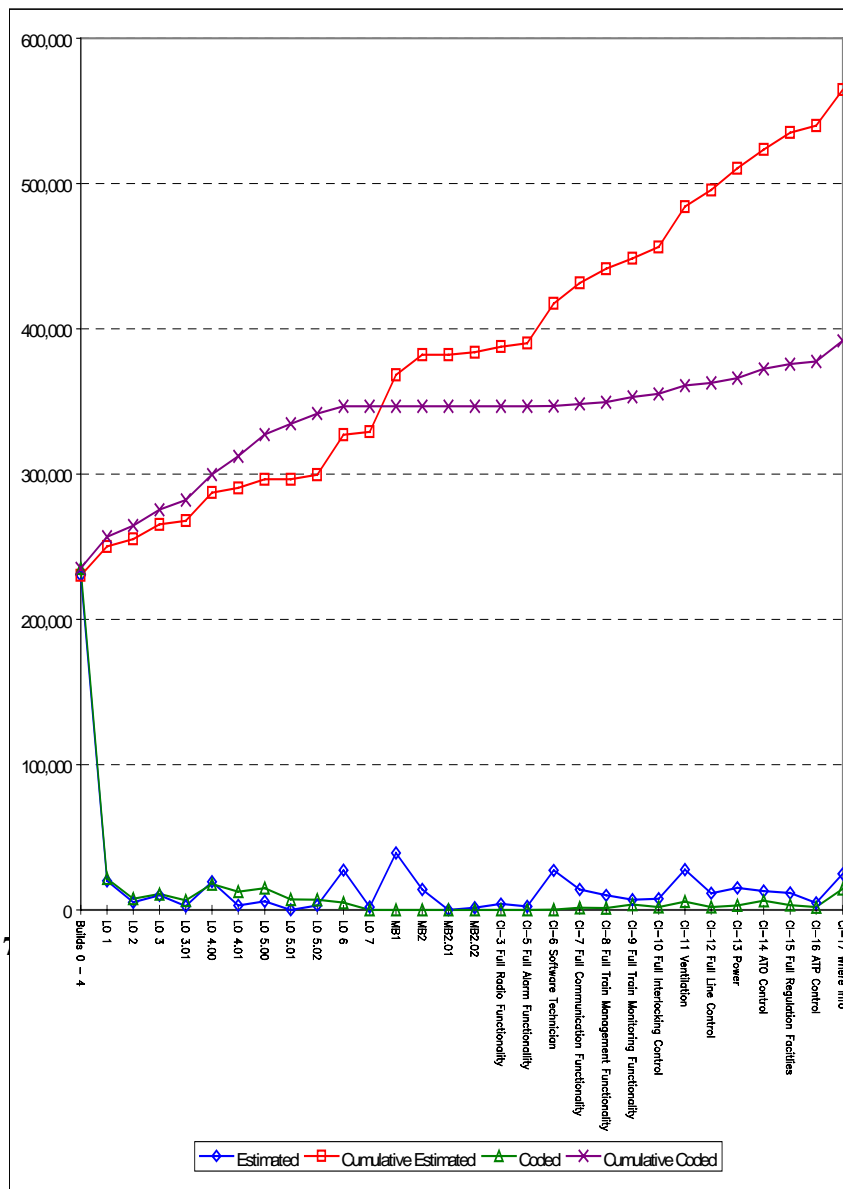


Figure 7

Progress since last report:  
 - Coded 897 SLOC for LO 5.02  
 - Coded 5,059 SLOC for LO 6

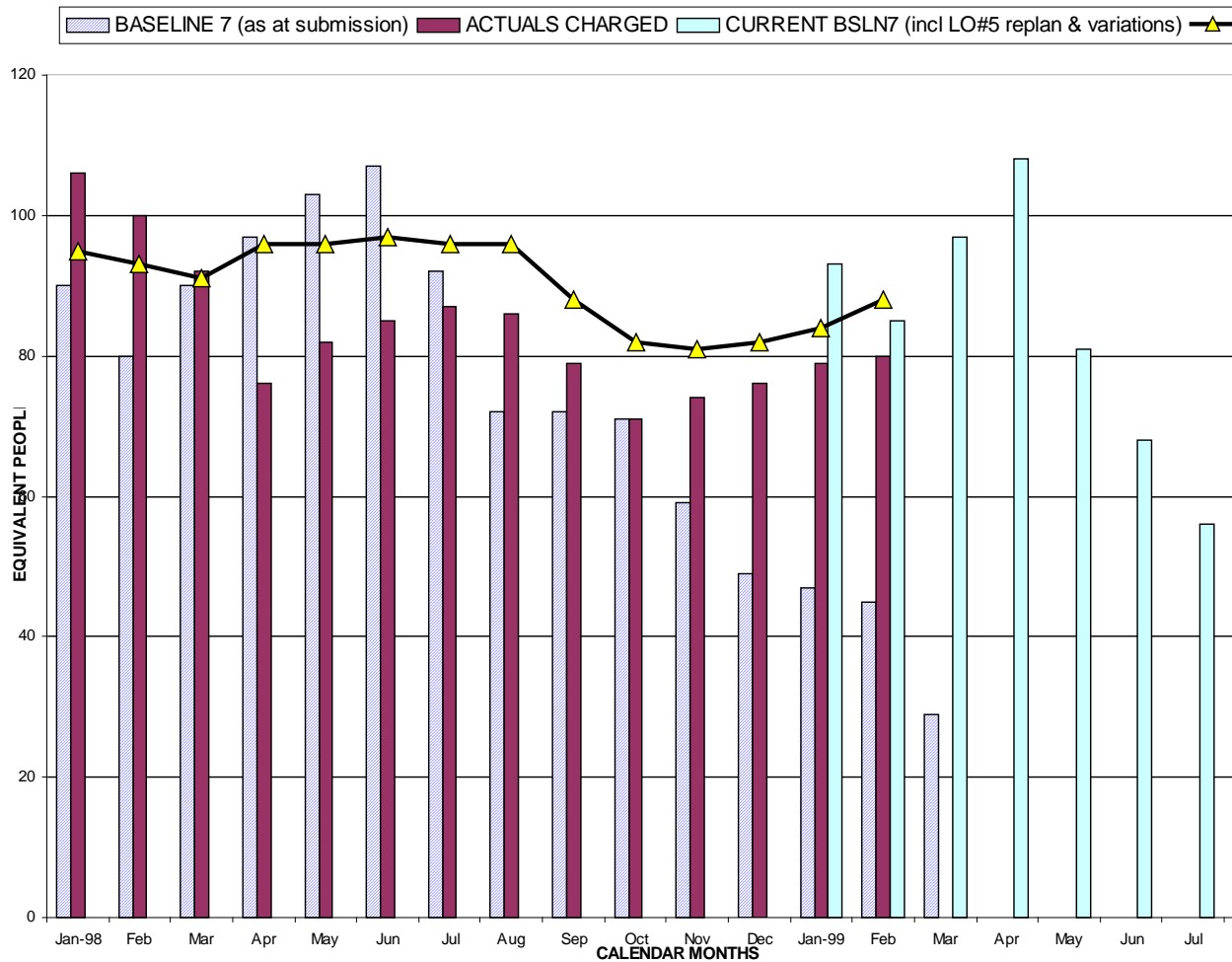
Figure 7 – MCS SLOC Metrics

### Activity Completion Curve

In addition to the SLOC metric, the number of activities planned to be closed in the schedule, or programme, on a monthly basis was a solid indicator of the work progress. As can be seen in figure 8 below, the Jubilee project kept track of the number of activities scheduled to be closed in a month. This metric translated into a dual curve, displaying early finish and late finish lines. Ideally the project should close all activities as planned on the early finish curve or at least as close as possible to the early finish line. Senior Management used this metric to obtain a quick assessment of project health.

### Resource Data

The hours invested in the project on a monthly basis was also used to monitor progress. These hours of work were translated into resource metrics. The planned resources are deducted from the scheduling tool, which extracted the resource requirements from the planned activities during the period of interest. This is collected on a monthly basis. The curve shown in Figure 8, shows the planned resources per month, the actual head count available per month, and the actual invested. It must be noted that those resource numbers are deducted from the hours planned versus hours invested including overtime, while the actual head count is collected from the actual staffing level.



	Jan-98	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Jan-99	Feb	Mar	Apr	May
BASELINE 7 (as at submission)	90	80	90	97	103	107	92	72	72	71	59	49	47	45	29	0	0
ACTUALS CHARGED	106	100	92	76	82	85	87	86	79	71	74	76	79	80			
HEADCOUNT	95	93	91	96	96	97	96	96	88	82	81	82	84	88			
CURRENT BSLN7 (incl LO#5 replan & variations)													93	85	97	108	81

(\* labour resource hours are divided by 150 per month = 1 person

(\* includes preliminary Millenium programme)

**Figure 8 – Jubilee Project Resource Histogram**

### Number of Requirements

The number of requirements is a fundamental element in the metrics system as it represents the basis of the Requirements Traceability Matrix or RTM. The customer will normally identify system requirements in a written document. This text can be separated into distinct system requirements and inserted into a database.

In the case of the Jubilee project, the DOORs requirements tool was used. The tool provides extensive capabilities. Functions available through use of the tool are:

- The production of requirements lists,
- The assignment of a requirement to a specific system Build,
- Integration and Testing (I&T) tests written against each Build or requirement,
- Test and Commissioning (T&C) tests against each Build or requirement and
- A record of the test results so a validation process can be followed.

This approach is key to provide full requirements traceability, from the requirements themselves to the final delivery. It provides a logical ground to discuss requirements' fulfillment and system acceptance.

In the case of Jubilee, the initial requirements document was broken down into 2,867 distinct requirements, which were in turn expanded for technical reasons, to 5,478 sub-requirements. Out of these came:

- 586 hardware sub-requirements
- 2,641 software sub-requirements
- 1,077 sub-requirements to be verified by inspection
- 3,545 sub-requirements to be verified by testing

The customer recognized the RTM Database as a key instrument in tracking the project requirements. It was maintained with all changes, interpretations, and clarifications to the requirements on a timely basis.

Thus, the LUL Project Representatives made the decision to adopt the RTM Database to replace the original requirements document.

#### Number of Dialogs

Graphical User Interface or GUI screens are referred to as "Dialogs". These Dialogs represent the visible link between the operators and the application software. The Dialogs are based on a style guide, which provides a road map for the engineers to code the GUI screens. Each Dialog is on average 1,200 SLOC.

This can be estimated in terms of "person-hours" to complete, as we know the productivity factor of software engineers to code, test and debug these software modules. The number of Dialog metrics has been used to provide time and cost estimates to the customer for change requests, as well as providing senior

management an estimate of time to complete project modules. The number of Dialogs can also be useful to the Project Management team to track the size of various Builds.

#### Number of Contract Deliverables on the Requirements List - CDRLs

Normally CDRLs comprise more than Project documentation, however in this paper CDRLs are referred to as documentation. Project documentation can be very extensive. This was the case for the Jubilee project where there were well over 120 documents produced and delivered. A core set of documents was identified as the set of documents needed by the HMRI (Her Majesty's Railway Inspectors) to commission the system. These represent or contain the information necessary to capture the information describing the design, delivery and operation of the system.

This set of documents (37 documents) had to be modified for each major Build in order to assist the team producing the safety case to HMRI. The documents were scheduled in the project schedule or programme, and tracked on a monthly basis. This served to provide the customer a feel for the progress of the work, and for Alcatel's Senior Management to monitor the health of the project on a monthly basis. Within the documentation metrics we successfully collected productivity metrics. For instance, for certain types of documents, the figure of 3 hours per page was used accurately to estimate production. An average number of pages per document type was also established.

#### Number of Data Tables and Data Elements

The Data metrics are very important on a project such as Jubilee. A large MCS such as Jubilee is heavily based on data and the data will inevitably change during the course of the project development and delivery. Establishing specific metrics on the cost of changing data elements in terms of number of data changes in a table, or number of table changes in a build greatly helps the customer identify the cost of any change.

#### Number of Integration Tests and Number of Formal Commissioning Tests

When a decision is made to deliver a large software product using an incremental delivery approach, it is important to consider the cost of testing. Testing becomes more costly due to the delivery technique; more regression tests are required. Testing application software, which becomes larger and larger with every iteration, becomes costly. There are two costs associated with the testing: the cost of writing test procedures, and the cost of actually conducting the tests. It is important to keep track of each of these costs.

The number of test procedures can be broken down into a number of test steps. This metric is useful to assess the size of the effort for each phase of the commissioning work. On a monthly basis it is interesting to note the progress made against the prediction.

#### Defect Metrics

The number of defects uncovered in the product is one of the most interesting metrics available to the Project Management Team. It provides information that may be critical in effecting changes in the Project organization. Defects are an integral part of any software system. There are many industry standards to consider when analyzing defects. The number of defects depends on many variables such as programming language, the type of software, and the level of complexity, etc. Whatever the standard used, the defects metrics provide quality indicators of the product, as well as providing indicators which may be useful in the identification of corrective actions necessary in the Project team.

For instance, on the Jubilee project, early in the analysis of the defect metrics, it was discovered that many of the defects were due to system design. When discussing the nature of the defects with the system design team, it became obvious that the level of experience was low. A corrective action was taken. More experienced staff were assigned to the system design team, and an immediate improvement was observed.

Defect analysis became a very beneficial exercise helping to improve the overall performance of the project. For Senior Management, defects metrics were helpful to assess the quality of the product being delivered.

The defect model, shown in Figure 9 below, provided an effective means of forecasting resources to maintain a certain level of system stability. Through tweaking of team size, defect closure rate, and number of new defects expected, the defects profile can be projected. Thus, by going through an iterative process one could fine-tune the model to derive the resource level necessary to resolve a specific number of defects within a timeframe. Actual number of defects raised and addressed and actual defect closure rate were collected and fed back into the projection model periodically to increase the accuracy.

Defects Projection		LO 5.00	LO 5.01	LO 5.02	LO 6	LO 6	LO 7	LO 7	Milln.	Milln.	Milln.	Milln.	Milln.	Milln.	
Data Date = 1999 Mar 12th		Nov-98	Dec-98	Jan-99	Feb-99	Mar-99	Apr-99	May-99	Jun-99	Jul-99	Aug-99	Sep-99	Oct-99	Nov-99	Dec-99
End of month (Cumulative)															
Projected # of Open Defects		437	454	459	258	375	335	245	140	372	302	252	192	109	17
Projected # of Resolved Defects		362	425	550	507	359	125	88	85	101	128	138	138	122	88
Projected # of Closed Defects		1968	1968	2218	2378	2737	3057	3217	3359	3559	3692	3842	4042	4202	4352
Targets for month															
Projected # of New Defects		0	80	100	80	64	46	33	34	80	90	110	140	60	25
Projected # of Defects resolved		0	63	69	117	86	86	123	139	200	160	160	200	143	117
Projected # of Defects verified		0	0	0	160	320	320	160	142	200	133	149	200	160	151
Resolution Resources															
Application Team		1.0	1.0	1.9	1.0	1.0	1.0	2.3	3.0	3.0	3.0	3.0	2.0	1.0	1.0
Data Team		1.0	1.0	1.6	1.0	1.0	1.0	2.0	2.0	2.0	2.0	2.0	2.0	1.8	1.0
HMI Team		1.0	1.0	2.8	1.0	1.0	1.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	1.0
STM Team		0.0	0.0	0.5	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Platform Team		1.0	1.0	1.5	1.0	1.0	1.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	1.0
Sys Eng Team		0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
SARs Team		1.0	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5
Total Resolution Resources		5.5	6.0	10.3	7.0	7.0	8.0	11.3	12.0	12.0	12.0	12.0	12.0	10.8	7.0
Verification Resources															
I&T Team		0.0	0.0	1.0	2.0	2.0	0.8	1.3	0.5	0.8	0.9	1.0	1.0	1.0	0.6
Test Prep Team		0.0	0.0	1.0	2.0	2.0	0.8	0.5	1.0	0.5	0.5	0.5	0.5	0.5	0.5
Total Verification Resources		0.0	0.0	2.0	4.0	4.0	1.6	1.8	1.5	1.3	1.4	1.5	1.5	1.5	1.1
Total Resolution Hours		880	960	1640	1120	1120	1600	1808	2400	1920	1920	2400	1720	1400	1400
Total Verification Hours		0	0	320	640	640	320	284	300	200	224	300	240	226	226
Total Defects Hours		880	960	1960	1760	1760	1920	2092	2700	2120	2144	2700	1960	1626	1626
Resolution rate (hr/defect)		14	14	14	13	13	13	13	12	12	12	12	12	12	12
Verification rate (hr/ defect)		3	2	2	2	2	2	2	1.5	1.5	1.5	1.5	1.5	1.5	1.5

#### Projection Change History

Jan 25/99

1. Week ending Jan 30/99 is based on actuals from Jan 23/99.

Feb 8/99

1. Updated projection model to resolve all showstoppers/high severity open defects & 50% med/low severity defects (277) by LO7. Remaining 166 med/low severity defects will be deferred to second half of the year. Note that all new LO defects projected defects are assumed to be closed by LO 7.

Mar 8/99

1. Week ending Mar 13/99 is based on actuals from Mar 6/99.

#### Definitions

\* # of Open defects at end of period = # of Open defects at start of period + # of New defects detected in the period - # of defects Resolved in the period.  
 \* # of Resolved defects at end of period = # of Resolved defects at start of period + # of defects Resolved in the period - # of defects Verified in the period.  
 \* # of Closed defects at end of period = # of Closed defects at start of period + # of defects Verified in the period.  
 \* Projected # of New defects for the periods is hard coded.  
 \* Projected # of defects Resolved = (Projected Total Resolution Resources X Projected Total Resolution Hours) / Projected Resolution Rate  
 \* Projected # of defects Verified = (Projected Total Verification Resources X Projected Total Verification Hours) / Projected Verification Rate  
 \* Open & Resolved defects include all defects with target release = LOX, SDSX, NULL, TBD.  
 \* Closed defects include defects with all target releases.  
 They exclude coding/design management type defects, and exclude RFQs

Figure 9 – Defects Project Model

### Number of Contract Changes or Scope Changes

The most common problem associated with a large software development project is the control of scope changes. The changes to the project scope are inevitable, and must be accepted in order to deliver a solid system to the operations, after all no one can completely determine the exact requirements early in the lifecycle of the project. The scope changes however must be very tightly controlled in order to ensure the success of the project. On the Jubilee project, the scope was protected by a rigid Configuration Control Change Process as part of the Configuration Management process. The Configuration Control Review Board or CCRB was the body which reviewed any change to the system requirements, assessed impacts, and decided on the implementation of the changes. As part of the CCRB, the contract administrator managed the costing and pricing of the changes. The number of scope change was an interesting metric, as it kept the customer abreast with the seriousness of the number of changes, as well as keeping all stakeholders fully informed with the impact of those contract/requirements changes.

### Earned Value (EV)

Probably the most well known metrics in Project Management is the Earned Value information. It is not the intent here to provide an extensive description of Earned Value, but rather a simple view on how it was used. The EV on the Jubilee project was closely associated with Work Packages (WP). Work Packages are approved by the Project Management and represent contracts with the WP managers. When WPs are closed, the difference between the initial budget and the actuals will contribute to the EV. What is interesting to note when analyzing the EV curves, is the trends as well as the major causes of the deltas between the budgeted work and the actual costs. This of course is subject to special attention at the Senior Management level.

We have seen above numerous metrics which were all used as part of the Jubilee project and which are also used on other projects within Alcatel Canada. The metrics are a key element in the project health analysis which is reviewed each month with the Senior Management Board. By collecting, presenting and analyzing numerous metrics on a project, it provides a certain level of independence. Meaning that if only one means of health measurement was used, it would be relatively easy to skew the measure and mislead the



management team. The more metrics that are used, the more difficult it is to mislead the reality. It is impossible to change all of the the metrics to bias the end analysis.

One of the most important factors associated with the collection of metrics is the long term benefit. The more metrics collected, the more precisely one can predict the work which has to be done. When considering that one of the most significant problems with software development is estimation, then it is easy to appreciate the value of such metrics:

- productivity factor for software development of control system with a high degree of complexity : 3.9 SLOC/hr
- productivity of closing defects: 19 hours/defect
- productivity of writing test procedures: 14 hours / test procedure

It then becomes obvious that the value of metrics are not only short term but provides great value in the long term to estimate future work.

## **LESSONS LEARNED**

Extensive preliminary planning by project management is necessary to identify what types of information will be most useful. These are the indicators that will tell the Project Manager how well the project is doing. Regular assessment of this information is the only way to maintain control. Historical data can be used to plan and develop projection models. Feedback from the collected information to projections will keep the model up to date.

A solid definition of what is measured is essential. Terminology can be a hindrance if the metrics are interpreted incorrectly. Each chart posted on the Metrics Wall contains definitions of the measurements collected.

A positive company culture must be present at all times. If metrics are seen to be used against a particular group, productivity will drop. This will produce a downward spiral effect. Metrics collection and use will not be supported, and subsequently the project will suffer. A positive environment, accompanied by

support for negative information will overcome this. If an area shows negative in the metrics, it must be examined to determine the cause. A lack of training may be the culprit. If so, training must be provided. There may be problems with existing methods or processes. There must exist within the company a willingness to change where necessary. Problem areas must be supported.

Metrics can be expensive to collect. The people assigned to metrics collection have to take the time to ensure the information is correct. This can involve a large number of people. Maintenance of this information is another factor. The use of automation, and an effective use of tools is essential.

Metrics must add value to a project. They have to be accurate. The project manager may have identified the metrics to be collected initially, however, support from all staff members is required. This can be in the form of preparing metrics data, or using this information during staff meetings to measure group success. Wider knowledge and use of this information produces more support and drive to project completion.

Finally, metrics data collection and analysis are error-prone activities. If you look at the data, draw your conclusions, and throw the data away, you will invariably lose faith in the conclusions later on. This will be particularly true if the conclusions are politically unpopular. In order to make a believable case for any metric analysis (believable even in the eyes of the person who does the analysis), it is necessary to save the raw data as well as having a methodical audit trail of the analysis process. If the conclusions are reconstructible from the data, they will be easier to defend and interpret<sup>6</sup>

---

<sup>1</sup> Software Quality Assurance Handbook, G. Gordon Schulmeyer, James I. McManus, Prentice Hall PTR, pp 403

<sup>2</sup> Controlling Software Projects, Tom DeMarco, Yourdon Press, pp3

<sup>3</sup> Controlling Software Projects, Tom DeMarco, Yourdon Press, pp 52

<sup>4</sup> Managing the software process, Watts S. Humphrey, SEI series in software engineering, pp 309

<sup>5</sup> Managing the Software process, Watts S. Humphrey, SEI series in software engineering pp 309

<sup>6</sup> Controlling Software Projects, Tom DeMarco, Yourdon Press, pp 53

# WHAT TO DO AFTER THE ASSESSMENT REPORT?

**Curtis Cook**  
**Computer Science Department**  
**Oregon State University**  
**Corvallis, OR 97331-3202, USA**

**Marcello Visconti**  
**Departamento de Informática**  
**Universidad Técnica Federico Santa María**  
**Valparaíso, CHILE**

## ABSTRACT

There are many software development process models that are used as a basis for generating assessment reports containing recommendations for software process improvement. For example, the CBA IPI assessment procedure is based on the Software Engineering Institute (SEI) Capability Maturity Model (CMM). An important question is: What to do about the recommendations in the assessment report? An obvious answer is generate an action plan to implement the recommendations. However, generating an action plan is a considerable undertaking as important decisions need to be made, questions raised, issues addressed and support gained. This paper presents a process framework for generating the necessary information and activities in the formulation of an action plan. A case study of a software development following the framework is presented.

**Keywords:** software process improvement, assessment, action planning, documentation

## Author Biographies

Curtis Cook is a professor in the Computer Science Department at Oregon State University. He received his undergraduate degree in mathematics from Augustana College and MS and Ph.D. in computer science from the University of Iowa. He has been a faculty member at Oregon State University since 1970. His research interests are software quality, software complexity metrics, software measurement, and program understanding.

Marcello Visconti is a professor in the Departamento de Informatica at Universidad Santa Maria in Valparaiso, Chile. He received his undergraduate and MS degrees from Santa Maria and Ph.D. in computer science from Oregon State University. His research interests are software process models, software quality, and software measurement.

## I. INTRODUCTION

Unfortunately many software development efforts have produced low quality products that are late and over budget. In an effort to improve this situation over the past 15 years, one branch of software engineering research has focused on the process. It is based on the premise that the software development is a process that can be controlled and improved. Hence improving the process should improve product quality and customer satisfaction.

Several software process improvement models for improving the development process and customer satisfaction have been developed. The Software Engineering Institute (SEI) Capability Maturity Model (CMM<sup>sm</sup>) [9,13,14] is easily the most popular software development process improvement model. The CMM was developed to assist the Department of Defense in assessing the quality of software produced by its contractors. It is a software process framework that defines five levels of process improvement or maturity levels for an organization. Each level gauges the organization's capability and has a set of key practices associated with it. Level 1 is low and level 5 is high. A questionnaire, interviews and other evidence is used in assessing the maturity level of an organization. To attain a particular level, all of the key practices for that level must be satisfied. These key practices also form the basis for process improvement as they provide the foundation for the activities at the next higher level. Other process improvement and capability determination methods include Trillium [8], SQPA (Software Quality and Productivity Assessment) [8], QMS (Quality Maturity System) [8], SPICE (Software Process Improvement and Capability dTermination) [7] and S:PRIME (Software Process Risk Identification Mapping and Evaluation) [16]. All of these assess the software development process against a model. The ISO 9001 is a different type of assessment process in that it uses an international standard as the basis instead of a model [8].

The IDEAL<sup>sm</sup> model [12] developed by SEI is an excellent framework for understanding the cycle of ongoing process improvement and how the various activities fit together. See Figure 1. The five phases of the IDEAL model

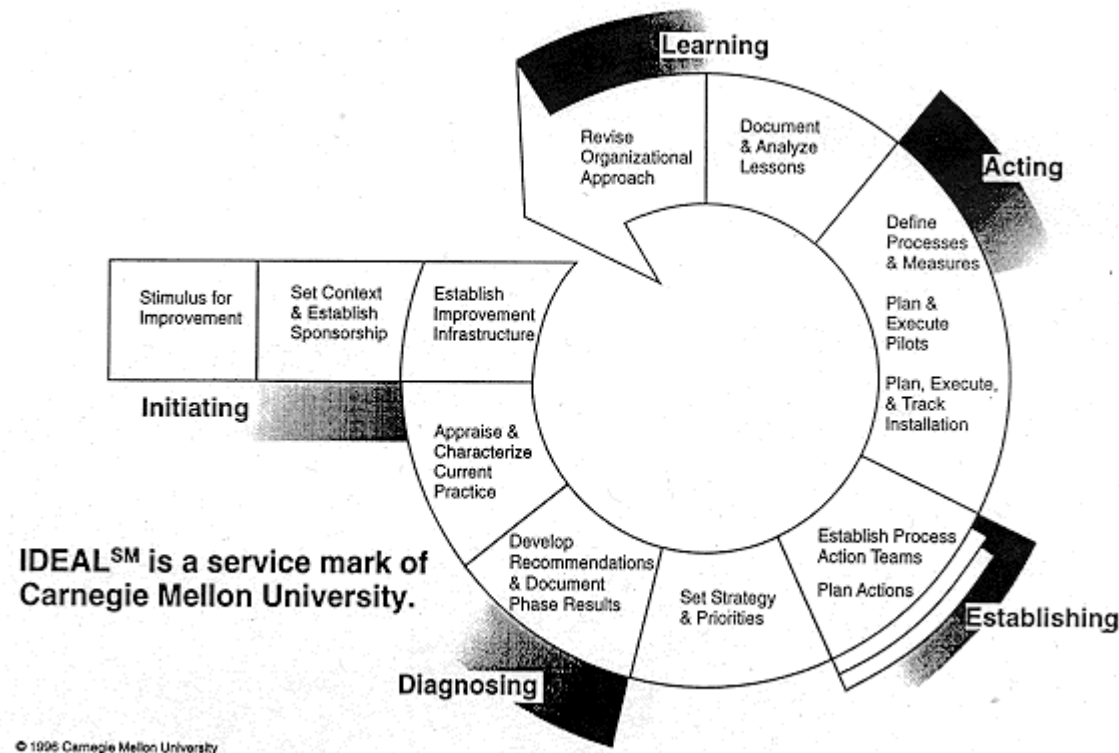


Figure 1: The IDEAL model

CMM<sup>sm</sup> is a service mark of Carnegie Mellon University.  
IDEAL<sup>sm</sup> and CBA IPI<sup>sm</sup> are service marks of Carnegie Mellon University.

are: Initiating, Diagnosing, Establishing, Acting and Learning. Activities in the Initiating phase are stimulus for improvement, obtain sponsorship, and establish the infrastructure. In the Diagnosing phase the current process is assessed and recommendations for improvement generated. Activities in the Establishing phase are set strategies and priorities, establish process action teams, and construct action plans. The Acting phase activities are carrying out the plans formulated in the previous phase and defining processes and measures. In the final phase, Learning, what was learned from an analysis of the process measures from the previous phase is documented and used in the formulation of a revised organizational approach. It in turn forms the basis for iterating the entire IDEAL process again.

Assessments from the CMM and the other model-based methods fit into the Diagnosing phase. For example CMM-based assessment and recommendations methods SPA (Software Process Assessment) and CBA IPI<sup>sm</sup> (CMM-Based Appraisal for Internal Process Improvement) [3] identify process strengths and weaknesses and attempt to foster a commitment of the personnel to process improvement. They are quite expensive and can take from 3 to 6 months to complete.

Rather than encompassing the entire software development process like CMM, we have developed a process model [19] that focuses on one particular software process – system documentation. System documentation refers to the documentation generated as part of the software development process such as requirements specifications, detailed design, or test plans. It does not include end-user documentation. The reason for focusing on system documentation is very simple: most defects discovered during software testing are documentation defects (requirements and design defects - defects in documentation that were introduced before any code was written). Empirical studies have shown that poor quality, out of date, or missing documentation is a major cause of defects in software development and maintenance [5,11,15,18]. Thus, documentation is a key component in software quality and improving the documentation process will have substantial impact on improving the quality of software. Other process models that focus on a single aspect of the software development process have been developed for testing [4] and quality assurance [20].

Our Documentation Process Maturity Model [6,19] describes the process maturity, capability and practices characterizing an organization that generates high quality documentation. A software system documentation process maturity model and assessment procedure have been developed. The model represents an ideal process and the assessment procedure determines where the organization stands relative to the model. An assessment report gives a documentation process profile indicating what practices the organization is doing well and what practices need improvement.

What to do about the assessment report is a question without a simple answer. According to the IDEAL process model, once the assessment has been completed the next step is the Establishing phase where plans for implementing some or all of the recommendations are formulated. Merely developing action plans for implementing the recommendations from the assessment is a gross simplification of this phase. In the Establishing phase important decisions need to be made, questions raised, issues addressed, and support gained. Some of the decisions, issues and questions include prioritizing the recommendations, relation of the recommendations to the business goals of the organization, and outsourcing or doing the planning internally. Support includes creating the infrastructure and organizational backing and gaining a buy in from the people who will be involved in the improvement effort. The importance of this last point is underscored by the fact that fostering a commitment from the personnel to improving the process is one of the primary objectives of an assessment procedure. For example, in the CBA IPI it is one of the two main objectives along with identifying strengths and weaknesses.

In this paper we present a process framework for generating the necessary information in the formulation of an action plan in the Establishment phase for our Documentation Process Maturity model. It leads the organization through the process of prioritizing (or modifying) the assessment recommendations, selecting the goals, and identifying the activities and information used in developing an action plan. It is important to note that an action plan details responsibilities for each task including identification of persons responsible, resources required, timeframe and deliverables. The process framework cultivates the support of the personnel who will be involved in the implementation of the recommendations by involving them in the process thereby giving them ownership of the action plan. They decide which recommendations have highest priority and identify what activities will be included in the action plan. We believe that giving ownership is an effective way of motivating the personnel to commit to supporting process improvement. Even though our process framework was developed for a particular software

process, we see no reason that it should not apply to each of the subprocesses of the entire software development process. Note that our approach to prioritize the recommendations is similar to that used in S:PLAN [10].

The next section describes our Documentation Process Model and assessment procedure in more detail. It provides the context for describing our process framework and case study. Section three presents the process framework for bridging the gap between the Diagnosing phase and the Establishing phase. It describes how the recommendations are transformed into activities that will become part of an action plan. We report the results of a case study to test our process framework in section 4. We assessed the system documentation process of a software development company (referred to as XYZ) and developed an assessment report containing recommendations for improving their documentation process. The results were reported to the organization. This report was then used as a basis for identifying the activities associated with and information needed for an action plan. Company XYZ personnel prioritized the recommendations and then selected the subset as goals for the action plan. The process helped them to identify the specific activities to be included in generating the action plan. The final section gives our conclusions and suggestions for further work.

## **I. DOCUMENTATION PROCESS MODEL AND ASSESSMENT**

The idea behind the Documentation Process Maturity Model is very simple: most defects discovered during software testing are documentation defects (requirements and design defects - defects in documentation that were introduced before any code was written). Empirical studies have shown that poor quality, out date, or missing documentation is a major cause of defects in software development and maintenance. Recall that documentation refers to system documentation used in software development and maintenance and does not include end-user documentation. Thus, documentation is a key component in software quality and improving the documentation process will have substantial impact on improving the quality of software. Our documentation process model provides the basis for the assessment of the current documentation process and guides the identification of key practices and challenges to improve the process.

The Documentation Process Maturity Model is a description of process maturity, capability and practices characterizing an organization that generates high quality documentation. See Table 1 for description of the model and Table 2 for a list of the key practices (in bold) and subpractices for each level. The model represents an ideal process and the assessment procedure determines where the organization stands relative to the model. The model and assessment procedure were influenced by the SEI's CMM in that key practices, indicators, and challenges are defined for each of the four levels of the model; for the assessment procedure a 67 question instrument is administered to each member of the project team. The tabulated questionnaire responses are used to generate an assessment report that gives a documentation process profile indicating what practices the organization is doing well and what practices need improvement.

There are several advantages to focusing on one particular software process area - system documentation - rather than encompassing the entire software development process like the CMM. First, the assessment procedure is less costly and time consuming. One attractive feature of our model is that it takes about 30 minutes to answer the 67 questions in the assessment questionnaire. It is a simple process to tabulate the responses to compute the score for each question and degree of satisfaction for the key practices. Second, it is easy to generate the recommendations – identify the small number of key practices needing improvement or that are missing. This sets the stage for the next step (Establishing phase) in improving the process; namely, investigate more thoroughly the key practice areas identified and formulate a action plan to improve them. We realize the trade-offs between our approach and a more comprehensive approach. The comprehensive approach would delve more deeply into documentation practices through a process audit by trained professionals involving interviews with project personnel, an extensive self-study

**Table 1.** Documentation Process Maturity Model - Summary

	Level 1 Ad-hoc	Level 2 Inconsistent	Level 3 Defined	Level 4 Controlled
Keywords	Chaos, Variability	Standards Check-off list Inconsistency	Product assessment Process definition	Process assessment Measurement Control Feedback Improvement
Succinct Description	Documentation not a high priority	Documentation recognized as important and must be done.	Documentation recognized as important and must be done well.	Documentation recognized as important and must be done well consistently
Key Practices	Ad-hoc process Not important	Inconsistent application of standards	Documentation quality assessment Documentation usefulness assurance Process definition	Process quality assessment and measures
Key Indicators	Documentation missing or out of date	Standards established and use of check-off list	SQA-like practices	Data analysis and improvement mechanisms
Key Challenges	Establish documentation standards	Exercise quality control over content Assess documentation usefulness Specify process	Establish process measurement Incorporate control over process	Automate data collection and analysis Continually striving for optimization

report prepared by the organization, and a more extensive questionnaire. This approach would produce an extensive report of findings and recommendations. We are trading an inexpensive and minimally inconvenient process that obtains general information and identifies problem areas for a very expensive and obtrusive process that obtains more detailed information. The general information can be used to identify areas where additional resources can effectively be committed for an in-depth investigation and recommendation. A final point in favor of our approach is that it provides a more in-depth evaluation of the documentation process than that provided by the comprehensive process models such as the CMM that evaluate the documentation process as part of the evaluation of the entire software development process.

## II. CREATING AN ACTION PLAN

The IDEAL model which presents a framework to guide the improvements is composed of 5 phases: Initializing, Diagnosing, Establishing, Acting and Learning. The purpose of this section is to introduce an approach to guide the transition from the Diagnosing phase (assessment) to the Establishing phase (action plan). The approach does not produce the actual action plan but gives a framework to generate useful information that an organization might need to build one, by providing a step-by-step method to help in the transformation of the assessment recommendations into a working action plan. This approach includes the following steps:

1. Analysis of recommendations presented in assessment report
2. Validation and prioritization of assessment recommendations
3. Selection of a set of goals
4. Derivation of actions from goals and related questions, and activities from actions

**Table 2.** Documentation Process Maturity Model Key Practices and Subpractices

Level	Key Practices	
1	---	<b>None</b> <ul style="list-style-type: none"> <li>• Consistent creation of basic software development documents</li> <li>• Consistent creation of basic software quality documents</li> <li>• Documentation generally recognized as important</li> </ul>
2	2.1	<b>Existence of documentation policy or standards</b> <ul style="list-style-type: none"> <li>• Written statement or policy about importance of documentation</li> <li>• Written statement or policy indicating what documents must be created for each development phase</li> </ul>
	2.2	<ul style="list-style-type: none"> <li>• Written statement or policy describing the contents of documents that must be created for each development phase</li> </ul> <b>Mechanism to check that required documentation is done</b> <ul style="list-style-type: none"> <li>• Use of check-off list</li> </ul>
	2.3	<b>Adherence to documentation policy or standards</b> <ul style="list-style-type: none"> <li>• Monitor adherence to documentation policy or standards</li> </ul>
3	3.1	<b>Existence of a defined process for creation of documents</b> <ul style="list-style-type: none"> <li>• Written statement to prescribe process for creation of documents</li> <li>• Mechanism to monitor adherence to prescribed process</li> <li>• Adequate time to carry out the prescribed process</li> <li>• Training material or classes about the prescribed process</li> </ul>
	3.2	<b>Methods to assure quality of documentation</b> <ul style="list-style-type: none"> <li>• Mechanism to monitor quality of documentation</li> <li>• Mechanism to update documentation</li> <li>• Documentation is traceable to previous documents</li> </ul>
	3.3	<b>Assessment of usefulness of documentation</b> <ul style="list-style-type: none"> <li>• Personal/group perception of usefulness of documents used</li> <li>• Mechanism to obtain user feedback about usefulness of created documentation</li> </ul>
4	4.1	<b>Measures of documentation process quality and usefulness</b> <ul style="list-style-type: none"> <li>• Collection of measures about usefulness of documentation</li> <li>• Tracking of documentation errors and problem reports to solutions</li> <li>• Recording of documentation process data</li> <li>• Recording of documentation error statistics</li> </ul>
	4.2	<b>Analysis of documentation process quality and usefulness</b> <ul style="list-style-type: none"> <li>• Analysis of documentation error data and root causes</li> <li>• Generation of documentation usage profile</li> </ul>
	4.3	<b>Process improvement feedback loop</b> <ul style="list-style-type: none"> <li>• Mechanism to feedback improvements to documentation practices or standards</li> <li>• Mechanism to incorporate feedback on usefulness of documentation</li> <li>• Mechanism to incorporate technology advances to the documentation process</li> </ul>

### 1. Analysis of recommendations presented in assessment report

This is the first activity after the assessment report has been released and the general recommendations have been presented. The goal is to give the assessed people an opportunity to express their views about the results in a open and free manner. The main purposes behind this step are that the participants be exposed to the recommendations and that everyone has an opportunity to express their views as well as to hear the views of others.



## **2. Validation and prioritization of assessment recommendations**

The main goal of this activity is to reach consensus on the priority ordering of the recommendations. That is which are of highest priority. To accomplish this we use a scheme in which each person has a number of points to distribute among the recommendations. In order to force the participants to make decisions we give them a number (1.5 times the number of recommendations) of points that can't be uniformly distributed among the recommendations. A similar approach is used by S:PLAN.

Once the points have been distributed and the results have been tabulated, the recommendations are sorted in order of most points received and are in turn compared against the dependency graph of the key practices derived from the maturity. This dependency graph shows the relationships of precedence between key practices indicating which practices are prerequisites for other practices according to the documentation process maturity model. The final activity of this stage is to choose a small subset of recommendations to pursue. This final step is performed by the group of assessed participants.

## **3. Selection of goals**

After a subset of recommendations have been chosen, the SEPG-like group (defined by the organization) has the responsibility to select from this subset the goals that will drive the improvement process in the next stages of the Establishing phase. This list of goals has to be sufficiently specific to allow the generation of concrete activities and tasks using the GQAA approach described below (e.g. "Define a process for the creation of documents" would be a specific goal, whereas "Improve the overall quality and productivity" would not).

## **4. Derivation of actions and activities**

The main goal of this stage is to derive information and specific activities that will address the goals selected in the previous stage. To do so we propose to use a GQM-like approach that, instead of deriving metrics, helps in the generation of activities and other useful information.

The Goal-Question-Metric (GQM) paradigm aims to tie measurement to overall goals of projects and processes because a measurement program can be more successful if it is designed with the goals in mind [1,2]. The GQM approach provides a framework involving three steps:

1. (Goals) List the major goals of the development or maintenance project.
2. (Questions) Derive from each goal the questions that must be answered to determine if the goals are being met.
3. (Metrics) Decide what must be measured in order to be able to answer the questions adequately.

The proposed approach, named Goal-Question-Action-Activity (GQAA) paradigm, captures the essence of GQM but applied with a different purpose: the idea is that an action plan can be more successful if the actions and activities are determined with the goals and relevant issues in mind. The proposed scheme has 4 major steps:

1. (Goals) List the major goals of the improvement process (e.g. recommendations or combination/modification of recommendations)
2. (Questions) List the major issues that determine the relevance of the goals being defined
3. (Actions) Derive from the goals and questions the actions that need to be taken to satisfy the goals and address the issues
4. (Activities) Derive from the actions the specific activities that need to be performed to accomplish the actions. Use the ETVX diagram to accomplish this.

The ETVX (Entry, Task, Verification, eXit) paradigm [17] facilitates the definition of activities by considering 4 aspects of an activity: entry criteria, task, verification, and exit criteria. See Figure 2 for more detailed information about the aspects.

In the GQAA, the Questions are the issues that are addressed along with each goal in order to assess its relevance, pertinence, importance, timeliness, etc. The idea behind explicitly addressing these questions is to perform a complete analysis of each goal in terms of its impact. Some of these issues to consider are:

- ✓ Current status: Determine clearly where the organization is currently with respect to each goal defined. The information derived helps in defining the gap between the current situation and the target.
- ✓ What? Determine what the organization is attempting to accomplish by defining a particular goal. The information derived helps in defining the expected scenario from satisfying a goal.
- ✓ Why? Determine the underlying motivations for pursuing each goal.
- ✓ How? Determine the way each goal is going to be achieved. The information derived helps in integrating the improvement actions into the current process.
- ✓ Who? Identification of the responsibilities involved in achieving each goal, in order to generate the proper support and sponsorship to increase the chance of success.
- ✓ When? Determine the timeframe for achieving each goal, to define a reasonable time for the expected results.
- ✓ Costs: Analyze all the different costs involved in achieving each goal, including hidden costs. The information derived will help avoid surprises in the future.
- ✓ Benefits: Derive from the motivation for achieving each goal the expected concrete benefits.
- ✓ Risks: Determine the barriers or different situations that could compromise the satisfaction of each goal, in order to determine what mitigation actions may be needed.

ENTRY	TASK	EXIT
<b>Policies</b> <b>Procedures</b> <b>Resources</b> <b>Funds</b> <b>Training</b> <b>Orientation</b> <b>Processes</b> <b>Sponsorships</b> <b>Responsibilities / Roles</b> <b>Databases</b> <b>Tools / Methods</b>	<b>Plans</b> <b>Actions</b> <b>Metrics</b> <b>Information</b>	<b>Goal Satisfaction</b> <b>Assets</b> <b>Complete Products</b>
	<b>VERIFICATION</b>  <b>Reviews / Audits</b> <b>Measurements</b> <b>Analysis</b>	

**Figure 2: ETVX diagram**

The actions represent the major decisions that will need to be made in order to pursue the goals and address the questions and issues. They can be thought of as lines of actions that define what should be done in the short, medium and/or long term to achieve the goal.

Once the actions have been defined, the activities are the concrete tasks that need to be specified and that will serve as the basis for the action plan that the company will devise. Every activity will be specified using the ETVX paradigm.

The following section describes the application of the GQAA approach in a case study involving an actual software development company.

### **III. CASE STUDY**

In this section we present a case study conducted at a small software house (referred to as XYZ) that produces mainly administrative software (MIS) and that has a relatively high level of automation of their software development process through the use of proprietary methods and a toolkit. XYZ employs approximately 30 software engineers some of whom are already using this highly automated software development process, while the rest have different levels of exposure to it. Nevertheless, they expect to have everybody using the tool in the very near future, as soon as a few projects been developed under the previous scheme are completed.

We conducted an assessment of the documentation process at XYZ. 11 people answered the questionnaires; the results and general recommendations are shown below in Table 3. The table uses the following scoring scheme: each key practice and subpractice is scored as either *very low* (VL), *low* (L), *medium* (M), *high* (H), or *very high* (VH) reflecting the degree of satisfaction of the given practice.

The general recommendations resulting from the assessment which are consistent with the results shown in Table 3 are:

1. Creation of basic software quality documents (test case design and test/quality plans)
2. Clear statement or description of what information must be included in each document
3. Mechanism to check that a software document contains information specified in the policy
4. Definition of the process to create specified documents
5. Mechanism to monitor adherence to the process
6. Control changes of the documentation
7. Monitor completeness and accuracy of documentation
8. Assessment of documentation quality
9. Mechanism to update documentation
10. Mechanism to assess documentation usability

With this information available we followed the step-by-step method described in section 3 to generate information XYZ could use to produce an action plan.

## **1. Analysis of recommendations presented in assessment report**

At the meeting each participant shared their opinions about the assessment recommendations, comments, disagreements, doubts, etc. A summary follows:

- (a) There was a general agreement with the recommendations from the assessment.
- (b) Some respondents answered many “don’t knows”, especially the newest employees, which might suggest a need for more internal dissemination of information and orientation about their software development process and the associated documentation.
- (c) There was some disagreement as to whether documentation contents are actually specified in their highly automated software development method and toolkit. The views expressed were highly dependent on the level of exposure to and involvement with the automated method.
- (d) Differences in the level of knowledge of automated software development method were reflected in the different perceptions of the documentation process. Not everyone in the company has the same familiarity with the automated software development process.
- (e) Finally, there was an overall acknowledgment that in their near future they will all be working with the technology so any process improvement endeavor should consider that.

## **2. Validation and prioritization of assessment recommendations**

After the participants had the chance to see and discuss the results as well as express their views, we asked them to distribute 15 points among the 10 recommendations. The 15 points could be distributed in any fashion as long as whole numbers of points are given to each recommendation. The rationale behind this activity is that the participants need to make a decision indicating which recommendations they agree with and which they think are irrelevant. For example, assigning no points to a recommendation would mean that recommendation is not relevant in the

**Table 3.** Assessment Results for each Key Practice and Subpractice at XYZ

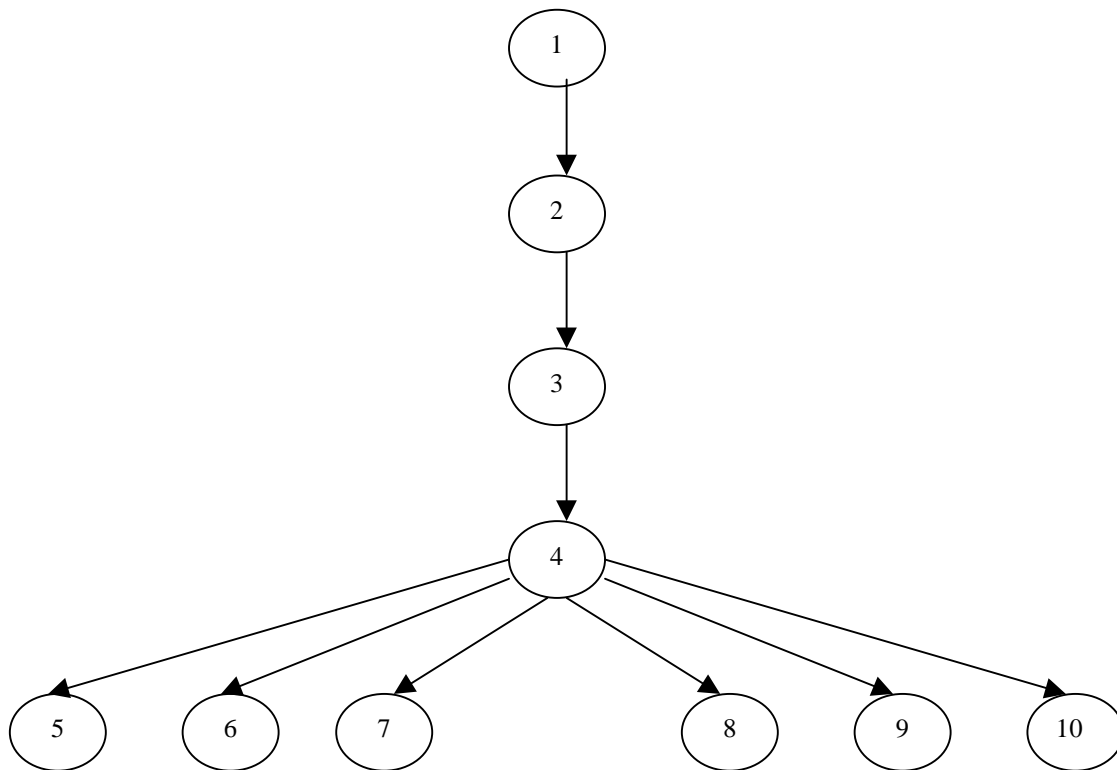
Level	Key Practices		
1	---	<b>High</b> H L VH	<b>None</b> • Consistent creation of basic software development documents • Consistent creation of basic software quality documents • Documentation generally recognized as important
2	2.1  2.2  2.3	<b>High</b> H H M <b>High</b> H <b>Low</b> L	<b>Existence of documentation policy or standards</b> • Written statement or policy about importance of documentation • Written statement or policy indicating what documents must be created for each development phase • Written statement or policy describing the contents of documents that must be created for each development phase <b>Mechanism to check that required documentation is done</b> • Use of check-off list  <b>Adherence to documentation policy or standards</b> • Monitor adherence to documentation policy or standards
3	3.1  3.2  3.3	<b>Low</b> L VL M M <b>Low</b> L L M <b>Medium</b> H L	<b>Existence of a defined process for creation of documents</b> • Written statement to prescribe process for creation of documents • Mechanism to monitor adherence to prescribed process • Adequate time to carry out the prescribed process • Training material or classes about the prescribed process  <b>Methods to assure quality of documentation</b> • Mechanism to monitor quality of documentation • Mechanism to update documentation • Documentation is traceable to previous documents  <b>Assessment of usefulness of documentation</b> • Personal/group perception of usefulness of documents used • Mechanism to obtain user feedback about usefulness of created documentation
4	4.1  4.2  4.3	<b>Low</b> VL H L VL <b>Low</b> VL L <b>Medium</b> L H	<b>Measures of documentation process quality and usefulness</b> • Collection of measures about usefulness of documentation • Tracking of documentation errors and problem reports to solutions • Recording of documentation process data • Recording of documentation error statistics  <b>Analysis of documentation process quality and usefulness</b> • Analysis of documentation error data and root causes • Generation of documentation usage profile  <b>Process improvement feedback loop</b> • Mechanism to feedback improvements to documentation practices or standards • Mechanism to incorporate feedback on usefulness of documentation • Mechanism to incorporate technology advances to the documentation process

improvement effort whereas assigning the 15 points to only one of the recommendations would mean that the given recommendation is the only action that should be taken. Since only 6 people participated in this activity there was a total of 90 points that were distributed as shown in Table 4.

With these tabulated results and the recommendations dependency graph shown in Figure 3 we asked the participants to choose a subset of recommendations to pursue. The recommendations dependency graph shows which recommendations are prerequisites for other recommendations. Given the dependency graph the participants were given the opportunity to consider modifying their selected subset of recommendations

**Table 4.** Distribution of points for each recommendation at XYZ

Recommendation	Points
1. Creation of basic software quality documents (test case design and test/quality plans)	17
2. Clear statement or description of what information must be included in each document	19
3. Mechanism to check that a software document contains information specified in the policy	5
4. Definition of the process to create specified documents	12
5. Mechanism to monitor adherence to the process	4
6. Control changes of the documentation	10
7. Monitor completeness and accuracy of documentation	2
8. Assessment of documentation quality	5
9. Mechanism to update documentation	4
10. Mechanism to assess documentation usability	12



**Figure 3:** Recommendations Dependency Graph

The final subset of 5 recommendations and their precedence is shown below.

Highest level of precedence:

1. Creation of basic software quality documents (test case design and test/quality plans)

4. Definition of the process to create specified documents
10. Mechanism to assess documentation usability

Next level of precedence:

2. Clear statement or description of what information must be included in each document
6. Control changes of the documentation

This is the end of the open session where the whole group of participants contributed to choosing which recommendations to pursue. The software engineering committee continued with the process. This software engineering committee is composed of the people involved directly in software process improvement at XYZ.

### 3. Selection of goals

The software engineering committee was in charge of analyzing the recommendations chosen by the group of participants and selecting an appropriate set of goals. These selected goals could be derived directly from the recommendations or from a combination or modification of them.

The committee selected the following goal to pursue:

**To continue with the definition of the software process while integrating the documentation into the process**

This one goal selected was in fact a combination of recommendations 1 and 2, and is the starting point in the application of the GQAA paradigm described in section 3 and fully carried out in the next steps.

### 4. Derivation of actions and activities

Starting with the goal selected in the previous stage, the next step in the GQAA paradigm is to address the questions and issues relative to the goal in order to assess its relevance, pertinence, importance, timeliness, etc.

The group discussion resulted in the following:

- ✓ Current status: Their software development technology is in transition to all future development using their tool.
- ✓ What?: Keep focus on software documentation process while improving the overall software process.
- ✓ Why?: They identified their need to increase quality and productivity, linked to their long term goals of profitability and personal and professional satisfaction.
- ✓ Who?: They made the decision to establish a SEPG using their current software engineering committee as starting point.
- ✓ How?: They decided to incorporate the documentation as part of their software process technology (methods and toolkit), because they defined automation as the key for success.
- ✓ When?: Start immediately.
- ✓ Costs?: Training, development (conceptualization and software construction), dissemination (internal and external), alternative investments.
- ✓ Benefits?: Project control and visibility, quality of life (personal and professional), free up effort and concentrate on playing their core roles at the company, strengthen their core competencies, on the basis of higher productivity and product quality and a stable operation in an unstable environment.
- ✓ Risks?: Most important: free up effort, financial, distraction from main stream business (produce low cost, high quality products); less important: impact on people (internal and external) and a major market change.

Based on analysis of the goal and the above issues related to that goal, the following actions were selected:

- ✓ Baseline actions:
  - ✓ Establish SEPG
  - ✓ Determine current status of documentation process in software development process

- ✓ Determine current status of perceptions within XYZ of documentation process activities in the software development process
- ✓ Then:
  - ✓ Formalize software documentation process while integrating it in software development process
  - ✓ Put it in practice at the company

Since the paper is intended to illustrate the use of the proposed method, it does not includes every activity for every action that has been selected. We include only the set of activities derived for the action “Formalize software documentation process while integrating it in software development process”. These are given below:

1. Management and people buy in.
2. Determining documentation currently being done in the automated software development process.
3. Determine documentation, if any, that needs to be added to the automated software development process.
4. Add these documents to the automated software development process.
5. Determine how adherence is currently being done.
6. Add adherence checking if needed.

For illustration purposes the ETVX paradigm was applied to only one of the above activities. Figure 4 shows the application of ETVX to the first activity, “Management and people buy in”. First the tasks were identified. Then the tasks were used to define the entry and exit criteria. Finally the verification activities were specified. It should be noted that these steps were performed iteratively; for instance, entry and exit criteria suggested additional tasks and verifications, and tasks suggested additional entry and exit criteria or verifications.

ENTRY	TASK	EXIT
<b>What’s to be sold</b> <b>Perceptions</b> <b>MH involved</b> <b>Timeframes</b> <b>Responsibles</b> <b>Documents involved</b> <b>Actions to work on people’s perceptions</b>	<b>Meeting for whole company</b> <b>Show process</b> <b>Throw ideas of change</b> <b>Get feedback</b> <b>Get/operate on perceptions</b>	<b>List of findings from meeting</b> <b>Task assignment</b> <b>Task monitoring/tracking</b> <b>SEPG schedule of meetings, tasks</b> <b>Meeting for whole company</b>
	<b>VERIFICATION</b>  <b>Feedback on meeting</b> <b>Feedback on individuals</b>	

**Figure 4:** ETVX diagram for activity “Management and people buy in” at XYZ

The case study discussed in this section illustrates the use of the framework for what to do after an assessment has been conducted and there is the need to begin the formulation of an action plan. The case study was limited to one goal, the questions related to that goal, one of the actions related to that goal, and the application of ETVX to one activity. The scope of the case study was severely limited because it was intended to be a proof of concept for the GQAA paradigm. The experience has been very positive in showing what can be gained from involving the assessed people in the analysis, validation, prioritization and selection of recommendations that lead to the definition of actions and activities that form the basis for the action plan. Based on this limited experience the scheme proposed appears promising because it is a simple yet powerful method for closing the gap between the Diagnosing phase and the Establishing phase of the IDEAL model.

#### IV. CONCLUSIONS

The IDEAL model provides an informative view of the software process improvement cycle. Most research (development of process models and assessment procedures based on these models) has focused on the Diagnosing phase. There seems to be far less research on the Establishing, Acting, and Learning phases. These phases are quite important as much of the success of a process improvement effort depends on how well these phases are carried out. These phases may be considered as a part of the validation of the process model and may provide insight into why high capability does not translate into high quality software products.

Some of the important questions raised by the Establishing, Acting, and Learning phase include the following: How to efficiently generate an action plan? How to generate an action plan that can be effectively used in the Acting phase? What are effective methods of carrying out the action plan? What is the important data to collect in the Acting phase for use in the Learning phase? How can the analysis of this data be effectively integrated into the process improvement cycle?

This paper has attempted to address part of the first question: How to efficiently generate an action plan? We have presented a method of bridging the gap between the Diagnosing and Establishing phases. Our method provides a process framework for generating the information and activities that will be needed in the formulation of the action plan. By giving personnel to be involved in the improvement process a major decision making role in the generation of information and activities, they assume ownership and hence commitment to the process improvement effort. Our case study indicated that our process framework was successful for a specific software development process – system documentation. Further research will be needed to determine whether the process framework is suitable for other specific software documentation processes such as testing or configuration management. We believe it should apply because there is nothing specific to documentation in the framework. Finally further research should consider how the framework might be modified in order to be applied to more general software processes.

#### V. REFERENCES

1. V. Basili and H. Rombach. The TAME project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, **14** (6) (1988) 758-773.
2. V. Basili and D. Weiss. A methodology for collecting valid software engineering data, *IEEE Transactions on Software Engineering*, **SE-10** (6) (1984) 728-738.
3. R. Basque. CBA IPI: How to build software process improvement success in the evaluation phase?, *IEEE Computer Society Software Process Newsletter*, **5** (Winter 1996).
4. I. Burnstein, T. Suwannasart and C. Carlson. Developing a testing maturity model: part I. *CrossTalk*, **9**(6) (1996) 21-24.
5. D. Card, F. McGarry and G. Page. Evaluating software engineering technologies. *IEEE Transactions on Software Engineering*, **13**(7) (1987) 845-851.
6. C. Cook and M. Visconti. New and improved documentation process model, in *Proceedings of the 14th Pacific Northwest Software Quality Conference*, Portland, Oregon, October 1996 (PNSQC, Portland, 1996), pp. 364-380.
7. K. El Emam, J. N. Drouin and W. Melo. *SPICE: the Theory and Practice of Software Improvement and Capability Determination* (IEEE Computer Society, 1998)
8. R. Grady. *Successful software process improvement* (Prentice-Hall, Upper Saddle River, New Jersey, 1997).
9. W. Humphrey. *Managing the Software Process* (Addison-Wesley, Reading, 1989).



10. INTEC-Chile. S:PLAN: Method for software process improvement action planning, in <http://www.intec.cl/> (Technological Research Corporation INTEC-Chile, Santiago, Chile, 1998).
11. B. Lientz and E. Swanson. Problems in applications software maintenance. *Communications of the ACM* , **24(11)** (1981) 763-769.
12. C. Myers. Managing resistance to software process improvement, in *Proceedings of VISION Conference*, Montreal, Quebec, Canada, October 1996 (CRIM-ASEC, Montreal, 1996).
13. M. Paulk, B. Curtis, M. Chrissis and C. Weber. *The Capability Maturity Model Guidelines for Improving the Software Process* (Addison-Wesley, Reading, 1995).
14. M. Paulk, B. Curtis, M. Chrissis and C. Weber. Capability maturity model, version 1.1. *IEEE Software*, **10(4)** (1993) 18-27.
15. J. Pence and S. Hon III. Building software quality into telecommunications network systems. *Quality Progress*, (October 1993) 95-97.
16. L. Poulin. S:PRIME, in *Proceedings of VISION Conference*, Montreal, Quebec, Canada, October 1996 (CRIM-ASEC, Montreal, 1996).
17. R. Radice. Getting to level 3 in the SEI's CMM, in *Proceedings of VISION Conference*, Montreal, Quebec, Canada, October 1996 (CRIM-ASEC, Montreal, 1996).
18. H. Rombach and V. Basili. Quantitative assessment of maintenance: an industrial case study, in *Proceedings of the IEEE Conference on Software Maintenance*, Austin, Texas, September 1987 (IEEE, Washington, 1987), pp. 134-144.
19. M. Visconti and C. Cook. A software system documentation process maturity approach to software quality, in *Proceedings of the 11th Pacific Northwest Software Quality Conference*, Portland, Oregon, October 1993 (PNSQC, Portland, 1993), pp. 257-271.
20. M. Visconti, Patricio Antiman and Patricio Rojas. Experiencia con un modelo de madurez para el mejoramiento del proceso de aseguramiento de calidad del software, *Novatica*, **125** (1997) 18-21. (in Spanish)

## **1999 PNSQC Conference 35-45 minute presentation**

**Title: "Steamrolling the organization with process, or is there a better way?"**

The Process Group  
P.O. Box 700012  
Dallas, TX 75370-0012  
Tel: 972 418-9541 • Fax: 972 618-6283  
E-mail: [help@processgroup.com](mailto:help@processgroup.com)  
[www.processgroup.com](http://www.processgroup.com)

### **Abstract**

Many organizations try to implement change. This includes everything from the introduction of a new tool or method, to a company-wide process improvement program. Often these attempts fail because too much is attempted too quickly with little thought to the most effective sequence of events. When new ideas are introduced they are either abandoned after a short time or adopted by only a few people.

Whenever an SEPG or process improvement team wants to deploy a change, there are some key principles that it should consider to be successful. This talk covers the use of an adoption curve that helps the team:

- Increase the speed of deployment, by determining who to work with and in which order.
- Reduce the risk of failure by building and deploying the solution in increments.
- Determine when a policy should be developed and an edict issued.

Examples of process deployment will be included in the presentation from observations of nine corporations since 1989.

### **Biographies**

Mary Sakry and Neil Potter are co-founders of The Process Group, a company that consults in software engineering process improvement. Mary has 23 years and Neil 13 years of experience in software development and management.

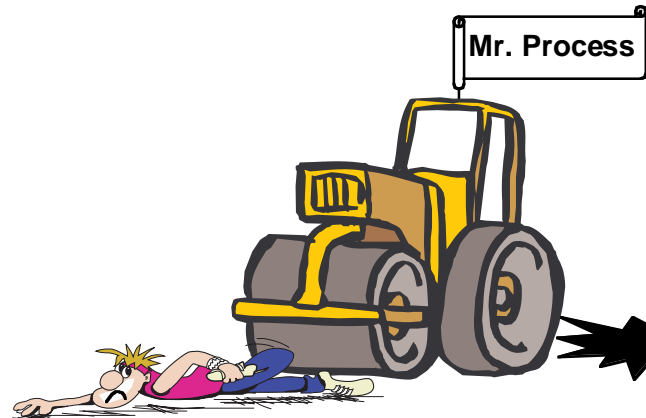
From 1988 Mary worked full-time on the Corporate SEPG within Texas Instruments (TI) to lead software process assessments across TI worldwide. The last year of TI was spent working with Neil in an SEPG for one division of TI.

For six years Neil was a Software Engineer in TI. In 1988 he created and managed a full-time SEPG spanning Dallas, England and India.

Neil and Mary have been working on process improvement for 10 years. They are SEI authorized lead assessors for CBA-IPI process and were in the first group of vendors that were authorized by the SEI.

# Steamrolling the Organization with Process, or is There a Better Way?

(based on 10 years in the process trenches)



Neil Potter  
Mary Sakry


---

*“Proving that the true skeptics are indeed truly skeptical achieves nothing, except that you’ve dented your pick and probably permanently diminished your credibility (and failed to appreciate the vital importance of building a fragile momentum).”*

[Peters, 85]

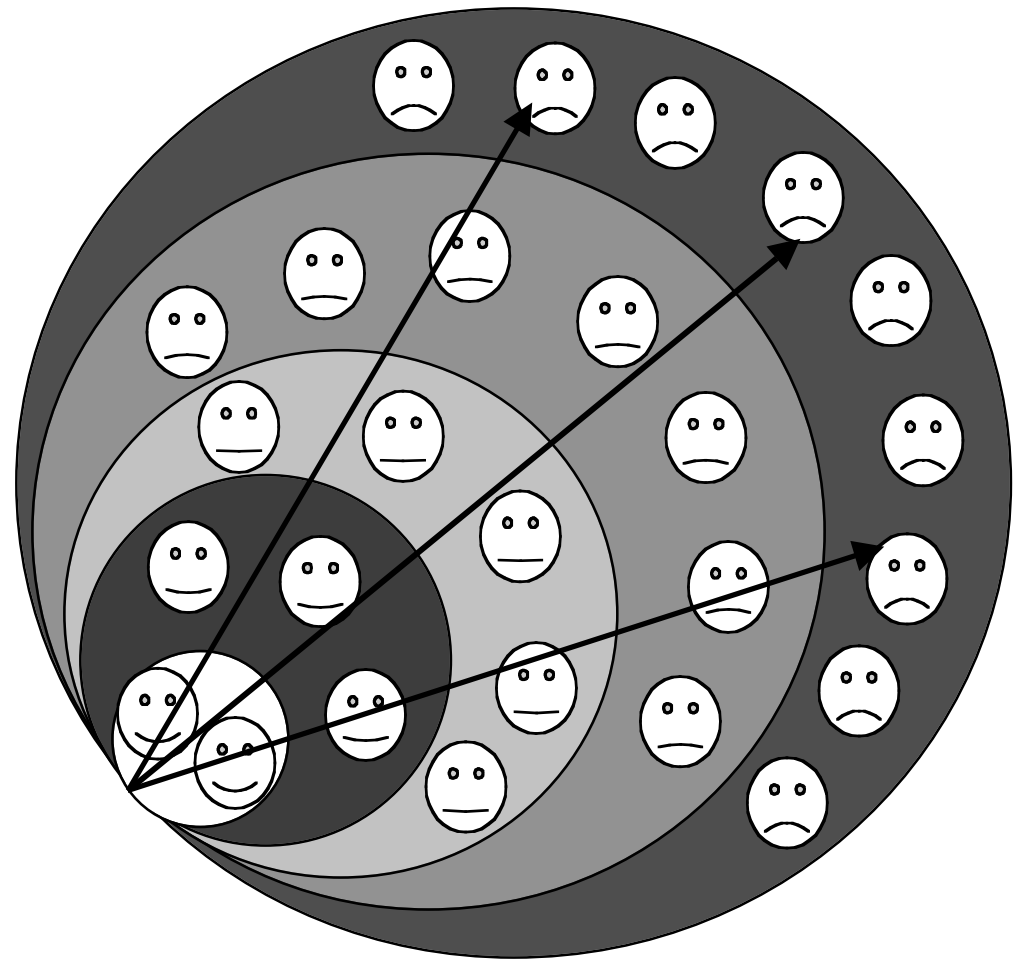
# The Problem

## Typical approach to process deployment:

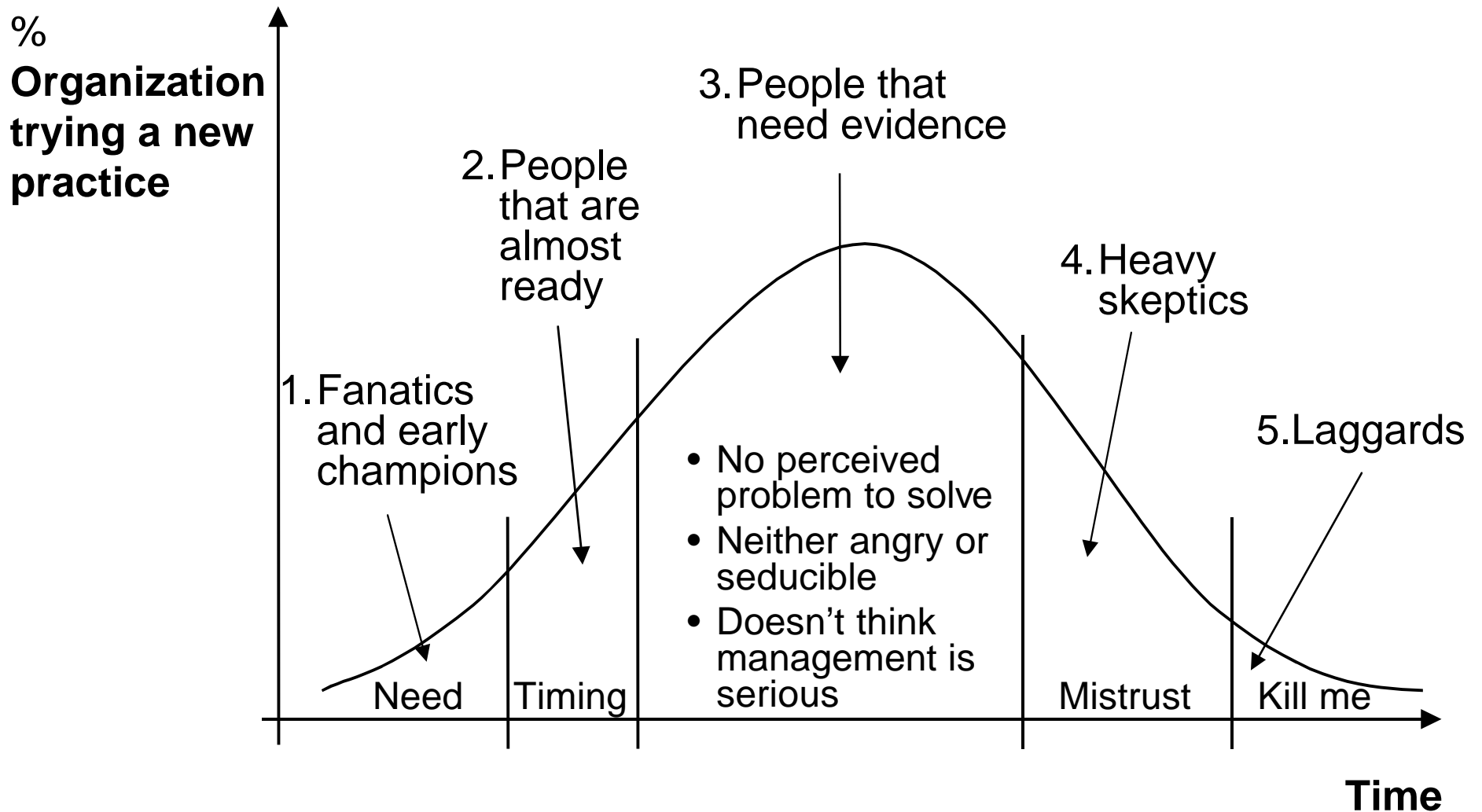
- Process improvement team is created
- A (big) process document is written
- The team assumes it is done and deployment is easy
- The process is “deployed” 
- The process is ignored, or significant resistance occurs
- The organization gives up or continues to struggle

# A Solution

- **A planned and staged approach:**
  - Builds momentum
  - Leverages success stories
  - Provides feedback to refine the solution
  - Easier to manage



# What Stages?



# How are the Groups Determined?

## 1. Interview to gather needs

- By department, project team or individuals

## 2. Sort interviewees by

- **Need** for the solution
- **Willingness** to try the solution

0 => poor match

Need now

Need later

Don't know they need it

No need & unwilling

Kill me!

# Three Uses of the Adoption Curve

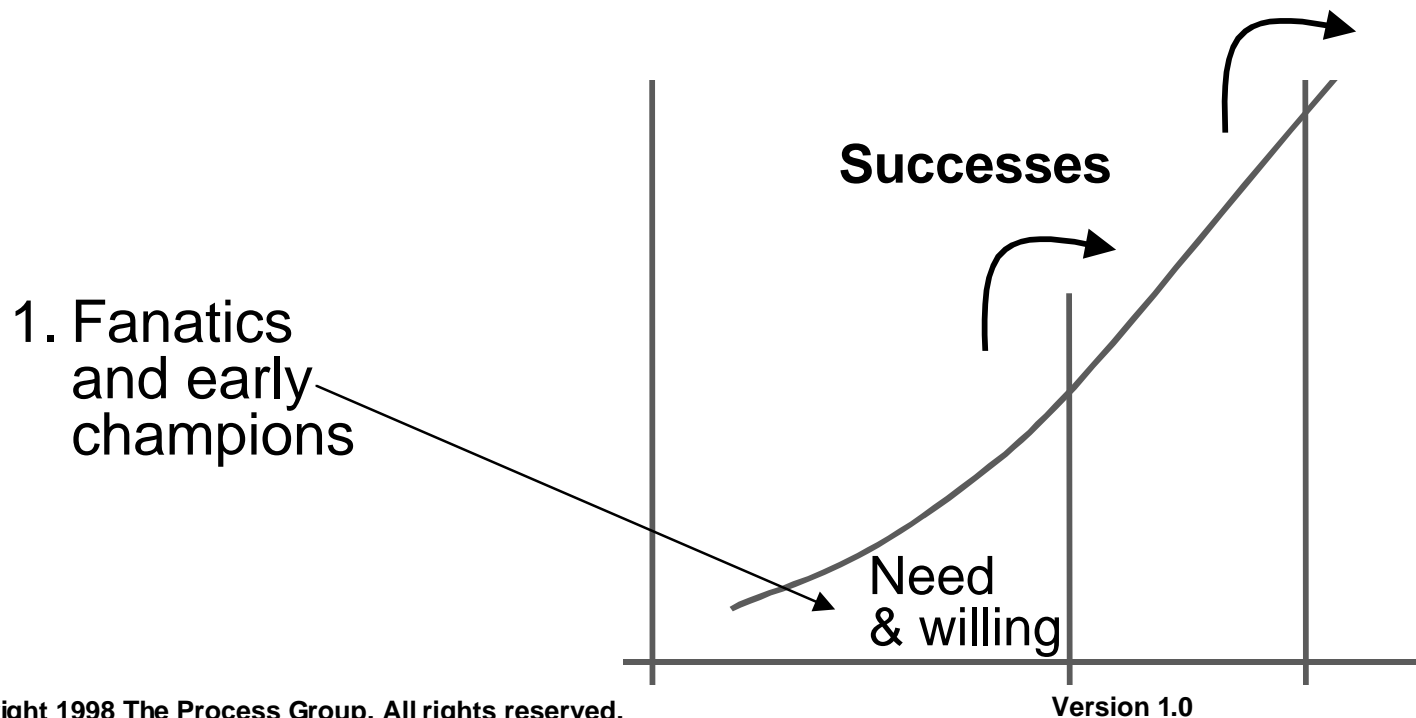
- 1. Increase the speed of deployment**
- 2. Reduce the risk of deployment failure**
- 3. Determine when a policy should be developed and an edict issued**



# Use 1: Increase Speed of Deployment

**Speed comes from:**

- Increasing motivation to adopt - based on need
- Decreasing resistance - based on willingness
- Using previous successes to influence the next group



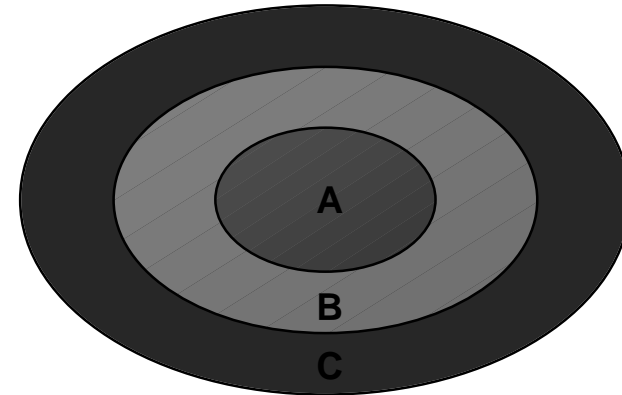
## Use 2: Reduce the Risk of Failure

**Solution increments:**

**A = Need now**

**B = Would like to have**

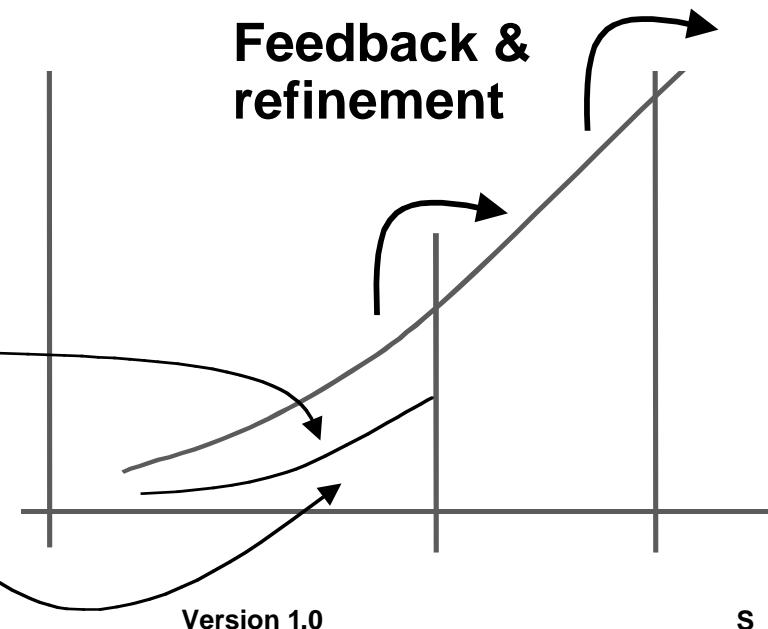
**C = Might need later**



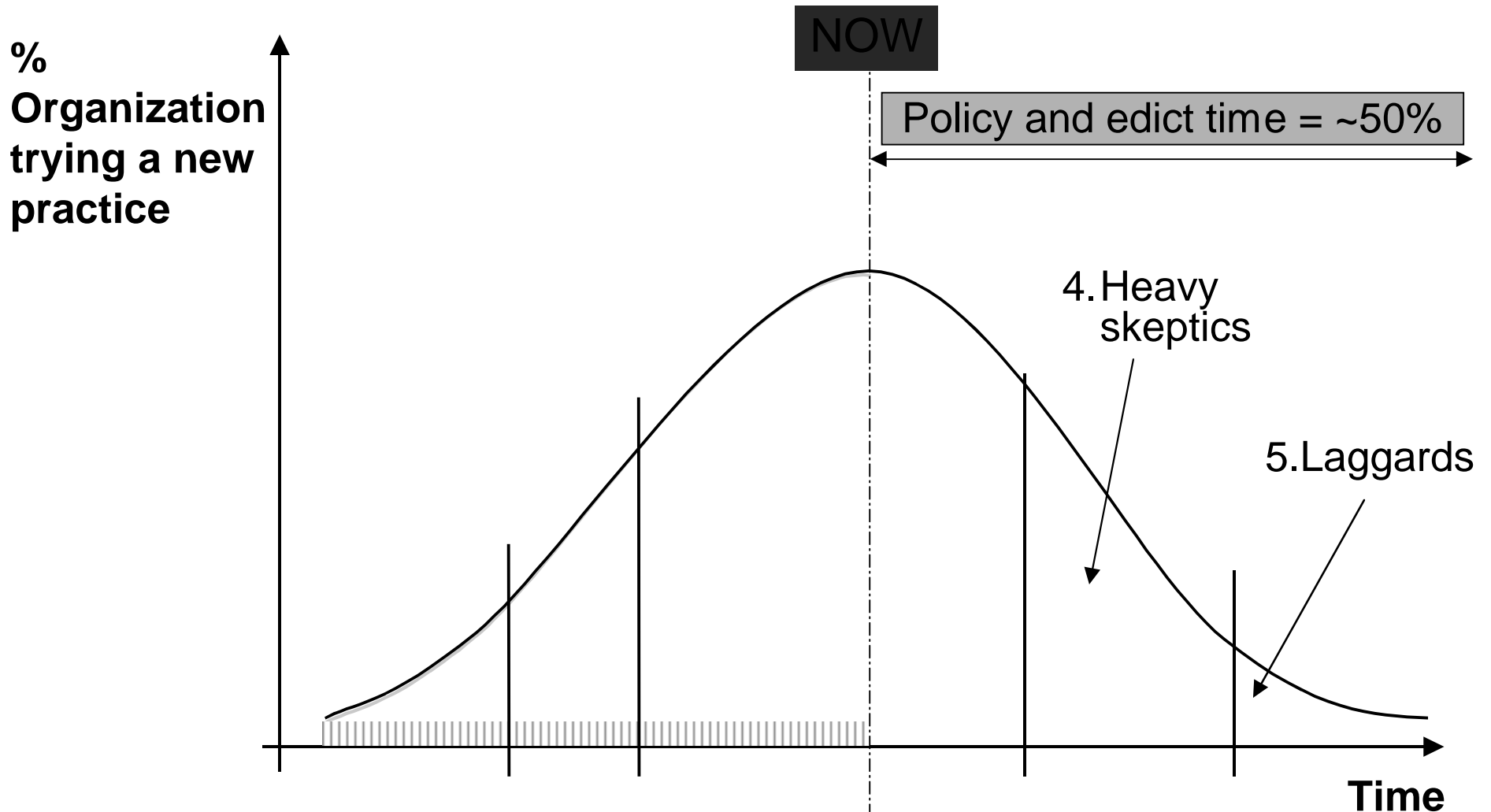
**For example:**

**Planning solution**

- A • Negotiation**
- A • Estimation**
- B • Metrics / tracking**
- C • Risk management**



## Use 3: Developing a Policy & Issuing an Edict



## Use 3: Developing a Policy & Issuing an Edict (cont.)

- **Policy states:**
  - When and where a practice should be used.
- **In the beginning:**
  - You might not have any idea!

**Wait until you get some  
experience and feedback**

- **Edict states:**
  - “Do it now, this is important.”
- **In the beginning:**
  - You don’t necessarily have proof or credibility.

**Wait until you get some  
experience and ownership**

# Summary



- **Don't use a steamroller on the organization**
- **Don't go after the hardest nut (laggard) first**
- **Focus on real needs (who needs what, when)**
- **The process provider needs to be flexible and provide appropriate, timely solutions**

- **Rogers, Everett M., *Diffusion of Innovation*, Free Press, New York, NY, 1983.**
- **Peters, Tom., Austin, Nancy., *A Passion for Excellence*, pp357, Warner Books, 1985.**
- **Company examples:**
  - Business machines company
    - » CA: Planning deployment
    - » CA: SEI Level 3 deployment
    - » NY: Planning deployment
  - Financial institution #1
    - » Lifecycle deployment
    - » Planning deployment
  - Financial institution #2
    - » SEI Level 3 process deployment
  - Government contractor
    - » Inspection deployment
    - » SEI Level 4 deployment
    - » Object oriented technology deployment
  - Aircraft manufacturer
    - » Assessment deployment
    - » SEI Level 2 deployment
  - Computer Aided Design company
    - » Inspection deployment
  - Silicon chip manufacturer
    - » Inspection deployment

The Process Group  
P.O. Box 700012 • Dallas, TX 75370-0012  
Tel. 972-418-9541 • Fax. 972-618-6283  
E-mail: [help@processgroup.com](mailto:help@processgroup.com)  
<http://www.processgroup.com>

**1999 PNSQC Conference 45-minute presentation**

**Paper**

**Title: "Steamrolling the organization with process, or is there a better way?"**

The Process Group

Tel: 972 418-9541 • Fax: 972 618-6283

E-mail: help@processgroup.com

*"Proving that the true skeptics are indeed truly skeptical achieves nothing, except that you've dented your pick and probably permanently diminished your credibility (and failed to appreciate the vital importance of building a fragile momentum)."*

[Peters, 85]

**Introduction.**

We have been coaching software development organizations for 10 years. In that time we have observed successes and failures as groups try to improve how they develop software. This paper provides advice for people who are trying to improve software development within their organization. Leveraging on our experiences, we will give you advice on how to make wise choices when helping people adopt new practices. Specifically, we will explain the adoption curve, what it is and how to use it. When you use your knowledge of the curve, you are better able to plan your approach as you work with people who need to adopt new practices. Using the curve you can avoid some of the pitfalls that have frustrated so many other process improvement professionals. These pitfalls include resistance to change, inappropriate mandates, wasted time, and overly complex process solutions that are eventually ignored.

This information is designed for Software Engineering Process Group (SEPG) members and other people who champion change. SEPG's provide a focal point for software process improvement in an organization. They plan and co-ordinate software process improvement. If you are trying to get people to improve things, this information will be useful to you.

When we say adoption, we are referring to changing behaviors. If you are currently doing something and you would like to improve it, then adoption encompasses the entire process from learning about a new idea or practice, trying it out, through to deciding that you want to continue using it. In the end, a new practice has been adopted when it is being used, and the user is getting the intended benefit. Using the adoption curve can aid you in helping people adopt new (or improved) practices, such as, a new estimation process, a new inspection process, a new configuration management tool, or an improved trouble reporting system.

Regardless of how big your organization is, it is wise to plan your approach. The adoption curve will help you select how to proceed, with the least resistance and most useful results.

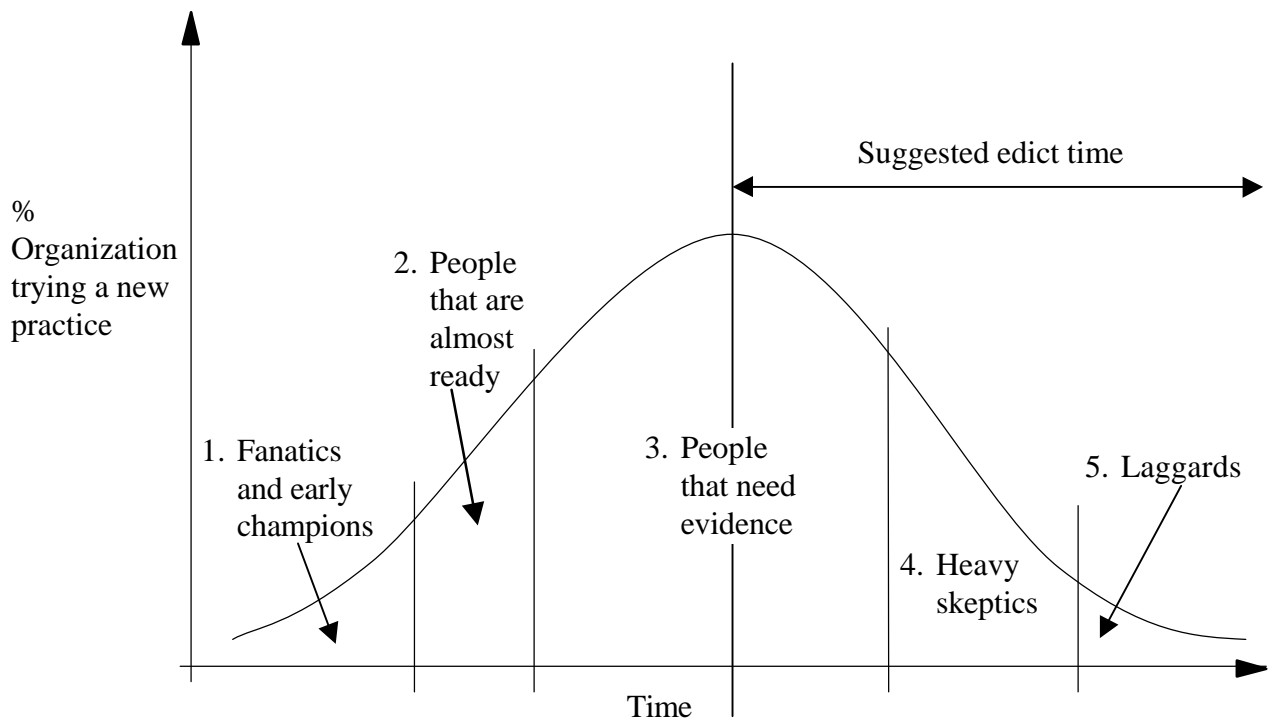


Figure 1

The adoption of any practice can be represented by the bell curve in figure 1 [adapted from Rogers, 1983]. Five distinct groups can be seen during the adoption of a new practice.

Group 1: At the leading edge you have a few who may be perceived as early champions or fanatics. These are the people that will try anything new, or ones that have a desperate need for your solution.

Group 2: After the fanatics there are people who would like to try the new practice but are not quite ready. The timing might be poor or further evidence might be needed. Beyond group two the challenges are greater.

Group 3: This group is typically the largest. This group needs considerable evidence from the first two groups before attempting the idea. Change is not seen as a priority, either because the current practices are comfortable, or there is no perceived problem to solve.

Group 4: This group consists of heavy skeptics who resist every change. There might be considerable mistrust regarding the motive for the change, or resentment built up from previous changes that have failed.

Group 5: This group contains the laggards, the people that would kill, or would like to be killed, before changing. Group five also contains people 'retired on the job'. A laggard sees a paycheck arrive in the mail whether the change occurs or not.

An individual can be placed in one of the categories by assessing his or her willingness and need for the proposed change.



### **Using the adoption curve.**

There are three primary uses of the adoption curve when deploying a change:

1. To increase the speed of deployment, by determining who to work with, and in which order.
2. To reduce the risk of failure by building and deploying the solution in increments. The early adopters can provide specific requirements for, and feedback on, early versions of the solution.
3. To determine when a policy should be developed and an edict issued.

#### Who to work with, and in which order

Always target the early adopters first. The early adopters are the people with willingness and need to try the new idea. If you pick people in the beginning that do not have a willingness or need, this is about all you will prove. For example, one company decided to do a process assessment of two sister organizations. One sister loved it; the other hated it and learned nothing. In hindsight the second sister should have been ignored. There was never a willingness or perceived need.

The people in group one will typically be more willing to help you improve any inadequacies in the idea or solution you are advocating. The solution you provide to the other groups will then be better tested and more robust. To increase the receptiveness of the next group, success stories from the previous one should be well publicized.

If you are the manager of a group of people, the adoption curve tells you who needs which solution, when they need it, and whom to avoid until the majority of the people have adopted the new practices.

#### Building the solution in increments

If your solution is complex you might want to consider breaking it up into components. The audience you have in mind for the complete solution will probably not need every component immediately. For example, if you are developing a planning process, you might have components for estimation, risk management, negotiation, tracking, metrics, reviews and approvals. Identify the components that the early adopters need and develop these first. Use the feedback from the early adopters to design and refine the next increment.

Each component of your solution might have a different group of early adopters. For example: Some of the managers will be the early adopters for the negotiation piece. Some of the developers will be the early adopters for the estimation piece. The adoption curve provides information to help one set priorities.

In one company, a group was developing a process for subcontract management. After studying the adoption curve it was realized that there were different audiences for each part of the process. The subcontractor selection piece had a different audience than the subcontractor tracking piece. The subcontract management process was broken into components and each component was developed and refined using a different set of early adopters. Some components, such as the subcontract auditing, were not needed by anyone until much later.

#### Determining when to develop a policy and issue an edict

Many process improvement teams develop policy statements at the start of their process development activity. A policy states under what circumstances a practice will be used. For example, a policy on risk management might say that the risk management process should be used on projects that cost greater than \$50K. At the beginning of developing a risk management process it might be very difficult to predict all the conditions where the practice would be most beneficial. It is much easier to develop a policy when some experience has been gained using the solution. One option would be to delay developing the policy statement until groups one and two have collected experience using the process.

Issuing an edict for a solution might be premature when only the early adopters have tried it. Since edicts cause resistance, delaying the edict until later in the adoption curve can reduce the amount of resistance. A good time to issue an edict is when approximately 50% of the organization are using the solution effectively. At the 50% point

the success stories can be used to influence the remaining people in groups three and four. It will be difficult, but not impossible, for them to resist in the face of such evidence. At 50% there is no majority opposing the idea.

You might decide to use an edict earlier if the consequences of not using the solution could lead to a severe business impact. You could even stagger the deployment of an edict. For example, one organization decided that all system interface descriptions should go through formal peer review, while the decision to mandate peer reviews on other aspects of the system was delayed until more peer review experience had been gained.

### **Whom to avoid.**

Some people believe that to implement change it is better to attack the most difficult groups first (the laggards and heavy skeptics) because they will need the most time to be convinced. There is also a common opinion that if you can 'crack' these groups the remainder will follow.

It is usually a mistake to start with the most difficult people. Ideas that are new to the organization take time to become credible. Successes from groups one and two will have a better chance of convincing a skeptic than all the preaching and memos in the world. If you were to work with the most difficult group first, you would not only encounter resistance, you would increase the likelihood of failure stories spreading across the organization. Bad news travels fast.

As for the laggards, ignore them. If you have individuals or teams that think your solution is the dumbest thing ever invented, ignore them (other than understanding their resistance). If you have a good solution you should be able to find early adopters. By the time you deploy the solution to groups one, two and three, some of the laggards will have changed positions or left the company. If they are still in the organization, either the evidence so far will have converted them, or they will comply, albeit superficially, under the edict. You may need to leave the conversion of laggards to management. An alternative career path for laggards might be necessary.

### **I don't have time for adoption curves!**

You may be feeling that using the adoption curve is too slow for your organization, which has set a goal to have 120 practices adopted in the next six months to be able to achieve the next process standard or maturity level. It is important to note that many organizations that rush adoption usually do not actually adopt the practices long-term. Someone might think that the practices are being used, "because the deadline says so." If you ask the developers and managers, you will find that some of the practices are being used, some superficially, but many not at all. One cannot master and internalize a new technique in a day, but one can pretend!

Not using the adoption curve can slow down the deployment of new ideas. The more resistance you encounter, the longer deployment will take. It is much quicker to work with people who do not resist.

Increasing the number of knowledgeable staff providing the expertise (often called process champions or a Software Engineering Process Group) can accelerate adoption. Some organizations use training companies so that more people learn the skills in parallel. Even so, the adoption and practice of the new skill takes time. A proportion of the audience will always believe that the new idea is a total waste of time.

### **Where to go from here**

In summary, we suggest you study your organization and use the adoption curve to facilitate your improvement program. Start with the early adopters and don't worry about the laggards for now.

### **Bibliography**

[Rogers, 83]

Rogers, Everett M., *Diffusion of Innovation*, Free Press, New York, NY, 1983.

[Peters, 85]

Peters, Tom., Austin, Nancy., *A Passion for Excellence*, pp357, Warner Books, 1985.

Company examples: The stories we tell during our presentation.

—Business machines company

- » CA: Planning deployment
- » CA: SEI Level 3 deployment
- » NY: Planning deployment
- Financial institution #1
  - » Lifecycle deployment
  - » Planning deployment
- Financial institution #2
  - » SEI Level 3 process deployment
- Government contractor
  - » Inspection deployment
  - » SEI Level 4 deployment
  - » Object oriented technology deployment
- Aircraft manufacturer
  - » Assessment deployment
  - » SEI Level 2 deployment
- Computer Aided Design company
  - » Inspection deployment
- Silicon chip manufacturer
  - » Inspection deployment

=====

=====

# Process Improvement as a Capital Investment: *Risks and Deferred Paybacks*

Warren Harrison<sup>\*</sup>  
David Raffo<sup>†</sup>  
John Settle<sup>‡</sup>  
Portland State University

## Abstract

Firms invest in process improvements in order to benefit from decreased costs and/or increased productivity sometime in the future. However, there are a large number of alternate improvements available, each of which may yield different levels of cost savings. To make things even more complicated, different alternatives may result in different savings over different periods of time with different levels of risk. Thus, we're faced with the question: "*in which opportunity should we invest?*" We present a well-accepted method of budgeting for capital expenditures from the financial community, and apply it to software process

*Warren Harrison* is a Professor of Computer Science at Portland State University. His Ph.D. is from Oregon State University. Warren's interests include software metrics and financial models of software process improvements.

*David Raffo* is an Assistant Professor of Business Administration at Portland State University. His Ph.D. is from Carnegie-Mellon University. David is interested in process simulation and process improvement.

*John Settle* is an Associate Professor of Business Administration at Portland State University. His Ph.D. is from the University of Washington. John's interests include capital budgeting issues in the software industry.

---

<sup>\*</sup> Department of Computer Science, Portland State University, Portland, OR 97207-0751

<sup>†</sup> School of Business, Portland State University, Portland, OR 97207-0751

<sup>‡</sup> School of Business, Portland State University, Portland, OR 97207-0751

# Process Improvement as a Capital Investment: *Risks and Deferred Paybacks*

Warren Harrison  
David Raffo  
John Settle  
Portland State University

## Abstract

Firms invest in process improvements in order to benefit from decreased costs and/or increased productivity sometime in the future. However, there are a large number of alternate improvements available, each of which may yield different levels of cost savings. To make things even more complicated, different alternatives may result in different savings over different periods of time with different levels of risk. Thus, we're faced with the question: "*in which opportunity should we invest?*" We present a well-accepted method of budgeting for capital expenditures from the financial community, and apply it to software process improvements.

## Introduction

The motivation for process improvements is typically reduced cost and/or increased productivity. For instance, early prevention of defects reduces the cost of rework later in the lifecycle and the practice of reuse improves productivity [Basili,1994; Dion,1993; Lipke,1992; McGarry,1993; Wohlwend,1993]. Of course, both the costs and returns of any particular process improvement can vary greatly. The costs of software process improvement have been reported to range between \$490 and \$8,862 per engineer, per year, with productivity gains ranging from 9% to 67% [Herbsleb,1994; Jones,1996]. The assumption is that the returns will outweigh the costs of implementing the process improvement.

Many different process improvements have been proposed. However, many organizations can only afford (or only choose) to implement one of many options. Therefore, it is important to be able to evaluate and compare different alternatives. The analysis and comparison of projects is known as *Capital Budgeting*. Most contemporary Capital Budgeting techniques utilize the concept of the "Time Value of Money". This has been addressed in various aspects of software engineering in the past, such as quality assurance [Slaughter 98] and software maintenance [Vienneau 95]. Perhaps the most common technique is referred to as *Net Present Value* (NPV). Briefly, the idea of Net Present Value analysis involves "discounting" a future cash flow at some discount rate, then

subtracting out the current cash outlay, resulting in the expected value of the investment in today's dollars.

This overall approach is well-suited to software process improvements because the costs are usually incurred "up front" in return for cost savings or increased productivity in the future.

To illustrate, assume a *hypothetical* investment of \$50,000 to introduce reuse into an organization. In return, we (*hypothetically*) expect to receive \$100,000 in benefits due to increased productivity two years later when the reusable components are actually utilized. Thus, applying a 10% discount rate to the benefits expected to accrue two years in the future, we obtain the following net present value (NPV):

$$\begin{aligned} \text{NPV} &= \text{PV}_{2,10\%}(100,000) - 50,000 \\ \$32,645 &= 100,000 / (1 + 0.10)^2 - 50,000 \end{aligned}$$

This represents a monetary value of introducing reuse given \$100,000 in increased productivity two years in the future (this is somewhat unrealistic since the yield from reuse would ostensibly accrue over many years, even though it may not start producing any benefits until two years after the initiative is funded).

It is clear we are adjusting the value of the effort by the amount of time necessary to start seeing a return from our investment. This is useful, if we wish to compare the reuse initiative with another opportunity with a different pattern of returns. For instance, instead of investing \$50,000 in reuse, we might be able to spend \$50,000 to acquire a CASE tool that yields \$100,000 in increased productivity after only a year:

$$\begin{aligned} \text{NPV} &= \text{PV}_{1,10\%}(100,000) - 50,000 \\ \$40,909 & \end{aligned}$$

Thus, the net present of the CASE tool is \$40,909 as compared to a net present value of \$32,645 for the reuse initiative. Obviously, given these hypothetical figures, the CASE tool would be the better investment of the two. However, since the discount rate should reflect the cost of raising new funds, and both NPVs are positive, the real message is that both efforts should be undertaken.

Addressing the issue of future returns is only a part of discounting returns. In most treatments of NPV, the discount rate is taken to reflect just the time value of money. However, cost of capital reflected in the discount rate also typically incorporates the impact of "risk" on the value of the investment. For instance, in the earlier example, if the CASE tool investment only had a 50% chance of yielding \$100,000 in increased productivity, but the reuse option was a sure thing, the analysis might very well be different.

Discount rates are (or should be) comprised of two parts: a base, “risk-free” rate which reflects the value of money to be received in the future with certainty, and a “risk premium” reflecting the uncertainty of the future pay-off. Obviously, the more variable the return, the more risky the investment, the higher the risk premium and thus, the greater the discount rate.

In the case of software process improvement, the risk derives from the fact that the benefits of process improvement are not a “sure thing”. The improvements may not be well-accepted by the developers, or if accepted, they may not be implemented correctly. Even if well-accepted and implemented appropriately, the project may not respond to a given process improvement due to a lack of opportunity. For instance, there may be few latent requirement errors to find, so a requirements review process itself may yield little improvement over not reviewing the requirements at all.

Risk means not only the risk of not having a positive return, but also the uncertainty in terms of *how much* of a return we will get. Herbsleb [Herbsleb,1994] observed productivity gains from software process improvement ranging from 9% to 67% with a median gain of 35%. Clearly, this is riskier than some other investment that yields a future return with certainty. Risk is addressed within Net Present Value Analysis, by using a discount rate  $k$ , which is related to the uncertainty of the project:

$$k = r_f + \phi$$

that is, the risk-free rate of return  $r_f$  (say the rate of return on Treasury Bills, currently about 5%), plus a risk premium  $\phi$ , which is a function of the *variability of the return*.

Naturally, an important issue is how one measures the “variability of the return”. In order to do this, we have to recognize that it is possible for different scenarios to occur in the future which impact the return of the process improvement. Also, we assume that we can, with some degree of accuracy, predict the likelihood of these scenarios occurring.

For instance, let’s revisit the hypothetical \$50,000 reuse initiative. We have (somehow) determined that each instance of reuse is worth \$5,000 in increased productivity. Further assume that based on data from other organizations, we believe that there is (*note these figures are completely hypothetical, and are not meant to represent any particular organization’s experience with reuse*):

- a 5% chance that none of the components will be reused,
- a 10% chance that 10 of the components will be reused,
- a 70% chance that 20 of the components will be reused,
- a 10% chance that 30 of the components will be reused and
- a 5% chance that 40 of the components will be reused.

This is summarized as follows:

Components Reused	Probability	Net Value (\$)
0	5%	(50,000)
10	10%	0
20	70%	50,000
30	10%	100,000
40	5%	150,000

Table 1

The “expected net value” is computed as the sum of each potential net value multiplied by the probability of its occurrence. Thus, the expected net value for this scenario is \$50,000, with a standard deviation of \$38,924. The coefficient of variation (standard deviation divided by the expected value), 0.77, can be viewed as a measure of the volatility of the return. This information can be considered a relative measure of the risk,  $Risk_{reuse}$  that would yield a risk-based discount rate  $k_{reuse}$ .

On the other hand, let’s consider spending the \$50,000 to purchase and introduce the CASE tool. Assuming that the return from the tool is a function of its adoption by the developers, we might hypothesize the following cases, returns, and probabilities of their occurrence:

Utilization	Probability	Net Value (\$)
None at All	10%	(50,000)
Modest	15%	0
Moderate	50%	50,000
Heavy	15%	100,000
Exclusive	10%	150,000

Table 2

The “expected net value” is still \$50,000, however the uncertainty of the outcome has been increased, with the new standard deviation being \$52,704, for a coefficient of variation of 1.05, which implies this is a much riskier project.

Thus:

$$Risk_{reuse} = 0.77$$

$$Risk_{case} = 1.05$$

and

$$\lambda_{Risk} = Risk_{reuse} / Risk_{case} = 73\%$$



The parameter  $\lambda_{Risk}$  implies that the reuse initiative is approximately 73% as risky as the CASE tool purchase. Thus, the risk premium component of the reuse initiative's discount rate should be 73% of the risk premium component of the CASE tool purchase's discount rate.

Computing the NPV of the two proposed process improvements will then give us some direction in selecting between the two. We can assume a two year payoff for the reuse initiative and a one year payoff for the CASE Tool purchase. We can also assume a risk-free rate of 5%, and a 20% base risk premium for the CASE Tool purchase. Because the reuse initiative is 73% as risky as the CASE Tool purchase, we'd assign a 14% risk premium for it. With these assumptions, we arrive the following NPVs:

$$\begin{aligned} NPV_{reuse} &= 100,000 / [1 + 0.05 + 0.14]^2 - 50,000 \\ &= \$20,922 \end{aligned}$$

and

$$\begin{aligned} NPV_{case} &= 100,000 / [1 + 0.05 + 0.20]^1 - 50,000 \\ &= \$30,000 \end{aligned}$$

As can be seen, even being a riskier project, the CASE Tool purchase appears to be the preferred investment, since its returns appears a full year earlier than the reuse initiative. However, if we were able to speedup the return from the reuse initiative to a single year, the outcome would be different:

$$\begin{aligned} NPV_{reuse} &= 100,000 / [1 + 0.05 + 0.14]^1 - 50,000 \\ &= \$34,034 \end{aligned}$$

and the reuse initiative would instead be the preferred improvement.

Obviously, a serious issue is the base "risk premium" that we glibly "assumed" was 20% for the reference project. If we are simply interested in ranking options with similar return patterns, the specific base risk premium is of less importance. However, if the timing of the returns vary, or if we're trying to establish if the process improvement actually sustains the cost of capital to the organization, this must be established with more care.

## Necessary Capabilities

In order to use a contemporary Capital Budgeting approach to compare process improvement opportunities, five capabilities must be present:

1. the ability to predict the cost of implementing the process improvement,
2. the ability to predict the expected return of the process improvement,

3. the ability to predict the timing of the expected returns,
4. the ability to quantitatively assess the “risk” involved in the process improvement returns,
5. the ability to establish a reference risk premium (or, equivalently, a reference cost of capital) for the firm's "typical" or reference project.

While more exhaustive treatments of each of these are beyond the scope of this paper, we will briefly address these capabilities here.

### Predicting Cost

Predicting the costs of a process improvement is not rocket science. However, it can be difficult to avoid missing some costs. To illustrate the method, we will consider inspections. O'Neil [O'Neill,1989; O'Neill,1995; O'Neill,1996] points out *the rollout and operating costs associated with software inspections include the training of practitioners and managers, the ongoing preparation and conduct of inspection sessions, the ongoing management and the collection of measurement data...* O'Neil goes on to observe that training requires approximately 12 hours per engineer, plus 4 hours for each manager. Preparation and conduct of a session entails 1-2 hours of preparation and 1-2 hours in session for each of the 5 people involved in an inspection of 250-500 lines of new code. He also notes that development of the inspections database will require about two months of effort (320 hours).

Using this data, we can see that the cost of introducing inspections on a 100,000 line project with 10 engineers and 2 managers will entail:

Activity	Total(hours)
Training, engineers (12×10)	120
Training, managers (4×2)	8
Preparation for 5 engineers @ 200 inspections (100,000/500 LOC)	2,000
In-session time for 5 engineers @ 200 inspections (100,000/500 LOC)	2,000
Database development	320
<b>Total</b>	<b>4,448</b>

Table 3

### Predicting Returns and Timing

Predicting the returns and their timing can be more difficult than predicting the costs. Nevertheless, there is no dearth of attempts at assessing the benefits of various software process improvements. For instance, in [McGibbon, 1996], the benefits of inspections (as well as the benefits of a variety of other process improvements) are modeled as a function of rework costs RC:

$$RC = R \cdot \sum d_i t_i$$

where  $d_i$  is the number of defects detected in phase  $i$ ,  $t_i$  is the amount of time in hours to detect and fix an error in phase  $i$ . And  $R$  is the average hourly rate to find and fix an error.

The key is either good historical data within your organization, or access to “industry standards” (which won’t quite fit). For instance, O’Neill observes that inspections will detect 10-20 errors per thousand lines of code. Thus, with our 100,000 line project, we can expect 1,000-2,000 errors to be found. What would the cost of these errors be if they were either (a) found later or (b) not found until after the product was released. This is particularly frustrating because little industry data exists as to the actual costs of correcting defects later in the lifecycle, and organizations seldom tend to keep track of such data.

It is also not always clear what the timing of the returns would be – for instance, should the benefit of finding an error early through inspections accrue when the error is found? When the product is undergoing testing (which is when the error might otherwise be found and corrected)? When the product is released? As our earlier examples have shown, deferring the returns can have a big impact on the perceived value of the process improvement when making investment decisions.

### **Assessing Risk**

Anyone who has ever spent time looking at past process data understands that real outcomes seldom fit the nice, mathematical prediction models developed by researchers. However, these models often provide a good basis for an expected outcome, with potential outcomes being clustered around that point. A good source of data on past experiences can go a long way towards understanding what the clustering looks like. Is it a peaked, low-risk distribution like the reuse initiative data showed earlier, or a flat, high-risk distribution like the CASE Tool purchase?

Lacking quantitative, organization-specific historical data, other sources of information can be tapped to assess risk. For instance, consultants, small, ad hoc experiments, expert judgement, etc.

### **The Reference Risk Premium**

It is presumed in this treatment that the firm is already using capital budgeting techniques, and has a basic understanding of what its reference cost of capital is. Determination of a firm cost of capital, although a critical step in deciding on investments and deserving of attention in software development contexts, is beyond the scope of this paper.

## The Role of Metrics Repositories in Establishing Costs, Returns and Risk

From this brief discussion, it is clear that the capabilities needed for capital budgeting of software process improvements are highly dependent on data. It is interesting to note that the decreased risk due to good historical data from a metrics repository can actually be used to quantify the value of a metrics initiative within an organization.

## Summary

In this paper, we have briefly described the application of traditional, time and risk based capital budgeting techniques to software process improvements. As we saw, these techniques can be valuable in choosing among alternatives. The major constraint to applying these techniques is access to the four **five?** capabilities we discussed. These are all highly dependent on historical data describing past experiences within the organization.

## References

- [Curtis,1995] Curtis, W., "Building a Cost-Benefit Case for Software Process Improvement," Notes from Tutorial given at the Seventh Software Engineering Process Group Conference, Boston, MA, May 1995.
- [Dion,1993] Dion, R., "Process Improvement and the Corporate Balance Sheet," *IEEE Software*, July 1993, pp. 28-35.
- [McGibbon, 1996] Thomas McGibbon, "A Business Case for Software Process Improvement", Data Analysis Center for Software State-of-the-Art Report, prepared for Rome Laboratory, September 30, 1996  
(WWW URL: <http://www.dacs.dtic.mil/techs/roi.soar/soar.html>).
- [Hayes,1995] Hayes, W., Zubrow, D., "Moving On Up: Data and Experience Doing CMM-Based Software Process Improvement," Presentation at the *Seventh Software Engineering Process Group Conference*, Boston, MA, May 23, 1995.
- [Herbsleb,1994] Herbsleb, J., Zubrow, D., Siegel, J., Rozum, J., Carleton, A., "Software Process Improvement:State of the Payoff," *American Programmer*, Vol. 7 no. 9, September 1994, pp. 2-12.
- [Jones,1996] Jones, C., "The Pragmatics of Software Process Improvements," *Software Process Newsletter* of the *Software Engineering Technical Council Newsletter*, No. 5, Winter 1996, pp. 1-4.

[Lipke,1992] Lipke, W., Butler, K., "Software Process Improvement: A Success Story," *CrossTalk*, Number 38, November 1992, pp. 29-31, 39.

[O'Neill,1989] O'Neill, Don. Software Inspections Course and Lab. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1989.

[O'Neill,1995] O'Neill, Don. "National Software Quality Experiment: Results 1992-1995." Proceedings of the Seventh Annual Software Technology Conference. Salt Lake City, UT, April 9-14, 1995. Hill Air Force Base, UT: Software Technology Support Center, 1995.

[O'Neill,1996] O'Neill, Don. "National Software Quality Experiment: Results 1992-1996." Proceedings of the Eighth Annual Software Technology Conference. Salt Lake City, UT, April 21-26, 1996. Hill Air Force Base, UT: Software Technology Support Center, 1996.

[Slaughter,1998]. S. Slaughter, D. Harter and M. Krishnan, "Evaluating the Cost of Software Quality", *Communications of the ACM*, August 1998, pp 67-73.

[Vienneau,1995]. R. Vienneau, "The Present Value of Software Maintenance", *Journal of Parametrics*, April 1995, pp. 18-36

# **Matching ISO 9000 Registration to Your Organization**

**Mark Johnson  
OrCAD  
13221 S.W. 68th Parkway  
Portland, OR 97223  
(503)968-4801  
mark.johnson@orcad.com**

## **ABSTRACT**

An ISO 9000 registration program is an example of a major process change effort for most software organizations. Before you undertake such an effort you need to understand what goals your executives really want to accomplish with the program. And you need to determine your organization's real operating principles. Then you will need to be creative in matching your process implementations to meet both those goals and operating principles. Your program will need to be structured to give you early feedback on successes and problems so that you can make course corrections or even change your fundamental approach, if needed.

This paper uses my experiences with ISO 9000 registration to explain the lessons I learned about effectively taking on a major process change project and adapting to match your organization. Although this paper is focused at ISO 9000 registration, I believe these lessons apply to any major process change effort you are asked to lead.

## **AUTHOR BIO**

Mark currently enjoys being part of a product development team at OrCAD. His primary interest is in helping teams be more productive and effective in their work. Mark is amazed to realize that he has over 25 years experience, including being a leader in ISO 9001 registration, process improvement using the CMM, and building teams. He has been active in the software quality community volunteering with conferences, presenting papers, and publishing articles.

This paper was first published in Software Testing & Quality Engineering magazine, volume 1, issue 4.

Copyright 1999, Mark Johnson

## Matching ISO 9000 Registration to Your Organization

Putting together an ISO 9000 registration program for a software development company is a major project. If you have found yourself tasked with coordinating ISO 9000 registration, some of the insights I have come to from my experiences might be of benefit for you. I've been through ISO 9000 registration efforts twice – while still at the same company. The first time, we structured the program in a classic manner as recommended by many consultants and 'how to' books. After expending a lot of energy but not making much progress over the course of a year, a management shake-up occurred and we tried again using a different approach. The second time we did achieve ISO 9000 registration, and I even felt like we had helped make the software development organization more productive.

The lesson that I learned from this was to first develop a true idea of what your executive manager(s) want to accomplish by getting ISO 9000 registration. Then, you need to have a keen understanding of how your organization really works, including where the decision making power lies and how things really get done. Finally, you need to be flexible in the approach you take, examine your progress, and be willing to make changes when you realize you are having problems.

### Understanding Management

Generally, you are not the person who decided that your organization should get ISO 9000 registration. You need to figure out what the person who *did* make that decision is actually expecting—and perhaps more importantly, is willing to support. ISO 9000 registration is the type of process change program that will not succeed without explicit executive management support.

When you ask what they are expecting, you will need to filter out the “executives are supposed to be upbeat and show support” portion of any answers. There is a rule of thumb that says you need to ask “why?” up to five times to get to the real motivation for an issue. Of course, you will need to do this in a manner that is not too obvious or irritating. There are two specific areas to explore during this conversation.

You need to explore what **external** factors might have led to this decision. Management may be vague at first—“We need to get ISO 9000 registration right away; it's important to our future business.” It may take a little probing to find out why Management is concerned about ISO 9000 all of a sudden: it turns out that your primary competitors are getting registered. The motivation may be even more immediate: Management discloses that your major customers have said they will only do business with ISO 9000-registered companies in the future.

It may have taken some effort to pry that information out, but now that you have, you're better informed about how to proceed with registration efforts. If your executive's primary focus seems to be satisfying external demands, the ISO 9000 registration effort is really about getting a 'check-off.' They are probably not proposing a major process improvement effort, but they'll want to go on record as doing ISO 9000 to improve quality.

You also need to explore **internal** factors that might motivate the interest in ISO 9000 registration.

If you pursue the above conversation with your executive one step further, you may find that there are other reasons why Management wants to pursue ISO 9000 registration. Management may tell you that they don't like the organization's unpredictable results. "Product X was six months late to market," they point out, "and we lost some major sales to our competitors." Management sees ISO 9000 registration as a known quality improvement program that would help them avoid those unpleasant surprises—especially if your organization's processes are seen as out of control or inconsistent. The appeal of ISO 9000 in this case is that it forces you to document your processes and audit to make sure they are being used.

Since ISO registration can point you in several directions, you may ask clarifying questions: Does Management think it is important that all your different product units in the organization follow the same set of processes, or would it be sufficient if each unit had its processes documented and made sure they were following them? You may find that Management's goal is to find the best practices in the organization and get everyone to use them—essentially using ISO 9000 as a tool to standardize processes across the organization. It's important to be clear about this goal, because getting organizational units to change the processes they use can be a major cultural change program in itself.

You will probably find that Management has a mix of reasons for deciding ISO 9000 registration is a good thing. You need to sort these out and decide which will be the guiding factors for your ISO 9000 program. In the example above, key customers have stated that all their suppliers must be ISO 9000 registered. At the same time, your executive is frustrated with the inconsistent results of product development. Although getting ISO 9000 registration will certainly satisfy your customer's demand, ISO 9000 and the standardization of processes may not improve the inconsistent product development results. You might be better off to do a minimal ISO 9000 implementation and then focus on understanding why product development is a problem. Of course, you will have to figure out how to sell that approach to your executive.

You also need to know where ISO 9000 registration and process improvement fit in the priorities of your executive and the organization. The organization and its executives will typically have prioritized lists of objectives. If ISO 9000 registration is one of many programs on the list of objectives, there will be limited energy and focus for it. And, if ISO 9000 is well down a prioritized list, this may indicate registration is a "nice to do" rather than a "must do." Another way to check for how much support ISO 9000 registration has is to listen to the standard public statements your management make within the organization. They usually try to provide a consistent message about what is important to them and to the organization. When you hear these statements, see where ISO 9000 is mentioned—or not mentioned, and use that to help you gauge the actual level of support.

As in the example above, using ISO 9000 as a process improvement vehicle frequently means driving process changes into the organization. What is your executive's attention span and willingness to help drive change in the organization? And what is the typical longevity of an



executive in your organization? Making fundamental changes in the way the organization works is a major long-term effort. Will they be there to support you along the way and keep a focus on this effort over the years it will take to really ingrain new processes?

My personal war story is that our first try at ISO 9000 registration was focused at “process improvement through standardization.” This was clearly the intent of our executive. As head of all product development activities he clearly had the position to command this and viewed it as one of his objectives in changing how work was done. This was a major process change for our organization and we spent a lot of time wrestling with resistance. When our executive left before the year was out, what progress we had made fell apart—and it was the end of our first attempt to reach ISO 9000 registration.

## **Understanding the Organization**

We had known subconsciously that the organization we were in was decentralized and valued autonomy, but we believed somehow that we could carry out our executive’s wish to overhaul the way the organization functioned. In analyzing what led to our first failure, we came to appreciate how the organization really worked, and how we could structure our ISO 9000 program to work with it. We inverted our approach from one of standardization and switched to working with each unit of the organization to help them identify the processes they *already* used, and how these mapped into ISO 9000’s requirements.

Understanding how your organization really works is at least as important as understanding what your executive would like to see accomplished. Goals and objectives set by your executive may change, but how the organization behaves will remain fairly constant. If you have been working for several years in an organization, you probably have collected enough information to be able to figure out how things really work and where the power lies. You may need to take some time to reflect on how you have actually seen processes function and evolve. For example, does the organization function top-down? Are management pronouncements closely followed without resistance? At the other extreme, are management directives ignored, or given lip service while people wait for them to die? Chances are your organization is somewhere in between. And in a large organization there will be variations from department to department.

If you are new to the organization you need to find a way to gather this understanding of how things work. One way is to select several people who are opinion leaders in the organization and ask them for their insights. If you believe they are comfortable with each other, getting them together as a focus group may be effective. They may feel more secure and willing to open up in a group of peers, rather than one-on-one with you. You need them to be honest and tell you what they and the organization really do. And in groups, one person’s comments often trigger memories or insights in another person.

What should you ask these people? Find out if there have been previous change or quality initiatives in the organization. Ask the focus group for their evaluation of how these initiatives were carried out, and how effective they believe the result has been. You can also look at the results of any project postmortems for hints. Are there standard processes that don’t sound like

they are really being followed? Ask the focus group about what you discovered. You can also ask them to recount war stories about things that they have seen go wrong in the organization.

## Program Alternatives

Now that you understand what your executive wants and how the organization works, you need to structure your ISO 9000 program. I've observed three different approaches:

- 1) Get registration with the minimum cost and disruption,
- 2) Use ISO 9000 to drive changes and standardization of processes throughout the organization, or
- 3) Help each unit of the organization understand and improve their processes.

If you have concluded that what Management really wants is to get an ISO 9000 registration certificate in order to satisfy a check-off item, then your best bet is to aim for a program with **minimum cost and impact**. In this case, the best approach might be to see if the scope of the ISO 9000 registration could exclude your *software development operation* and focus on the manufacturing and distribution parts of the company. If you can't exclude software development, you could buy a "how to" book that includes a CD-ROM with twenty process templates. You can modify these process documents to fit your organization and adopt them as a veneer on top of how you really do work. You then train people on what the processes are and how they should handle questions during ISO 9000 registrar audits.

If you believe that your executive wants to use ISO 9000 to drive process changes and standardization, and you believe that the organization will go along with such efforts, you can structure a traditional registration effort. You get a team of people together representing a cross-section of the organization, and they document the processes the organization is to use. Then you train the organization on these processes and put an auditing program in place to make sure people are following the processes and that the processes are effective. I will warn you that this is much easier said than done. From my personal experience I can tell you that even those people who support your efforts will have a hard time agreeing on common processes and getting around to using them *all* the time. So you will need to allow enough time and resources to do more than one cycle of developing and implementing a process. Start with general, high level process descriptions, and then add to them only as you find necessary to make the process more effective.

If you realize that your organization is made up of autonomous units, and if Management is willing to support the cost and time to help these units improve their processes, then approach ISO 9000 work on a unit by unit basis. What we did was to put a superstructure of general processes in place at the organization level. This gave us something to point the registrar at as a default, with the units having broad latitude to interpret these processes or create their own. We then worked with each unit to help them discover how the processes they already used met most of ISO 9000's requirements. Any process improvements were focused on what that unit considered important to their work.

Practical reality is that you will actually want to mix and match these alternatives as you approach ISO 9000's 20 clauses. For example, you might feel that it would add value to focus process improvement on ISO 9001 Clause 4.14, Corrective and Preventive Action, by putting processes in place that A) emphasize root cause analysis and B) ensure that the results are fed into process improvement.

On the other hand, if no one really feels Clause 4.11, Control of Inspection, Measuring, and Test Equipment, will help your organization, you might do a minimum disruption implementation such as documenting the simple process you currently have for validating and rolling out new versions of the software test tools.

## **Cross Purposes**

What if you are facing a mismatch between what your executive wants to do with ISO 9000 registration and what you believe the organization will support? If the organization wants to do more process improvement with ISO 9000 than your executive does, you can ask your executive to expand the ISO 9000 program, and you can work directly with the organization to expand process improvement over time.

On the other hand, if your executive wants to drive process changes and standardization with ISO 9000, but you don't think the organization will support standardization, you are facing a tough situation. Once you are convinced this is the case, you are going to have to go back to your executive, present your evidence and conclusions, and explore alternatives. If getting ISO 9000 registration is still needed for external reasons, you can propose doing the minimal approach, and then addressing process change as a separate program. If your executive is still determined to use ISO 9000 for process standardization, you are really facing a major organizational change program, rather than ISO 9000. You will need to ensure your executive's support for a long-term, high effort organizational transformation, and you should seek expert help on organizational change. Or by understanding why Management wants to standardize processes, you might be able to convert them to the idea of allowing autonomy for organizational units within a larger structure of ISO 9000 registration.

## **Problems**

Life isn't perfect...and you will have problems. A key problem is that you may have chosen the wrong program alternative—as you go along, you need to be honest with yourself in evaluating your progress, and be willing to make changes to your program.

To help you in evaluating whether you are on track, you should structure the work so that there are a series of short cycles containing all the steps in implementing ISO 9000. These steps are:

- 1) define and document,
- 2) train and implement, and
- 3) audit use and records.

You will need to be creative when picking the right parts of the software development process to use in these short cycles. And avoid striving for perfection or too much detail when trying to make the first attempt at defining or documenting a process.

When a cycle is completed, you can do a review and see if there were hang-ups. Look for which parts of the cycle were hardest to complete, were late, or didn't meet original expectations. Then analyze why these parts of the cycle had difficulties. See if the results say anything about your original evaluation of management's objectives and support, the organization's operating principles, or your approach to ISO 9000 registration. Then you can adjust your approach to registration as needed. You could find, as we did, that process improvement through standardization wouldn't work in your organization. Or you could find yourself on the flip side of this—you've decided that all that was wanted was a minimal impact program. In reality, however, Management will support a major effort, and the organization is ripe for change.

There is an old joke that "a camel is a horse that was designed by a committee." If you are working with a group of people to document processes for the organization to use, be aware of their motivations. Frequently the people who volunteer to help you will be those who believe that the existing processes could be improved by making some changes and adding more detail. Others in the organization may or may not believe their processes need these improvements. But if your process documentation grows into a four-inch binder full of things that people have never done before, you are in trouble. Be glad to accept help from volunteers, but make sure they are the people who will actually be using the processes they are documenting. (And be like the editor of a magazine. In writing this article, I first put down everything I thought I should say; then the editors helped me focus the content and get down to what really mattered.)

### **The Bottom Line**

Of course, all this talk of reading Management and the organization sounds pretty political—and in trying to do this, you don't want to be cynical about your work or your organization. What I am really trying to say is that if your bottom line is to get something done, **work within your organizational context**. I think of this like the martial art Judo. When you are faced with a large and strong opponent, you don't try to overpower them. Instead, with your observations and agility, you use their strength to meet your *own* objectives.

### **Reference:**

An example of asking "Why?" 5 times can be found in The 5<sup>th</sup> Discipline Fieldbook: Strategies and Tools for Building a Learning Organization, Peter M. Senge(Editor), et al. Article "The Five Whys" by Richard Ross, pages 108-112.

# Starting a Requirements Management Initiative

*Mike Young, Don Moreaux, and Kris Hartung  
Hewlett-Packard Company  
11311 Chinden Blvd., Boise, Idaho 83714-0021  
Mike\_Young@boi.hp.com  
Don\_Moreaux@boi.hp.com*

## **Abstract**

Requirements become the basis for all of a project's subsequent management, engineering and assurance activities. Consequently, how effectively requirements are managed can have a dramatic influence on project resources and schedules, as well as product quality.

This paper describes one HP Division's experiences in starting an initiative that significantly changed the way in which its managers, engineers and marketing specialists understand and implement *product requirements management*. The experiences detail how, by maturing our requirements management practices, our organization was able to address the following issues: 1) information redundancy and synchronization, 2) inconsistent documentation format and content, 3) inefficient requirements elicitation, communication and review procedures, and 4) lack of customer focus. Recommendations are offered for those organizations that plan to start a similar initiative. Finally, the paper discusses plans to improve our requirements practices on future projects.

## **Introduction**

Hewlett-Packard's Personal LaserJet Division produces a family of software, printer, and multi-functional products that are designed for home office, small office, and personal use environments. In the past, we've had small teams and simple products and have been able to produce highly reliable solutions with a simplistic and uncontrolled documentation and requirements management process. We recently changed our product development strategy, focusing on component development and re-organizing into a matrix organization to create a more common solution and lessen duplicate work. This has created parallel development of components and a fairly complex reporting/working structure with multiple communications paths.

To address this increased complexity of our organization and products, we determined it was necessary to invest significantly in creating a solid set of product requirements and some means of effectively managing them. The purpose of this paper is to share what we have learned as an organization during the evolution of our product requirements management process. We describe the difficulties that we encountered with our previous approach, and how we solved these problems with our implementation of a more efficient and sophisticated tool-based approach. Finally, we outline some additional benefits we realized that we hadn't even considered prior to the initiative and discuss our plans for further improvement in developing the next generation of products.

## The Process Prior to Starting the Initiative

We have always begun new product development by assigning a “program manager” to investigate the feasibility of a new product idea. While extensive research of customer needs and market opportunities did occur prior to the initiative, the information was typically only captured in a high-level data sheet for the product. A large jump was made from the data sheet to low-level engineering design of each subsystem--first hardware, then firmware (embedded software), and lastly software began fitting their designs into the already-defined work of the other teams. The system specification occurred piecemeal throughout the development cycle of a product. For the most part, our requirements management process consisted of a data sheet, sporadically created and updated documents, and a process of requirement elicitation that we had reduced to informal management-driven communication between marketing and our research and development lab.

The major problem with this ad hoc requirements process was that it lacked well-*defined* and *documented* requirements. The end result was that we addressed many issues too late in the process to make changes, products tended to be very technology focused, and each subsystem became an “add-on” rather than an integral piece of a fully architected system designed to meet specific customer needs. Schedules between teams were difficult to coordinate, and we integrated the complete system for the first time at the very end of the development phase. We were still able to produce high-quality products that met customer needs, but doing so required frequent heroics and long test cycles.

## The Initiative

### Defining the Problem

The only thing worse than leaving the development process alone is to try to roll out a solution that doesn't solve a known problem. When we performed defect “root-cause” analyses on three separate project teams, they all pointed to the lack of requirements as the top cause of defects. Resolving these defects required changes in product design or substantial rework of algorithms or test code. This and the overall frustration level converged to help us realize that we needed a well-defined document strategy and requirements management process.

The initiative we undertook attempted to solve the following key problems that we saw in our past product development. We felt this would greatly increase our effectiveness as an organization and our ability to deliver quality products on-time.

- 1. Information Redundancy and Synchronization.** Product documentation included information that each author owned along with information that they believed affected their share of the product design. Consequently, the same information was recorded in many documents because every author was documenting part of every other author's work. With numerous places for the same information, it became out-of-synch quickly and no one knew which was the real data.
- 2. Inconsistent Documentation Format and Content.** Because authors often combined their own innovations with data from previous projects, everyone considered themselves the sole caretakers and distributors of that area of information. Reviews often became personal because documents were *personalized*. Document reviewers did not feel responsible for making sure that the information was correct because they thought that this was the author's job.

3. **Inefficient Requirements Elicitation, Communication and Review.** No one knew which were the "official" documents for the program, where to get the latest version of each, and whether they were up to date. In fact, typically documents were reviewed early in the project then not updated or put under change control until the product release. During product development and testing activities, word-of-mouth communication was the predominant mode of exchanging accurate information. This was typically a "pull" model as well, so that if other developers or testers didn't know to ask, they didn't find out about changes until integration or test execution. By relying on verbal communication, there was no definitive way of knowing whether we were really building what we initially set out to build. With a newly-reorganized matrix organization that needed to do parallel development and work out many dependencies, this would not suffice. Many developers were conscientious about having documents reviewed up front, but they typically didn't get updated until the product release and often lived in their local directory somewhere.
4. **Lack of Customer Focus.** Because documents were always "technical descriptions," either of how the system worked or of how it was built, we did not seriously consider having a method for describing and documenting the customers' needs and how they would use the product. Products were more driven by technology than by customer needs. There was ample validation by real users through usability and beta testing, but it was always too late to make serious changes when problems were found.

## **TBI's Caliber-RM Tool: Formulating the Process**

As we began to develop a new generation of products, we started with a single requirements engineer working with multiple program managers to elicit and record what the product requirements were. This initial close work with the program managers proved crucial in getting their buy-in and understanding of what requirements management is all about.

The initial deliverable was a single document based on a customized IEEE specification for a "Product Requirements Document." We collected requirements by gathering information from numerous sources—specifications and User's Manuals from previous products, requirements that other people had written down for other LaserJet products, and discussions with individuals. The main idea was to "get something in writing" because gathering and organizing information is often the hardest part of implementing a new process (it is easy to engage others to correct what is already there, but can be hard to acquire new information from them).

We made it a policy that all product specifications would be part of this document. While this provided a single source of data and a consistent writing style, we very quickly found we could never get or keep the entire document up-to-date or the changes communicated throughout the organization. This and the desire for a central archive of data caused us to search for a tool that would help guide our activities and processes.

While we were searching for a requirements management tool, another LaserJet division had already started to manage their requirements by piloting a tool called Caliber-RM from Technology Builders, Inc. (TBI). After conducting a thorough analysis of requirements management tools earlier in the year, they chose Caliber-RM for its ease of use and its view of each requirement as a "database element" rather than a part of a document. This "database" approach promotes flexible reporting, shows traceability between requirements, and makes it easy to associate other fields with a requirement. We also liked the ease-of-use, the full change history for each requirement, the automatic e-mail notification to "responsible persons" when requirements change, and the fact that team members can provide feedback on requirements through a threaded discussion group feature accessible through a Web Access component.

The fields we used for each requirement are listed in Table 1. Having the same fields for each requirement and using pull-down choices where possible solved our "data inconsistency" problem. With the information entered into a central repository and split out by these fields, we assured that the right information was captured and highly visible and accessible to everyone. Various views of the data can be created through reporting as well--by product, by status, by priority or ranking, or filtering the data like generating a list of just the recorded open issues.

**Table 1:** Information tracked for each requirement

Field	Description/Values
Requirement Name	Short identifier used for summary reports
Owner	The individual who owns and keeps the requirement up to date
Status	The state of the requirement in the development cycle (Proposed, Approved, Scheduled, Implemented, Postponed)
Priority	Overall marketing value of the requirement (Must, High Want, Want) Note: this is influenced by the Product Ranking and Difficulty fields
Description	Free-form field; includes the user need and the general feature description
Device Requirements	Description of the implications to the embedded software and hardware
Host Software Reqts	Description of the implications to host software
Open Issues	Free-form list of questions or ideas that need to be addressed
Closed Issues	Capture of how and when each open issue was closed
Applicable Products	The products to which the requirement applies
Product A Ranking	The requirement's importance to Product A's customer
Product B Ranking	The requirement's importance to Product B's customer
Difficulty	The effort required to implement the features and satisfy the requirement (Unknown, Easy, Difficult, Very Difficult)
Settings	Default values; range of values for a requirement parameter
Management Issue	Checkbox to identify that the issues require manager action
Risk	The risk to the program schedule (Unknown, Low, Med, High)
Source	The origin of the requirement - a team, individual, or previous product

Since reports are generated automatically, the information is viewed as "belonging to the product" rather than "belonging to the individual owner." Writing requirements is much faster since requirement owners no longer have to focus on writing and formatting a long document, but on simply filling out the appropriate fields. Furthermore, by using the security aspects of Caliber-RM, our change management process has improved significantly by restricting the write access of each requirement to its owner.

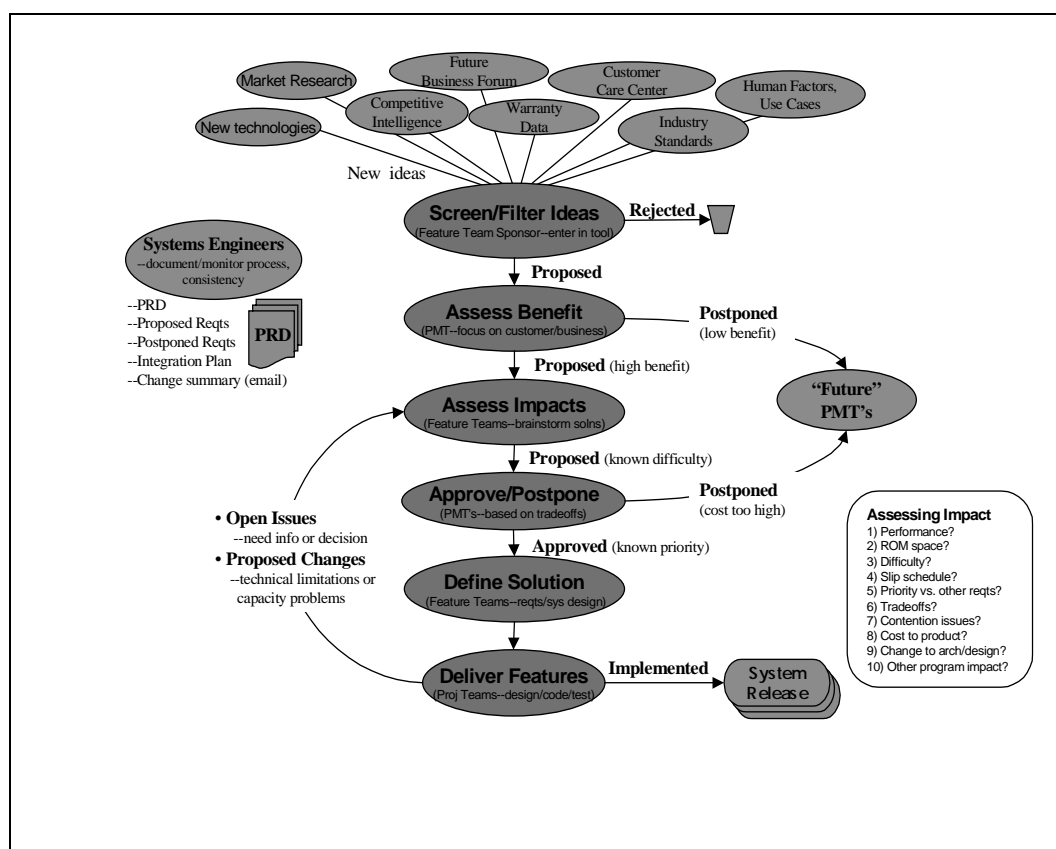
## Organizing around Communication: Rolling out the process

"Success or failure of a system depends on whether the relationships among its human components are as good as they can be--even more so than the relationships among its technical components." Human engineer Cavett Roberts says "85% of success in any field depends on knowing humans and human relationships." (7). Getting the right people to work closely together was the key to a successful rollout of our requirements management process. We started with a team of **requirements engineers** who provided oversight, support, and a consistent approach. We needed a screening/decision-making team who could assess customer benefits and determine whether to approve new requirements. Already existing **Program Management Teams (PMT's)** took on this role. Finally, we needed a team who could analyze all the incoming requirements, determining impacts and coming up with design alternatives to meet the requirements. Cross-discipline **feature teams** were formed to do this impact analysis and system design. They also allocated the requirements across each discipline. Figure 1 shows the process flow between these three teams to enable the requirements process. It focuses mainly on the state of each requirement--progressing from *proposed* to *approved* to *scheduled* to *implemented*. It also includes the sources from which most requirements came. The key to the success of this process was



making sure ownership existed at all levels. Reports were used to generate "to do lists" for each group to assure that requirements progressed as necessary.

**Figure 1: Requirements Management State Model, Roles and Responsibilities**



## Controlling Change

Currently, our product requirements are documented, controlled, and include project objectives, features, and performance characteristics. Commitments resulting from the requirements are negotiated with the affected groups and individuals. We make changes to controlled project requirements through the approved process in Figure 1.

For the formal requirements phase of the lifecycle, we always tried to err on the side of "keep it simple" so did not focus on controlling change to the requirements. In fact, we typically didn't even communicate changes to the whole organization. Requirements were being added and changed so quickly by so many that it would have been overwhelming for anyone to have to wade through all requirements changes. Since all of the data was in a robust tool, we did capture a full change history throughout. The feature teams actually managed the whole process during the first phase, including brainstorming/collecting ideas, assessing benefit, and designing possible solutions.

After we transitioned out of the requirements phase by holding a large-scale review of all of the 100+ pages of requirements, we fully employed the process in Figure 1 for two main reasons:

1. *To make sure any changes were fully communicated.*
2. *To protect the feature teams from turmoil in requirements.* Engineers tend to spend time finding solutions to any problem they're aware of, rather than assuring the customer need exists first. Putting the Program Management Teams in charge of this "screening" process allowed them to

engage the engineering community on the feature teams on an as-needed basis to understand impact and possible designs.

We're working towards a weekly email generated from the tool that shows all changes to any requirements in the past week.

## Impact

This section details the major impacts that this initiative had on our organization at the program level. Typically, our program managers coordinate the various functional areas, such as marketing, engineering, testing, manufacturing, quality, and support. Specifically, this section will focus on the positive influences the initiative had on the program's subsequent management, engineering and assurance activities. In some cases the affects were expected, in others they ended up being favorable surprises.

The most surprising affect occurred during the planned periodic review sessions of the Product Requirements Document (PRD). Membership during those review sessions was open to the entire program team. The main purpose during those reviews was to discuss new requirements for the product, as well as review and determine the priority and status (Proposed, Approved, Designed, Scheduled, Rejected, ...) of existing requirements. While there were long debates over rejecting or postponing high priority requirements, the decisions in most all cases ended up being final. This behavior was a big departure from the "it's in, it's out, it's in again" decision making we were previously accustomed to when these decisions were made without the benefit of a program level review.

We were also surprised to discover that the way in which we categorized our requirements had two very interesting side effects. Our initial plan was to organize the requirements into groups, or requirement types, based upon product attributes. These types included such product attributes as Availability, Compatibility, Disposability, Functionality, Maintainability, Performance, Security and more. By doing so, we had hoped we could more reasonably ensure requirements coverage of the product.

The first consequence of this decision was that we found it very difficult to track requirements because some categories, such as Functionality, contained large numbers of requirements while categories such as Security and Safety were hardly used. The second consequence was that this categorization made it difficult to assign ownership for requirements, a necessary element of good requirements management.

Since the engineers on this program were already organized into cross-discipline engineering teams, called "Feature Teams", it became obvious to us that changing the requirement types to match product features solved two problems. By creating requirement types from our product feature set we were able to balance the distribution of the requirements. Since each of the Feature Teams controlled the design and implementation of its unique product feature, the issue of ownership was resolved.

Incorporating our Test Design Specifications into the requirements tool along with the product requirements provided additional benefits in the area of product assurance . The tool allowed us to automatically create and manage a Traceability Matrix (Figure 2), showing the relationship between the test requirements and product requirements.

**Figure 2**  
**Traceability Matrix**

	2.3 Custom Paper Size - PSWT4092	2.4 Number of Copies - PSWT4093	2.5 Paper Orientation - PSW	2.6 Pages per Sheet (N-up) - P
16 PostScript Level	↑			
17 Viewable Self-Test				
18 Self-Test Value				
19 Maintain Country				
20 Change Fuser Temperature				
21 Booklet Printing			?	
22 Manual Booklet Printing				
23 Bitmap Watermark				↑
24 Text watermarks				↑
25 Font Symbol Sets				
26 4-Corner Symbol				

Because the tool automates the ability to keep track of changes to either of these two pieces of information, test designers were immediately made aware of changes to requirements that might impact their testing. Additionally, test planning was now taking place much earlier in the development cycle than it ever had been on previous programs.

Finally, we believe that we significantly reduced our documentation overhead by almost half. This was accomplished in a number of ways. Having a requirements management tool in place, providing easy network access provided a focal point for our organization. Everyone knew how to find the information, allowing them to reference the information instead of reproducing it in other documents. Report generation is automated, no more hours spent formatting requirements documents in word processors.

## Recommendations

### Use a Tool To Help Manage Requirements

There are a wide selection of Requirements Management Tools available on the market today. While an organization can get by without one, it was our observation that the benefits justified the cost of the tool we selected. Having the ability to:

- ◆ Track and Record History of Changes,
- ◆ Limit Write Access with Security Controls,
- ◆ Automatically Generate a Traceability Matrix,
- ◆ Create Customizable Requirements Forms, and
- ◆ Easily and Quickly Generate Customizable Reports

will save substantial amounts of time in the course of your project's requirements management effort.

### Have a Dedicated Requirements Engineering Resource

Eliciting and writing well-formed requirements is not a simple set of tasks. It involves a special skill set and a full time commitment to be done well. In our opinion, requirements engineers become proficient largely through experience. Requirements engineers also help manage changes to requirements when write access to the requirements is limited to them alone.

Currently, we write our requirements in natural language. Although requirements written in natural language are inherently prone to problems with ambiguity, inaccuracy, and inconsistency, the informality of the language makes it a good candidate for specifying high-level general requirements. However, this informality makes differences in the "formal" versus "popular" words and phrases that are difficult to use in precisely describing complex, dynamic systems. Even words with strict definitions, such as the IEEE-defined *error*, *fault*, or *failure*, are often used incorrectly.

- Give special attention to each word and phrase to avoid ambiguity and imprecision.
- Use the simplest language that is appropriate to the purpose of the requirement.
- Use the imperative voice, and avoid weak phrases such as "held to a minimum."
- Avoid generalities in place of number values (for example, "largest").
- Avoid words with relative meanings (for example, "normal" or "easy").

Eventually product requirements get "decomposed" into more detailed engineering-level requirements documents such as software requirements specifications as well as test plans and procedures. Having a comprehensive documentation strategy, one that specifically identifies the hierarchy, ownership and content will greatly reduce the effort it will take to manage the suite of documents that will flow from the product requirements. An example of our test documentation strategy is given in Figure 3.

### Test Documentation Strategy

**PRD**  
Owner: PMT  
User: Feature Teams  
Contents: Description of User Needs, and Product Behaviors

**Program Test Plan**  
Owner: Test Program Manager  
User: eSW & SW Test Leads(LTL) and Test Integration Leads (R&D)  
Contents: Program Quality Goals, Product Test Contacts and Liaisons, Program Test Issues

**Test Process Handbook**  
Owner: R&D Test ES/ LTL ES  
User: Project Team, LTL  
Contents: Test Design Strategy, Roles & Responsibilities, Defect Tracking Plan

**Configuration Management Plan**  
Owner: eSW & SW Project Teams  
User: Project Team, LTL  
Contents: Configuration Plan of Record, What, When, Where and How Items will be configured

**eSW & SW Test Plan(s)**  
Owner: Shared between Test Leads (LTL) & Test Integration Leads (R&D)  
User: Test ERS Owners, Test Leads, Test Integration Leads  
Contents: High Level Description of Testing Scope, Approach, Resources, and Schedule

**Test Design Specification (a.k.a Test ERS)**  
eSW, SW, System, ...  
Owner: Shared between Test Leads (LTL) & Test Intrgr. Leads (R&D)  
There will be specific owners for each ERS element  
User: Test Designers - both internal and external to HP  
Contents: Requirement ID, Test Name(s), Test Objective, Pass/Fail Criteria, Test Issue, Test Owner, Test Parameters, Test Setup Criteria

**Test Case Spec.**  
Owner: Test Designers & Analysts  
User: Test Execution Lead  
Contents: Inputs, Outputs, Setup Requirements, Test Configurations, Intercase Dependencies

**Test Procedures**  
Owner: Test Designers & Analysts  
User: Test Execution Lead and Test Operators  
Contents: Setup and shutdown instructions, Steps to execute, Contingencies

**Logging and Reporting Documentation**

Requirements Coverage Link

May be placed in Caliber

Not to be placed in Caliber

# Conclusions

For our next project, we plan to “start at the top” of the requirements management process by focusing on “use cases” for the product. In order to get an early start on requirements management for this initiative, we gathered information from every source that we could think of (documents from predecessor products, descriptions of competitive product offerings, and insights from sister divisions producing products). Rather than focus on the detailed specification first, we should make user needs more visible, and they should be driven by what we are trying to solve for the customer (based upon how customers are currently doing their jobs and how we can streamline that process). By describing the flow of events that we need to accomplish, we can drive the actual product requirements. By driving product requirements, we mean that user needs and use cases *become* the product requirements and shape the look and feel of product features rather than features being added just because they are a “cool technology” or a product of one engineer’s personal preferences. This is helping us change from being a technology-driven organization to one that is market-focussed and uses technology only to respond quickly to feedback from customers.

We need to establish owners early in the requirements management process. The key to the success of our initiative was assigning engineers to a particular feature area as the focal points for keeping the requirements up-to-date and coordinating all issues. This streamlined communication ensured that engineers were making progress. Proactively identifying the feature leads and their supporting teams at the beginning of new product development, rather than reactively waiting for a hole to appear in the process, would make a significant difference in the way we manage requirements.

For our next project, we will try to manage requirements early enough to include hardware requirements. The majority of the development organization consists of software development (host-based software and embedded software). This was the natural place to focus our development to gather requirement information that was readily available. For the next generation, we would like to have more influence on third-party companies delivering the hardware. This will involve specifying “quality goals” from an HP perspective, rather than relying mostly on the specifications that these companies provide. Currently, we have a process of “improving hardware quality,” but we do not understand how to do this according to HP desires.

We want to add more sources of input to the requirements process. A number of established organizations in HP’s LaserJet organization could strongly influence requirements if we established the proper communication channels. We successfully included Human Factors and Market Research input; however, we could benefit by including information from the Customer Care Center (customer support), beta testing (from previous products), warranty information, and system test information. In addition, R&D is very tied to the “future product marketing” organization, which looks at future trends and conducts market research, but never obtains feedback from “current product marketing.” This feedback includes messages that are pushed with the media and major customer accounts, and major features or blocking issues that we should add/fix to be able to establish higher sales.

Next time we will use shared requirements. LaserJet devices share many requirements across the whole line of products, from low-end monochrome to high-end color (requirements like PJL, PCL6, localization needs, RFI testing, manufacturing test needs, etc.). Caliber-RM 2.0 (introduced mid 1999) added the ability to share requirement descriptions between projects, thus creating a “parent-child” relationship between the two. This eliminates duplicating information by re-entering and synchronizing data between projects.

Our project requirements should form the basis for estimating, planning, performing, and tracking the project’s activities throughout its life cycle. At our current speed of developing products, we recognize the value of effectively estimating and tracking activities. Using requirements as a source for this task is an area we are now seriously investigating.

# References

1. J. Grady, *System Integration*, CRC Press, Boca Raton, FL, 1994.
2. J. Brackett, et. al., "Software Requirements Engineering," Software Engineering Institute Technical Course CE-SRE-01, Carnegie Mellon University, Pittsburgh, PA, 1992.
3. IEEE Standard 830-1994, *Recommended Practice for Software Requirements Specifications*, 1995.
4. W. Wilson, "Writing Effective Natural Language Requirements Specifications," *Crosstalk: The Journal of Defense Software Engineering*, February 1999.
5. B. Willis, M. Young, L. Simmons, Internal HP Requirements Documentation, 1998.
6. Technology Builders, Inc., *Caliber-RM Requirements Management User's Guide*, 1998.
7. Harold Kurstedt, "Human Components--The Greatest Challenge to System Success," *Proceedings of 13<sup>th</sup> Annual International Conference on Systems Engineering*, August 1999.

# **Requirements Traceability Improvement in Small Software Organizations**

T. Varkoi & T. Mäkinen

*Tampere University of Technology  
Information Technology, Pori, Finland*

## **ABSTRACT**

Requirements traceability improvement can support small software organizations to enhance their project management and improve customer satisfaction. This paper describes requirements processing and traceability development in seven small software organizations, which are participating in a joint project for software process improvement (SPI). The project uses ISO/IEC 15504 TR (SPICE) as the software process assessment and improvement framework. SPICE defines the process purpose, base practices and related work products for each process. Several of the processes deal with system and software requirements.

Our focus is on process improvement. First we analyze how SPICE treats traceability in processes and practices. Then we present the actual SPICE assessment results of the SPI project with respect to the goals that SPICE sets for traceability. We collect the improvement ideas, and define general guidelines for small organizations how to improve traceability in software development.

## **PRINCIPAL AUTHOR**

Mr. Varkoi started his software career with paper mill applications in 1983 after a B.Sc. degree in Forest Industry. Later he worked as an Application Engineer in the area of production management systems. After Computer Science studies at Tampere University of Technology he graduated M.Sc. in 1989. He then joined Jamont, one of Europe's leading tissue paper producers, and was appointed IT Manager for the Nordic Area. Mr. Varkoi was invited to Honeywell, Varkaus Automation Center in 1995 and became Development Manager for the Production Management Systems including responsibility for software process improvement. To conclude his post-graduate studies in computer science Mr. Varkoi moved to Pori School of Technology and Economics (Tampere University of Technology) where he is currently working as a project manager in a project to improve software processes with the local software companies.

Timo Varkoi  
Pori School of Technology and Economics  
P.O. Box 300, FIN-28101 Pori, Finland  
Email: Timo.Varkoi@pori.tut.fi

# **Requirements Traceability Improvement in Small Software Organizations**

T. Varkoi & T. Mäkinen

*Tampere University of Technology  
Information Technology, Pori  
P.O. Box 300, FIN-28101 Pori, Finland  
Phone: +358 2 627 2844  
Fax: +358 2 627 2727  
Email: Timo.Varkoi@pori.tut.fi*

## **ABSTRACT**

Requirements traceability improvement can support small software organizations to enhance their project management and improve customer satisfaction. Software requirements can be e.g. customer needs, operational or technical requirements, or quality criteria. Traceability of the requirements means that each requirement is identified, unique, and verifiable.

This paper describes requirements processing and traceability development in seven small software organizations, which are participating in a joint project for software process improvement (SPI). The project uses ISO/IEC 15504 TR (SPICE) as the software process assessment and improvement framework. SPICE defines the process purpose, base practices and related work products for each process. Several of the processes deal with system and software requirements.

Our focus is on process improvement. First we analyze how SPICE presents traceability in processes and practices. Then we discuss the actual SPICE assessment results of the SPI project with respect to the goals that SPICE sets for traceability. We collect the improvement ideas, and define general guidelines for small organizations how to improve traceability in software development.

## **INTRODUCTION**

Requirements traceability can save software projects. If requirements are properly tracked throughout the whole software lifecycle, we are able to better understand the status of the project, what has been done, and what still needs to be done. Software requirements can be e.g. customer



needs, operational or technical requirements, or quality criteria. Traceability of the requirements means that each requirement is identified, unique, and verifiable. Traceability is needed to ensure that each of the requirements is satisfied by the work products, and on the other hand, each work product is built to satisfy a set of pre-defined requirements.

This paper describes requirements processing and traceability development in seven small software organizations, which are participating in a joint project for software process improvement (SPI). The SPI project aims to enable small and medium sized software enterprises to develop their operations using international software process models. The goals of the project include improvement of customer satisfaction and competitiveness; management of business growth; improvement of personnel competence and motivation; development of working methods; and improvement of knowledge and skills by software engineering training. The project is based on the co-operation of the participating enterprises e.g. by sharing software process improvement experiences and best practices among the participants. The main activities in the project are process assessments, improvement planning, and consultation and training to support the improvement activities.

The project uses ISO/IEC 15504 TR (SPICE) as the software process assessment and improvement framework. Several of the processes deal with system and software requirements. The paper describes the chain of requirement specifications from customer requirements elicitation through software design to testing and integration of the software according to SPICE. The emphasis is on customer requirements.

Traceability of the requirements demands that each requirement shall be identified, unique and verifiable. The capability dimension of SPICE adds demands on management of the work itself and the management of the work products.

Pressman [5] gives definition of traceability: The ability to trace a design representation or actual program component back to requirements. This definition contains the idea of backward traceability: from work products to requirements to ensure that work products fulfil the requirements. It is equally important to have forward traceability from requirements to work products to be able to ensure that all requirements have been satisfied.

Ramesh [6] has identified two distinct groups with respect to requirements traceability practice: low-end and high-end traceability users. Low-end users view traceability simply as a mandate from project sponsors, whereas high-end users view traceability as an important component of a quality systems engineering process. The process perspective is discussed in system development context. Lower process capability of low-end users is described as "ad-hoc practices and methodologies" and higher capability of high-end users as "standardized methodologies and procedures" respectively. Mechanism for process improvement is seen as a facilitating characteristic for adoption and use of traceability.

Our focus is on process improvement. First we analyze how SPICE presents traceability in processes and practices. Then we discuss the actual SPICE assessment results of small organizations with respect to the goals that SPICE sets for traceability. We collect the

improvement ideas and define general guidelines for small organizations how to improve traceability in software development.

Small organizations are typically low-end users of traceability. Customer requirements are collected, but changes are not systematically managed. Requirements are not identified, which impedes traceability. Though positive exceptions exist where traceability is taken into account.

In SPICE the traceability improvement is related to improving the process capabilities primarily of the engineering processes and their base practices on level 1, and to supporting it with the level 2 management practices. The essential improvement targets common to most small organizations are:

- Identify the requirements and track changes
- Establish documentation and version control
- Reviews of customer requirements
- Checklists to assure quality of work products

The paper consists of four chapters. First SPICE structure is discussed including capability and process dimensions. Concept and relations of traceability in SPICE is analyzed. Second chapter presents the project and the SPICE assessment findings related to traceability. Thirdly we summarize the process improvement ideas that small organizations can use to improve their process capability and especially requirements tracking. The last chapter shortly describes the plans of the SPI project to help companies in achieving their improvement objectives.

## **1. FRAMEWORK**

The project uses ISO/IEC 15504 TR (SPICE) as the software process assessment and improvement framework. Several of the processes deal with system and software requirements. This chapter describes the chain of requirement specifications from customer requirements elicitation through software design to testing and integration of the software according to SPICE. The emphasis is on customer requirements.

The technical report of type 2 ISO/IEC 15504 (SPICE: Software Process Improvement and Capability dEtermination) published in 1998 provides a framework for the assessment of software processes. Process assessment has two principal contexts for its use: process improvement and process capability determination.

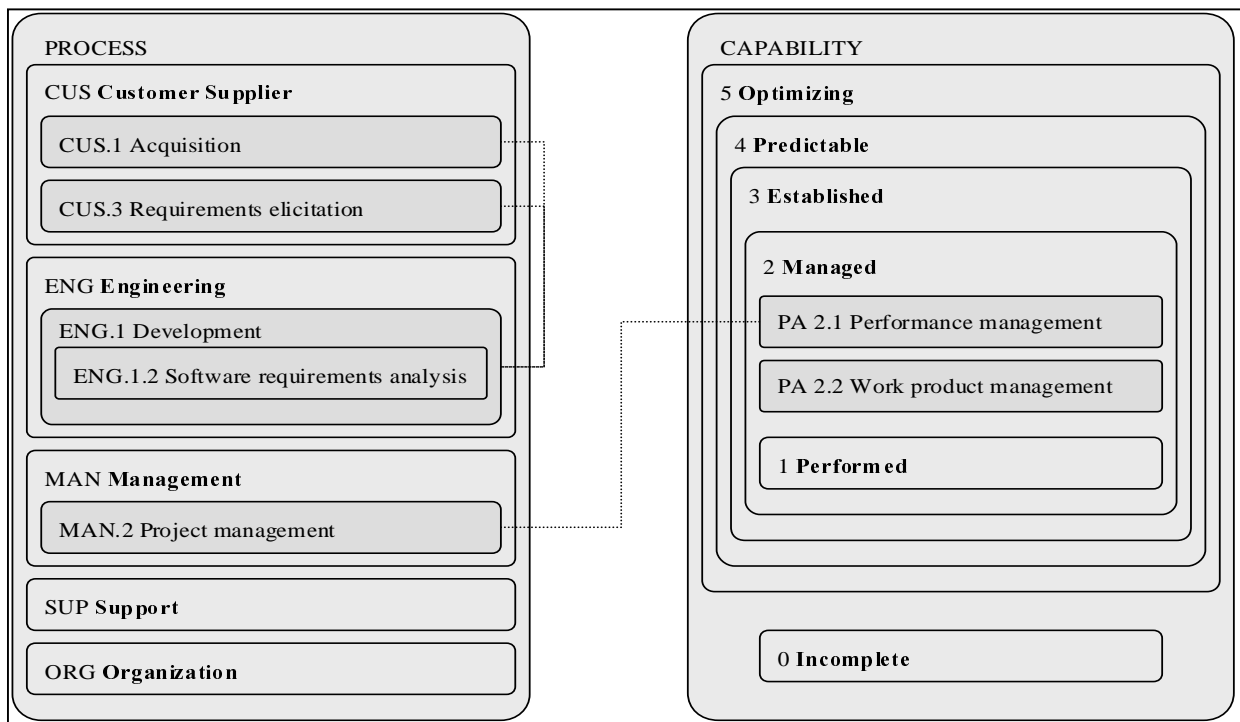
SPICE is composed of nine parts (see Appendix A for the list of all parts of the technical report). Part 2 [2] defines a two-dimensional reference model for describing processes and process capabilities used in a process assessment. Part 5 [3] provides an exemplar model for performing a process assessment that is based upon and directly compatible with the normative reference model defined in part 2. The assessment model, Part 5, extends the reference model, Part 2, through the inclusion of a set of indicators of process performance and capability. Part 5 also

describes base practices (BP) and management practices (MP). Base practices are Level 1 software engineering or management activities that address the implementation or establishment of process attributes. Management practices are upper capability level management activities or tasks that address the implementation or establishment of other process attributes.

The process dimension of Part 2 defines 40 processes and groups them into five process categories: CUS, ENG, SUP, MAN and ORG (Fig. 1.1). For example ENG.1.2 Software requirements analysis is a process which belongs to Engineering (ENG) process category. Some of the processes are components of another processes; e.g. ENG.1.2 is a component of ENG.1 Development. A process may be associated with other processes. For example the requirements for ENG.1 Development may be provided by the operation of CUS.1 Acquisition process or CUS.3 Requirements elicitation process.

Part 2 capability dimension consists of six capability levels from 0 Incomplete to 5 Optimizing (Fig. 1.1). Each level has from zero to two process attributes (PA). For example PA 2.1 Performance management attribute and PA 2.2 Work product management attribute are the process attributes of Level 2 Managed. A process may support the performance of one or two process attributes; e.g. MAN.2 Project management process supports performance of PA 2.1 in those instances where it is invoked. The capability level of a process instance is determined by how completely (Not, Partially, Largely, or Fully achieved) it fulfills the criteria defined by process attributes. For example, if a process instance is rated at level 3, its level 3 process attributes are Largely or Fully achieved and all the lower level process attributes are Fully achieved.

**Figure 1.1** SPICE process and capability dimensions.



The SPICE process model defines the process purpose, base practices and related work products for each process. Several of the processes deal with system and software requirements. Table 1.1 describes SPICE primary life cycle processes and their requirements traceability related outcomes. Table 1.2 describes traceability related SPICE supporting life cycle processes and their purposes.

**Table 1.1** SPICE primary life cycle processes and their requirements traceability related outcomes.

<b>PROCESS</b>	<b>PROCESS OUTCOMES ENSURING REQUIREMENTS TRACEABILITY</b>
<b>Requirements elicitation</b> (CUS.3)	<ul style="list-style-type: none"> <li>agreed customer requirements will be defined</li> <li>a mechanism will be established to incorporate new customer requirements into the established requirements baseline</li> </ul>
<b>Development</b> (ENG.1)	<ul style="list-style-type: none"> <li>intermediate work products will be developed that <b>demonstrate that the end product is based upon the requirements</b></li> <li><b>consistency will be established between requirements and designs</b></li> <li>evidence (for example, testing evidence) will be provided that <b>demonstrates that the end product meets the requirements</b></li> </ul>
<b>System requirements analysis and design</b> (ENG.1.1)	<ul style="list-style-type: none"> <li>requirements of the system will be developed that <b>match the customer's stated needs</b></li> </ul>
<b>Software requirements analysis</b> (ENG.1.2)	<ul style="list-style-type: none"> <li>the requirements allocated to software components of the system and their interfaces will be defined to <b>match the customer's stated needs</b></li> <li><b>consistency will be established between system requirements and design and software requirements</b></li> </ul>
<b>Software design</b> (ENG.1.3)	<ul style="list-style-type: none"> <li><b>consistency will be established between software requirements and software designs</b></li> </ul>
<b>Software construction</b> (ENG.1.4)	<ul style="list-style-type: none"> <li><b>consistency will be established between software requirements and design and software components</b></li> </ul>
<b>Software integration</b> (ENG.1.5)	<ul style="list-style-type: none"> <li>verification criteria for software items will be developed that <b>ensure compliance with the software requirements allocated to the items</b></li> <li><b>consistency will be established between software requirements and software items</b></li> </ul>
<b>Software testing</b> (ENG.1.6)	<ul style="list-style-type: none"> <li>acceptance criteria for integrated software will be developed that verify <b>compliance with the software requirements</b></li> </ul>
<b>System integration and testing</b> (ENG.1.7)	<ul style="list-style-type: none"> <li>acceptance criteria for each aggregate will be developed to verify <b>compliance with the system requirements allocated to the units</b></li> <li>an integrated system demonstrating <b>compliance with the system requirements</b> (functional, non-functional, operations and maintenance) and validation that a complete set of useable deliverable components exists, will be constructed</li> </ul>

**Table 1.2** SPICE processes supporting requirements traceability.

<b>SUPPORTING PROCESS</b>	<b>PROCESS PURPOSE</b>
<b>Configuration management</b> (SUP.2)	<ul style="list-style-type: none"> <li>to establish and maintain the <b>integrity of all the work products</b> of a process or project</li> </ul>
<b>Verification</b> (SUP.4)	<ul style="list-style-type: none"> <li>to confirm that each software work product and/or service of a process or project <b>properly reflects the specified requirements</b></li> </ul>
<b>Validation</b> (SUP.5)	<ul style="list-style-type: none"> <li>to confirm that the <b>requirements for a specific intended use of the software work product are fulfilled</b></li> </ul>
<b>Joint review</b> (SUP.6)	<ul style="list-style-type: none"> <li>to maintain a common understanding with the customer of the progress against the objectives of the contract and what should be done to help <b>ensure development of a product that satisfies the customer</b></li> </ul>
<b>Audit</b> (SUP.7)	<ul style="list-style-type: none"> <li>to independently determine <b>compliance of selected products</b> and processes <b>with the requirements</b>, plans and contract, as appropriate</li> </ul>

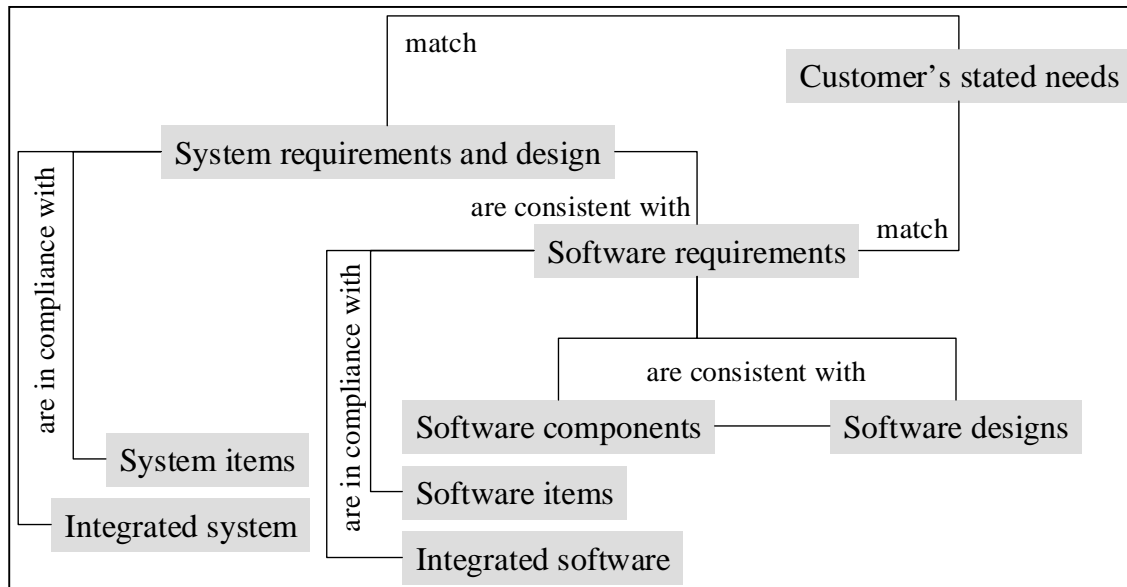
The capability dimension of SPICE adds demands on e.g. management of the work itself and the management of the work products. Requirements traceability related process attributes are on SPICE Level 2 and they are presented in Table 1.3.

**Table 1.3** SPICE level 2 process attributes and their requirements traceability related outcomes.

<b>PROCESS ATTRIBUTE</b>	<b>ATTRIBUTE OUTCOMES ENSURING REQUIREMENTS TRACEABILITY</b>
<b>Performance management</b> (PA 2.1)	<ul style="list-style-type: none"> <li>the <b>objectives</b> for the performance of the process will be identified (e.g. <b>quality</b>, time-scale, cycle time and resource usage)</li> <li>the performance of the process will be managed to produce <b>work products that meet the defined objectives</b></li> </ul>
<b>Work product management</b> (PA 2.2)	<ul style="list-style-type: none"> <li>the <b>requirements</b> (functional and non-functional) <b>of the specified work products</b> of the process <b>will be defined</b></li> <li>the <b>dependencies</b> among the controlled work products <b>will be identified</b></li> <li>the <b>work products</b> will be verified and adjusted to <b>meet</b> the defined <b>requirements</b></li> </ul>

The chain of requirement specifications from customer requirements elicitation through software design to testing and integration of the software according to SPICE is described in Figure 1.2.

**Figure 1.2** Chain of requirements in SPICE.



## 2. OBSERVATIONS

The SPI project provides the participating companies process assessments, training and consultation. SPICE Part 7 [4] describes eight-step continuous software process improvement cycle. First step is to analyze organization's needs. This is done using BootCheck quick assessment tool [1]. The results support preliminary process improvement program planning and guide in selecting processes for SPICE assessment. The assessment output is used in detailed process improvement action planning.

During the SPI project several software projects of the participating companies have been assessed. Each company was assessed separately and the assessment results were reported in feedback sessions and in detailed assessment reports. Assessment results and observations point out that requirements management is based mostly on individual effort and that documented traceability practices are nearly non-existent. Even though it might be possible to manage very small projects without established requirements management, most often the outcome is an obscure scope of the customer project, slipping timetables and decreased profit.

The SPICE assessments give lots of detailed information and improvement ideas for the SPI in the organizations. This paper documents a collection of the most typical findings related to requirements management and traceability in Appendix B. The process relations were already described in chapter 1. Presented findings are limited to relevant level 1 base practices and level 2 management practices of some of the primary life cycle processes.

Level 2 management practices presented for each process are:

- MP 2.2.2 Manage the documentation, configuration management and change control of the work products.
- MP 2.2.3 Identify and define any work product dependencies.

Processes and base practices that are used as an example are:

- CUS.3 Requirements elicitation process
  - CUS.3.BP1 Obtain customer requirements and requests
  - CUS.3.BP4 Manage customer requirements changes
- ENG.1.2 Software requirements analysis process
  - ENG.1.2.BP1 Specify software requirements.
  - ENG.1.2.BP8 Evaluate the software requirements.
- ENG.1.3 Software design process
  - ENG.1.3.BP3 Verify the software design.
  - ENG.1.3.BP5 Establish traceability.
- ENG.1.6 Software testing process
  - ENG.1.6.BP2 Develop tests for integrated software.

Findings of all the assessments are grouped together to protect anonymity of the assessed organizations.

In general the assessment results show that companies do collect customer requirements, but changes are not systematically managed. In addition the requirements are not identified, which impedes traceability. Some positive exceptions exist where traceability is taken into account, though improvements in identification and verification are still needed.

### **3. RECOMMENDATIONS**

In this chapter we discuss our recommendations to improve traceability in small software organizations. Improvement objectives are based on the assessment findings and ideas collected during the SPICE assessments in the organizations.

Use of naming conventions is essential in providing full control over the requirements between processes. The key recommendation is first to create a consistent naming practice for the requirements and then to define and apply practices to support requirements traceability throughout the project.

The actual findings and recommendations are documented in the SPICE assessment reports. Reports were delivered to the assessed organizations as part of the assessment output. Recommendations form the basis to be used in detailed process improvement action planning. A summary of the recommendations is listed in Appendix C. Presented recommendations and ideas

are limited to the same processes as with findings in Chapter 2. Improvement objectives are here grouped by capability levels 1 and 2, also to protect anonymity of the assessed organizations.

In SPICE the traceability improvement is related to improving the process capabilities primarily of the engineering processes and their base practices on level 1, and to supporting it with the level 2 management practices. The essential improvement targets common to small organizations are:

- Identify the requirements and track changes
- Establish documentation and version control
- Reviews of customer requirements
- Checklists to assure quality of work products

These traceability improvements would help reaching SPICE level 2 capability in the engineering processes. Beyond the level 2, the organizations should consider documentation and resourcing of the traceability related processes. Traceability measurement should also be planned.

For instance, Sallis et al. [7] present simple requirements traceability measurements:

$$\begin{aligned}\text{Percentage of requirements traceability} &= (RI / R) * 100 \% \\ \text{Percentage of non-agreed introduced requirements} &= (RE / R) * 100 \%\end{aligned}$$

where  $R$  is the number of agreed requirements,  $RI$  is the number of agreed requirements implemented and  $RE$  is the number of implemented requirements not in the agreed set. These measures correspond to the backward ( $RE$ ) and forward ( $RI$ ) traceability discussed earlier.

#### 4. NEXT STEPS

Requirements traceability has a role in helping small software companies to reach their business goals. Tracked customer requirements support better project management and improved customer satisfaction. The key issue for the SPI project is to encourage companies in improvement planning and actions. The assessment results and recommendations will be further discussed in each organization.

The SPI project will provide training and consultation for the organizations to improve their requirements management. Requirements management improvement will be integrated in the improvement planning, which originally is based on the improvement priorities each organization has set. The small organizations' priorities are described in [8]. Training includes requirements traceability related courses and workshops. The idea of workshops is that the participating organizations can exchange their experiences and ideas about the subject. Consultation is related to actual traceability improvement actions in the companies. Traceability improvements will be confirmed with follow-up assessments.



## References

- [1] BOOTSTRAP Institute: *BootCheck assessment tool*, <http://bootstrap.ccc.fi>.
- [2] ISO/IEC TR 15504-2:1998 Information technology - Software process assessment - *Part 2: A reference model for processes and process capability*.
- [3] ISO/IEC TR 15504-5:1998 Information technology - Software process assessment - *Part 5: An assessment model and indicator guidance*.
- [4] ISO/IEC TR 15504-7:1998 Information technology - Software process assessment - *Part 7: Guide for use in process improvement*.
- [5] Pressman, Roger S., *Software Engineering: a practitioner's approach*, McGraw-Hill 1992.
- [6] Ramesh, B., *Factors Influencing Requirements Traceability Practice*, Communications of the ACM, Vol. 41, No. 12, Dec. 1998.
- [7] Sallis, Philip J. et al., *Software engineering: practice, management, improvement*, Addison-Wesley 1995.
- [8] Varkoi, Timo & Mäkinen Timo & Jaakkola, Hannu: *Process Improvement Priorities in Small Software Companies*, Proceedings of the PICMET'99, Portland, Oregon, 1999.

## Appendixes

### A. Parts of the ISO/IEC 15504 TR (SPICE)

- Part 1: Concepts and introductory guide (informative)
- Part 2: A reference model for processes and process capability (normative)
- Part 3: Performing an assessment (normative)
- Part 4: Guide to performing assessments (informative)
- Part 5: An assessment model and indicator guidance (informative)
- Part 6: Guide to qualification of assessors (informative)
- Part 7: Guide for use in process improvement (informative)
- Part 8: Guide for use in determining supplier process capability (informative)
- Part 9: Vocabulary (informative)

## B. SPICE assessment findings and comments

**Table B.1** Comments and assessment findings related to SPICE process CUS.3.

<b>CUS.3 Requirements elicitation</b>	<b>Comments and assessment findings</b>
<b>Level 1, Base practices</b>	
CUS.3.BP1 Obtain customer requirements and requests.	<ul style="list-style-type: none"> <li>• Supplier has experience of previous similar projects</li> <li>• Customer did not provide any written material; requirements drop continually; customer has accepted memos where requirements are written</li> <li>• Basis was customer's present system; requirements are shortcomings of the running system</li> <li>• Prestudy was done during sales phase, but it is not used by the project</li> <li>• Customer provided business requirements</li> <li>• Sales phase took two years, offer was made in a hurry, collected material is of no use to the project</li> <li>• Customer requirements are not collected systemically in the sales phase</li> <li>• Customer has presented functional diagrams, operations are under development</li> <li>• Functional specification was given by the customer</li> <li>• Customer requirements document has been written based on minutes of the meetings</li> <li>• Customer has a requirements specification, but it is based on another technology</li> <li>• Requirements are attached in the offer, not identified</li> <li>• Customer requirements were changed during the project</li> <li>• Project was aware of roughly 70% of the requirements in the beginning</li> <li>• Six months sales phase provided lots of valuable material for the project</li> </ul>
CUS.3.BP4 Manage customer requirements changes.	<ul style="list-style-type: none"> <li>• Requirements modification procedure was not specified</li> <li>• Each meeting with the customer provided new/changed requirements</li> <li>• The large amount of changes was difficult to manage</li> <li>• Successive changes created inconsistency</li> <li>• No exact set of requirements</li> <li>• Schedule has slipped due to customer</li> <li>• Not specified in the contract; project group makes proposals to steering group</li> <li>• The risks of the changes were discussed but not documented</li> <li>• Contract prohibits any changes, change procedures are missing</li> <li>• Changes are discussed in meetings</li> <li>• Supplier makes the changes, customer is informed</li> <li>• Ineffective risk analysis, total work load is estimated</li> <li>• Specification contains items that are not based on customer requirements</li> <li>• Requirements are not reviewed</li> <li>• Changes must be accepted by project managers of both</li> </ul>

	customer and supplier <ul style="list-style-type: none"> <li>Changes to the accepted set of requirements are discussed and approved</li> </ul>
<b>Level 2, Management practices</b>	
MP 2.2.2 Manage the documentation, configuration management and change control of the work products.	<ul style="list-style-type: none"> <li>Change requests are not recorded systematically, requests are mainly in e-mailboxes</li> <li>Document structure not defined</li> <li>Changes are agreed in meetings</li> <li>Small changes updated to specification without control, large changes noted in minutes</li> <li>Documents are distributed by e-mail</li> <li>Specification is updated on-line with customer</li> <li>Changes are noted in the minutes of a meeting and followed up in next meeting</li> <li>Document template contains field for acceptance - used occasionally</li> <li>Managed by document date, the last one is valid</li> <li>Fixed folder structure for project work products, documents delivered to customer can be tracked</li> <li>Tools are provided to manage documents</li> <li>Changes to previous version are shown underlined, each version is it's own file</li> <li>Color is used to show status of items in the specification</li> </ul>
MP 2.2.3 Identify and define any work product dependencies.	<ul style="list-style-type: none"> <li>Traceability is missing, information is scattered</li> <li>Under project managers responsibility</li> <li>Consistency of customer requirements should be improved</li> <li>Work products are controlled, date tells the last version</li> <li>Versions have different document numbers</li> <li>Review comments are written directly to the document</li> <li>Necessary changes are made based on the review comments</li> </ul>

**Table B.2** Comments and assessment findings related to SPICE process ENG.1.2.

<b>ENG.1.2 Software requirements analysis process</b> <b>Comments and assessment findings</b>	
<b>Level 1, Base practices</b>	
ENG.1.2.BP1 Specify software requirements.	<ul style="list-style-type: none"> <li>Requirements have been categorized: functions, characteristics, interfaces, restrictions and constrains</li> <li>Clear traceability missing</li> <li>Reference by chapter numbers, unique identifiers are not used</li> <li>Specification is made afterwards based on notes</li> <li>Specification does not fully correspond to implementation</li> <li>Customer requirements have been identified partially</li> <li>Standards have been followed</li> <li>Identification is partial</li> </ul>
ENG.1.2.BP8 Evaluate the	<ul style="list-style-type: none"> <li>No unique identification for a requirement</li> </ul>

software requirements.	<ul style="list-style-type: none"> <li>• Partial naming conventions for requirements</li> <li>• Requirements that are impossible to implement are not recorded</li> <li>• System level needs for change are known</li> <li>• Consistency is controlled</li> </ul>
<b>Level 2, Management practices</b>	
MP 2.2.2 Manage the documentation, configuration management and change control of the work products.	<ul style="list-style-type: none"> <li>▪ Identification is not used with requirements</li> <li>▪ Number of changes is not tracked</li> <li>▪ Updated version of the document is available on the server, no assurance that all know about changes</li> </ul>
MP 2.2.3 Identify and define any work product dependencies.	<ul style="list-style-type: none"> <li>▪ Changes are documented randomly</li> <li>▪ Traceability to customer requirements is missing</li> <li>▪ Peer review results are not recorded</li> </ul>

**Table B.3** Comments and assessment findings related to SPICE process ENG.1.3.

<b>ENG.1.3 Software design process</b>	<b>Comments and assessment findings</b>
<b>Level 1, Base practices</b>	
ENG.1.3.BP3 Verify the software design.	<ul style="list-style-type: none"> <li>• Designs are not collected systematically</li> <li>• Documents and user interfaces are designed, not detailed enough for implementation</li> </ul>
ENG.1.3.BP5 Establish traceability.	<ul style="list-style-type: none"> <li>• Traceability to functional specification is partial, no references to functions</li> <li>• Traceability cannot be verified</li> <li>• Naming conventions exist, inconsistent with specification</li> <li>• Traceability does not exist</li> <li>• Forward traceability is missing</li> <li>• Backward traceability is missing</li> </ul>
<b>Level 2, Management practices</b>	
MP 2.2.2 Manage the documentation, configuration management and change control of the work products.	<ul style="list-style-type: none"> <li>▪ Number of changes is not tracked</li> </ul>
MP 2.2.3 Identify and define any work product dependencies.	<ul style="list-style-type: none"> <li>▪ Inadequate traceability</li> <li>▪ Implementation of changes is not controlled</li> </ul>

**Table B.4** Comments and assessment findings related to SPICE process ENG.1.6.

<b>ENG.1.6 Software testing process</b>	<b>Comments and assessment findings</b>
<b>Level 1, Base practices</b>	
ENG.1.6.BP2 Develop tests for integrated software.	<ul style="list-style-type: none"> <li>• Satisfactory for test items, acceptance criteria not defined</li> </ul>

	<ul style="list-style-type: none"> <li>• Some test cases created in advance</li> <li>• Developed only partially</li> <li>• Test cases included in the test plan</li> <li>• Bulk tests are not performed</li> <li>• Adequate test material in use</li> </ul>
<b>Level 2, Management practices</b>	
MP 2.2.2 Manage the documentation, configuration management and change control of the work products.	<ul style="list-style-type: none"> <li>▪ Not defined</li> <li>▪ Inadequate, unclear practice</li> <li>▪ Principals in handbook, practice according to tester</li> <li>▪ After acceptance naming changes required by configuration control are made</li> <li>▪ Practice is followed but not documented</li> <li>▪ Regression testing is partially described</li> <li>▪ Version level is updated</li> <li>▪ Corrections are made according to procedure, errors are classified (location, reason, phase, severity)</li> </ul>
MP 2.2.3 Identify and define any work product dependencies.	<ul style="list-style-type: none"> <li>▪ Use cases are used to test integrated modules, performance of tests is controlled</li> <li>▪ Demo system contains test material</li> <li>▪ Erroneous data is not used in tests</li> <li>▪ All test cases will be performed, new ones are created if necessary</li> <li>▪ Production platform was taken into account in tests</li> <li>▪ All uncorrected errors are tracked</li> <li>▪ Interface testing is performed</li> </ul>

### C. SPICE assessment findings and comments

**Table C.1** Key improvement objectives of SPICE process CUS.3.

<b>CUS.3 Requirements elicitation process</b>	<b>Key improvement objectives</b>
<b>Level 1, Performed process</b>	<ul style="list-style-type: none"> <li>• Agree in advance with the customer about the procedures to change requirements</li> <li>• Identify the requirements and track changes</li> <li>• Systematic processing for customer requirements</li> <li>• Establish documentation, identification and version control</li> <li>• Checklists to support requirements elicitation</li> <li>• Establish status control for requirements</li> <li>• Management of changes</li> <li>• Reporting to customer</li> <li>• Organize development meetings with customers</li> <li>• Improvement of forward/backward traceability (references, identification)</li> <li>• Traceability in project documentation</li> <li>• Definition of traceability procedures and responsibilities</li> <li>• Customer requirements review</li> <li>• Use of customer's language</li> <li>• Analysis of the original reason behind a requirement</li> </ul>

<b>Level 2, Managed process</b>	<ul style="list-style-type: none"> <li>• Improve traceability: documentation must tell that each functional requirement is derived from one or more customer requirements</li> <li>• Acceptance is clearly noted in minutes of meetings and requirements lists</li> <li>• Model for work estimation and planning</li> <li>• Proposal to customer of necessary resources</li> <li>• Risk analysis</li> <li>• Checklists to assure quality of work products</li> <li>• Checklist to assess source material coverage</li> <li>• Goal setting and control</li> <li>• Clarify role of the sales force in acquiring requirements and in passing them to a project</li> <li>• Quality criteria for documents and review</li> </ul>
---------------------------------	---

**Table C.2** Key improvement objectives of SPICE process ENG.1.2.

<b>ENG.1.2 Software requirements analysis process</b>	<b>Key improvement objectives</b>
<b>Level 1, Performed process</b>	<ul style="list-style-type: none"> <li>• Define acceptance criteria</li> <li>• Naming and identification of requirements</li> <li>• Define contents of a requirements specification</li> <li>• Describe update procedures and change management</li> <li>• Update and track list of changes</li> </ul>
<b>Level 2, Managed process</b>	<ul style="list-style-type: none"> <li>• Describe the material needed from the customer</li> <li>• List risks</li> <li>• Document quality criteria, quality plan</li> <li>• Communicate changes</li> <li>• Documented quality requirements</li> <li>• Plan actions</li> </ul>

**Table C.3** Key improvement objectives of SPICE process ENG.1.3.

<b>ENG.1.3 Software design process</b>	<b>Key improvement objectives</b>
<b>Level 1, Performed process</b>	<ul style="list-style-type: none"> <li>• Identify the requirements and track changes</li> <li>• Transaction diagrams of the interfaces</li> <li>• Document software designs</li> <li>• Define development process</li> <li>• Define internal connections and interfaces</li> <li>• Software design documentation for maintenance and operation of the software</li> <li>• Use of the design in implementation</li> </ul>
<b>Level 2, Managed process</b>	<ul style="list-style-type: none"> <li>• Put reviews in the project plan</li> <li>• Document and report reviews</li> <li>• Introduce change history, change control</li> <li>• Define design goals and resources</li> <li>• Quality criteria of the design</li> </ul>

**Table C.4** Key improvement objectives of SPICE process ENG.1.6.

<b>ENG.1.6 Software testing process</b>	<b>Key improvement objectives</b>
<b>Level 1, Performed process</b>	<ul style="list-style-type: none"> <li>• Creation of test cases in the design process</li> <li>• Definition of acceptance criteria, e.g. performance criteria, and non-functional quality criteria</li> <li>• Document principles of regression testing</li> </ul>
<b>Level 2, Managed process</b>	<ul style="list-style-type: none"> <li>• Observation of external interfaces in testing</li> <li>• Agree quality criteria for tests with the customer</li> <li>• Identify work products in process description</li> <li>• Error documentation and processing</li> <li>• Error classification</li> <li>• Adequacy of test material</li> <li>• test results could be sent to customer</li> <li>• Improve coverage of the test cases</li> </ul>

## **Taming Requirements Through Metrics**

Dr. Linda Rosenberg, NASA, GSFC

Requirements are the foundation for every project, yet little focus is placed on their development and management. There are tools for requirements storage and tracing, but even theses can be overwhelming. The bottom line is, how do we manage requirements? The answer: throughout the life of the project and with the application of metrics. This talk will focus on using metrics to gain control, and hence better manage, requirements.

Dr. Linda Rosenberg is an Engineering Section head at Unisys Government Systems in Lanham, M.D. She is contracted to manage the Software Assurance Technology Center (SATC) through the System Reliability and Safety Office in the Flight Assurance Division at Goddard Space Flight Center, NASA. The SATC primary responsibilities are in the areas of Metrics, Assurance Tools and Techniques, Risk Management, and Outreach programs. Although she oversees all work areas of the SATC, Dr. Rosenberg's area of expertise is metrics. She is responsible for overseeing metric programs to establish a basis for numerical guidelines and standards for software developed at NASA, and to work with project managers to use metrics in the evaluation of the quality of their software. Dr. Rosenberg's work in software metric outside of NASA includes work with the Joint Logistics Command's efforts to establish a core set of process, product and system metrics with guidelines published in the Practical Software Measurement. In addition, Dr. Rosenberg worked with the Software Engineering Institute to develop a risk management course. She is now responsible for risk management training at all NASA centers. Dr. Rosenberg holds a Ph.D. in Computer Science from the University of Maryland, an M.E.S. in computer science from Loyola College, and a B.S. in Mathematics from Towson State University.



# **An Operational Model Supporting The Generation of Requirements That Capture Customer Intent**

Markus K. Gröner                      James D. Arthur  
Department of Computer Science  
Virginia Tech  
Blacksburg, VA 24061-0106

{groener, arthur}@vt.edu

**Abstract:** Product quality is a direct reflection of how well it meets the customer's needs. Hence, the ability to capture customer requirements correctly and succinctly is paramount. Unfortunately, within software development frameworks the gathering of requirements is one of the more ill-defined and least structured processes. To address this inadequacy, our paper (a) proposes a refinement (or decomposition) to the requirements generation phase that admits to detailed examination and promotes a better understanding of where requirements generation goes wrong, (b) using the results of a preliminary study, identifies where and how miscommunication and requirements omission occur, and (c) outlines a more structured process that can be applied early in the requirements generation process to minimize requirement faults.

**Keywords:** Requirements Identification, Requirements Generation, Customer Intent, Requirements Engineering, Software Engineering, Software Methodology

## **Author biographies**

**Markus K. Gröner** is currently pursuing his Ph.D. at Virginia Tech (VPI&SU). He received his B.S. and M.S. in Computer and Information Sciences from the University of South Alabama (USA) in 1991 and 1994. His research interest is rooted in Software Engineering with a special emphasis on (a) establishing an error-resistant communication framework between Clients and Software Engineers, and (b) improving Software Requirement Specifications through better IV&V methods. Mr. Gröner has held teaching and research assistantships throughout his graduate career. He is a member of both the ACM and IEEE societies. Mr. Gröner can be contacted at Virginia Tech, Department of Computer Science, 660 McBryde Hall, Blacksburg, VA, 24061. His email address is groener@vt.edu; his telephone number is (540) 231-5853.

**James D. Arthur** received the B.S. and M.A. in mathematics from the University of North Carolina, the M.S. from the University of Houston, and the M.S. and Ph.D. from Purdue University. He has been involved in software engineering research for the past fifteen years. His research interests encompass requirements engineering, IV&V, software quality assessment, user environments, and parallel computation. He is currently an Associate Professor of Computer Science at Virginia Tech in Blacksburg, VA. Dr. Arthur can be contacted at Virginia Tech, The Department of Computer Science, 660 McBryde Hall, Blacksburg VA, 24061. His e-mail address is arthur@vt.edu; his telephone number is (540) 231-7538.

# **An Operational Model Supporting The Generation of Requirements that Capture Customer Intent**

## **1. Introduction**

A successful software development effort, i.e. one resulting in a quality product delivered on schedule and within budget, depends on a well-defined software development process. Such a process is guided by procedures and methods outlined in models like the Waterfall Model [1] and the Spiral Model [2], as well as development approaches such as rapid or throw-away prototyping [3] [4]. The objective in using these procedures and methods is to promote product correctness and to provide qualitative guidance during development. Our research and experience have shown, however, that no matter how rigorous the development process is, the delivered product is only as good as the set of requirements by which it is defined. More specifically, a delivered software product may possess all of the quality aspects induced through an effective software engineering process, yet not fulfill the most important objective -- perform as intended by the customer.

The primary task of the requirements engineer is to capture accurately the requirements expressed by the customer. To do so, the real challenge emerges -- recognizing inconsistent, wrong, and/or implied requirements. Contributing to that challenge is the observation that often the customer (a) has only a minimal understanding of the many functions that the proposed system is to support, (b) lacks experience in identifying and expressing desirables as requirements, and (c) states desirables in terms that lead to misleading or incorrect inferences by the requirements engineer [5]. In similar ways, the requirements engineer also contributes to the challenge. Consequently, we often fail in or find it very difficult to achieve our task of capturing requirements that *truly* reflect the customer's intent and expectations.

Models and methodologies that assist *both* the requirements engineer and the customer in the generation process are fundamental to capturing requirements accurately. Unfortunately, finding such a model or methodology that is *effective* is difficult indeed. Existing models like Participatory Design (PD) [6] [7] [8] and Joint Application Design (JAD) [9, 10] expand on conventional software development lifecycle models. In particular, they focus on reducing the disparity in domain knowledge between the requirements engineer and customer, respectively. Both PD and JAD prescribe a closer working relationship between the requirements engineer and customer -- which, expectantly, should result in the generation of a better set of software development artifacts. The major obstacle to the acceptance of PD and JAD by industry, however, has been its high cost in terms of time commitment for both the customer and development personnel [7]. Nonetheless, the contributions of these (and other) approaches, *and* their apparent pitfalls, underscore the need for a cost-effective approach to improving the requirements generation process, and ultimately, the quality of the product.

In response to the above observations, this paper focuses on the following question: "*How does one evolve requirements that capture the intent of the customer?*" The results of an initial study and on-going research indicate that one answer lies in providing structure and guidance during the requirements definition phase. Reflecting those results, we propose (a) a framework that provides structure to the requirements definition process, and (b) an integrated methodology that

guides the generation process. The framework partitions the conventional requirements definition phase into three well-defined mini-phases:

- A *setup* mini-phase that governs all activities up to and including the initial meeting between the customer and the requirements engineer,
- A *requirements capturing* mini-phase that focuses on the identification, recording and refinement of requirements, and
- A *transformation* mini-phase that guides the process by which the requirements and context information produced in the preceding mini-phase are transformed into a software requirements specification document (SRS).

The methodology consists of protocols, procedures and methods that assist in requirements elicitation, recording and analysis. In particular the methodology focuses on two aspects of requirements generation:

- Providing “tried and true” guidance that has a high probability of success, and
- Incorporating procedures for identifying characteristics indicative of problems and suggesting correction methods.

In effect, the framework and methodology have been purposefully designed to (a) add structure and control to the requirements definition phase, (b) support the accurate capturing of requirements, and we conjecture, (c) reduce the cost of an effective requirements generation process.

The structure of the remainder of the paper is as follows. Section 2 provides an introduction to the framework overlaying the conventional requirements definition phase. The figure provided and the corresponding discussion illustrate *where* guidelines and protocols fit within the framework and describe *how* they support improved requirements generation. Section 3 presents a detailed discussion of the second mini-phase: *requirements capturing*. Section 4 describes the three primary components of our proposed methodology and discusses how they span the phases of requirements definition. Results from a prior study are used to illustrate the potential utility of the methodology. Lastly, section 5 provides a summarization of our proposed model and a discussion of current work in progress.

## **2. The Operational Model: Framework, Guidelines and Protocols**

Given the conventional Waterfall Model (or one of its many variants), five development phases are most prominent: requirements, design, implementation, testing, and maintenance. Moreover, we often find one or more of these phases further decomposed to emphasize important sub-components. For example, it is common to find design split into a high- and a low-level design phase, as well as implementation divided into coding and unit-testing. The requirements generation phase is composed of two primary activities - requirements definition and requirements analysis. Because of inconsistent terminology [11], however, requirements generation is often equated with only requirements analysis. This view, unfortunately, promotes the misconception that requirements definition is the less significant of the two. Clearly, such implications run counter to the observations that “a system is only as good as the requirements

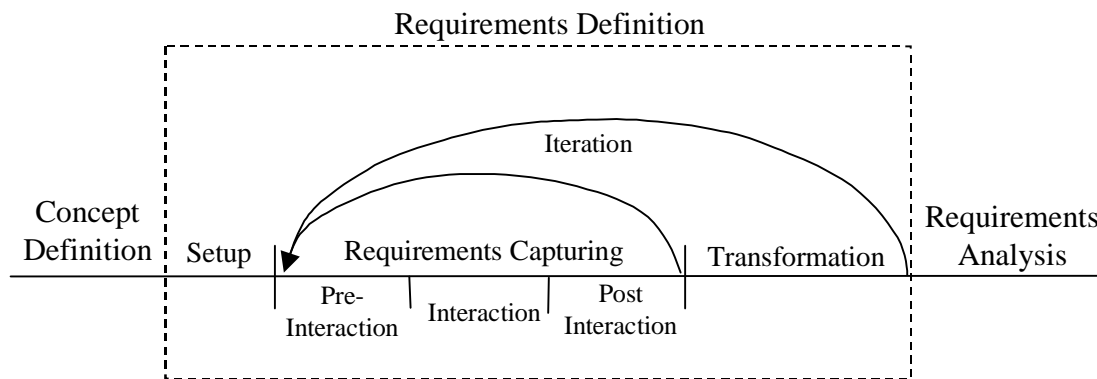
from which it is built” and “requirements are only as good as the extent to which they reflect the customer’s needs and intent.”

To address the misconception noted above and to promote the generation of requirements that better reflect the customer’s intent, we propose a refinement of the requirements definition phase that imposes structure and methodology. The three major components of the refinement are (a) a structured framework that partitions the requirements definition activity into three mini-phases, (b) a set of guidelines and protocols to suggest and structure activities within the framework, and (c) procedures and methods that assist in requirements definition.

To start, this section provides an overview of the proposed framework. Following the overview, we outline the roles of guidelines and protocols as related to structuring the framework.

## 2.1 The Proposed Framework

As illustrated in Figure 1, the proposed framework for requirements definition is comprised of three mini-phases: the setup, requirements capturing, and transformation. Furthermore, requirements capturing is partitioned into three additional components: pre-interaction, interaction and post-interaction. The structure of the framework underscores an important aspect of requirements definition: it consists of several distinct activities with each having its own independent set of objectives. The framework structure also illustrates that the requirements definition process is not necessarily linear in activities. That is, requirements definition is (or should be) an iterative process that constantly refines and elaborates on existing requirements.



**Figure 1:** Proposed Framework

An outline of the three mini-phases is provided below.

### *The Three Mini-Phases*

- (1) The “setup” phase is the first of the mini-phases. It governs all activities leading up to and including the initial meeting between the customer and the requirements engineer. In particular, the setup mini-phase outlines each participant’s initial responsibilities prior to beginning requirements capturing. For example, the customer is to provide a high-level overview of the existing and proposed system; the requirements engineer is to instruct the customer about the process underlying requirements generation.

- (2) The second mini-phase, “requirements capturing”, focuses on those activities where requirements are being identified, recorded and refined. Within this mini-phase the customer and requirements engineer prepare documents for meetings, attend well-structured and controlled meetings, examine artifacts from those meetings, and decide if additional meetings are necessary. The requirements capturing mini-phase is iterative in nature, allowing the refinement of requirements through progressive elaboration and reification [12]. Artifacts produced during this mini-phase include an unstructured software requirements list (USRL) and supporting contextual information.
- (3) The “transformation” mini-phase is third in our partitioning and guides the process by which the USRL and contextual information (defined during the requirements capturing mini-phase) are transformed into a software requirements specification document (SRS). Initially requirements are grouped together based on similarity characteristics; connections between those groups are then established. Among other criteria, requirements are examined for completeness, accuracy, understandability, and their relationship to other requirements. If major problems are identified during this phase, an additional iteration through the requirements capturing mini-phase is initiated.

In an effort to minimize duplication of effort between the transformation mini-phase and requirements analysis, the analysis phase should exclude or significantly limit the effort allocated to overlapping activities.

The framework presented above outlines an organization of activities necessary for effective requirements generation. Those organization and attendant activities are not, however, sufficient. In particular, clear and unambiguous guidance reflecting the objectives and constraints within and among the individual activities must be provided. The guidelines and protocols described below are integral elements of our operational model, and provide that needed guidance.

## 2.2 Guidelines

Guidelines are used to familiarize the participants with the requirements definition process, and to *suggest* activities that help ensure a successful requirements definition process. To help educate the participants, guidelines include overviews of

- the requirements definition process, its components, and activities,
- what results can be expected when going through the process,
- how the participants are involved in the process, and
- how each participant should prepare for the definition process.

More generally, the guidelines provide suggestions for supportive activities that (a) span all mini-phases, or (b) focus on individual ones.

*Cautionary Note:* Guidelines must be written in a language that can be understood by all participants. The use of technical terminology should be limited and must be explained in a glossary. In effect, guidelines are a reference for all participants throughout the entire requirements definition process and must be written accordingly.

## 2.3 Protocols

Protocols define the structure imposed on the framework. Unlike guidelines that offer guidance (adherence is optional) to the participant in the process, protocols establish constraints that define the structure (adherence is required). More specifically, given the objectives of any particular mini-phase, protocols help to ensure that they are met by keeping all participants operating within well-defined “structural boundaries.” Similar to guidelines, certain protocols permeate the framework while others are specific to individual mini-phases. For example, protocols specific to the requirements capturing mini-phase state that before every interaction meeting (a) the subject and objective of that meeting must be established, and (b) all participants and their roles must be identified.

Clearly, protocols state “what must be done.” We acknowledge, however, that they are only words on paper. If they are not followed, one *does not realize* their beneficial contributions.

## 3. Phases

In section 2 we introduce the framework and outline its three mini-phases. We now present a more detailed description of the framework’s first three mini-phases: (1) setup, (2) requirements capturing, and (3) requirements transformation. The following discussion focuses on problems identified through experience and research that can occur during these mini-phases. We introduce procedures and methods designed to prevent or eliminate the problems associated with the requirements capturing process. Although the problems discussed in this paper are not specific to any one particular domain, they do represent some of the more critical ones. We contend that even after an application domain is established, little (if any) tailoring of our proposed procedures and methods will be required.

### 3.1 Setup

As shown in Figure 1, the setup mini-phase follows concept definition and focuses on defining the working relationship between customer and requirements engineer. The objectives of the setup mini-phase are (a) to introduce the customer to the requirements definition process, (b) to provide the requirements engineer with an overview of the customer’s problem domain and needs, and (c) to help prepare the customer for involvement in the requirements definition process. The following five items outline a set of activities and documents designed to achieve these objectives.

1. *Guidelines*: The customer is given a set of guidelines that introduce the requirements generation process and define the working relationship between the customer and the requirements engineer.
2. *Domain overview*: The customer provides an introduction of his/her work environment and system application domain to the requirements engineer. This is achieved through presentations, guided tours and/or meetings with company personnel. The requirements engineer, who should have some initial knowledge of the domain, acquires additional familiarity with the customer’s problem domain.

3. *Process overview:* The requirements engineer provides the customer with (a) an overview of the software development, (b) a description of the requirements definition process, and (c) instructions on how to prepare material for the upcoming requirements capturing process.
4. *System overview:* The customer provides the requirements engineer with documentation to (a) the current system, and (b) the proposed system, including possible feasibility studies, et cetera. Furthermore the customer presents an overview of what he/she believes to be the more critical or difficult parts of the system.
5. *Q/A session:* The Q/A session provides the customer and requirements engineer with an opportunity to follow up on elements of the setup mini-phase that are still unclear. The customer may have further questions pertaining to parts of the process. The requirements engineer may ask the customer for clarification on items discussed or presented.

At the conclusion of the setup mini-phase the customer is expected to have gained an appreciation of the requirements generation process and knows how to prepare (material and questions) for the subsequent mini-phases. The requirements engineer is expected to have gained familiarity with the application domain of the customer and the needs for the new system. Following the setup mini-phase the customer and requirements engineer should be able to answer “yes” to the following questions:

- 1) Have the rudiments of the problem been communicated?
- 2) Are the fundamental issues understood?
- 3) Do we understand how to prepare for the upcoming interaction session?

*We emphasize that preparing the customer well for the process is the responsibility of the requirements engineer.*

## **3.2 Requirements Capturing**

The requirements capturing mini-phase, as shown in Figure 1, follows setup. The objectives of the requirements capturing mini-phase are (a) to provide a structure within which requirements are accurately identified, and (b) to support the refinement of identified requirements through iteration. How these two objectives are achieved is documented in the following three sub-sections. The discussions reflect a further decomposition of the requirements capturing mini-phase into three working sub-phases: (1) pre-interaction, (2) interaction, and (3) post-interaction. In reaching the overall objective of requirements capturing we describe objectives and tasks associated with each sub-phase. We also explain how refinement of requirements is realized through iteration over the requirements capturing mini-phase.

### **3.2.1 Pre-Interaction**

The pre-interaction sub-phase is initially entered from the setup mini-phase, and subsequently by iteration from the post-interaction mini-phase. The objective of the pre-interaction sub-phase is (a) to examine issues stemming from either the setup or post-interaction mini-phase, (b) to resolve those issues, and (c) to prepare for the interaction meeting. Part (c) is achieved by

documenting solutions to resolved issues and by recording problems and questions on issues that are not resolved.

*Entering from setup:*

Entering pre-interaction from setup occurs only once during the requirements capturing process. Initial tasks to complete include:

- establishing the subject matter (system components) and objective (level of detail) of the upcoming interaction meeting,
- identifying the major participants and their roles,
- conveying to all participants the issues to be addressed during the interaction meeting, and
- ensuring that all participants clearly understand how to prepare for that meeting.

*Entering from post-interaction:*

Entering pre-interaction from post-interaction occurs with each iteration of the requirements capturing mini-phase. The initial tasks to address are:

- establishing the subject matter for the upcoming interaction meeting,
- identifying the reason for iteration, including one or more of:
  - unresolved issues from previously identified requirements
  - incomplete or missing high level requirements, requiring further requirements identification (breadth)
  - lack of detail in previously identified requirements (depth)
- establishing the expected detail at which the material is to be presented, e.g., overview, mid-level, or detailed.
- identifying additional participants (possibly domain experts) and their roles,
- conveying to all participants those issues to be addressed during the interaction meeting, and
- ensuring that all participants clearly understand how to prepare for the meeting.

After completing the initial set of tasks, the participants (usually the customer and requirements engineer) of the upcoming meeting prepare the appropriate material. During this preparation period, participants are required to:

- separately address unresolved issues and document their solutions,
- record unresolved issues that require further investigation during the interaction meeting, and
- indicate their readiness for the interaction sub-phase - Michael Fagan [13] suggests that it is better to postpone an inspection session than to enter it unprepared.



In general, communication between customer and requirements engineer is more focused on: (a) asking simple questions of understanding, and (b) determining meeting times and places.

The overarching goal of the pre-interaction mini-phase is to ensure that each participant carefully prepares for the ensuing discussions so that requirements can be succinctly conveyed during the interaction mini-phase.

### 3.2.2 Interaction

The tasks and problems we have discussed in the previous sections deal primarily with preparation, and making sure that all participants understand what is expected of them. Being well prepared contributes to more successful interaction sessions, but it does not prevent participants from making mistakes. Our experience and research has shown that most errors in communicating and capturing requirements are committed during interaction sessions.

The main objective of the interaction sessions is to enable the requirements engineer to accurately identify and record system requirements as expressed by the customer. In identifying requirements, the problems and associated tasks facing the requirements engineer differ from those of the customer because of the difference in their respective roles: the customer's role is to *convey* information relevant to system development, and the requirement engineer's role is to *capture* information and context accurately. Differing roles and differing session goals require specialized tasks to help answer questions and solve problems. We cannot therefore provide simple guidance for the interaction sub-phase, but instead, must rely on a combination of guidelines, protocols, and a methodology to detect and prevent problems. In the remainder of this "interaction session" discussion we introduce some of the inherent problems addressed by protocols and our proposed methodology.

As outlined in section 2, structure imposition and enforcement is the focus of guidelines, and more specifically, protocols. In this section we discuss four protocols that help provide a defining basis for interaction meetings.

- *Subject of Meeting:* Protocol requires that the subject matter for each interaction meeting be identified before that meeting can occur, and thereby, provides a discussion focus for the customer and requirements engineer.
- *Objective of Meeting:* As discussed in the pre-interaction sub-phase, the meeting objective for every interaction meeting must be established *a priori*. This, too, is mandated by the protocol. During the interaction session the customer and the requirements engineer must work toward meeting that objective.
- *Iteration Reason:* We have previously stated that after the initial entry into the interaction phase, all subsequent meetings stem from the need to "iterate" on some aspect of the problem. Protocol prescribes that one or more of the following motivations guide each subsequent interaction session:
  - (a) unresolved issues – such issues are reflective of requirements being stated incorrectly, incomplete, or incomprehensibly. These unresolved issues are documented and conveyed to all participants. Some, or all, of the unresolved

issues and their solutions are topics for discussion during the next interaction session.

- (b) progressive elaboration – not all requirements have been identified in previous interactions. The set of requirements is still incomplete and does not describe the complete system. Further requirements need to be identified.
  - (c) refinement – requirements have been identified but further detail on these requirements needs to be established.
- *Length of Meeting:* Protocol stipulates that meeting lengths must be kept to comfortable length. The actual length of a meeting should not be more than one to two hours, often depending on the “difficulty level” of the material being discussed. Meetings must be effective and meet their objectives. Fagan [13] [14] and others [5] [15] [16] have discussed that effectiveness can only be achieved if those involved have a fresh mind and are not “tired of” the discussion. Having several shorter meetings of no more than one to two hours each helps one maintain a focus on the subject of the meeting, as time for “unnecessary” discussion is limited.

The items outlined above relate to the model’s structure and are controlled by protocol. By design, they should complement rather than oppose each other. Problem categories that are not part of the structure but are more related to activities are addressed through methodological procedures and methods. We outline some of those categories below.

- *Problem Recognition:* During an interaction session, the requirements engineer, and to a lesser extent the customer, must: (a) recognize when problems occur, (b) be able to categorize the problem, and (c) based on that categorization, identify a solution to the problem. Using the procedures and methods defined in our proposed methodology, the customer and requirements engineer are able to do just that. During the interaction meeting it is important that when problems are encountered, they can be adequately addressed and resolved before proceeding further with requirements capturing. Identified problems not actively addressed *do not* resolve themselves.
- *Constraint Recognition:* All systems have constraints bounding the requirements. Most often these are time, money, and environment. Additional constraints are often imposed by the system, the customer, or the requirements engineer. For example, the requirements engineer might decide to use an Off-the-Shelf (OTS) component before the requirements capturing process is complete. For the sake of argument, suppose that the OTS component requires system inputs based on a pre-defined and “unnatural” format. Unfortunately, the requirements engineer is then forced to write requirements that fit the dictates of the OTS component. Hence, the customer and requirements engineer must recognize when unnecessary and inappropriate constraints are being imposed, and then, work to remove them.
- *Controlled Interrupts:* Our research has shown that dialog interruptions (and disruptions) are some of the more commonly occurring problems, and also the easiest to detect. In our study, the customers and requirements engineers did not seem to be aware of the interruptions. Yet, when analyzing interaction sessions we found a detrimental impact on requirements caused by those interruptions. The adverse impact stems from (a) an unintentional change in discussion topic, (b) early termination of a

detailed discussion, and (c) the speaker forgetting what was going to be said. It is prudent that the customer and requirements engineer have methods at their disposal to assist in recognizing detrimental interrupts and for handling them in a controlled fashion. We call this “controlled stop-and-go”, in which the interrupted person takes active notice of the interruption, and in doing so, takes *intentional* steps to ensure that discussion continues along the same line after the interruption is handled.

We note, however, that not all interruptions have negative impact. For example, interruptions in the form of questions can contribute to improving the discussion by indicating to the speaker that topics need to be better articulated. We therefore believe that *actively* noticing and controlling interruptions has more advantages than disallowing interruptions, which seem to be “natural” to human communications.

- *Explicit/Implicit Requirements:* Explicit requirements are those requirements *stated clearly* during the requirements capturing phase. On the other hand, implicit requirements are those *implied* through parts of the conversation. If correctly capturing explicit requirements is difficult, determining implicit requirements must be a game of luck. A structured communication process that encourages the iterative refinement of requirements will certainly set the stage for identifying explicit *as well as implicit* requirements.

To summarize, from an interaction perspective we must not place constraints on *what* is discussed between participants. Instead, we need to constrain (or structure) the framework within which communications takes place. In doing so we allow communications to occur “naturally” within a well-defined boundary, while meeting the interaction session objectives. We then employ specialized procedures to identify problems when they do occur, and through our methods resolve them.

### 3.2.3 Post-Interaction

The post-interaction sub-phase follows the interaction sub-phase (see Figure 1) and is the last sub-phase in requirements capturing. The objectives of the post-interaction sub-phase are: (a) to determine if the objectives of the immediately preceding interaction session have been met, and (b) to check if all exit criteria have been met to enter the transformation mini-phase (section 3.3).

To determine if the objectives of the interaction session have been met, the customer and requirements engineer review the artifacts (USRL and context information) from previous interaction sessions. More specifically, the artifacts are examined to ensure that:

- (1) the subject matter identified and the focus of the preceding interaction sub-phase have been adequately addressed, and
- (2) the results of that interaction session are documented at the appropriate level of detail.

During that examination, deviations from these objectives are documented as *unresolved issues*. The severity of those deviations is assessed and a decision is made either to re-iterate on those issues or to move towards the transformation mini-phase.

If the artifacts meet the above two objectives, then they have met the first of several exit criteria for the post-interaction sub-phase. The remaining exit criteria include:

- checking the USRL and related context information for completeness,
- determining if the requirements are well-stated,
- determining if the requirements are understood clearly by the customer – this includes an interaction session during which the customer reads the requirements and gives his/her verbal interpretation of them to the requirements engineer, and finally
- *ensuring that the requirements, without exception, meet the customer's intent.*

When all exit criteria are sufficiently met the transformation mini-phase is entered.

### 3.3 Transformation

As shown in Figure 1, the transformation mini-phase follows successful completion of the requirements capturing mini-phase. Furthermore, it is the last mini-phase that is part of the refined partitioning of the requirements definition phase within our framework, and precedes the conventional requirements analysis phase. The objective of the transformation mini-phase is to transform the USRL and related contextual information into an SRS. This objective is met by a two-step approach: (1) structuring requirements of USRL into an SRS based on the contextual information, and (2) determining if the SRS is complete. In the following section we describe the details of each of these two steps.

The task of the first step is to take the requirements of the USRL and the contextual information captured during requirements identification and create an SRS. This requires that the requirements engineer identifies functional-related inter-relationships among requirements and group them according to that functional cohesiveness.

In the second step of this approach the requirements engineer must check for completeness of the newly created “structured requirements specification.” The following questions are used to guide completeness assessment:

1. For each functional group of requirements, has the source of its inputs and recipient of its output been identified?
2. Does the SRS give the complete “picture” of the system and are all parts of the SRS well understood?

During any part of the transformation process, if the requirements engineer discovers any problems, then based on the severity of those problems, the requirements engineer can

- (a) request an extra iteration through the requirements capturing mini-phase (severe error),  
or
- (b) resolves the problems in a less formal manner, e.g., a short meeting with the customer (minor error).

## 4. Methodology

The previous sections of this paper present the proposed framework, and describe the mini-phases and activities within this framework. As part of this discussion, we have introduced, without elaboration, the notion of procedures and methods. The rationale behind the procedures and methods is to direct the requirements generation process and assist in problem identification and resolution. These procedures and methods, as well as protocols, comprise our proposed methodology. In this section we present the objectives of the methodology and show how we achieve those objectives through the use of protocols, procedures and methods.

The objectives of the methodology are to enforce the structure needed for the requirements generation process, and to guide the customer and requirements engineer in problem identification and resolution. To achieve these objectives we have designed a two-prong approach:

- (1) define the structure (and constraints) through protocols, within which the customer and requirements engineer must interact, and
- (2) identify and resolve problems using procedures and methods, respectively.

*Analogous to medicinal practice, our methodology provides one set of “preventive” measures and another set of “curative” ones.*

### 4.1 Structure through Protocols

For the most part interaction between the customer and requirements engineer transpires without problems and with positive results. There are instances, however, where such interactions lead to situations that have the potential to adversely impact requirements generation. To minimize the possibility of problems occurring, our methodology includes protocols designed to “constrain” the interaction latitude provided to the customer and requirements engineer. In actuality, it is the protocols that define the framework’s structure and boundaries to which one must adhere. Remaining within the boundaries of that structure helps focus the customer and requirements engineer on the tasks they must perform, which in turn, helps achieve the objectives of the requirements generation process.

Examples of structure imposed by protocols are:

- (a) participant role identification – Protocol mandates that before any meeting occurs, all participants and their respective roles be identified. Role identification includes:
  - establishing the responsibilities and tasks of each participant during the interaction session (e.g., leader, recorder, presenter, etc.),
  - ensuring that a single person is designated as “in charge”,
  - assigning multiple roles to one participant during one interaction session (e.g. the leader is also the presenter), and
  - switching roles between participants in different interaction sessions.

(b) identification of subject matter and objective of meeting – Protocol also dictates that the subject matter and meeting objective must be identified and conveyed to all participants. Adherence to this protocol:

- minimizes the occurrence of ineffective, “ad-hoc” meetings, and
- sets preparation expectations for all session participants.

## 4.2 Problem Identification and Resolution

Because of the complexity of the interaction during the requirements generation process we cannot hope (or even expect) to provide a constraining structure that addresses all potential problems. As an initial step we have ascertained representative procedures and methods that help identify and resolve problems that still occur within the constraints of the protocol-imposed structure.

### *Problem Identification*

Our experience and research have shown that, even within a structured environment, problems can still surface and, before being recognized as such, have a detrimental impact on the requirements generation process. Hence, enabling the recognition and identification of problems *as they occur* is an important part of our methodology. We achieve this by providing the customer and requirements engineer with procedures that help to identify problems quickly. The procedures reveal problems through sets of questions that, as part of the process, should be asked periodically. Examples of such questions include:

- Have “unnatural” constraints been placed on requirements?
- Are participant interruptions having an adverse impact on requirements?

Associated with each question are a set of indicators that enable one to accurately measure and answer the question.

### *Problem Resolution*

Problem identification is only the first step in improving the requirements generation process; the second step is problem resolution. Based on problem characteristics, procedures direct the customer and/or requirements engineer toward one or more methods that help resolve the identified problem. These methods consist of specific tasks enabling the resolution of problems. For example the “pre-conceived mindset” presents a problem in which either the customer or requirements engineer unduly constrains the requirements generation process by focusing exclusively on a single “Solution” approach. Using procedures and methods we identify and resolve this problem by:

- 1) Asking the question, “Are pre-conceived ideas constraining requirements generation?”
- 2) Recognizing the indicator: *Requirements are reflecting design characteristics*, and
- 3) Applying the Method: *Enforce an initial identification of multiple alternatives to solving a problem.*

## 5. Summary and Work in Progress

A successful software development effort is one that produces a quality product within budget and on time. An integral part of such an effort is a set of well-defined processes that structure and guide development activities within an appropriate framework. Because product quality is a direct reflection of how well it meets the customer's needs, the ability to capture customer requirements accurately and succinctly is paramount.

In this paper we have presented a refined model of the requirements generation process that promotes the accurate capturing of requirements. The major components of that model are:

- (a) a framework reflecting a refinement (or decomposition) of the requirements generation phase into mini-phases – each mini-phase has its own unique set of objectives and activities associated with it,
- (b) guidelines and protocols designed to structure the interaction process and to guide the customer and requirements engineer in their discussions, and
- (c) a methodology consisting of protocols, procedures and methods to address potential problems and to identify and resolve existing problems.

### *Work in Progress*

The work identified in this paper is “work in progress.” What is being proposed is a set of ideas and concepts that are constantly evolving. Clearly much work is still remaining. Our current effort is focused on:

- (1) developing a better understanding and details of the framework, guidelines and protocols,
- (2) identifying additional problem categories and refining existing ones,
- (3) continuing to develop the methodology, i.e., the procedures and methods,
- (4) determining how to
  - develop an adequate USRL and the contextual information, and
  - form functional grouping based on functional cohesiveness, and
- (5) identifying steps to transform the USRL and associated contextual information into an SRS.

Finally, we need to apply our proposed model to an actual requirements engineering effort to substantiate and/or validate our approach.

## References

- [1] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1991.
- [2] W. Perry, *Effective methods for software testing*. New York, NY: John Wiley & Sons, Inc., 1995.
- [3] J. L. Connel and L. B. Shafer, *Structured Rapid Prototyping: An evolutionary approach to software development*. Englewood Cliffs, New Jersey: Yourdon Press, Prentice Hall Building, 1989.
- [4] I. Sommerville, *Software Engineering*, 5th ed. Reading, Mass.: Addison-Wesley Publishing Co., 1996.
- [5] D. P. Freedman and G. M. Weinberg, *Handbook of walkthroughs, inspections, and technical reviews: evaluating programs, projects, and products*, 3rd ed. New York, NY: Dorset House Publishing Co., Inc., 1990.
- [6] E. Bravo, "The hazards of leaving out the users," in *Participatory Design: Principles and Practices*, D. Schuler and A. Namioka, Eds. Hillsdale, New Jersey: Lawrence Erlbaum Associates, Inc. Publishers, 1993, pp. 3-11.
- [7] J. Greenbaum, "A Design of One's Own: Towards Participatory Design in the United States," in *Participatory Design*, D. Schuler and A. Namioka, Eds. Hillsdale, New Jersey: Lawrence Erlbaum Associates, Inc. Publishers, 1993, pp. 27-37.
- [8] P. Mambrey, R. Oppermann, and A. Tepper, *Computer und Partizipation: Ergebnisse zu Gestaltungs- und Handlungspotentialen*. Opladen: Westdeutscher Verlag GmbH, 1986.
- [9] E. Carmel, R. D. Whitaker, and J. F. George, "PD and Joint Application Design: A Transatlantic Comparison," *Communications of the ACM*, vol. 36, pp. 40-47, 1993.
- [10] P. Mambrey and B. Schmidt-Belz, "Systems Designers and Users: Fictions and Facts," in *Systems Design for, with, and by the user*, U. Briefs, C. Ciborra, and L. Schneider, Eds.: North-Holland Publishing Company, 1983, pp. 61-69.
- [11] A. M. Davis, *Software Requirements: Objects, Functions, and States*, Revision ed. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1993.
- [12] R. E. Nance, "The Conical Methodology: A framework for simulation model development," presented at Methodology and Validation, Orlando, Florida, 1987.
- [13] M. E. Fagan, "Design and Code Inspection to Reduce Errors In Program Development," *IBM Systems Journal*, vol. 15, 1976.



- [14] M. E. Fagan, "Inspection Software Design and Code," *Datamation*, 1977.
- [15] R. G. Ebenau and S. H. Strauss, *Software Inspection Process*. New York, NY: McGraw Hill, 1993.
- [16] R. O. Lewis, *Independent verification and validation: a life cycle engineering process for quality software*. New York, NY: John Wiley & Sons, Inc., 1992.

# Requirements for Test Automation

**Douglas Hoffman**

Software Quality Methods, LLC.  
24646 Heather Heights Place  
Saratoga, California 95070-9710  
Phone 408-741-4830  
Fax 408-867-4550  
doug.hoffman@acm.org

## Abstract

Automating testing is like any software development project, and as such we need to articulate the requirements. Test automation should not be done piecemeal or in an ad hoc fashion because the resulting work products will become more and more expensive to use, eventually becoming obsolete or unsupportable. The presentation describes many special considerations for automation requirements and a method for understanding and organizing them.

## 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 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), a 750 member Task Group of the ASQ. 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. He is also a member of the ACM and IEEE. Doug 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 a MS in Electrical Engineering and BA in Computer Science.

Although his current focus is in software test automation, his experience includes extensive consulting, teaching, managing, and engineering in the computer and software industries. He has fifteen years experience in creating and transforming software quality and development groups, and twenty years of management experience. His work in corporate, quality assurance, development, manufacturing, and support organizations makes him very well versed in technical and managerial issues in the computer industry.

---

# Requirements for Test Automation

PNSQC '99

Douglas Hoffman  
Software Quality Methods, LLC.  
24646 Heather Heights Place  
Saratoga, California 95070-9710  
Phone 408-741-4830  
Fax 408-867-4550

Copyright © 1999, Software Quality Methods, LLC. No part of these graphic overhead slides may be reproduced, or used in any form by any electronic or mechanical duplication, or stored in a computer system, without written permission of the author.

Douglas Hoffman

Copyright © 1999, SQM, LLC.

1

---

## Requirements Considerations

- Software Under Test
- Tools and environment
- Test management
- Organizational situation
- Automation architecture

---

Douglas Hoffman

Copyright © 1999, SQM, LLC.

2

---

## Automated Software Tests

- Able to run two or more specified test cases
- Able to run a subset of all the automated test cases
- No intervention needed after launching tests
- Automatically sets-up and/or records relevant test environment
- Runs test cases
- Captures relevant results
- Compares actual with expected results
- Reports analysis of pass/fail

---

## Levels of Automation

- Fully automated software testing
- Semi-automated software testing
- Manual software testing

---

## Questions About Requirements<sup>†</sup>

- Is the product stable?
- To what extent are oracles available?
- Who is the client for test information?
- Will the product have through multiple releases?
- How good is the source control management?
- Do you need requirements traceability?
- How capable are the current testers?
- How cooperative are the programmers?

<sup>†</sup> Kaner, C. (1998) "Avoiding Shelfware: A Manager's View of Automated GUI Testing"

Douglas Hoffman

Copyright © 1999, SQM, LLC.

5

---

## Product Considerations

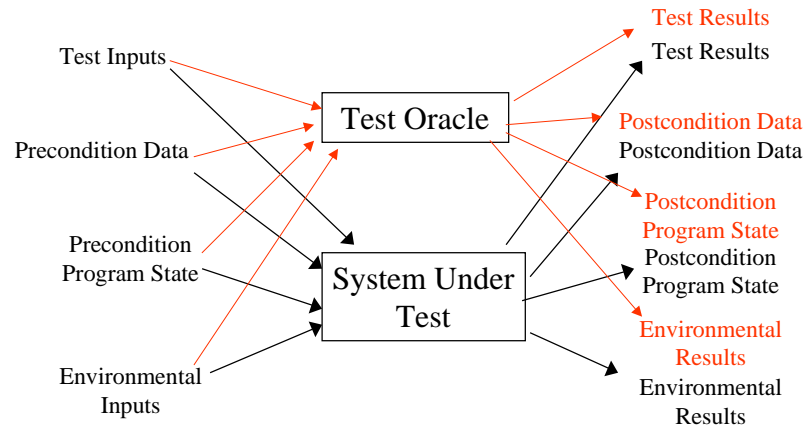
- Components of SUT
- Access points for inputs and results
- Important SUT features and capabilities
- SUT environments
- Test data elements
- Oracle availability

Douglas Hoffman

Copyright © 1999, SQM, LLC.

6

## Testing With An Oracle



Douglas Hoffman

Copyright © 1999, SQM, LLC.

7

## Automation Requirements

- What are your wants and needs?
- Where is automation practical?
- Where does automation pay off?
- What are the expected benefits?
- Criteria for make / buy decision?

Douglas Hoffman

Copyright © 1999, SQM, LLC.

8

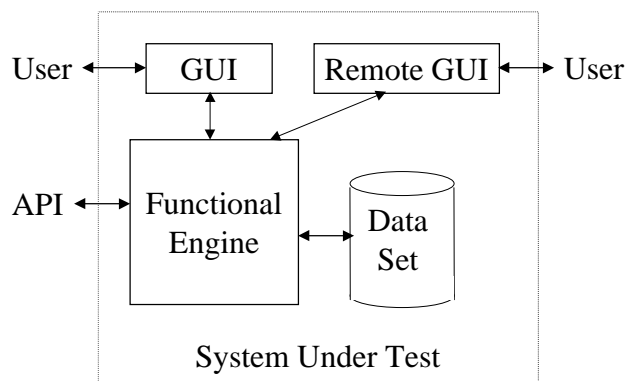
---

## Automation Architecture

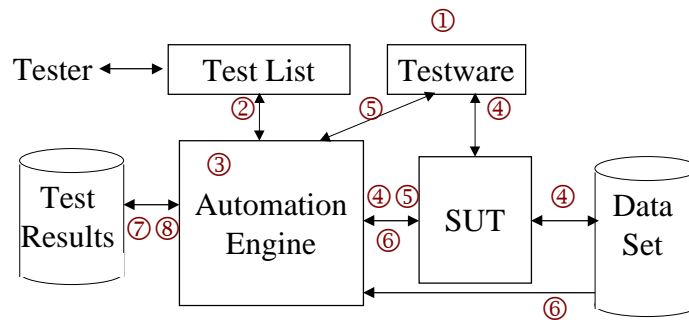
- Model for SUT and environment
- Break down software testing problem
- Decide on location(s) of automation
- Decide on level(s) of automation
- Describe automation architecture

---

## A Model For SUT



## Automated Software Testing Process Model



## Process Model Example

1. Testware creation, version control, and configuration management
2. Selecting the subset of test cases to run
3. Set-up and/or record environmental variables
4. Run the test exercises
5. Monitor test activities
6. Capture relevant results
7. Compare actual with expected results
8. Report analysis of pass/fail



---

## Factors in Test Tool Selection<sup>†</sup>

- Capability
- Reliability
- Capacity
- Learnability
- Operability
- Performance
- Compatibility
- Non-Intrusiveness

<sup>†</sup> Bach, J. "Test Automation Snake Oil", 1999. Published in Windows Tech Journal, 10/96, and proceedings of the 14<sup>th</sup> International Conference and Exposition on Testing Computer Software

Douglas Hoffman

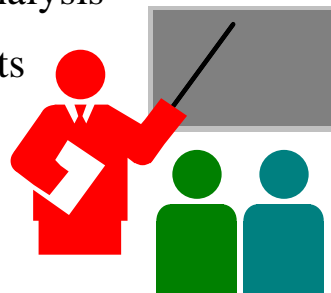
Copyright © 1999, SQM, LLC.

13

---

## Summary

- Requirements begin with analysis
- Identify critical requirements
- Model the SUT
- Model the testing process
- Analyze available tools



Douglas Hoffman

Copyright © 1999, SQM, LLC.

14

---

## References

- Bach, J. “Test Automation Snake Oil”, 1999. Published in Windows Tech Journal, 10/96, and proceedings of the 14<sup>th</sup> *International Conference and Exposition on Testing Computer Software*
- Hoffman, D. (1999) “Test Automation Architectures: Planning for Test Automation,” *Quality Week 1999*
- Kaner, C. (1997) “Improving the Maintainability of Automated Test Suites,” *Quality Week 1997*
- Kaner, C. (1998) “Avoiding Shelfware: A Manager’s View of Automated GUI Testing”

# Ensuring Outsourced Software Success

*By Daniel Clemens*

## **Abstract:**

Approaching the outsourcing process from a customer's point of view, this paper addresses several of the major points that a customer should consider when attempting to outsource a software project.

Topics covered include: finding an outsourcing partner, evaluating the partner, risk management, aligning the motivations of each party, what to do when things go wrong, establishing verifiable milestones, contract issues and more. There are examples from real world projects, along with clear-cut advice on areas where emphasis in the relationship could pay big dividends for a more successful project and partnership.

## **Author's Bio:**

Daniel Clemens was the founder of MicroCrafts and the Company's president for nearly 10 years. He has been an entrepreneur for more than 20 years, and has owned and operated a number of engineering consulting businesses since 1981. Mr. Clemens has been designing hardware since 1978 and has been writing or managing software since 1980. He has operated at all levels of the software development process from front line engineer to lead engineer, to project manager, to the manager of an engineering company. He brings a practical, very non-theoretical perspective to his approach to engineering. Additionally, Mr. Clemens has participated at the board level in several other high tech companies.

Mr. Clemens led MicroCrafts through an exponential growth period of four years, during which time the company grew from the original two employees to more than 85 employees. MicroCrafts received numerous fast growth awards, among them making the Inc 500 list in 1998. MicroCrafts has helped its customers achieve success through solid engineering services for more than a decade.

8700 148<sup>th</sup> Avenue NE, Redmond, Washington 98052

# Ensuring Outsourced Software Success

*Author: Daniel Clemens*

## What is Software Outsourcing

When people ask me what I do for a living, I sometimes find it difficult to explain to the “non-nerd.”

My company provides outsourced software development for other companies. Some folks make the assumption that this is essentially the “body shop” model, i.e., consulting, contracting, where a company needs a software developer and our firm provides that developer. The body shop model is also analogous to a “temp agency.” It boils down to a firm providing a worker for your use on a temporary basis - a worker that you must supervise and direct. Our firm, and firms like ours, provide a service to our customers. The service is the ability to independently develop software and deliver a finished project without the need for supervision or close guidance by the customer.

There is a large difference in the two models. The body shop approach places all the management requirements on the customer, including managing the individual developer(s) and project, developing and tracking the process, etc. The outsourced approach places the responsibility on the software development company to deliver the final product to the customer. The final product may be just a component of a larger project; a deliverable is identified and described, and the outsourcer is responsible for that deliverable.

## Why Outsource

The outsourcing decision is one that should be made as early in the product development cycle as possible. Identifying an outsourcing partner, working through the project details, and just getting the project started usually takes longer than expected.

If the partner you find is competent, the arrangement structured correctly, and the motivations properly aligned, outsourcing can be extremely beneficial for both parties. However, it can also be a complete disaster if the relationship and the partnership are not tended to properly.

Following are some highlights of why you might outsource a project:

- **Cost Effectiveness**

Executed correctly, an outsourced project can actually cost *less* than doing it in-house. Sometimes the political and bureaucratic processes at your company may cost much more than finding a firm that has domain knowledge and expertise, and motivating them to “crank it out.”

Another large concern in the cost decision is time-to-market costs. Often the cost of missing a market window may far outweigh the costs of outsourcing the project to an expert firm. An example of this would be a firm that sells children’s software. The winter holiday season has the largest sales volume of the entire year for this type of software. The company must have its product “in the channel” by mid-October in order to be on the shelves in time for the holiday crunch. The cost of outsourcing may pale in comparison to the loss in revenue if they were to miss the mid-October deadline.

- **Predictability**

If chosen correctly, the outsourcing firm may have a much higher degree of predictability than an internal development source. The outsourcing firm is usually free from the “feature creep” associated with internal projects because they usually are working within a financially limited contract. In other words, if you’ve structured the deal correctly, they don’t get paid to bloat the project. Generally, internal development sources don’t have the same motivations and are usually harder to control in this sense.

It’s important to find an outsourcing firm that has a track record for accurately predicting their work. Later in this paper, I’ll show you some ways of determining an outsourcing firm’s track record for predicting projects.

- **Project Management**

Presuming the outsourcing firm has a successful track record, they will (by necessity) be good at managing projects. Project management is critical to any project that is larger than one developer. Some single developers are not able to manage themselves and require project managers.

- **Resources**

Your firm may not have enough resources or may lack the specific domain expertise required to develop a project. This is an excellent reason to outsource. Plus, the developer may have an experienced team of people who have worked together and have a proven track record.

### **What NOT to Outsource**

Generally, it is felt that you should not outsource what I call your company's "secret sauce." The prospect of losing control of the secret sauce may put your company at a level of risk that is unacceptable. For example:

Your firm sells software to architects. Assuming this is all that your firm does, you would probably be wise to build an internal development staff for your company, if you intend to have a long sales life of a particular product. It may be very advantageous for you to outsource the initial development and transfer the "core" technology to an internal staff. The main concern in this case is that an outside firm essentially controls the chief sources of your firm's revenues.

There is likely to be a series of peripheral work that can easily be done by an outsourcing firm, which will not compromise the secret sauce. Identify these components early, and find a partner who can help you develop them.

### **Partnership**

Both parties should view the resulting work (performed by the developer) as a partnership. Usually, the developer is hoping to have follow-on work with the customer, and the customer hopefully is getting great value for their investment. Unless there are unusual circumstances, the relationship should continue.

Another important consideration with respect to partnership is that when the project is struggling, it is much easier to ask a "partner" to go the extra mile than to ask a "vendor" to provide extra services. Although the distinction may seem insignificant, the connotation of a partnership is that the parties are dependent on each other for the success of their mutual projects; this is important and should be valued by both parties. It takes two parties to have a partnership. You should make certain the developer is desirous of having a true partnership. If the developer views the relationship as one of vendor-customer, you should continue looking for a different developer.

### **Selecting a Partner**

#### **Finding a Partner**

As it turns out, finding a partner can be a little harder than it may seem. Generally, developers are good at developing software, but may not be great at marketing their services. Furthermore, good developers usually have more than enough work to keep them busy and have little need to market their services. This leads to a minor problem for new customers who are searching for a development partner.

The larger development firms will have marketing efforts in place and, therefore, will be easier to find. By larger, I am talking about 40+ person firms. There are many very capable and reputable firms with fewer than 40 people that may have exactly the specialized skills or expertise that your firm needs for a project.

Hands down, the best way to find a development partner is via reference. Some of the better methods of networking to obtain a reference include:

- If you're part of a large firm, you may have access to recommended vendor lists. Seek them out and follow up with reference checking.

- You probably have “friends” in the industry. Colleagues who have moved on to other jobs or folks you’ve worked with at your previous jobs are great targets.
- There are usually software industry associations in your state or local area. Find one; attend a meeting and talk to the folks there about who they think may be best suited for your type of project.

If you’re part of a smaller firm, check with your attorney, accountant, web site manager, and other people who provide you with services. Chances are that they know of someone who has had an experience with a software outsourcing firm.

There are also some terrific resources on the Internet. Some of the better sites that can provide information and links to the web sites of outsourcing firms are:

The Software Contractors Guild, <http://www.scguild.com/>  
 The Outsourcing Institute, <http://www.outsourcing.com/index.htm>  
 Software & Information Industry Association (formerly SPA), <http://www.siiia.net/>  
 Software Quality Engineering (SQE), <http://www.sqe.com>  
 Software Association of Oregon, <http://www.sao.org/index.htm>  
 Washington Software Alliance, <http://www.wsa1.org/>

Lastly, you can get references or even find partners through attending industry events such as this one (PNSQC). Conferences, seminars and trade shows provide great opportunities to network with people, allowing you to gain information about possible companies to work with. Presenters (such as myself), attendees, and exhibitors can provide you with names of people and companies that may be able to help you. As an example:

Metamor Software Solutions is a company composed of six development centers around the country including Boston, Rochester (NY), Seattle, San Jose, Salt Lake City and San Diego. They have more than 350 people dedicated to providing high quality, software engineering services. The company’s areas of expertise are very diverse; it is very likely that they have the resources to help you deliver your project with good quality, on time and on budget. If they can’t do the project, they would probably be able to help you find the right resource to do it for you.

You just gained at least two references: myself, Metamor Software Solutions, and the additional references that they could supply you with if needed.

## **Top 10 Things to Consider**

### **1. Reputation**

Next to the ability to develop software, a development firm’s reputation is probably their most valuable asset. They should be proud of their reputation, their track record as a company, and their body of work. Further, you should be able to actually see and examine some of their work. Ask them for a list of some of the software they have developed and if possible, acquire some of it and give it the once over.

### **2. Demonstrated Capability**

Do they have the people necessary to complete your project? Do those people have the required skill sets? Are they available? When will the right people be available?

### **3. Approach to Business**

A developer’s approach to business is a bit more difficult to ascertain than it might seem. As you might imagine, there are many different ways to conduct a business and maintain the relationships that come with it. You should look at the history of the developer’s customers. Do they have repeat business?

Ask the developer what their approach to business is. If they describe a relationship building and partnership fostering arrangement, they probably have the right approach.

Ask them what sort of work they *like* to do. Ask them how your project fits with the type of work they like. If there isn't a good crossover between your project and the work they are pursuing, ask them how that will affect their ability to complete the work. Carefully consider the developers who take work that doesn't fit the model of what they are pursuing.

#### **4. Past Performance**

If you've read any mutual fund prospectus, you are probably familiar with the phrase "Past performance doesn't indicate future success." In the development business, I think past performance is a pretty sound indicator of whether or not a developer has a fighting chance of doing the right thing on your project.

Ask the developer to describe their company's past successes. What made the successes successful? What could have been done better?

Perhaps a great question to ask the developer is "What criteria do you use to measure the success of your projects?" The proper answer here is something close to "on time delivery," "on or under budget," "high quality and a happy customer," and so on.

Probably as important as the successes are any failures or challenges the developer has experienced. Ask them to describe some projects that have not gone as well as they had hoped. What went wrong? What was the final outcome of the project? Do they still have a relationship with the customer? What did they learn from the experience?

#### **5. References**

Ask for and check with at least three references. You should be particularly interested in references that have utilized the developer for similar projects as yours. If possible, ask the developer for references by firms that are similar in size to yours. Additionally, if possible, ask the developer for references of companies that have had experience with the proposed team or personnel who may be working on your project.

There are situations where developers have done work for companies under a non-disclosure agreement (NDA). Some of these NDA's are strict enough to prohibit the disclosure of the relationship between the customer and the developer. While these NDA's should be respected, it is often possible for the developer to call a representative of the company and ask them for permission to have you speak to them about their experience with the developer. Be wary of developers who readily disclose what you would consider confidential information or too easily stretch the limits of an NDA. If they are willing to do it with someone else's confidential information, they may be just as willing to do it with yours.

When you get the reference list together, it would be helpful for you to write a short script to interview each reference with. This will make your questions consistent with all the references and make it easier for you to avoid skipping around.

It is important to determine your mindset prior to doing the reference checking. Have you already made up your mind that you are going to use the developer? If so, then your reference checks will most likely be a "rubber stamp" and you might consider saving yourself the time and effort of conducting them. It's best to have the mindset of "Let's learn more about this developer and how they maintain partnerships." and make the reference calls *before* you have made any decisions.

Checking references on a company is very similar to checking references on an individual when they are applying for employment. Here are a few questions you might consider:

- \* What are they like to work with?
- \* Are they reliable?
- \* Did you enjoy working with them?

- \* Would you use them in the future for another project?
- \* How many projects have you used them on?
- \* What personnel did you work with?
- \* What was the best part of working with them?
- \* What do they need to improve on?
- \* Did you have any surprises? If so, how were they handled?
- \* Did your project meet your quality expectations?
- \* Could you describe the project they developed for you?

Some questions to avoid:

- \* What rates did they charge you?
- \* What were their weak areas during contract negotiations?

In general, avoid the specifics about the financial dealings.

## 6. Metrics and Processes

Without some methodical process in place to measure, predict and monitor the development process, the developer will have little hope of meeting the goal of delivering the project. Ask the developer to describe the process to you. Make sure you understand exactly how they transition from one stage to another. Where do they think the weak spots are? How long have they been using this process? Is it an externally approved process (CMM or ISO) or an internal “home brew”?

I will risk being pummeled by the software development purists here. My opinion is that the exact process is not as important as having a process that has a demonstrated successful track record. To illustrate this, I’ll pose a question: If developer A had just implemented CMM Level 3 (a predictable process and probably audited by a third party) and developer B had been using an internal process for the last 10 years that has been consistently delivering projects on time and on budget, which would you feel most comfortable with?

Developer A has the challenge of all their people being trained on a new process; there are bound to be potential problems. Developer B has a great deal of corporate memory and inertia associated with their process. All other things being equal, I would choose Developer B for the project.

Again, the issue here is having a predictable, repeatable process in place that the developer (and you) can rely upon. The buzzwords and nomenclature surrounding the process, while important for communication, is of less importance.

Some things critical to the process you should consider:

- \* How do they measure progress?
- \* How do they verify the progress is actually progress? (See “Living the Lie”)
- \* What mechanisms do they use to establish milestones?
- \* Are there standard documents for each phase of the project?
- \* What tools do they use to facilitate their process?
- \* What do they think are the weaknesses of their process?
- \* When did they implement this process?
- \* What are the successes of their process? Failures?
- \* How do they train their people on their process?

## 7. Resources

In today’s market for software engineers, perhaps the \$64,000 question is “Does the developer have the personnel necessary to complete your project?” Keep in mind that if the developer is successful, their people resources should be in demand and their time commitments may be tightly scheduled. When you ask the developer if they have the right people available, they should be able to tell you a window of time when the personnel will be available for your project.



A key issue in this area is the experience level of the personnel working on your project. You need to know how long they have been with the firm; are they actually employees of the firm; have they worked together on other projects, and so on. Ask the developer about the personnel and ask to meet them if possible. If you're uncomfortable with the personnel, ask the developer to change them if possible. These people are the main vehicle for delivering your project. It is likely that you will be working with them for quite some time in the future. (You should enjoy working with them if possible.)

Beware of the developer who will use a "bait and switch" tactic. They will parade out the superstars to woo your project only to put the 3rd string benchwarmers on your project. Ask the developer to commit early as to who the key players will be. Let the developer know that you understand that they may not be able to commit the entire team because of conflicting schedules; but you want to know who the key players are and would like them committed early. Keep in mind, the delays in actually starting your project could have a very negative impact on the developer's ability to staff the project with the originally committed people.

For example, say you inquire in January about a 6-month project for three people. You tell the developer you want to start in March, and you interview the key people in February. For some reason, your company delays the actual start of the project until May. The developer will likely have moved the key people to another project by then because they would prefer to have those people working on projects as opposed to waiting. Be realistic about your time commitments and let the developer know how you are progressing in this area so they can accommodate your needs better.

#### 8. **Size and Stability**

Of all the questions I've been asked by customers, potential customers, consulting clients and business people (regarding picking a developer), size and stability is definitely a tough one. The question usually takes the form of "Is a larger firm better than a smaller firm?" or "They've only been in business for two years. Is that long enough?"

The short answer to nearly all of these questions is: it depends. Yes, size does matter in this case. And, yes, there are many advantages to a larger firm and ones that have been in business for many years. Here are some of the "party line" bullet items you'll hear and see in the brochures:

- \* We can put more people on the project when necessary.
- \* We can move people on and off the project as needed.
- \* Our size reduces risk to the project.
- \* We've been in business for 185 years.
- \* Our firm has a proven track record. And so on.

While all valid, these statements don't insure that your project will be any more successful with a larger firm than with a smaller one. My bias in this area is to pursue a firm with the passion and interest to work on your project like it was the best project they had worked on in a long time.

Some advantages to smaller firms are:

- \* Your project may be the most important project in the company.
- \* It is likely that a principal of the firm will be directly involved with your project.
- \* The best people in the company may be working on your project.
- \* You may be the most important customer to the firm.

As you might imagine, there are some disadvantages to smaller firms. Among them:

- \* Risks of your project requiring the entire company's resources to deliver.
- \* Risks of having too few customers. If another customer causes trouble for the firm, they

- may be unable to deliver on your project.
- \* Personnel choices will be dramatically reduced.

The perfect answer for your firm and your project will depend on your comfort level. In today's market for software developers and if your project is very bleeding edge, a small, newly started firm may be the best bet for your project. On the other hand, if your project is based on stable technology and requires personnel who don't have extremely specialized skill sets, a larger more stable firm may be the best choice for you. You need to do your homework and make sure the firm matches the other criteria discussed here along with the size of the firm.

#### 9. **Technical Expertise**

Does the developer have the actual expertise necessary to complete your project? Does the type of expertise needed for your project actually exist anywhere at this time? Is the developer going to be *learning on your nickel*?

The developer should be very straightforward with you about the issue of technical expertise. You should be very direct about asking questions regarding what type of technology is required to deliver your project and the direct experience of the developer in this area. As with other issues, the actual personnel working on your project should be evaluated for the technical expertise necessary, not just the developer's overall expertise.

There are generally two types of expertise: actual experience implementing the exact type of project you are requesting and reasonable domain knowledge about the subject matter. Let's say your project is to develop a printer driver for your firm's new color printer. In order to exploit your printer to the fullest, the driver must have some heavy-duty algorithms developed. Clearly, it is unlikely that the exact experience with your new printer exists because you just developed it; but you should easily find a company with some serious domain knowledge in the development of printer drivers.

Keep in mind that there may be no domain experts and no direct experience implementing something similar to your project. An example might be a new type of widget for an unreleased operating system. It is unlikely you'll find domain experts for your new widgets, and the experience level for the new operating system will be very low throughout the industry. In these cases, you will need to rely on the developer's demonstrated track record of helping early adopters of technology to be successful. Inquire about projects that fall into these categories and be diligent about following up on the references.

#### 10. **Pricing Structure**

How does the developer price their services? You might be surprised to find out how this works. You might be even more surprised when you try to compare two or more developers' bids on a project.

There are many ways to price a project. Many of the pricing structures rely on the type of engagement, i.e., fixed bid, time and materials, etc. Ask the developer to provide you with a standard time and materials rate sheet early in the process of learning about them. The rates may be flat for all personnel or they utilize a ladder of rates for different skill sets or personnel levels. If they use the latter, ask them to give you a rough breakdown of the skills required for the various levels. You will need this information to compare developers' bids.

It is also important to obtain the hourly rate sheets for the personnel, even if you are engaging in a fixed bid contract. There will likely be components of the work that will be done under time and materials arrangements, such as, change orders or additional unanticipated work. Further, it will be helpful to have the rate sheet and pricing structure to double check the method of fixed bidding the project.

## **Requesting a Bid or Proposal**

Prior to engaging the developer on any project, you should request and carefully review at least one proposal prepared for your project. Your firm sometimes controls the process of soliciting bids. (There are plenty of books on the subject.) It is not my intention to tell you how to solicit and compare bids, but rather to tell you what to ask for in the bid and hopefully to help you evaluate the results.

A point of reference here should be the quality of information you provide to the developer prior to the proposal request. This will also give you a good idea of the developer's willingness to work on your project. If the information you provide is sketchy (e.g. table napkin drawing and lots of arm waving), you should expect equally sketchy estimates and plenty of protection for the developer in the resulting proposal. The best situation for your firm, the developer, and the relationship you're building is to disclose everything you know about the project. Be open and forthright with the developer about the risks you see, the potential problems, potential delays in any interlocked work and so on. These issues will effect the proposal and the bid, and will come into play at some time.

By providing the developer with all the information early, you will have the best shot at getting accurate information. If you have more than one developer in the bid process, it is critical that you provide substantially similar information to all of them. It should be obvious that if developer A knows that the hardware for the project will be delayed by three months and developer B is not privy to that information, it will be very difficult to equally evaluate the resulting proposals.

When you request a proposal from a developer, you should always request a time and materials (T&M) estimate *and* a fixed priced estimate. The developer may resist preparing the proposals this way, but there are several good reasons for taking this approach, including:

- **Risk Assessment**

By examining the differences between a fixed bid and a T&M bid, you will be able to ascertain what the *developer* perceives are the risks in the project. Remember the main difference between a T&M and fixed bid project is the assumption of risk by the parties. In a T&M project, the bulk of the risk is borne by the customer. While in a fixed bid project, the risk is primarily borne by the developer.

By getting both a T&M bid and a fixed bid, you will be able to see the delta between what the developer believes is a straight hourly rate to produce the project and what they perceive as a reasonable rate to assume the risk of the project going poorly.

There should definitely be a spread between the two bids. The fixed bid could be as much as two times the T&M bid and still be within reason. If the fixed bid and the T&M bids are extremely close it should indicate to you that either the developer is extremely confident that the work can be completed with minimal risk or the developer is unaware of the risks. The former is good news; the latter is a disaster waiting to happen. Possibly the only way to determine which it is would be to speak frankly to the developer after looking at the bids.

- **Determine Approach**

The developer may believe there are better ways to "skin the cat" than the straight T&M arrangement. By requesting the fixed bid, you provide them the opportunity to provide you with a new viewpoint on delivering the project. In the T&M model, the motivations are somewhat misaligned. On the surface, the developer is not motivated to finish the project in a timely fashion, but rather to continue billing as much time to the customer as possible. If the developer is in a fixed bid situation, they are motivated to truly minimize the cost of the project. Thus, by giving them the opportunity to minimize the cost (and perhaps increase their profits), you may see a potential solution that you were not aware existed.

An example of this may be that the developer has internally developed libraries of code that they feel would save a great deal of time on your project. However, the process of weaving this into the T&M bid may not be something they are motivated to do. A fixed bid will actually motivate them to take this approach and may end up being the best situation for both parties.

- **Expose Conflicts**

While this is somewhat similar to the information in the paragraph above, I think there is some additional information of interest here. As previously mentioned, the motivation of the parties is somewhat misaligned with the straight T&M project. While the fixed bid is not a perfect solution, it does put the responsibility on the developer to take a harder look at the project. The tendency with T&M proposals may be to under bid them, either intentionally or unintentionally, by not making enough effort to truly understand the project. The developer has little true responsibility to accurately or conservatively bid a T&M project. In fact, the developer who accurately and honestly bids a T&M project may be penalized by not being awarded the contract because they are not the lowest bidder. This creates a conflict of interest in the relationship prior to starting.

By asking the developer to prepare both a fixed bid and a T&M bid, the developer will (by necessity) need to put in a fairly significant effort to understand the project. The presumption here would be that by expending the effort necessary to prepare a fixed bid, the accuracy of the T&M bid would be greatly increased.

When you request the proposals from the developer, there are a number of things you should be requesting along with a schedule and price. Here are a few of them:

- **List of Assumptions Being Made**

In every proposal, there are assumptions being made by both parties. They range from the absolutely trivial (a particular button will be green) to extremely serious (the program will *never* run on Windows NT). It is absolutely *critical* that the developer exposes and describes the assumptions they are making when preparing the proposal. Actually, the best method of addressing this is to encourage the developer to inquire with you and your team during the proposal process about their assumptions. If the developer calls and asks “Will all these buttons be green?” and your response is “yes,” the assumption goes away. Likewise, if they ask, “Is it okay if this program doesn’t run on Windows NT?” and your answer is “yes,” the assumption is mitigated. In a perfect project, the assumptions wouldn’t be necessary, but it is very unlikely that all the loose ends can be tied up prior to actually executing the project.

From your perspective, it is important that you understand the assumptions clearly and completely when evaluating the bids. Some developers may bring up assumptions that others do not. This will be a good indication that there are ambiguities in the information provided to the developers. You should make certain that you fully understand the implications of the assumptions that each developer is making. In my experience, these assumptions are the largest and most dangerous “bear traps” in any project; they have the potential to haunt the project from start to finish. It is best if both parties are fully appraised of all assumptions and able to address them in the open.

- **List of Developer’s Assessment of Risks**

There are always risks to a project. The developer should be able to determine the associated risks from their perspective. These may range from “Programmer X is taking a 4 week vacation in July.” to “Third-party software library represents a tremendous liability to the project.”

It is equally important to understand what the risks are *and* what the developer perceives the risks are. Are the risks things that they feel could be “show stoppers,” completely stopping the project in its tracks? Or are the risks simply things that they believe will have an impact, but are controllable? It is not always easy to determine the extent of the risk at the outset, but it is important that the developer have an idea of what the risks are and that both parties are aware of them prior to beginning the project.

- **Disclosure of Reliance**

Is the developer relying on any other parties or products to meet their obligations? Are they going to license third-party software, and if so, what is the state of it? Will the schedule be relying on a third party?

While these items should be listed under the assumptions and risks, it is often helpful to break them out separately because it will call attention to the need to mitigate them. These items are of particular interest because they may be out of the direct control of both parties. For example, if the developer says they are relying on a third party to deliver some hardware in order to meet the development schedule, who is responsible for making certain the third party is meeting their required schedule?

- **Proposal Delivery Date**

Get a confirmed date for the proposal to be ready and delivered to you. Make sure you have adequate time to review and follow up on the proposal with any questions that you may have prior to any internal deadlines.

Be certain the developer acknowledges the delivery date and feels they can meet it comfortably. Additionally, ask the developer to make someone available for follow up questions after the proposal has been reviewed.

**Should you pay for a proposal?**

Good question. While this may seem a little like putting a fox in charge of the hen house, sometimes it could be a very good investment in your project. Here are a few thoughts to consider:

- Good developers are very busy.  
The developer's workload may be such that they simply don't have enough time to spend doing a great job on a free proposal. If this is the case, the resulting proposal may actually end up being a poorly executed T&M only proposal. This proposal could actually be worth less than nothing and could be a severe liability to your project. In a perfect world, people who are responsible to execute the resulting work would prepare all the proposals; this isn't always possible. By paying for the proposal, you may have the opportunity to be a bit more specific about who is working on the team and their skill sets.
- Your project may not be attractive.  
It is possible that your project is not the "ideal" project for the developer. They may place a lower priority on bidding it or actually doing the work. This could result in a lower quality of people working on the project and perhaps the proposal. The resulting work may not be as good as you might expect.
- The proposal price may be nominal.  
If you have a large project, the actual cost of the proposal may be very nominal. For example, if the project is over \$1 million, the proposal price of \$25,000 may not be entirely significant. On the other hand, if the developer is a relatively small company, putting enough resources on the proposal process to complete it may be very burdensome. The developer may be reluctant to do so without some worthwhile compensation.
- You can "try before you buy."  
This is a great opportunity for you to see how the developer treats their customers and their responsibility to meet deadlines and delivery requirements. If you grant a contract to the developer to generate a proposal, and they don't meet the deadlines, are tough to work with, or just don't feel right, it is a lot easier to back out at that time than after you have engaged them to do the project.
- You may be able to request a continuous team effort.  
By paying for a proposal, you may be able to request the developer establish a team of people who will prepare the proposal *and* be available to continue on with the project should you award it to them. This can have a tremendously positive effect on the project because the people who do the proposal know they must be accurate and careful since they will need to live up to their commitments in the future. In addition, it can help both parties by having the education process of your project be incurred only a single time by a single set of people, resulting in a positive effect on the schedule.

## **Risk Management**

There are many risks involved in developing software. Not all of the risks are related to the actual project. Many of the risks are to the companies involved and to the individuals who are running the projects. If possible, try to understand and communicate the risks you think are relevant and important. While it isn't necessary to dwell on them, understanding and communicating the risks will lead to a stronger partnership.

### **Risks to the Partnership**

What would the impact be on your firm if the developer completely failed at delivering the software? What if they delivered, but it was one year late? What would the impact be on the developer if they fixed bid your project and had similar overruns? Could they afford to continue working on the project?

Both parties should consider these types of questions early and carefully. On one hand, you may be a crack negotiator and negotiate a terrific fixed bid contract for your project, only to find that you have backed the developer into a situation that they cannot deliver on that will result in them making the unpleasant choice of going out of business or working on your project. The opposite situation is equally distasteful. Imagine you are a weak negotiator and the developer convinces you that a T&M project is the best arrangement for you, and they have drastically under bid the contract. Shortly into the project, you find that the project will overrun by about 300% of the original bid.

I've seen both of these scenarios happen; the situations were very unpleasant for all parties involved.

This is one area where an experienced, successful developer (presumably) will be very cautious and careful. The developer should fully understand the commitments they are making and should be able to provide the resources necessary - financial or otherwise - to meet their commitments. However, as the old saying goes, "You can't squeeze blood from a turnip." If the developer is over extended, it may be just a matter of time before the project comes apart at the seams.

You need to ask the hard questions when you are making the decision of whether to use a developer and what type of contract to arrange. What will happen if the project goes over schedule by 100%? 200%? 300%? What if it goes over budget by the same percentages? Will your firm be able to handle it? If the developer is "on the hook" for the overruns, will they be able to handle it? How will it affect your relationship?

### **Risks to the Project**

Although I covered some of these risks in the paragraph above, it might be worthwhile to consider a few more. In general, when most people think about the risks involved in outsourcing a project, they think about the risks to the actual project.

The main risks at the macro level are cost and schedule. The third macro risk is the features or the functionality of the finished project.

Inevitably, in all projects that are having troubles, the question is raised something like this: "We can make the schedule if we cut feature X." or "We can save \$25,000 if we cut feature Y." These represent the tradeoffs in what I call the "control points."

### **The Control Points**

There are three major control points within any project:

1. Cost (How much?)
2. Schedule (When?)
3. Functionality (What?)

It is usually a reality within project management that changing one of the control points will have an influence on the others. For example, if you want to cut the cost of the project, you will most likely need to either extend the schedule or reduce functionality. Conversely, if you want to decrease the time to market

(pull the schedule in), you could potentially do this by increasing the cost (by adding more resources) or decreasing the functionality required.

With few exceptions, there is no *free lunch* regarding these control points. I'm sure that everyone has seen the very visible project slippage of Microsoft on something like Windows 95 or Windows 2000. In both of these cases, the culprit for the slippage of schedule was functionality. Microsoft either underestimated the actual time to implement all the functionality or they continued to add functionality beyond the original estimates. Even for the software giant with virtually unlimited resources, they are not able to pull the schedule in appreciably.

Understand and get comfortable with the control points on your project. When you start the project, you should have a reserved budget for cost, schedule, and functionality. Determine your driving motivator, i.e., controlling cost, meeting a schedule, minimal functionality, etc., and adjust your other control points to compensate.

### **Contract Issues**

For both partners' benefit there should be a written contract for the work being undertaken. The contract is the vehicle under which the parties conduct their business and it should cover the technological, legal and business issues for both the client and the developer.

There are a number of basics that are essential in every contract. The "big three" are:

- What are we building?
- When will it be done?
- How much will it cost?

Without these three issues identified and clearly addressed in the contract, you will have a hard time achieving success with your project. These three issues are the measure of success for any project.

#### **What?**

The "What" is generally covered in a document called a "Statement of Work" or "Functional Specification." (For the purpose of this paper, this document will be referred to as FuncSpec.) There are varying degrees of completeness to the FuncSpec, ranging from a drawing on a dinner napkin to a book-type document. Furthermore, it is possible that you have asked your partner to develop the FuncSpec. In any case, the FuncSpec must be adequate for an *independent* third party person to easily make an assessment of whether or not the developer has met the requirements. Think about the scenario where eight months from now the developer says the project is finished, but you disagree. Does this document contain sufficient details to refute the developer's claims?

Often times, there is ambiguity in the FuncSpec. The degree of complexity associated with the average software project virtually prohibits these documents from being as detailed as they should be. Further, it is often better to leave some areas a little ambiguous to provide for resolution by the developer once they have gained enough knowledge to solve the problem or potentially make suggestions about better ways of addressing the problem. This ambiguity can be a double-edged sword. While a client can use the sword to claim the project is incomplete, the developer can easily use the ambiguity to claim the project did not include the requested level of complexity.

Identify areas in the project that you feel are critical and make certain that you have specified the behavior and the appearance, carefully and completely. Spend the time to make sure that the areas (you feel) essential to the success of your project are documented adequately.

Another critical area to make sure you have nailed down is the requirements for the operating environment and specifications for performance. What operating systems will the product run on? What other programs must it interact with? What data sources must it be compatible with? What is the minimum machine configuration it will run on (CPU speed, memory requirements, and hard disk storage requirements)?

## **When?**

This is often the \$64,000 question and perhaps one of the most difficult to answer. In my practice, I have had customers equate the development of software to many things, including voodoo, custom home construction, bridge construction, airplane manufacturing, and shipbuilding. Nearly all of these comparisons are valid (Okay, maybe voodoo is a stretch.) because they generally are descriptions of building a unique product as opposed to an assembly line. They also describe a situation where you are potentially building something for the first time. That is where the similarities end. You could easily look at each of the examples above and identify many projects that have run amok with cost overruns and schedule problems. Software maintains the unique position of being notoriously late on delivery schedules.

Why is software so hard to predict? This question could fill volumes and is beyond the scope of this paper, but I think it's important to understand the basics as they can have a tremendous impact on determining and controlling your project's schedule. The short answer is that the software developer has very little control over a great deal of the project. This sounds a bit strange. But you must consider that usually the project requires an operating system (Windows 98, Windows NT, Windows 2000, Linux, Mac OS, etc.); often times, it requires additional support libraries (database access, graphing libraries, reporting libraries, sound libraries, etc.); it almost always requires an installation program, which is usually developed by a third party; and on some projects, there is an external device involved (device drivers). This combination of variables presents a challenge that is very difficult to predict and grasp.

To further complicate the issue, often times, the project needs to be compatible with a beta version of an operating system in anticipation of its release - sometime in the future. This is a situation where the operating system - the very foundation on which your project must run - is in a state of flux. All these challenges before work on your actual project begins. The actual problem-set of addressing your project's needs begins after thinking through all the environmental issues. The developer must gain at least the basic domain knowledge of your project and the problem(s) it is trying to solve. Then, the developer must predict the amount of work to complete the task. Usually, the software you are asking them to develop is something that they have not developed before, further complicating the problem.

Predicting a schedule is very important and should be done for all projects. As a customer, you should be aware of what the developer believes is an achievable schedule. Further, if you use the methods of motivating (described later in this paper), you should be able to identify where the developer is confident in their predictions and where they are perhaps less confident.

## **How much?**

The "how much" figure is nearly always a factor of the "What" and "When." Additionally, it is perhaps the most worried over, controlled, and negotiated factor in a contract. Although it is important to control costs, I think it is best to focus on the "What" and "When" *before* focusing on the cost. Here is why:

- Unless you are licensing some proprietary technology, the cost is very likely directly linked primarily to the "What" factor and secondarily to the "When" factor. If you don't have these two issues fairly well in hand, the natural reaction by the developer is to increase the cost to cover the potential risks associated with unspecified work.
- Increasing the "What" in any way will likely increase the cost. Work carefully to make certain that you have communicated an accurate description of what it is you are trying to build. The better this is described and understood by the developer, the more accurate the cost estimates will be.

You should try to identify features of your product individually and determine what features are absolutely essential to have a successful product. Prioritize the other features, making the hard decisions *early* in the process as to which features you think can be dropped or deferred to future versions. Doing this early in the process and with the help of the developer can have a tremendously positive effect on the cost of the project.



- Schedule is a tough subject. It's important to understand that there are limits to accelerating the schedule. The classic example is to consider a pregnant woman. It is supposed to take nine months for her to have a baby. You can't put nine women on the job and get the baby born in one month. There is a critical path here; bearing a child is a one-woman job. It is essential that you and the developer understand the critical path of the project and how the functionality interacts with the schedule. Sometimes it is possible to make minor changes in the functionality to gain a positive impact on the schedule.

I am not advocating that you completely ignore the cost issue until you have the schedule and specification hammered out. I'm simply saying that focusing on the cost as the primary concern will not achieve your goal of controlling the cost. I have had customers who said things like, "I have a budget of \$75,000. How much of a printer driver can you get done by June 30?" The customer is telling me that they have two primary goals here, cost and schedule. However, it may be impossible to meet either of the goals without knowing what the printer driver requirements are. These projects send chills down the spine of every seasoned developer. A much better approach to this type of problem would be to say something like, "I need a printer driver for the FlameJet 4000 that supports the full LaserJet 2000 feature set. I've got an aggressive schedule of beta on June 30, and I have budget restrictions. Is it possible to meet the schedule with these required features?" Then proceed from that point to ascertain the viability of the project and schedule. When you're feeling comfortable about the developer's grasp of the project and schedule, pursue the cost issue.

### **Additional Contract Concerns**

#### **Intellectual Property**

The contract should identify the disposition of intellectual property (IP) and the control of the property. Is the developer free to use the IP on future projects with other customers? Does the developer have their employees under contract to prevent IP problems when they leave? Does the developer have conflicts of interest in the IP area, i.e., customers with similar products?

#### **Ownership of the Resulting Work**

Depending on the type of work you are engaging, you might be surprised to learn that your firm does not actually own the resulting work of the developer. It is common practice for a developer to license the resulting work to a customer (your firm) and retain the rights to resell the software in some form to other customers. This technique can be a cost-saver for your firm if the developer intends to resell the technology to others and is discounting or paying your firm royalties on the future sales. However, the issue should be clearly addressed in the contract and both parties must be comfortable with the arrangement.

Generally, a time and materials arrangement is a "work for hire." Essentially, this means the customer owns all the work unless specifically called out in the contract. Other types of contract engagements (fixed bid, blended, etc.) can also be work for hire, but they can just as easily be licensing arrangements.

#### **Licensing of Other Technology**

Today, developing software frequently requires interaction with third-party software. It is common practice for the developer to utilize libraries and tools that ease the burden of developing the software, and should save your firm time and money.

Some things to consider with respect to the use of these other technologies:

- Which party pays for the technology?  
This is fairly simple to understand. Often these tools are inexpensive and may be insignificant in the overall cost of the project. However, there are tools that may be very expensive. The cost of the technology should be clearly identified in the contract. It is a good idea to identify the significant technologies in the contract and give the developer an authorization to use technologies that they can identify as meeting the quality, licensing, cost, schedule, and other requirements.

- Does the developer own any interest in the technology?  
It is common that a developer will have libraries or software components that they have developed on previous projects. Often the developer will be using these components to gain an advantage on delivering your project early or faster. Be careful to understand the licensing requirements. Ask your developer if the software being developed for your firm will be used on other projects for other customers. There should be a fair and mutually beneficial arrangement made in these situations.
- What are the distribution arrangements?  
Are there restrictions on the distribution of the technology? This could be a concern if your project is for distribution to a large customer base; you may be surprised to find out there are onerous costs associated with a wide distribution. Obviously, if your project is for use by your accounting department, which contains only three people, then you really don't need to worry too much about unlimited distribution licenses; but you should understand the distribution restrictions and comply with them. Further, it is essential that the developer knows *in advance* what your intentions are to help them understand the issues.
- Are the licenses in your firm's name?  
Depending on who "owns" the license (your firm or the developer), the contract with the technology vendor should be clearly identified. If your firm intends to distribute the software, it is very likely that your firm's name should be on the licensing agreements. Be sure to understand the issues *before* the developer integrates the technology into your project.
- What is the support policy of the technology?  
When your project is at the *almost ready to ship* phase, and you discover a major show-stopping bug in this third-party technology, what is the vendor's policy on fixing the bug? I have seen numerous projects held up by a minor third-party vendor not wanting to fix a bug outside of their release schedule. Again, this may be acceptable to your project, but you should understand the vendor's policies in this area prior to integrating their technology.

An additional concern is the upgrade policy of the technology. Generally, these libraries or tools are solving a problem that your firm needs solved in the long term. Often these problems are moving targets, and upgrades will be forthcoming. What is the vendor's policy on upgrades? Do they guarantee backward compatibility? Will they continue to support your version?

- Who suggested using the technology?  
While this may seem a minor point at the early stage of the project, it could become very divisive. The issue here is if you or your firm suggests or insists that the developer utilize a certain piece of technology, and the developer ends up having significant problems with the technology, it may impact the schedule and cost. It may be advisable to identify this in the contract.  
  
On the other hand, if the developer suggests the use of the technology, they should be held responsible for making certain it meets the requirements of the project and schedule. Again, this should be spelled out in the contract.
- What is the quality of the technology?  
Is this technology something that your firm feels comfortable placing your name on? How do you assess this issue? I have seen several projects that are quality pieces of work, but have a low quality component that drags the rest of the work down with it. Further concern is the track record of the technology, its revision history, its installed customer base, and so on. If this technology represents any significant portion of your project, either you or the developer should get comfortable with the technology's history and level of quality.

### **Payment options**

How are the payments made and under what conditions? This is often linked to the type of contract. In a time and materials arrangement, the developer will usually desire Net 30 or similar arrangements. In fixed bid arrangements, there are usually some types of milestones that are connected to payment arrangements.

The payment arrangements should be spelled out clearly in the contract. I strongly suggest that payment terms be connected to a demonstration of completed work. This usually takes the form of a milestone. In other words, the developer reaches a milestone, the milestone is verified, and the payment is made. In the absence of the milestone requirement, it is all too easy for the project to get off schedule without the parties being aware of it.

Another issue with payment is the authorization chain for granting payment. It is essential that the approval of payments be linked to someone at your firm who has knowledge of the project and is responsible for approving the progress of the project. Having the developer obtain payment directly from your firm's accounts payable department, without intervening approval by a project monitor, is a recipe for disaster.

### **Methods of motivation**

There is an expansion of this topic in following paragraphs, but I think it's important to note that the contract must identify any bonuses, motivational issues and, in particular, payments that are connected with out of the ordinary performance by the developer. The actual methods of motivating should also be spelled out clearly in the contract along with any special requirements such as milestone dates, quality levels, tough problems to be solved, etc.

### **Legal Mumbo Jumbo**

I think it's important to put you on notice that I am not an attorney. Although I can give you advice and share my opinion on contract issues, I would recommend that each of your contracts utilize an attorney who works on your behalf to consummate each contract. There are numerous legal issues that should be spelled out in the contract that I will not go into here, including venue of law, arbitration, legal fee retribution, notices, etc.

Usually, these contracts involve fairly large sums of money (greater than \$25,000). It would be prudent for you and your firm to have an attorney review all contracts on your behalf.

### **Methods of Motivating**

It is important to understand that each party on the project has different motivational needs and desires. Earlier, I referred to the late delivery of Microsoft Windows 95 and Windows 2000. Microsoft placed a higher priority on functionality and quality than they did on schedule. As a customer of and partner with a developer, your firm must identify the motivators that are critical to the project's success. Is it the schedule and time to market? Is it the cost? Or could it be the functionality? It is not possible to have all three be priorities; occasionally you can combine two of them, but usually you can boil it down to a single priority.

Communicate these priorities to your developer. Make sure they are aware what your priorities are and ask them how you can have a positive impact on what you consider the most important. There are always options, and the developer is perhaps the best positioned to help you control the project's priorities.

The developer's motivation must coincide with the main motivator for your project. Although this sounds obvious, it rarely happens. Here are some examples that I've seen:

- The customer has a definite need to control costs. The project is a driver for an input device. The contract is structured as a time and materials arrangement, and the customer wants to manage the project thinking that by managing it they will save money.

In this case, the customer would have been better off going to a body shop. They are using the developer incorrectly. The motivations for the developer are completely misaligned with those of the

customer. The developer doesn't have control of the project; the customer is directing the developer. This leads the developer into the position of "being paid by the hour" at the customer's direction. Instead of controlling costs, quite the opposite will occur.

The best arrangement in this case is a fixed bid contract. The customer and the developer will agree on a cost that is workable to both parties, and the developer will be responsible to meet the feature set at the costs agreed upon. The developer should be left to manage the project as best they can.

- The customer, absolutely positively, must have the product done by June 1. They don't care about the cost (within reason), and they are willing to strip features and functionality to meet the deadline. The contract is structured as a fixed bid because the customer has had bad experiences in the past with time and material arrangements. The developer is now charged with meeting what is most likely an impossible schedule, but at the same time, is very likely having to manage a great deal of risk because of the fixed bid arrangement. The risk is primarily borne by the developer, and the risk to the project is extreme. The motivation of the developer is to control costs, not to get the project out.

The best arrangement in this case is a time and materials contract with additional rewards for delivering the project to meet the schedule. If the customer has concerns about runaway costs, they can put "not to exceed without permission" clauses into the contract. By having the project be time and materials, and having the rewards for meeting the schedule, the developer is motivated to meet the schedule.

- The customer has no clear direction for the project, but needs to get moving on it. This is like saying, "We don't know where we are going, but we don't want to be late getting there." We had a client like this for about two years. They invested a great deal of money into building demo products, modifying the software for trade shows, and (in general) did not proceed towards building a finished project. The contract arrangement was a time and materials arrangement. Although we told them repeatedly that they were not spending their money wisely, they didn't understand what we were telling them. The best arrangement for both parties here would have been a series of contracts or milestones in which the customer and the developer identified the next steps in the process of completing the project. The customer didn't understand what it took to actually deliver the product, and the developer is not motivated to force this on them.
- The customer wants to develop a client-server system that is probably 20,000 man-hours of development work, wants it done in two months, and has about \$50,000 to spend. You do the math! (This is an actual example of a potential customer from a few years back.)

Okay... so some things never change and you can't figure out solutions to motivate each party all the time. Understanding that there are conflicts in motivation is critical to predicting where problems will come up.

There are essentially four types of working arrangements for dealing with developers. Below are the four models, a short description, and some pros and cons of each.

#### 1. **Time and Materials**

This is the classic hourly rate model. The developer works for an hour and bills your firm for an hour. This model affords the developer the greatest freedom from risk. If the developer makes mistakes, your firm pays for it. If the developer slips on a schedule, your firm pays for it.

T&M is best suited for loosely defined problems, challenging problems, difficult schedules, research projects and so on. The key learning here is that T&M is *not* well suited for long term, well-defined projects, or projects that have reasonable time schedules. The developer's motivations are not aligned properly with those of the customer.

#### 2. **Fixed Bid**

The simplest form of a fixed bid arrangement is that the developer will develop product X for

the cost of Y and the delivery date of Z. Rarely is it quite this simple, but it could be. The developer is assuming a great deal of risk in these arrangements, and they should be asking to be compensated for that risk. These types of arrangements generally look great to the customer because of the limit in the cost arena. However, if the developer has trouble meeting the obligations, there are risks to the relationship and the project as I've discussed in the risk section.

Fixed bid arrangements are best used for well-defined work that has a fairly reasonable schedule. This is not to say that well-defined work with aggressive schedules wouldn't be good for a fixed bid. I think it would be better to utilize a fixed bid with the bonus upon delivery system. Fixed bid is ill advised for loosely defined work, work that is likely to change during the development (most work changes, but I'm speaking here of dramatic changes), work that has experimentation requirements, and so on.

### 3. **Cost Plus**

Cost Plus entails the developer to expose their actual costs and, from there, negotiate a profit margin. These arrangements are good for projects that have varying needs with respect to people working on the project. Generally, these arrangements are also similar to the T&M type of contract in that the motivations for the developer are to bill hours and not necessarily to complete the work.

### 4. **Hybrids**

You can mix and match the above combinations into quite a potpourri. I've seen arrangements where the initial stages of the project were T&M and the final stages were fixed bid. I've seen projects where it was entirely T&M until a cost ceiling was reached and then it changed to a cost plus arrangement.

The main point is the importance of being flexible and creative. Determine how you want to motivate the developer based on what stage the project is in. Arrange the contract to meet the needs of the motivation at that time. Here are some examples:

- During the initial stages of design and development, utilize a T&M contract. This allows the developer to spend the necessary time to get it right early. They are not bearing a tremendous amount of risk, and their motivations are to deliver the best quality designs they can.
- After the design is finished, utilize a fixed bid arrangement until the project is "code complete." After the developer is finished with the designs and has had the time to *do it right*, they should be willing to sign up to the risk of a fixed bid implementation phase.
- When the project is code complete, switch to a cost plus arrangement. This is typically known as the debug phase of the project, when the developer is finding and fixing bugs. It is important to be certain that the motivation is for the developer to finish this part of the project. Keeping them on a T&M arrangement during this stage creates a conflict of the developer billing hours to fix their own problems. In this case, they are motivated to *not* fix bugs during earlier stages and defer them to the final stage. This is detrimental to the project. By switching to a cost plus arrangement with a very small or non-existent profit margin, the developer will be motivated to complete the project and not defer bugs into this stage.
- Consider bonuses for meeting schedule or quality goals. Utilizing a cost plus and bonus approach can be very beneficial to both parties and reduces the risk for both parties. For example, the developer has clear deliverables and schedules identified. The schedule completion dates have bonuses associated with meeting the schedule. The developer proceeds under a cost plus basis until deliverables are achieved. When both parties agree the goals have been met, the developer receives the bonuses.
- Consider using a sliding scale bonus. One of our customers came up with this approach a few years back, and I think it aligns the motivations for the parties quite nicely. In this case, they needed to have

a project completed by a magic date (for a trade show demo). They requested an estimate from us; the estimate came in around \$200,000. Then, the customer suggested that we put a “not to exceed figure” on the contract. Generally, I am opposed to *not to exceed* contracts because they are really fixed bid contracts in disguise and, therefore, put excess risk on the developer without the accompanying rewards.

In this case, the customer said the not to exceed figure was \$300,000; if we could meet the scheduled deadline, they would split the savings of our actual time billed subtracted from the \$300,000. In other words, if we billed \$225,000 to meet the deadline, the difference would be \$75,000 and we would receive a bonus of one half or \$37,500. This arrangement worked out very well for both parties. The customer aligned our motivations to get the project out on time, and we were motivated to do it efficiently.

One last point here: when in doubt about what the motivations should be, focus on schedule. In most reasonable sized projects, the schedule actually will drive the overall cost of the project and the feature set. Obviously there are limits here; but in general, if you have 10 people working on a project and you can control the total amount of time they are actually involved in the project, you will have a control on the overall costs. If you can keep the schedule roughly on track, the costs will remain roughly under control.

### **Verification of Progress and Success**

You’ve done your homework, picked the developer, and the project is underway. Everything is going smoothly, or is it? The only way to know for certain is to develop a method for measuring the progress and, just as importantly, the success associated with the progress.

The tendency for most developers and customers alike is to establish intermediate milestones or checkpoints throughout the project where evaluations and measurements are taken. The challenge with these milestones is that during the early stages of the project, the milestones are extraordinarily difficult to assess as being complete. Since everyone is typically interested in the project being a success, the milestones are usually met and (most often) on schedule!

The real challenge is to work with the developer to establish easily measured milestones and checkpoints throughout the project. I remember a project where the milestones in the early stages of the project were as follows:

1. Functional Spec Complete - July 1
2. Design Complete - July 25
3. Coding Complete - September 11
4. Debugging and Testing Complete - October 15
5. Final Shipment to Customer - November 1

The project progressed nicely. The first two milestones were hit right on the money. The third milestone began to slip in mid-August. Milestone 3 was complete sometime in November. The testing was complete in January, and the project shipped to the customer sometime in early February.

What’s wrong with this picture? How do you measure whether the Functional Spec is complete? The potential for missing items in the functional spec is great. All the great minds on the product review the functional spec and decide it is complete. However, when someone sees the product partially functional a couple months later, they say “Where is the modulating blue sliding door feature?” Upon which the developer says “It’s not in the functional spec.” The customer says “We can’t ship this product without the modulating blue sliding door feature; it has got to be in there!” The project slips.

The Design Complete milestone is equally difficult to measure. How do you know the design is complete and will work? Who is qualified to make this decision? Remember that depending on the motivations of each party, the developer may want the design completed before it really is.

The later milestones are easier to measure because the product is beginning to work. People can “touch” it and “feel” it. The bug reports measure and can show the number of bugs found, fixed and ignored.

How can you avoid these problems? The key is to avoid establishing milestones that are not measurable. Much easier said than done. In the example above, instead of having a milestone that is functional spec complete, it might be better to establish a set of “typical use cases” that are the top 30 to 50 main uses for the project. These use cases are then tested against the functional spec. The functional spec is then used to walk through the use cases that describe what the user is trying to do with the program. The same use cases can be used to exercise the developer’s design. The developer should be required to utilize the use cases to walk through their design, demonstrating that the design is functional and can meet the requirements of the product.

The exercise of generating use cases is less of a challenge than measuring the progress of a milestone. As a customer you should be able to articulate and describe the use cases with the help of the developer. These use cases will most likely be descriptions of how you envision the product to work.

### **What to Do When Problems Arise**

There is a sinking feeling in your stomach - you just know the project is not going well. Is it “tanking” or just struggling? You need to make a true and painful assessment of the state of the project. This is where your concrete milestones and deliverables are critical.

#### **Identify Problems Early**

The sooner you can identify the problems, the more time and options you have to actually affect change on them. To use a personal example, I have been river rafting and kayaking for about 25 years. I used to own an outfitting business in Idaho, and we trained hundreds of river rafting guides over the years. One of the most important principals of running difficult rapids is to make your choices early and make two strokes at the top of the rapid instead of 200 at the bottom. By the time you reach the bottom of a rapid, the current and water is a formidable obstacle to overcome. This is very true in project management as well.

When the problem is identified early, there are plenty of options to dealing with it. If, for example, the first milestones of the project are slipping, it may be time to add additional resources or reduce the functionality. Make this decision early and stick with it. Slippage early in a project will become multiplied during the later stages.

Utilize the milestones and the verification mechanisms you and the developer put into place prior to starting. Don’t “live the lie.” It is very easy to fall into the trap of *feeling good* because it makes it easier to sleep at night. Be objective. Have a coworker look over the project details with you and give you an honest assessment. If the data indicates you have trouble, get it out on the table with the developer and ask them to propose solutions to meet the goals of the project.

#### **Exploit the Partnership**

At the first sign of trouble, you will find out what your partnership is made of. If you have chosen a true partner, the partner should come to you and say, “We’ve got trouble and here is what we want to do about it.” If you go to them and they resist or don’t agree, the red flags should go up. If the partners are interested in a relationship that survives a single project, then there should be considerable effort applied to solving a problem on the project.

#### **Re-evaluate Motivations**

This is a great time to take a look at the motivations for both parties. If the project is in trouble and the developer is struggling, perhaps the motivations are not correct. It is also possible that by changing the motivations, the project will get back on track sooner. It is sometimes very difficult to look at a project and say “The developer created this problem. Why should we increase the bonuses or motivations for them to correct it?” The answer lies in your firm’s motivations. Do you have schedule issues you cannot let slip? Is the feature functionality something that cannot be sacrificed? If so, you need to swallow your pride and figure out a way to motivate and re-energize the developer to move forward and deliver what you need. Take a serious look at the state of the project and determine if the developer can actually complete the work. If you believe they can, you should make certain the motivations for them are adequate and properly aligned.

### **Avoiding “Living the lie”**

“Living the lie” is our internal term for the euphoria associated with people believing a project is doing better than it really is. Sometimes the entire team and the customer are actually ignoring the warning signs, and they really want to believe the project is doing well. It is *critical* for everyone to objectively look at the project and discuss the state on a frequent and regular basis.

Software developers are notorious for under estimating the amount of work involved in a software project. My experience is that most software developers use a very unsophisticated estimating mechanism. It goes something like this: If they can see a fairly easy solution to the problem, or if they have solved the problem before, it will take them between two hours and two weeks (This also takes the form of *trivial* or *easy*). If the solution is not obvious, and they think it will take some serious work to figure out the solutions, then the estimate is usually two months.

Seriously, the issue here is having *concrete* and agreed upon measurement mechanisms for determining the true state of the project. The natural tendency for a software developer is to be optimistic about the state of the project. This can be infectious. You want to believe the project is doing well, and it is human nature to not question good news. Think about it. Developer X comes to you and says, “The project is doing great. We’re right on schedule.” Your natural tendency is to say “Great! Thanks for the update.” What we should really do is say something along the lines of “Great! Let’s review the data carefully, understand why you feel that way, and try to insure we stay on schedule.”

My theory with software developers and their views of schedules goes something like this: In the developers mind, the project is in one of three states: ahead of schedule, on schedule, or behind schedule. If they feel the project is ahead of schedule, it’s probably really on schedule. If they feel it is on schedule, it’s probably behind schedule. And if they feel it’s behind schedule, it is probably *really* behind schedule.

The real truth is to make certain that you have established true mechanisms of measuring the state of the project. Make the developer demonstrate the functionality that they claim is working. This sounds relatively simple, but as it turns out, it is very difficult. The initial stages of a project are the most important to measure *accurately* and are also the most difficult. Consider that when the project is approaching the finish line, a majority of the software is functioning; it is fairly easy to identify areas that are not performing correctly and others that are missing. During the early stages of the project the deliverables may be design documents or functionality documents. These deliverables are relatively easy to *identify*, but fairly difficult to confirm that they are correct. How do you verify that the presented design is sound and can be built? Challenge the developer to prove to you that the work they have produced and the milestone they have delivered are of the quality necessary to continue the project and meet the schedule and cost objectives.

Remember, without a hard and fast truth about the state of the project, you will be “living the lie.”

### **Co-Development**

Co-development is generally a two-headed monster - loosely defined as “We’re doing part of the project and you’re doing part.” There are some definite pitfalls to avoid. Generally, this takes the form of your firm developing a proprietary component for integration with the outsourced components. For example, your firm sells architectural design software for skyscrapers. Your firm has domain expertise in the analysis of the structure of the buildings, but you want an outsourcer to design and develop the user interface for Windows 2000. The “interface” between your developers and the outsourced developers (both in terms of the software interface and the people interface) must be tightly defined and closely monitored.

A very common occurrence is for the project schedule to begin to slip; when you start to examine the reasons, you will find that there are serious problems or misunderstandings between the developers of each team. The parties need to develop an interface specification, which should be revisited and maintained frequently.



Of *extreme* importance is some type of agreed upon mechanism for deciding when each party is done with their component and how it can be verified. In the example above, the user interface software would rely extensively on the underlying structure analysis software. If the structure analysis software was incomplete in any way, it is likely that the user interface software will *appear* to be incomplete.

Here is a typical scenario of how this plays out, using the example above:

The structure analysis component (SAC) is supposed to be complete and debugged on Dec 1. The outsourcing team has been working in parallel on the user interface (UI) and is relying on the SAC to be ready for integration on Dec 15 (They didn't believe that the SAC team could hit their deadline.). On Dec 1, the SAC team declares the SAC is complete and has a ship party. The UI team is running a little behind and starts looking at the SAC on Dec 7. Everything appears to be in order, but the interface is extremely complicated and it is difficult to tell for certain if it is truly 100% finished.

The UI team begins the integration process and, around Jan 1, starts to discover that there are major parts of the SAC that are incomplete (programmers call this "stubbed out") and some that are malfunctioning. The UI team has a deadline to present an Alpha version of the software to the senior management of your firm on Jan 5. They make the presentation, but every few minutes they say something like "This function is not complete because the SAC is broken." Or even worse, they say, "The application crashes because the SAC is crashing." This now has the potential for some serious damage to the project, each company, the relationship, and the careers of the people involved.

How can you avoid co-development problems:

- Avoid Co-Development. Seriously - if possible, just avoid it.  
If the project is complicated and you must do it, you should take extra precautions to make certain your firm and the developer are well prepared for the undertaking.
- Make intelligent and informed decisions about where to divide the work.  
Often you may find that dividing the work at a slightly different spot may make it significantly easier to define the interface between the teams. Further, with a simpler interface, you increase the likelihood that it can be tested and verified to be working correctly.
- Understand the impact each component has on the project.  
In the example above, understanding that the SAC is required to be largely functioning correctly for the project to be stable or demonstrable would avert a great deal of trouble for both teams. Finding the critical components and path early in the project, and ensuring they are under control, will go a long way towards helping teams to avoid blaming each other for problems.
- Two words: Acceptance Criteria  
Perhaps one of the most critical items when doing co-development projects is acceptance criteria. It is essential that the teams agree on the method for accepting each other's work. I like to refer to this as an "interlock." When Team A's work relies on Team B's efforts to be successful, Team B should be in a position to accept Team A's work. Not to have it dumped on them and their feet held to the fire to deliver to the original schedule. The interlock occurs when Team B says, "Yes, we accept your work and we can rely on it to build upon and meet our schedule."

Writing, maintaining and living with acceptance criteria is a bit of a challenge. It doesn't have to be complete black magic, but it does take work. Often the major pitfall is that it is *easier to reject than to accept*. For example, Team A thinks they're finished, but Team B says function X doesn't perform correctly. Instead of accepting the work and giving Team A a new deadline for function X, they simply reject the work because it is easier. Sometimes Team B is behind schedule and blaming it on Team A feels better than just saying "Team A did a great job and delivered on time, but we just aren't ready yet."

By having a concise, clear and agreed upon acceptance criteria, you can avoid the finger pointing. Team A says they are done and proves it by demonstrating that it meets the acceptance criteria. There may be some discussions on the finer points, but the larger issue is that Team B can generally proceed with their efforts.

### **Closing Remarks**

This paper really just scratches the surface of a great number of issues associated with developing software and the outsourcing process. There are a number of great books on the subject of developing software and managing projects. Many of these are technical in nature and may not be all that readable for the “non-nerd”. However, I’d like to list a few that I found helpful.

*Software Project Survival Guide*, by Steve C. McConnell

*Debugging the Development Project: Practical Strategies for Staying Focused, Hitting Ship Dates, and Building Solid Teams*, by Steve Maguire

*The Mythical Man-Month: Essays on Software Engineering*, by Frederick P. Brooks Jr.

In closing, if there are only a few things that you take from this paper, I believe they should be the following:

1. Choose your development partner carefully.
2. Align the motivations.
3. Set verifiable milestones.
4. Make sure you get the contract in line with the project.
5. Avoid “living the lie.”
6. Avoid Co-development.
7. If you can’t avoid Co-development, make sure you have very tight acceptance criteria.

I would be interested in your comments or questions. You can drop me a note at:  
[danc@microcrafts.com](mailto:danc@microcrafts.com) or via fax: (425) 250-0113

## **Remediation of a Y2K Problem in a CMM-1 environment**

### **A case for project over process.**

David Butt, Brian Hansen, Tiel Jackson and Sam Sanchez

Oregon Health Sciences University, Portland, Oregon

#### **Introduction**

The maintenance, repair and enhancement of a mid-scale software system can be challenging even in ordinary times. Typical mid-size organizations are supported by in-house software developed, in some cases, over decades. Over time, they lose access to the experience of the original creators of the systems as well as their subsequent maintainers. Existing documentation, what there is, may be scattered and out of date. Programmers leave and the verbal documentation goes with them. Additional personnel may improve the situation after they gain mastery of these systems, but it is a hard-fought battle. When an unstable production environment is added to the mix, data processing workers may spend all day "putting out fires" with no time left for routine system upgrades, let alone responding to a changing business environment or the challenge of the internet. Remediating the unanticipated problems caused by the end of the century puts new strains on the data processing department which can threaten to overwhelm the entire enterprise. How many simultaneous battles can we expect to fight and win?

Before starting our Y2K project, we had recognized that our software engineering processes were a textbook case study at CMM level 1<sub>1</sub> and were motivated to devote some serious effort to a deliberate transition towards CMM level 2. What actually happened is the topic of this paper. We found that we had to consciously put that motivation on a back burner. Our experience was that whenever there was a conflict between project goals and process goals we gave precedence to the project over the process. We feel that we chose wisely, given the circumstances.

The authors take the stance that the Y2K crisis is technical in nature and its solution is essentially a technical issue. As such, given the time limitations imposed by the hardest of project deadlines (January 1, 2000), the project can best be addressed using the technical processes already in place. If the existing culture is CMM level 1, then you transition to the year 2000 within that culture; a simultaneous effort to jump a CMM level *and* solve the Millennium Bug is probably doomed to failure.

We felt from the beginning that the key to success in this project was to identify the multiple goals involved in the effort and to prioritize them. Clearly, year 2000 compliance was the foremost goal, but the safety of user data and avoiding user disruption were very important as well. Improving on existing documentation was of great importance, not so much for posterity, but for use during the project. Well-designed system libraries, improvements in the user interface, moving away from programs written in pre-Dijkstra<sub>2</sub> FORTRAN, improvements in robustness, and, yes, moving up to a higher CMM level all had to compete for priority and the scarce resources available.

#### **Background**

A data management system has evolved over the 1985-1998 time frame. Its basic purpose is to manage patient care for a large population of very sick children in the Pacific Northwest. Data encompasses: what treatments are provided; when, where and to whom; and also how the clinical programs are funded (by individual insurance, Medicaid and block grants). Two independent inventory management subsystems are included. The Care Management System is informally known as CMS.

CMS began as an assemblage of FORTRAN programs operating on flat files on a Harris computer. By 1990, the environment was a TurboImage data base (programs still FORTRAN) on an HP3000 computer. All programs

were typical batch, mainframe applications under JCL. By 1998, about half of the workload was interactive and about 25% of the applications software was written in C. There were 78 data tables in eight intertwined databases and 120,000 lines of code. Many programs ran to 3000 (or even 10,000) lines of "GO TO" code with minimal comments. Subroutines, often quite lengthy, were not stored in libraries, but cut and pasted from similar programs and adapted for reuse. The databases were not in third normal form. Dates were entered in YYMMDD format with so little input quality assurance that "Monday" was accepted as a valid date.

This system has had four significant authors; usually one at a time with minimal overlap during transition. The current owner and author had been in place for almost two years. He was familiar with not more than 40% of the code.

Experienced programmers know that while software does not "wear out," it does become old and brittle. The environment, both computing and business, changes. The human knowledge of how to use the system is diminished by every employee departure. Existing users tend to use a smaller "palette" of options, forgetting how to access areas of peripheral functionality and, consciously or unconsciously, avoiding problem areas even after the problem has been fixed. New users learn mostly from existing users.

All of these factors were at play in our environment. We have reason to believe that they are all too typical in others as well.

### Our approach

Our evaluation of priorities led us to make both immediate and longer-term changes. We implemented an SQL interface to the databases and put a stop to the practice of copying, altering and re-compiling code to accomplish routine data queries. We also added one full-time and one half-time software engineer whose efforts were dedicated to the Y2K upgrade project. With the redirection of half of our existing software engineer's time to the Y2K project, this gave us the equivalent of two full-time software engineers on the project. The project manager ran interference with both management and with the user community, leaving the team free to concentrate on the project.

With this team in place we were ready to plan our Y2K strategy. We made an early decision that the value of our databases far exceeded the value of the programs. They were literally the institution's long-term memory. This led us to a decision to use four-digit years for all future input, computation, storage and output. Accomplishing the transition from two-digit to four-digit years meant that every program and nearly all data tables had to be modified prior to deployment. The opposite approach (known as "windowing") leaves the databases with a two-digit year and implements logic in software (at run time) to assign two-digit dates to the appropriate century. The windowing approach can require some very complex programming logic when sick kids survive to be centenarians.

Lacking a generally agreed upon map of the territory, we undertook an inventory of all programs, capturing the relationships between programs and databases diagrammatically. This had never been attempted before and it proved invaluable. We concurrently placed a copy of existing code into a reasonably modern software engineering test environment, employing RCS and makefile technologies. This was our first benchmark.

The second benchmark was known as "control testing." On a program by program basis, we executed the copied code to verify that, prior to making extensive changes to program code, we could replicate the current, working system. This testing was quite informal and no test scripts were retained.

The test environment required the creation and maintenance of two copies of all existing databases; a two-digit and a four-digit year version. Software was written to keep the databases synchronized on a daily basis. This was known as "shadowing".

Control testing was only the second in a series of progress benchmarks; the next was to upgrade the programs to use four-digit years. This was a nearly mechanical process, and constituted a surprisingly small proportion of the overall effort. Before testing the upgraded programs with users, we had to pass the next benchmark called "crash testing." Like the previous testing, this was a programmer-only process, quite informal; essentially undocumented and no test scripts were retained. The crash testing did however ensure that only the programmers saw the obvious problems. Many of these problems were independent of the changeover from two-digit to four-digit years, arising

from file protection problems and similar factors introduced by the complexity of our test environment. The rest were pre-existing errors that the users had either never reported or that the programmers had never gotten around to fixing.

Up to this point, the new members of the team had very limited comprehension of the real purpose of these programs. User testing (our next benchmark) gave them the chance to interact directly with the users. Programs were tested informally by expert users and no attempt was made to capture these tests in the form of scripts.

Previous practice had been that bugs were either fixed on the spot or forgotten until they were reported again. User testing in the Y2K project gave us the obligation (and the resources) to reopen some dormant issues. Users identified several problems and functional deficiencies (some long standing and some new). These were analyzed and the highest priority items were fixed. A budget of 10% was specifically allocated to this.

Rather than deploying the system all at once and inviting the danger of multiple bugs in multiple program areas, we opted for a staged deployment strategy. We used the diagrams developed earlier in the project to identify program clusters which were independent or which had only limited connections with the other clusters. All programs within a cluster were deployed simultaneously.

The most serious political issue over the course of the project was to gauge the necessary and sufficient volume of user testing prior to deployment. The users were a factor here; a user community with a serious commitment to the user test process would have had a reduced risk of failure during deployment. While we attempted to test all functions before deploying a sub-system, some functions (especially undocumented features known only to expert users) slipped through. As a result, the deployment process introduced errors more than once. On two occasions, there was some data corruption and users were involved in the reconstruction process. On one other occasion, some functionality was withdrawn for two weeks and again, users were inconvenienced. We held an informal inquest immediately after each incident and the cumulative price of our abbreviated testing strategy was not much more than four or five user-days. We figured we could have halved the risk exposure overall if we had invested an extra month in testing time. The bottom line is that the costs involved in instituting a more comprehensive test plan would most likely have far outweighed the benefits.

Our original plan was to ignore the specific date test issue until we had completed deployment of the four-digit year programs and databases. At about the 75% completion point, however, we were granted a few weeks of access to another HP3000 and did a significant amount of clock advance testing. The testing was sufficient to convince us that although we had done an excellent job of converting to four-digit years we had broken code in several instances where data tables were accessed using fixed numeric offsets. This test phase was not ideally synchronized with the overall project plan. When we started these tests, only two (of seven) major systems had been fully deployed.

After completion, with all programs and databases in the four-digit year environment, we returned to the testing issues. The test plan had to answer two simple questions:

- “Does the revised program generate exactly the same results as the pre-existing program for May 13, 1999 (to take a date at random) and for other non-critical dates?” and
- “Do the revised programs produce valid reports from artificial data entries with critical date fields such as January 1, 2000; February 29, 2000 and July 1, 2000 (the start of our financial year)?”

We called these “baseline” and “clean” testing respectively. The baseline test had been answered to our satisfaction by what we called “holographic testing.” The Y2K project goal was to replicate existing program functionality in the four-digit year environment. Our holographic testing, also known as “flicker” testing was designed to demonstrate that upgraded programs produced *identical* results to the original two-digit programs, with the exception that date fields contained four instead of two digits. Errors in the pre-existing software are not detected using this testing technique.

The clean test required some system clock manipulation and this was also required as part of a legal “due diligence” defense plan. With user assistance, we developed four plausible scenarios that covered about 60% of the CMS user workload. These scenarios included scripts for making appointments (for example, prior to a critical

date) and ensuring that the appointments were handled correctly after the critical date. We ran each test up to five times with the critical date inserted at a different stage in the real time schedule. Similar scenarios were developed for inventory management and for the payment authorization process. All of these scenarios were played out through each critical date (calendar year-end 1999; leap-day 2000; and, where appropriate, financial year end 6/30/1999 and 6/30/2000. We were looking for two distinct error possibilities:

- The vendor's operating system calls could be in error. Such functions are not invoked consistently throughout CMS and errors of this type would parallel the inconsistencies. For example, it would be possible for the FORTRAN programs to be more error prone than the C programs or vice versa.
- Date arithmetic could be in error locally because different programs had the in-line computations modified independently. The February 29, 2000 date was a perfect example where we had to take extraordinary precautions to locate each instance.

The entire family of operating system calls was tested on the spare computer early in the project. We considered this a closed issue. Local program errors are a good-news, bad-news situation. Being local, they are likely to escape any test plan lacking 100% coverage. On the other hand, the user impact is limited and once detected, problems are easy to fix.

The administrative overhead on this project has been restricted to the following:

- 1) A twenty-five page project plan that we kept current to provide us with a common vocabulary, an agreed methodology and a sufficiently broad "critical path."
- 2) The program/database charting that we physically moved from one cork-board to another as we deployed the program clusters.
- 3) A 60 minute project meeting every week.
- 4) A one-page report of the above meeting including: done-last-week, to-be-done-next-week, new-information, decisions-made, and unresolved-issues.
- 5) A simple spreadsheet with "level of achievement" in each cell. Achievement was computed along two orthogonal dimensions. In one dimension, 20 component packages were assigned different weights based on lines of code. In the other dimension, progress was tracked from 0% (pre-benchmark #1) through 85% (after user testing) to 100% (post-deployment). Percent completion on the project was computed from the weighted sum of each partial completion; it tracked people-time within 10% through completion of the project.
- 6) Test runs from the Due Diligence Defense Plan, kept filed in separate folders for easy access.

An initial project goal was to maintain full user functionality and to devote ten percent of the time budget to fixing bugs and making minor enhancements. No metrics are in place to quantify the benefits (if any) from this expenditure. We are however confident that the benefits were significant as seen from the end-user perspective. System reliability also appears to have been increased, though, again, no metrics are in place to validate this impression.

The entire project was done between July 1998 and June 1999. The expense and effort were within budget after making allowance for a month spent on transition to the (newer) test mainframe that we inherited in April.

#### Deliberate Departures from best Software Engineering Principles

There were several occasions when our prejudices were "conservative" in the sense that we wanted to conserve at all costs a fragile structure that did actually work. The opposite, "radical" prejudice would have been to apply strict engineering principles that we had used (in another life) or learned about from professional acquaintances who claimed to live by them. Throughout the project, we argued the point continuously and the conservative view inevitably won. The following points relate to technical issues.

- Good professional practices would have begun with a comprehensive database and software re-architecture. We did not have the resources to do this even if the organization as a whole had the will to address the issue of specification. They did not.
- We retained the pre-Dijkstra structure of the code for fear of breaking it every time we removed a backward GoTo.
- We resisted the temptation to enforce uniform function libraries for calendar date manipulation. Some of the in-line code just proved inseparable from its source.
- An early ambition was to use automated code translation tools to convert from FORTRAN to C, yielding a uniform source language environment as a byproduct of the project. We could have done it but the cost was higher than we had hoped and the real goal of improved code legibility proved unachievable.

We followed a similar bias in our resistance to organizational innovations. If we had been more “process oriented”, we might have devoted as much as 50% overhead (a programmer year) to more formal project management; documented standards, checklist and templates; metrics and a stricter verification process. We could have created a permanent test environment and developed a regression test plan (i.e. replicating some or all of the previous tests in addition to creating new tests prior to each release). We could have documented every decision we made with a view to benchmarking our software engineering processes. Process was not our goal; we rejected all of the above as inappropriate for our culture, resources and time frame.

These radical/conservative biases can be put into a CMM framework and a CMM Level 2 assessment (viewed from 20,000 foot elevation) is enclosed as Figure 1.

GOALS	Key Process Area	COMMITMENT TO PERFORM					ABILITY TO PERFORM			METRIC	VERIFICATION		
		Policy	Process Definition	Methods	Standards & Checklists	Templates	Tools	Training	Budget	Metrics & Evaluations	Project Mgmt. Review	Top Mgmt. Review	External Review
RM1	baseline	■■■	■	■			■	■■■					
RM2	consistency	■■■	■	■	■		■	■■■	■	■	■		
PP1	estimation	■	■	■				■■■	■		■	■	
PP2	documentation	■	■	■	■		■	■■■	■	■	■		
PP3	negotiation		■	■				■■■	■		■		
PT1	tracking	■	■■■	■■■	■		■	■■■	■	■	■		
PT2	correction	■	■	■			■	■■■	■				
PT3	negotiation		■	■				■■■					
QA1	planning	■	■	■	■			■	■				
QA2	standards	■	■	■				■	■				
QA3	information flow	■	■	■				■■■	■				
CM1	planning	■■■	■■■	■■■	■		■	■	■				
CM2	identify assets	■	■	■	■		■	■	■				
CM3	change control	■	■	■	■		■	■	■				
CM4	information flow	■	■	■				■	■				

Figure 1: CMM Level 2 self assessment

The sixteen goals of CMM Level 2 are listed down the page. These relate to Requirements Management (RM), Project Planning (PP), Project Tracking (PT), Quality Assurance (QA) and Configuration Management (CM), respectively. All of these were considered relevant except for QA4 which covers a situation that never arose. The goals are briefly described in the table below.

RM1	System requirements allocated to software are controlled to establish a baseline for software engineering and management use.
RM2	Software plans, products and activities are kept consistent with the system requirements allocated to software.
PP1	Software estimates are documented for use in planning and tracking the software project.
PP2	Software project activities and commitments are planned and documented.
PP3	Affected groups and individuals agree to their commitments related to the software project.
PT1	Actual results and performance are tracked against the software plans.
PT2	Corrective actions are taken and managed to closure when actual results and performance deviate significantly from the software plans.
PT3	Changes to software commitments are agreed to by the affected groups and individuals.
QA1	Software quality assurance activities are planned.
QA2	Adherence of software products and activities to the applicable standards, procedures, and requirements is verified objectively.
QA3	Affected groups and individuals are informed of software quality assurance activities and results.
QA4	Noncompliance issues that cannot be resolved within the software project are addressed by senior management.
CM1	Software configuration management activities are planned.
CM2	Selected software work products are identified, controlled and available.
CM3	Changes to identified software work products are controlled.
CM4	Affected groups and individuals are informed of the status and contents of software baselines.

The twelve key processes of the CMM are arrayed across the page.

- Commitment to perform at level 2 requires policies, process definition, repeatable methods, standards, checklists and templates. We stopped about half way down the list. We consistently invented policies, process definitions, methods and metrics a few days before we implemented them – a form of “just in time delivery.” Such processes were judged to be sufficient for the job, no attempt was made to “institutionalize” them and there is no guarantee that they will be applicable to future projects.
- Ability to perform involves access to appropriate tools, training and funding. We considered ourselves adequately trained to begin with, our tools were neither more nor less than we required for the job and funds were deployed where we felt we needed them.
- We tracked only one simple metric. If we needed an excuse to celebrate, we had the percent completion metric to justify it. More than this would have been overkill.
- Verification was also neglected. The team was small enough that any of us could simply state that a goal had been met and we all accepted it without question. Relations with upper management were on the same informal basis.

Empty areas are those goal/process combinations that we consciously or unconsciously ignored. The heavy shaded (3 square) areas are where we rate ourselves highly; whether because that is the way we always did it or because we adopted new processes during the project. The intermediate (1 and 2 square) shading fills in the spectrum. The



grading is entirely subjective and we make no claim that our highest “grade” would constitute anything approaching a passing grade from a CMM assessor.

There are several time dimensions involved in the chart. Commitment and Ability precede Measurement and Verification. Requirements and Planning come earlier than Tracking, Quality Assurance and Configuration Management. A quick look at the chart shows that we are stronger on the top-left than we are on the bottom right. A more mature group might show a more even distribution.

There is an entirely different time dimension. Those cells in the figure 1 having three squares relate to areas where we had processes in place early in the project; we used two squares where such processes are still evolving, and one square to denote those areas in which we would like to institute more formal processes in the near future.

If we were weak on formal software engineering processes, there was positive adherence to a few quite radical quality engineering innovations:

- We held meetings with agendas and minutes; we enjoyed them.
- We did inspections (as described by Fagan<sub>3</sub>) on the test plan scenarios and we remained good friends.
- We introduced one metric (see above) without observing any undesirable side effects or false motivations.
- We made mistakes and we learned from them.
- We held postmortems without blaming anyone for what went wrong.

It should be obvious from the above, that the project took priority over the process and that we made limited progress in terms of an enhanced CMM rating. Where our cultural preferences coincided with the model, we used the appropriate tools. We had very little patience with formalism that promised benefits beyond our deadline.

## Conclusions

The argument of this paper has been that the Y2K crisis is technical in nature and its solution is essentially a technical issue that could best be addressed using the processes already in place. If the culture in place is CMM level 1, then you transition to the year 2000 within that culture. The bottom line is that programs have been built for decades using CMM level 1 processes and it takes time to begin implementing level 2 processes.

With level 1 software there is always an unanswered question "will it work today?" It was an unanticipated insight to realize that by upgrading the software to work with four-digit years throughout, we had made the January 1, 2000 question almost exactly equivalent to the May 17<sup>th</sup>, 1999 question (to take a date at random). There were minor exceptions to this, caused by in-line leap-year and fiscal-year calculations.

Mark Paulk<sub>4</sub> asks why the organization should be interested in using the Software CMM? His answer is irrefutable - “. . . to improve process, with a direct tie to business objectives and a willingness to invest in improvement.” We feel we should make some justification for setting such a low priority on such a laudable goal. Is it that the goal was irrelevant to the health sciences or that it applies only to software in the post FORTRAN era? Our answer is that the inflexibility of our deadline justified the narrow focus. The software was brittle and fragile; the team was strong and competent. We knew where the emphasis had to be; we put the Project ahead of the Process. If we had had the luxury of flexible schedules or abundant resources, we may have benefited from more emphasis on process. If the software had been stronger or the team weaker, then we would have adjusted our priorities accordingly.

Our Y2K Project has shown us that you can achieve technical goals on time and under budget while making a significant improvement in the system functionality and reliability. Ironically, we achieved this by using the same CMM level 1 methods that brought us to the Y2K problem in the first place.

## References

1. The Capability Maturity Model - Carnegie Mellon University, Software Engineering Institute 1993 Papers TR-24 and TR-25.
2. Dijkstra, Edsger W, "Go To Statement Considered Harmful", Communications of ACM Vol. 11, pages 147-148
3. Fagan, Michael E, "Design and Code Inspections to Reduce errors in Program Development", IBM Systems Journal, Vol 15 (1976), pages 182-211.
4. Paulk, Mark, "Using the software CMM in Small Organization"; The Joint 1998 Proceedings of the Pacific Northwest Software Quality Conference and the Eighth International Conference on Software Quality, Portland, Oregon, October 13-14, 1998, pp. 350-361.

**1999 PNSQC Conference 45-minute presentation**

**Paper**

**Title: Planning for Project Surprises - Coping with Risk**

The Process Group  
P.O. Box 700012  
Dallas, TX 75370-0012  
Tel: 972 418-9541 • Fax: 972 618-6283  
E-mail: [help@processgroup.com](mailto:help@processgroup.com)  
[www.processgroup.com](http://www.processgroup.com)

**Abstract**

Whether or not you think your project (software product or process improvement effort) is plagued with problems, you might need effective risk management to keep you out of trouble. Risk management is similar to performing preventive health care and buying insurance for your project. It involves identifying potential problems (risks), analyzing those risks, planning to manage them, and reviewing them.

This presentation offers a practical risk management process. It is simple, effective and takes 60 to 120 minutes to run. The risk process presented can be applied to any type and size of software development project or process improvement project.

The presentation will cover:

- Risk Identification
- Risk Analysis
- Risk Management Plan
- Comprehending Risk in a Project Schedule
- Risk Review
- One-page Summary of the Risk Process

**Biographies**

Mary Sakry and Neil Potter are co-founders of The Process Group, a company that consults in software engineering process improvement. Mary has 23 years and Neil 13 years of experience in software development and management.

From 1988 Mary worked full-time on the Corporate SEPG within Texas Instruments (TI) to lead software process assessments across TI worldwide. The last year of TI was spent working with Neil in an SEPG for one division of TI.

For six years Neil was a Software Engineer in TI. In 1988 he created and managed a full-time SEPG spanning Dallas, England and India.

Neil and Mary have been working on process improvement for 10 years. They are SEI authorized lead assessors for CBA-IPI process and were in the first group of vendors that were authorized by the SEI.

## **1999 PNSQC Conference 45-minute presentation**

### **Paper**

#### **Title: Planning for Project Surprises - Coping with Risk**

The Process Group

Tel: 972 418-9541 • Fax: 972 618-6283

E-mail: [help@processgroup.com](mailto:help@processgroup.com)

Whether or not you think your project (software product or process improvement effort) is plagued with problems, you might need effective risk management to keep you out of trouble. Risk management is similar to performing preventive health care and buying insurance for your project. It involves identifying potential problems (risks), analyzing those risks, planning to manage them, and reviewing them.

Many software development groups run projects without any consideration for the problems that might occur. Risk management is insurance for software projects, and can help reduce your cost and efforts when trouble strikes. It can also help you prevent problems. When risk management techniques are used, you can prevent problems and anticipate others to make the project run smoothly.

#### **When to Perform Risk Management?**

The risk process, described below, should be performed at the beginning of a project, at the beginning of major phases in a project (e.g., requirements, design, coding, and deployment) and when there are significant changes (e.g., feature changes, target platform changes and technology changes).

At the end of this paper we have provided you with a copy of our risk process. It is simple, effective, and takes 90 to 120 minutes for projects that are 12-60 person-months. Projects smaller than 12 person-months take 40-60 minutes. You can control the length of the session by controlling the scope you pick. Most sessions usually take less than two hours.

There are four steps to risk management: risk identification, risk analysis, risk management planning, and risk review.

#### **Risk Identification**

To identify risks, we must first define risk. Risks are potential problems, ones that are not guaranteed to occur. When people begin performing risk identification they often start by listing known problems. Known problems are not risks. During risk identification you might notice some known problems. If so, just move them to a problem list and concentrate on future potential problems.

Risk identification can be done using a brainstorming session. The brainstorm typically takes 15-30 minutes. Be sure to invite anyone who can help you think of risks. Invite the project team, customers, people who have been on similar projects, and experts in the subject area of the project. Limit the group size to nine people. In the brainstorming session, people call out potential problems that they think could hurt the project. New ideas are generated based on the items on the brainstorm list.

During the brainstorm, consider the following items:

- Weak areas such as unknown technology.
  - new, or new to the team (e.g., development tools, target machine.)
- Things that are critical or extremely important to the effort.

- such as the timely delivery of a vendor's database software, creation of translators, or a user interface that meets the customers' needs.
- Things that have caused problems in the past.
  - such as loss of key staff, missed deadlines, or error-prone software.

Example risks are: "We may not have the requirements right," "The technology is untested," "Key people might leave," "The server won't restart in situation X," and "People might resist the change." Any potential problem, or critical project feature, is a good candidate for the risk list.

Once you have created a list, work with the group to clarify each item. Duplicate items can be removed.

### Risk Analysis

The first step in risk analysis (process step 4) is to make each risk item more specific. Risks such as, "Lack of Management buy-in," and "People might leave," are a little ambiguous. In these cases the group might decide to split the risk into smaller specific risks, such as, "Manager Jane decides that the project is not beneficial," "Database expert might leave," and "Webmaster might get pulled off the project."

The next step is to set priorities and determine where to focus risk mitigation efforts. Some of the identified risks are unlikely to occur, and others might not be serious enough to worry about. During the analysis, you will discuss each risk item to understand how devastating it would be if it did occur, and how likely it is to occur. For example, if you had a risk of a key person leaving, you might decide that it would have a large impact on the project, but that it is not very likely.

In the process below we have the group agree on how likely it thinks each risk item is to occur, using a simple scale from 1 to 10 (where 1 is very unlikely and 10 is very likely). The group then rates how serious the impact would be if the risk did occur, using a simple scale from 1 to 10 (where 1 is little impact and 10 is very large). To use this numbering scheme, first pick out the items that rate 1 and 10, respectively. Then rate the other items relative to these boundaries. To determine the priority of each risk item, calculate the product of the two values, likelihood and impact. This priority scheme helps push the big risks to the top of the list, and the small risks to the bottom. An example is given below.

Risk Items (Potential Future Problems derived from the brainstorm)	Likelihood of Risk Item Occurring	Impact to Project If Risk Item Does Occur	Priority (Likelihood x Impact)
New operating system might not be stable	10	10	100
Communication problems over system issues	8	9	72
We may not have the requirements right	9	6	54
Requirements may change late in the cycle	7	7	49
Database S/W might be late	4	8	32
Key people might leave	2	10	20

Now that the group has assigned a priority to each risk, it is ready to select the items to manage. Some projects select a subset to take action upon, while others choose to work on all of the items. To get started, you might select the top 3 risks, or the top 20%, based on the priority calculation.

### Risk Management Planning

There are two things one can do to manage risk. The first is to take action to reduce (or partially reduce) the likelihood of the risk occurring. For example, some projects that work on process improvement make their deadlines earlier and increase their efforts to minimize the likelihood of team members being pulled off the project due to changing organizational priorities. In a software product, a critical feature might be developed first and tested early.

Second, we can take action to reduce the impact if the risk does occur. Sometimes this is an action taken prior to the crisis, such as the creation of a simulator to use for testing if the hardware is late. At other times it is a simple backup plan, such as running a night shift to share hardware.

For the potential loss of a key person, for example, we might do two things: 1) Plan to reduce the impact by making sure other people become familiar with that person's work, or 2) Reduce the likelihood of attrition by giving the person a raise, or by providing daycare.

Here is another example, using the priority scheme described in the process below:

<b>Risk Items (Potential Future Problems derived from the brainstorm)</b>	<b>Likelihood of Risk Item Occurring</b>	<b>Impact to Project If Risk Item Does Occur</b>	<b>Priority (Likelihood x Impact)</b>	<b>Actions to Reduce Likelihood</b>	<b>Actions to Reduce Impact</b>	<b>Who Should Work on Actions</b>	<b>When Should Actions be Complete</b>	<b>Status of Actions</b>
New operating system might not be stable	10	10	100	Test OS more	Ident. 2nd OS	Joe	3/3/99	
Communication problems over system issues	8	9	72	Develop sys. interface document	Add replan milestone	Cathy	5/6/99	
We may not have the requirements right	9	6	54	Build prototype of UI	Limit distribution	Lois	4/6/99	
Requirements may change late in the cycle	7	7	49	Prototype top 10 reqs.	Limit distribution	Cecil	1/2/99	
Database S/W might be late	4	8	32	Check with supplier	Develop a backup plan	Joe	2/2/99	
Key people might leave	2	10	20	Make sure Jim is happy	Earmark Fred as a backup	Pete	3/4/99	

### **Risk Review**

You will want to review your risks periodically so you can check how well mitigation is progressing. You can also see if the risk priorities need to change, or if new risks have been discovered. You might decide to rerun the complete risk process if significant changes have occurred on the project. Significant changes might include the addition of new features, the changing of the target platform, or a change in project team members. Many people incorporate risk review into other regularly scheduled project reviews.

In summary, risk management is the planning for potential problems, and the management of actions taken related to those problems.

## Risk Management Process

### 1. Determine scope of the risk session.

### 2. Select the team and moderator.

The moderator explains the risk process to new team members.

### 3. Identify risks (potential future problems).

- **Brainstorm areas of risk, e.g.,**
  - Weak areas such as unknown technology.
    - new, or new to the team (e.g., development tools, target machine.)
  - Things that are critical or extremely important to the effort.
    - such as the timely delivery of a vendor's database software, creation of translators, or a user interface that meets the customers' needs.
  - Things that have caused problems in the past.
    - such as loss of key staff, missed deadlines, or error-prone software.
- **Remove invalid or irrelevant.**

*Current problems should be treated as problems, not risks.*

### 4. Analyze risks.

- **For each risk item:**
  - Does the team understand the risk item?
    - If necessary, split into separate risk items, e.g.,
      - Disk may overload under condition X.
      - Disk may overload under condition Y.
  - Discuss and determine its scope:
    - What would the consequences be if this risk item did happen?
  - Determine what the **impact** would be if the worst happened, using a scale of one to ten.
  - Determine how **likely** it is that the risk item will occur, using a scale of one to ten.
  - Determine the **priority** of the risk items and thus which to work on (**impact x likelihood**).

### 5. Plan to mitigate risks.

- Select the most important risk issues, such as the top 2 or 3, or top 20%.
- Brainstorm on actions that could be taken to reduce the **likelihood** of the risk item occurring.
- Brainstorm on actions that could be taken to reduce the **impact** if the risk item does occur.
- Decide which actions to pursue.
- Select a person to be responsible for each action chosen.
- Document the information in the risk management plan.

### 6. Review risks.

- **Establish how often risks should be reviewed (once a month is typical).**
  - Risk reviews can be incorporated into existing project status and phase reviews.
- **Update the list based on risk review sessions.**

© 1998, The Process Group, All rights reserved.

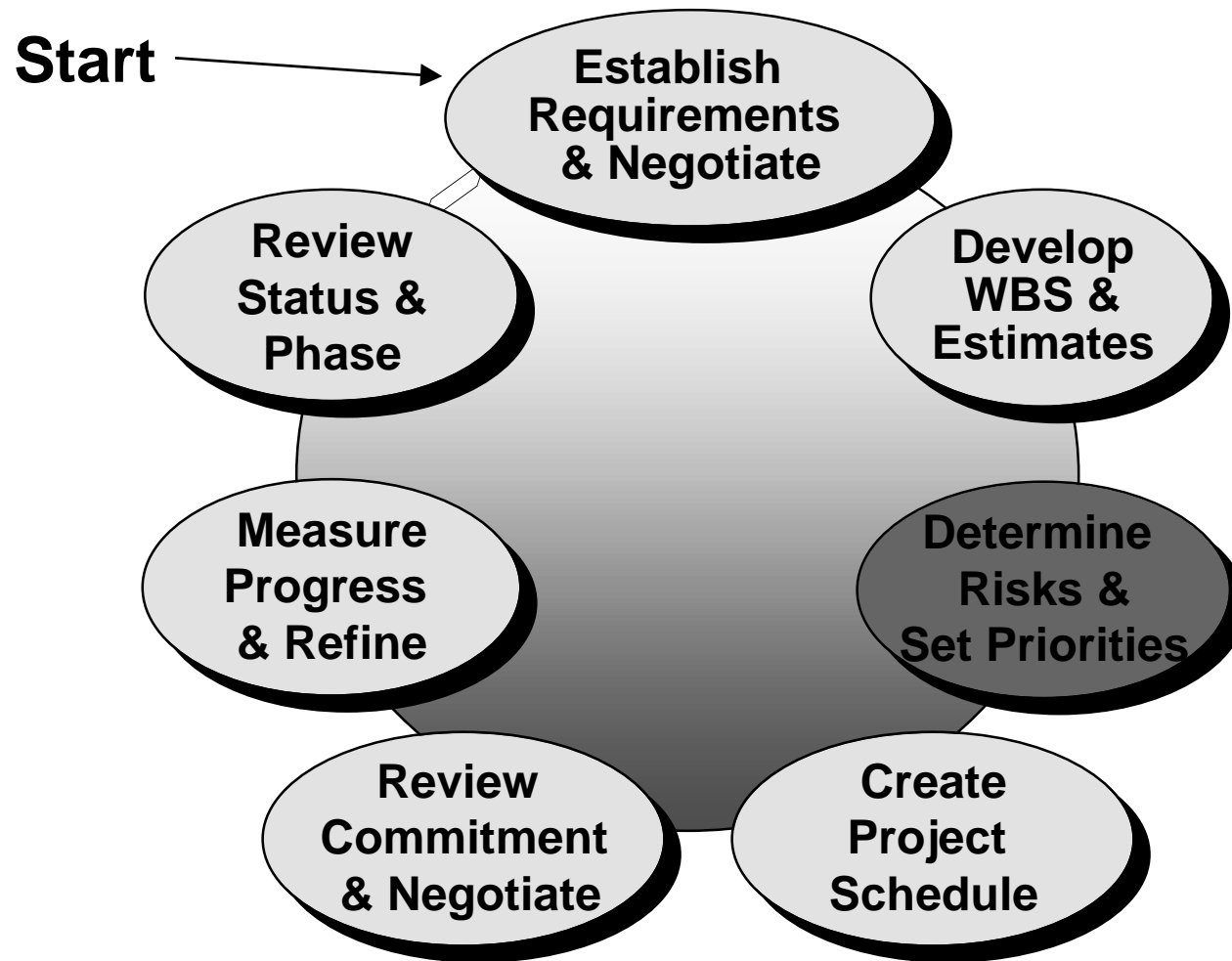
# Planning for Project Surprises - Coping with Risk

Neil Potter  
Mary Sakry

The Process Group  
P.O. Box 700012 • Dallas, TX 75370-0012  
Tel. 972-418-9541 • Fax. 972-618-6283  
E-mail: [help@processgroup.com](mailto:help@processgroup.com)  
<http://www.processgroup.com>



# Determine Risks and Set Priorities

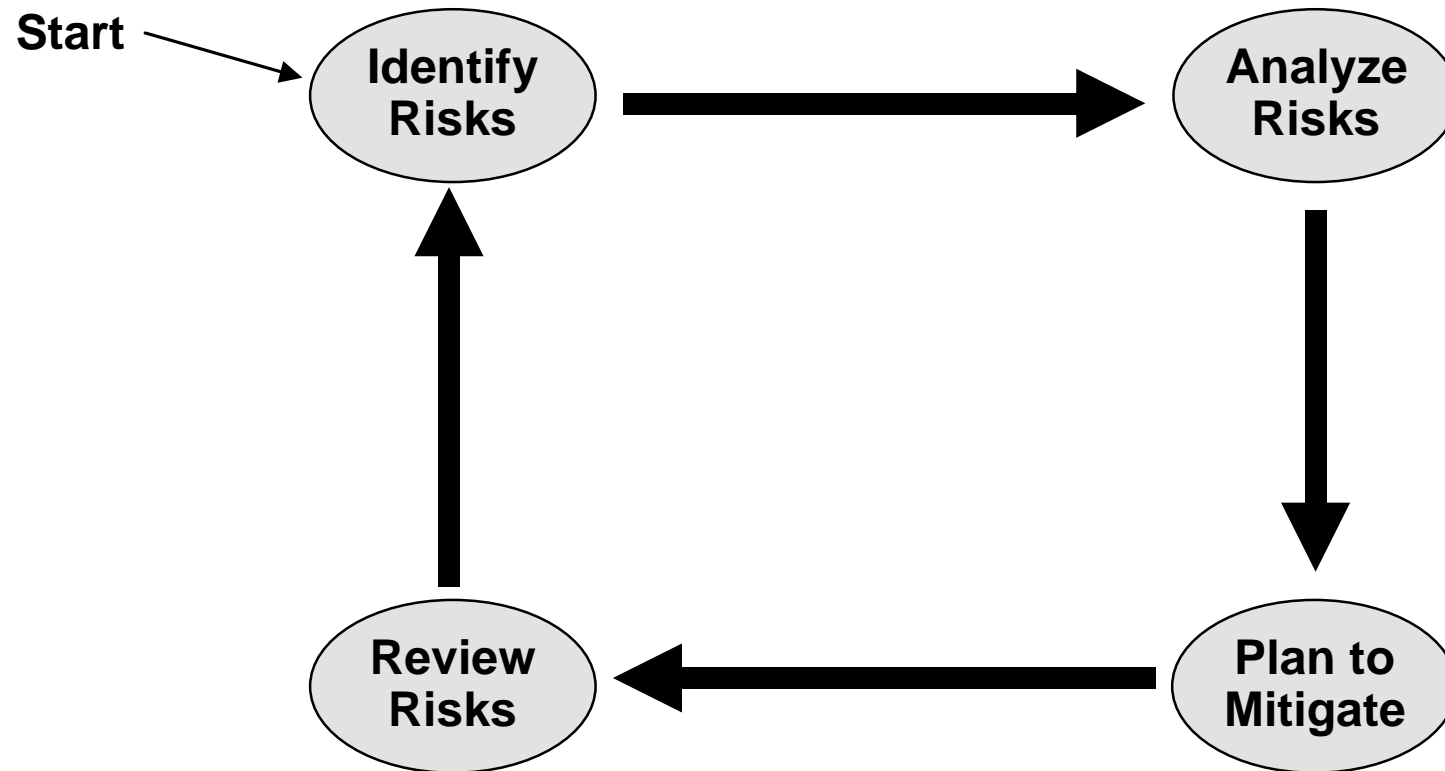


# What is a Risk?



**A risk is anything negative that could happen**

# Risk Management Cycle



**Risks should be identified and managed throughout the project**

# Risk Identification



- **Create a list of potential future problems**

- Risk does not include current problems that are about to happen, that are certain to occur
- Current problems should be tracked on a problem list

## *Suggestions*

- **Project manager creates a list**
- **Brainstorming works well with the project team**
- **Include anyone who may help you think of more risks**
  - People who have done similar projects and tasks
  - Technical experts
  - Customer champions

## What to Look For

- **Weak areas such as unknown technology**
  - new, or new to the team (e.g., development tools, target machine)
- **Things that are critical or extremely important to the effort**
  - such as the timely delivery of a vendor's database software, creation of translators, or a user interface that meets the customers' needs
- **Things that have caused problems in the past**
  - such as loss of key staff, missed deadlines, or error-prone software

# Possible Risk Sources - I

## Product Engineering

### 1. Requirements

- a. Incomplete
- b. Unclear
- c. Unverifiable
- d. Unstable
- e. Infeasible
- f. Unprecedented

### 2. Design and Implementation

- a. Infeasible
- b. Performance
- c. Untestable

### 2. Design and Implementation - continued

- d. Constraints
- e. Documentation

### 3. Integration and Test

- a. Product Integration
- b. System Integration

### 4. Engineering Specialities

- a. Reliability
- b. Safety
- c. Security
- d. Human Factors

Based on material presented at the 1992 Software Engineering Symposium at the SEI at Carnegie Mellon University.

# Possible Risk Sources - II

## Development Environment

1. Work Environment
  - a. Quality Attitude
  - b. Uncooperativeness
2. Development Process
  - a. Informality
  - b. Suitability
  - c. Product Control
  - d. Communications
  - e. Process Control
  - f. Familiarity
  - g. Training

## 3. Development System

- a. Availability
- b. Capacity
- c. Suitability
- d. Usability
- e. Familiarity
- f. Reliability
- g. System Support
- h. Deliverability

Based on material presented at the 1992 Software Engineering Symposium at the SEI at Carnegie Mellon University.

# Possible Risk Sources - III

## Development Environment (cont.)

### 4. Management Process

- a. Planning
- b. Monitoring
- c. Project Organization
- d. Personnel Management
- e. Management Experience
- f. Program Interfaces
- g. Quality Assurance
- h. Configuration Management

## Program Constraints

### 1. Resources

- a. Budget
- b. Schedule
- c. Facilities

### 2. Contract Constraints

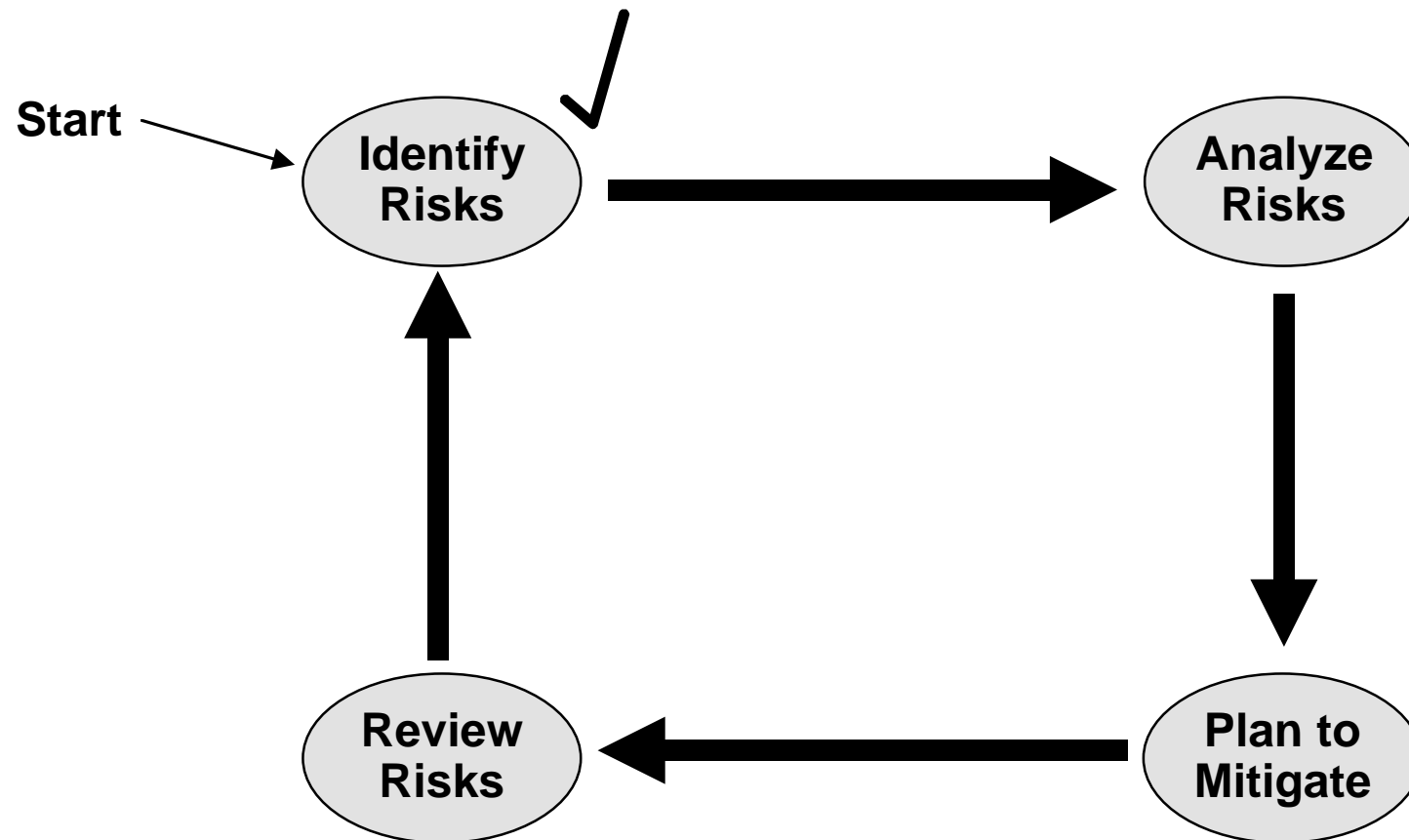
### 3. Externals

- a. Associate Contractors
- b. Vendors
- c. Corporate Management

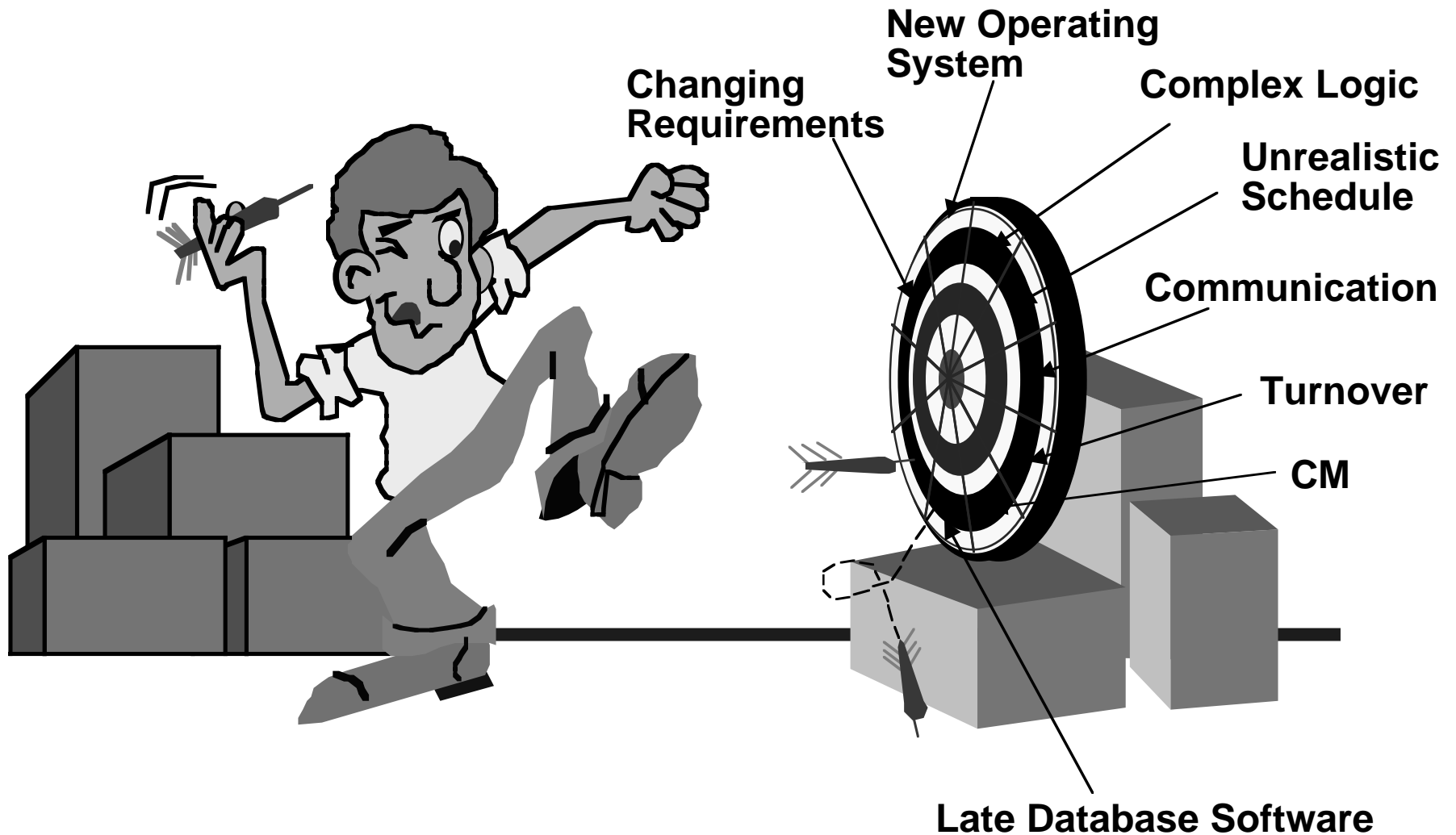
Based on material presented at the 1992 Software Engineering Symposium at the SEI at Carnegie Mellon University.



# Risk Management Cycle



# Selecting Which Risks to Address



# Risk Analysis

## For each risk item:

- Does the team understand this risk item?
  - » If necessary, split into risk items
    - Disk may overload under condition X
    - Disk may overload under condition Y
- Discuss and determine its scope:
  - » What would the consequences be if this risk item did
- Determine what the impact would be if the worst happened, using a scale of one to ten
  - » This could vary by phase or time-frame
- Determine how likely it is that the risk item will
  - a scale of one to ten
- Determine the priority of the risk items using x likelihood

# Risk Analysis Example

## **Risk Item:**

We may not have the requirements right

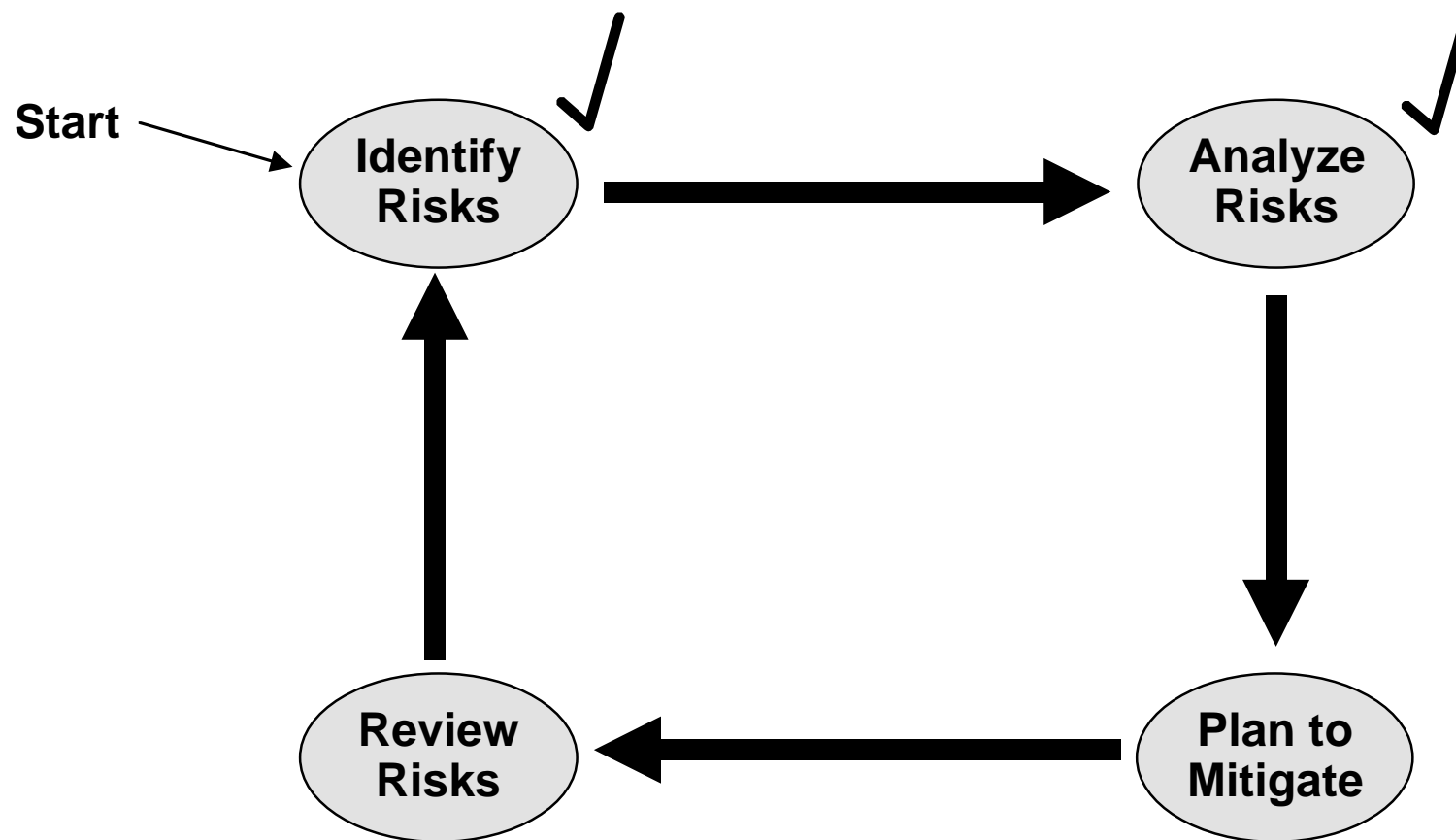
## **Possible consequences if this risk occurs:**

- There are no competitors for this product, so we will just irritate our customers (not lose them)
- We might delay the start up of our next project due to excessive drain on manpower fixing the problems
- We might deliver the wrong product
- We might deliver a product that is not used much

# Setting Priorities Example

Risk item (Potential Future Problem)	Likelihood	Impact	Priority Likelihood x Impact
New operating system might not be stable	10	10	<b>100</b>
Database S/W might be late	4	8	<b>32</b>
Key people might leave	2	10	<b>20</b>
Communication problems - system issues	8	9	<b>72</b>
We may not have the requirements right	9	6	<b>54</b>
Requirements may change late in the cycle	7	7	<b>49</b>

# Risk Management Cycle



# Risk Planning

- Select the most important risk, issues, or the top 2-3, or top 20%
- Brainstorm on actions that could be taken to the likelihood of the risk item occurring (risk mitigation)
  - Make actions specific
- Brainstorm on actions that could be taken to the impact if the risk item does occur (risk contingency planning)
- Decide which actions to pursue
- Select a person responsible for each action chosen
- Document the information in risk management plan

# Risk Planning Example

Risk Item:

We may not have the requirements right

Possible actions to reduce likelihood of risk item:

- Develop a prototype to understand the requirements better before proceeding
- Schedule more time for reviews that include the customer
- Rewrite requirements to be testable

Possible actions to reduce impact of risk item if it does

- Delay the release of the product
- Plan an iterative incremental release to allow for refinement

Responsible person: Joe ~~Think~~ Knows it



# Risk Management Plan Summary

Action items to reduce likelihood

Action items to reduce impact

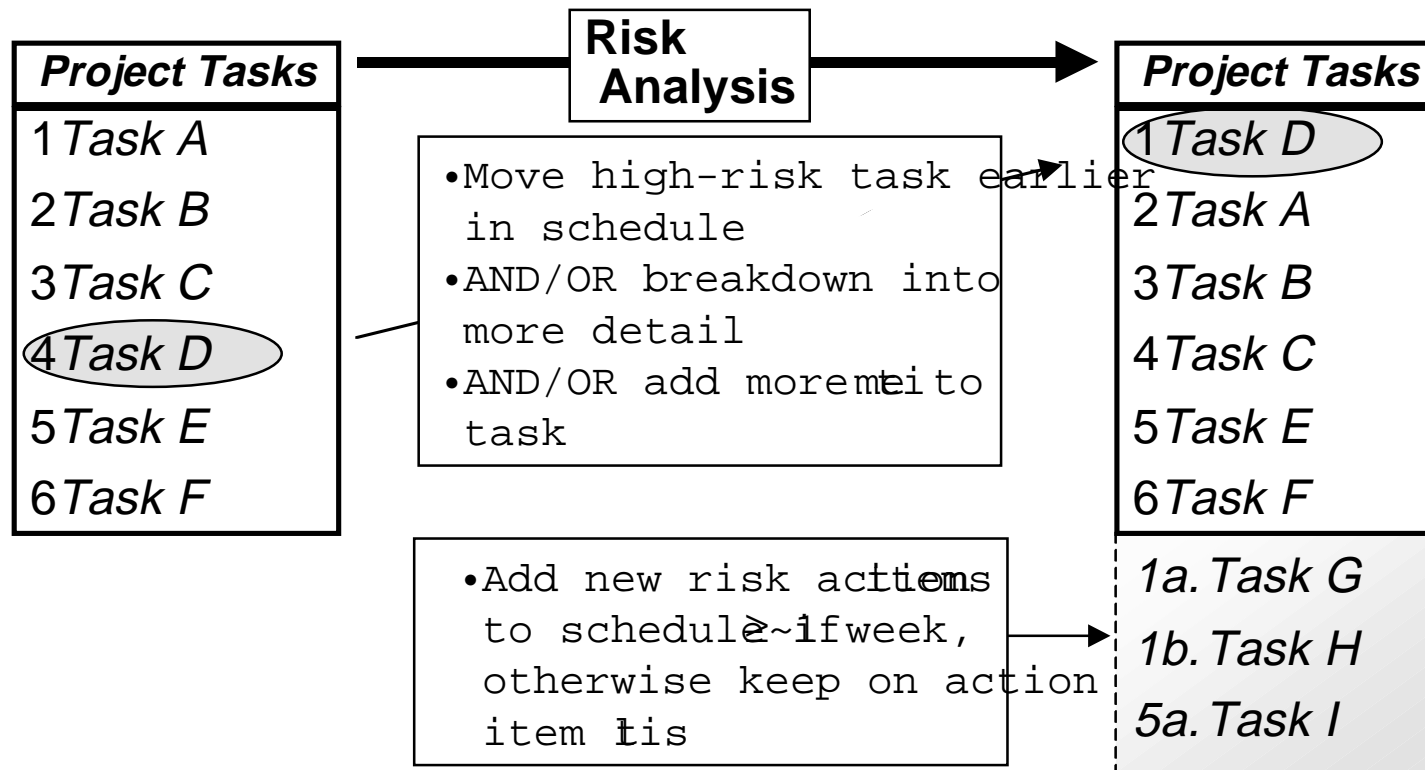
Status of action items

Risk Item (Potential Future Problem)	Lkli.	Imp.	Pri.	Action-Likelihood	Action-Impact	Who	When	Status
New operating system might not be stable	10	10	100	Test OS more	Ident. 2nd OS	Joe	3/3/95	
Communication problems over system issues	8	9	72	Develop sys int. doc	Add replan milestone	Cathy	5/6/95	
We may not have the requirements right	9	6	54	Build prototype of UI	Limit distribution	Lois	4/6/95	
Requirements may change late in the cycle	7	7	49	Prototype top 10 reqs.	Limit distribution	Cecil	1/2/95	
Database S/W might be late	4	8	32	Check with supplier	Dev. plan	Joe	2/2/95	
Key people might leave	2	10	20	Make sure Jim is happy	Earmark Fred	Pete	3/4/95	

Action items will likely be more numerous and detailed

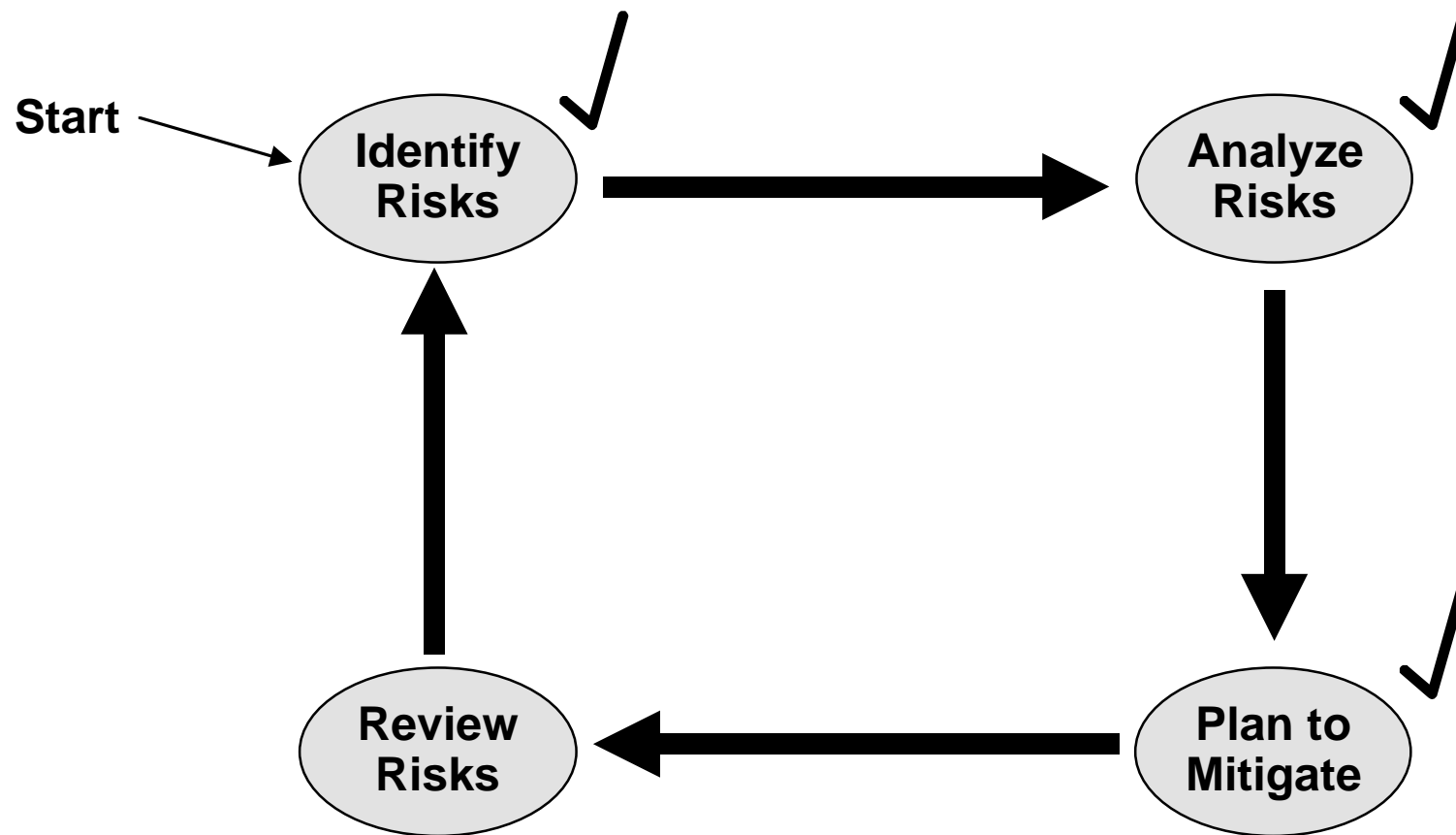
Keep track of the top 20% or top 2-3 risks

# Comprehending Analysis Results in the Project Plan



**The project team should direct its attention to the higher risk, rather than the easier (lower risk) tasks (within customer and schedule constraints)**

# Risk Management Cycle



# Review Risks

## Purpose:

To determine if the identified risks have changed and update the plan accordingly

- **Be sure that** periodic risk reviews **are held to monitor the risks identified**
- **Establish** how often **risks should be reviewed (once a month is typical)**
- **Risk reviews can be incorporated into** existing **project status and phase reviews**
- **Update the list based on risk review sessions**

## Summary

**Risk management helps the project anticipate and plan for problems along the way. It includes identification, analysis, planning and review.**

# Software Risk Management

Linda Westfall

The Westfall Team

westfall@idt.net

PMB 383, 3000 Custer Road, Suite 270

Plano, TX 75075

972-867-1172 (voice)

972-943-1484 (fax)

**Abstract:** This paper reviews the basic concepts, terminology, and techniques of Software Risk Management. It teaches readers how to identify and analyze software risks on their projects. Readers then learn techniques for planning and acting to mitigate risks so that the overall impact of those risks on their projects is minimized.

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

Very active professionally, Linda Westfall is Chair Elect of the American Society for Quality (ASQ) Software Division. She has also served as the Software Division's Program Chair and Certification Chair, and on the ASQ National Certification Board. Linda wrote the Software Metrics and Software Project Management sections of the ASQ Software Quality Engineering course and co-authored the ASQ Software Metrics course.

**Key Words/Phrases:** Software Project Management, Risk Management, Risk Identification, and Risk Analysis.

# Software Risk Management

## Defining Software Risk Management

There are many risks involved in creating high quality software on time and within budget. However, in order for it to be worthwhile to take these risks, they must be compensated for by a perceived reward. The greater the risk, the greater the reward must be to make it worthwhile to take the chance. In software development, the possibility of reward is high, but so is the potential for disaster. The need for software risk management is illustrated in Gilb's risk principle. "If you don't actively attack the risks, they will actively attack you" [Gilb-88]. In order to successfully manage a software project and reap our rewards, we must learn to identify, analyze, and control these risks. This paper focuses on the basic concepts, processes, and techniques of software risk management.

There are basic risks that are generic to almost all software projects. Although there is a basic component of risk management inherent in good project management, risk management differs from project management in the following ways:

### Project Management

Designed to address general or generic risks

Looks at the big picture and plans for details

Plans what should happen and looks for ways to make it happen

Plans for success

### Risk Management

Designed to focus on risks unique to each project

Looks at potential problems and plans for contingencies

Evaluates what could happen and looks for ways to minimize the damage

Plans to manage and mitigate potential causes of failure

Within risk management the "emphasis is shifted from crisis management to anticipatory management" [Down-94]. Boehm defines four major reasons for implementing software risk management [Boehm-89]:

- Avoiding software project disasters, including run away budgets and schedules, defect-ridden software products, and operational failures.
- Avoiding rework caused by erroneous, missing, or ambiguous requirements, design or code, which typically consumes 40-50% of the total cost of software development.
- Avoiding overkill with detection and prevention techniques in areas of minimal or no risk.
- Stimulating a win-win software solution where the customer receives the product they need and the vendor makes the profits they expect.

## Defining Risk

So, what are risks? Risks are simply potential problems. For example, every time we cross the street, we run the risk of being hit by a car. The risk does not start until we make the commitment, until we step in the street. It ends when the problem occurs (the car hits us) or the possibility of risk is eliminated (we safely step onto the sidewalk of the other side of the street).

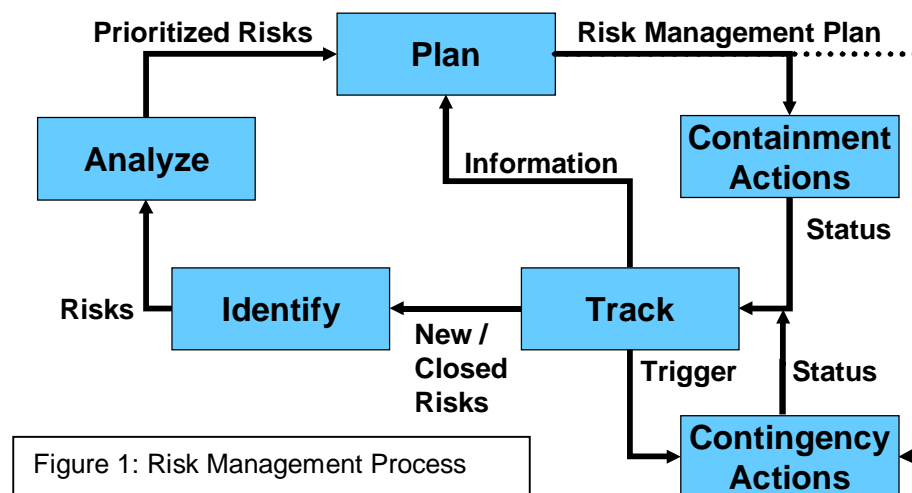
A software project may encounter various types of risks:

- **Technical risks** include problems with languages, project size, project functionality, platforms, methods, standards, or processes. These risks may result from excessive constraints, lack of experience, poorly defined parameters, or dependencies on organizations outside the direct control of the project team.
- **Management risks** include lack of planning, lack of management experience and training, communications problems, organizational issues, lack of authority, and control problems.

- **Financial risks** include cash flow, capital and budgetary issues, and return on investment constraints.
- **Contractual and legal risks** include changing requirements, market-driven schedules, health & safety issues, government regulation, and product warranty issues.
- **Personnel risks** include staffing lags, experience and training problems, ethical and moral issues, staff conflicts, and productivity issues.
- Other resource risks include unavailability or late delivery of equipment & supplies, inadequate tools, inadequate facilities, distributed locations, unavailability of computer resources, and slow response times.

## Risk Management Process

Figure 1 illustrates the risk management process. This process starts with the identification of a list of potential risks. Each of these risks is then analyzed and prioritized. A risk management plan is created that identifies containment actions that will reduce the probability of the risk occurring and/or reduce the impact if the risk turns into a problem. The plan also includes contingency actions that will be taken if the risk turns into a problem and the associated triggers (indicators that the risk is turning into a problem). The containment part of the plan is then implemented and actions are taken. The tracking step involves monitoring the status of known risks as well as the results of risk reduction actions. If a trigger indicates the onset of a problem, the corresponding contingency plans are implemented. As new status and information are obtained, the risk management plans are updated accordingly. Tracking may also result in the addition of newly identified risks or in the closure of known risks.



The risk management process is an on-going part of managing the software development process. It is designed to be a continuous feedback loop where additional information and risk status are utilized to refine the project's risk list and risk management plans.

Let's use the crossing the street analogy to examine the risk management process. First we identify the risk: we want to cross the street and know there is a possibility of traffic. We analyze the risk. What is the probability of being hit by the car? How much is it going to hurt if we are hit? How important is it that we cross this street at this time? We look both ways, we see the on-coming car, and we judge its rate of speed. We form a plan to reduce the risk and decide to wait until the car has passed. We implement the plan and wait. We track the situation by watching the car and we see it pull into a driveway. We change our plan and proceed across the street. We step onto the curb across the street and stop thinking about crossing the street (i.e., we close the risk).



## Risk Identification

During the first step in the software risk management process, risks are identified and added to the list of known risks. The output of this step is a list of project-specific risks that have the potential of compromising the project's success. There are many techniques for identifying risks, including interviewing, reporting, decomposition, assumption analysis, critical path analysis, and utilization of risk taxonomies.

One technique for identifying risks is interviewing or brainstorming with project personnel, customers, and vendors. Open-ended questions such as the following can help identify potential areas of risk.

- What new or improved technologies does this project implement?
- What interfaces issues still need to be defined?
- What requirements exist that we aren't sure how to implement?
- What concerns do we have about our ability to meet required the quality and performance levels?

Another risk identification technique is voluntary reporting, where any individual who identifies a risk is encouraged and rewarded for bringing that risk management's attention. This requires the complete elimination of the "shoot the messenger" syndrome. It avoids the temptation to assign risk reduction actions to the person who identified the risk. Risks can also be identified through required reporting mechanisms such as status reports or project reviews.

As the product is being decomposed during the requirements and design phases, another opportunity exists for risk identifications. Every TBD ("To Be Done/Determined") is a potential risk. As Ould states, "The most important thing about planning is writing down what you *don't know*, because what you don't know is what you must find out" [Ould-90]. Decomposition in the form of work breakdown structures during project planning can also help identify areas of uncertainty that may need to be recorded as risks.

Process and product assumptions must be analyzed. For example, we might assume the hardware will be available by the system test date or three additional experienced C++ programmers will be hired by the time coding starts. If these assumptions prove to be false, we could have major problems.

As we perform critical path analysis for our project plan, we must remain on the alert to identify risks. Any possibility of schedule slippage on the critical path must be considered a risk because it directly impacts our ability to meet schedule.

Risk taxonomies are lists of problems that have occurred on other projects and can be used as checklists to help ensure all potential risks have been considered. An example of a risk taxonomy can be found in the Software Engineering Institute's Taxonomy-Based Risk Identification report that covers 13 major risk areas with about 200 questions [SEI-93].

## Risk Analysis

During the risk analysis step, each risk is examined to determine:

- Likelihood: the probability that the risk will result in a loss
- Impact: the size or cost of that loss if the risk turns into a problem
- Timeframe: when the risk needs to be addressed (i.e., risk associated with activities in the near future would have a higher priority than similar risks in later activities)

Additionally, the interrelationships between risks are examined to determine if compounding risk conditions magnify losses.

The list of risks is then prioritized based on this assessment. Since resource limitations rarely allow the consideration of all risks, the prioritized list of risks is used to identify risks requiring additional planning and action. Other risks are documented and tracked for possible future consideration. Based on

changing conditions, additional information, the identification of new risk items, or the closure of existing risks, the list of risks requiring additional planning and action may require periodic updates.

The following is an example of risk analysis. During our analysis, we determine that there is a 30% probability the Test Bed will be available one week later than schedules and a 10% probability it will be a month late. If the Test Bed is one week late, the testers can use their time productively by using the simulators to test other aspects of the software (loss = \$0). The simulator can be utilized for up to two weeks. However, if the Test Bed delivery is one month late, there are not enough productive activities to balance the loss. Losses include unproductive testers for two weeks, overtime later, morale problems, and delays in finding defects for a total estimated loss of \$100,000. In addition to the dollar loss, the testing is on the critical path and not all of the lost testing time can be made up in overtime (loss estimated at two week schedule slippage).

Boehm defines the Risk Exposure equation to help quantitatively establish risk priorities [Boehm-89]. Risk Exposure measures the impact of a risk in terms of the expected value of the loss. Risk Exposure (RE) is defined as the probability of an undesired outcome times the expected loss if that outcome occurs.

$RE = \text{Probability(UO)} * \text{Loss (UO)}$ , where UO = Unexpected outcome

Given the example above, the Risk Exposure is  $10\% \times \$100,000 = \$10,000$  and  $10\% \times 2 \text{ calendar week} = 0.2 \text{ calendar week}$ . Comparing the Risk Exposure measurement for various risks can help identify those risks with the greatest probable negative impact to the project or product and thus help establish which risks are candidates for further action.

## Risk Management Planning

Taking the prioritized risk list as input, plans are developed for the risks chosen for action. Specific questions can be asked to help focus the type of planning required.

*Is it too big a risk?* If the risk is too big for us to be willing to accept, we can avoid the risk by changing our project strategies and tactics to choose a less risky alternate or we may decide not to do the project at all. For example, if our project has tight schedule constraints and includes state of the art technology, we may decide to wait until a future project to implement our newly purchased CASE tools. Things to remember about avoiding risks include:

- Avoiding risks may also mean avoiding opportunities
- Not all risks can be avoided
- Avoiding a risk in one part of the project may create risks in other parts of the project.

*Do we know enough?* If we don't know enough, we can plan to "buy" additional information through mechanisms such as prototyping, modeling, simulation, or conducting additional research.

*Can we transfer the risk?* If it is not our risk or if it is economically feasible to pay someone else to assume all or part of the risk, we can plan to transfer the risk to another organization. An example of this is to contract with a disaster recovery firm to provide backup computer facilities that will allow continuation of the project in case a fire or other disaster destroyed the project's work environment.

*Should action be taken now?* If immediate action should not or cannot be taken, or is not economically feasible, we can plan to live with the risk, establish a trigger, and monitor that trigger for indications that future action is needed. A trigger is a time or event in the future that is the earliest indication that the risk will turn into a problem. For example, if there is a risk that outsourced software will not be delivered on schedule, the trigger could be whether the critical design review was held on schedule. A trigger can also include relative variance or threshold metrics. For example, if the risk is the availability of key personnel for the coding phase, the trigger could be a relative variance of more than 20% between actual and planned staffing levels. Risks that are assigned triggers can then be set at a "monitor only" priority until the trigger occurs. At that time, the risk analysis step should be repeated to determine if risk reduction action is needed.

There is a trade-off in utilizing triggers in risk management. We want to set the trigger as early as possible in order to ensure that there is plenty of time to implement risk reduction actions. We also want to set the trigger as late as possible because the longer we wait, the more information we have to make a correct decision.

If we decide to attack the risk directly, we typically start with creating a list of possible risk reduction actions that can be taken for the risk. Two major types of risk reduction actions should be considered. There are actions that reduce the likelihood that the risk will occur. There are also actions that reduce the impact of the risk should it occur. These may include immediate actions such as establishing a liaison with the customer to insure adequate communications, conducting a performance simulation, or buying additional equipment for the test bed to duplicate the operational environment. In other cases, they may include contingency plans like a disaster recovery plan, contacting a consulting firm to establish a fallback position if key personnel are not available, or selecting an alternative design approach if the new technology is not delivered as promised.

From the list of possible risk reduction actions, we must select those that we are actually going to implement. When considering which risk reduction activities to select, a cost/benefit analysis must be performed. Boehm defines the Risk Reduction Leverage equation to help quantitatively establish the cost/benefit of implementing a risk reduction action. [Boehm-89]. Risk Reduction Leverage measures the return on investment of the available risk reduction techniques. Risk Reduction Leverage (RRL) is defined as the difference between the Risk Exposure before and after the reduction activity divided by the cost of that activity.

$$RRL = (RE_{\text{before}} - RE_{\text{after}}) / \text{Risk Reduction Cost}$$

If the Risk Reduction Leverage is less than one, it means that the cost of the risk reduction activity outweighs the probable gain from implementing the action.

Each selected action in the risk reduction plan must include a description of the action and a list of tasks with assigned responsibilities and due dates. These actions must be integrated into the project plan with effort and cost estimations and schedules being adjusted to include the new actions. For example, new tasks such as creating prototypes or doing research must be included in the plan

## **Taking Action**

During the action step, we implement the risk reduction plan. Individuals execute their assigned tasks. Project effort estimations are adjusted to take into consideration additional effort needed to perform risk reduction activities or to account for projected additional effort if the risk turns into a problem. Some tasks may be moved forward in the schedule to ensure adequate time to deal with problems if they occur. Other tasks may be moved back in the schedule to allow time for additional information to be obtained. Budgets must also be adjusted to consider risk reduction activities.

## **Tracking**

Results and impacts of the risk reduction implementation must be tracked. The tracking step involves gathering data, compiling that data into information, and then reporting and analyzing that information. This includes measuring known risks and monitoring triggers, as well as measuring the impacts of risk reduction activities. The results of the tracking can be:

- Identification of new risks that need to be added to the risk list.
- Validation of known risk resolutions so risks can be removed from the risk list because they are no longer a threat to project success.
- Information that dictates additional planning requirements
- Implementation of contingency plan

Many of the software metrics typically used to manage software projects can also be used to track risks. For example, Gantt charts, earned value measures, and budget and resource metrics can help identify and track risks involving variances between plans and actual performance. Requirements churn, defect

identification rates, and defect backlogs can be used to track rework risks, risks to the quality of the delivered product, and even schedule risks.

## Conclusions

With ever-increasing complexity and increasing demand for bigger, better, and faster, the software industry is a high-risk business. When teams don't manage risk, they leave projects vulnerable to factors that can cause major rework, major cost or schedule over-runs, or complete project failure. Adopting a Software Risk Management Program is a step every software manager can take to more effectively manage software development initiatives. Risk management is an ongoing process that is implemented as part of the initial project planning activities and utilized throughout all of the phases of the software development lifecycle. Risk management requires a fear-free environment where risks can be identified and discussed openly. Based on a positive, proactive approach, risk management can greatly reduce or even eliminate the need for crisis management within our software projects.

## References

- [Boehm-89] Barry W. Boehm, *Tutorial: Software Risk Management*, Les Alamitos, CA, IEEE Computer Society, 1989.
- [Gilb-88] Tom Gilb, *Principles of Software Engineering Management*, Wokingham, England: Addison-Wesley, 1988.
- [Ould-90] Martyn Ould, *Strategies For Software Engineering: The Management of Risk and Quality*, Chichester, England, John Wiley & Sons, 1990.
- [SEI-93] Marvin J. Carr, Suresh L. Konda, Ira Monarch, F. Carol Ulrich, Clay F. Walker, *Taxonomy-Based Risk Identification*, CMU/SEI-93-TR-006, Pittsburgh, PA, Software Engineering Institute, 1993.

# **Solving the Software Quality Management Problem: The Next Step**

James L. Mater  
asLasm  
rtamen

## **ABSTRACT**

This paper is based on direct experience with the software quality management problems of hundreds of software, systems and information technology companies and organizations. We identify the requirements for successful software quality management and the evolution of the models for this task. We have developed a new model which allows the profit and loss manager to overcome the inherent problems in managing this function while reducing his expenses for quality management and the costs of poor quality. The conclusions and recommended actions in this paper move the industry thinking forward on this vital issue.

## **Table of Contents**

ABSTRACT .....	2
Table of Contents .....	2
Introduction .....	2
Solving the Problem: Software Quality Policy .....	3
Solving the Problem: Monitoring and Enforcing Quality Policy .....	4
Management of the Quality Function: The Industry's Weak Link .....	5
Successful Software Quality Management .....	8
The Evolution of Software Quality Management .....	9
The Future of Software Quality Management .....	11
Conclusions .....	12

## **Introduction**

In our capacity as an independent software testing lab we have worked with hundreds of different software and systems companies and with the information systems groups of small and large companies. Our work provides us with a unique opportunity to observe the struggles organizations go through in attempting to solve their software product quality and quality management problems. This paper presents our observations and thinking about the basic quality management problem and a new solution for the industry.

There is a lack of clear understanding and value placed by the profit and loss (P&L) manager<sup>1</sup> on software quality management. Software projects and products (and companies) fail to deliver quality products because quality management is not treated as a strategic, critical aspect of the product development process equal to requirements, design and code development. The basic problem is that software quality management is not properly “owned” within organizations. Software quality management historically has been delegated to a software quality assurance function that is considered a technical function - one that few P&L managers would ever consider that they should “own” directly. Traditional industrial business philosophy treats product quality management, quality assurance, and quality control functions as major corporate functions, reporting to the P&L manager. However, this approach has not yet been adopted by most software organizations. The software business is such a recent “discipline” that the issue of product quality management<sup>2</sup> remains a mystery, especially to a P&L manager without software training or experience.

The software quality problem –i.e., the quality of software products delivered to customers - is not a technical one. It is achieved through a combination of good customer understanding (requirements) and good product development processes.

Any number of good software development processes and techniques are proven and available. The industry knows how to build good quality products that meet the feature, cost, and schedule needs of customers; that are easily maintained and upgraded; and that are reliable. Any claim that we need better processes or better tools to solve the quality problem is a myth and skirts the real issue of P&L manager responsibility.

What the industry typically doesn’t have is a combination of discipline and organizational structure required for consistently delivering successful products – i.e., a well defined, well executed quality management function. Responsibility for this must start at the P&L manager level. Until The P&L manager thinks long and hard about the quality requirements for his products; until he has clearly communicated his conclusions; until he actively monitors product quality; and until he is willing to act on that information to enforce his policy, this problem will remain unsolved.

### **Solving the Problem: Software Quality Policy**

The P&L manager has two critical responsibilities relative to software quality. First, he has the responsibility to set and communicate clear policy and to empower his resources to carry out that policy. And second, he has the responsibility to insure that his policies are implemented.

---

<sup>1</sup> We use the term P&L manager to refer to the executive ultimately responsible for both the revenue and expenses for the product organization. In larger companies this is likely to be a Division General Manager or President. In smaller companies it is likely to be the CEO or President.

<sup>2</sup> Product quality management consists of the quality management function (insuring good quality policies are in place and enforced), the quality assurance function (developing and implementing practices and processes that insure quality products are produced), and the quality control function (actual testing of products to insure conformance to customer requirements).

He needs to monitor quality on an on-going basis and take action as needed to keep the organization on track.

The P&L manager must give serious thought to software quality policy. Is the policy to be first to market with the right features at the right price and fix reliability issues later? Is it to have the most reliable product available in its class? Is it to aim at the low end of the market that will accept poorer quality at a lower price? Are there critical safety or customer issues that demand perfection – 100% reliability (for example, medical instruments, defense systems, avionics components, etc.)? Is the company committed to a zero defect policy?

This is not a task that can be delegated. Only the P&L manager can set this policy because everyone else in the organization will orient his or her (individual) quality goals and the design / implementation of the software to this policy. In our view, the policy should reflect serious consideration and commitment and carry the weight of the P&L manager. It should have lasting value and it should be unambiguous to those implementing policy.

Is there really a value in thinking about and articulating such a policy? We think the answer is absolutely YES. The cause of all quality problems in the software industry is the lack of clear direction from the P&L manager and the will to enforce such policy. How can a P&L manager hold his team accountable for meeting quality standards if these are unstated? How does a decision to ship a product get made when there are no clear criteria for making such a decision? How can a development team be disciplined for causing exorbitant support headaches when no one ever directed that minimizing support costs was a critical issue at the time of software design? How can a product manager set quality goals for a product when the standard corporate policy seems to vary from day to day and product to product?

Before enforcement of the right policy can happen, the right policy must be set and articulated. A P&L manager that does not step up to this issue is negligent in leading his organization.

### **Solving the Problem: Monitoring and Enforcing Quality Policy**

Once a quality policy has been put in place, the second major issue is monitoring product quality to insure that the policy is carried out. For the P&L manager this means establishing a good quality management function that provides him and his organization with good information about the quality of the products under development and enforces his quality policies. If the P&L manager only finds out after a product has shipped that customers are dissatisfied, then his policies and the enforcement of them have failed.

The proactive P&L manager needs to answer the question: will the products in development be delivered on time, on budget and with the quality required to succeed in the market? If the P&L manager is unusual and has put in place the right organization and people with the right direction, then the problem is very simple to solve. He need merely ask for the information and it will be available in some form which gives an accurate view of the quality of products in development.

For most organizations and P&L managers, this is a dream. There may be a test team in place to measure product quality but it is probably buried in the development organization and has inexperienced staff reporting to an inexperienced test manager. The right information seldom reaches the right people in time. There is no quality management as an independent

function but only a lower level quality control function (the test team) that has minimal understanding of corporate quality policy and issues.

What the P&L manager needs is a quality management team that:

- is independent of the development team;
- is empowered with the authority of the P&L manager;
- is working with the product on a day-to-day basis;
- has the skills to thoroughly evaluate the product against explicit or implicit criteria and can ferret out the evaluation criteria<sup>3</sup> from whatever internal sources are available (or raise a flag if adequate product requirements do not exist);
- can professionally<sup>4</sup> provide documented information to both the development team and the P&L manager;
- clearly understands the P&L manager's business problem and is assisting in solving these above all else;
- operates very efficiently and effectively.

After years of working with various software organizations we have come to believe that it is difficult, if not impossible, for a business organization to put this definition in place internally.

### **Management of the Quality Function: The Industry's Weak Link**

The management of product quality is the executive function that takes ownership of the process for delivering products of the quality required by the market. The function starts with a good understanding of the requirements (good product requirements), moves to a development process which is designed to deliver predictable results based on the requirements; and ends with a quality control process (testing) which validates that the product indeed meets the defined requirements. The development process needs to include explicit quality assurance steps to succeed. Most company executives concentrate on the requirements and development aspects and treat the quality assurance activities as an afterthought.

In most organizations that we have worked with there is not a designated quality management function. Some have a software test department. Some have a quality assurance department referred to as software QA that is really a software test group. Invariably, and despite protests to the contrary, this department (we'll refer to it as software QA since this is the

---

<sup>3</sup> Most organizations call these criteria requirements. These are the specifications that the organization believes a product must meet in order to satisfy a customer need.

<sup>4</sup> By "professionally" we mean that the team provides information in a form, at a time, and in a way that is perceived as non-threatening, objective and valuable. There is no appearance of a hidden bias or agenda. In short, the test team is respected and listened to by all parties. This is not usually the case with test teams.



most popular title we see in the industry) is often the weak link in the chain. The symptoms of this weakness are manifested in companies in various ways:

- ***The software quality assurance function itself is typically a "hot potato" that no senior manager wants to own.*** It is moved around from engineering to manufacturing to operations and back to engineering. It seesaws between a centralized and decentralized function every couple of years. Two companies we recently interviewed had just finished dissolving the central QA function and deploying the engineers to the product teams. In both cases, a great deal of disruption was required. In both cases this comes after deciding that the central function was not working after 2-3 years of effort to make it an effective business tool. In another case, the VP who had been “given” QA was all too happy to hand it off to us.
- ***There is discontinuity in the management of the QA function itself.*** It is difficult to find and keep a good manager in software testing or software QA. Managers move out of the function if they are really good (often hired away for more money) or fired if they are ineffective. In any case, it is rare to find stable management of the software QA or test function.
- ***There is no encouragement (and it rarely happens) that highly respected developer move to software QA.*** In fact, the opposite is true. Many companies are proud of the fact that they can use software QA as an entry point and training ground for development. The most attractive career path available to the QA engineer is to move to development. One of our major customers has a terrible time keeping good test leads. They are hired right out of college and have been screened for good development skills. As soon as they become effective test leads they are moved into development. This works well for the development organization but leaves software QA with a continually inexperienced staff.
- ***There is a constant turnover in QA staff.*** The consequence is that the QA organization never matures to the same level of skill and professionalism as the development teams. It is not uncommon for a company to be proud of the fact that they have had a stable QA organization for one or two years. Yet if they looked at the stability and maturity of the development team (typically five years or more) this pride should evaporate against the recognition that the QA team is not even close to adequate for the task.
- ***The use of developers as testers.*** A major customer of ours contacted us recently needing help with a critical project – their division management had just fired all of the QA engineers in an attempt to fix the quality problem. The company’s ISO9000 model stated that developers should actually do all of the quality assurance and final acceptance testing themselves but this group just didn’t have the bandwidth. Although we believe that developers need to own the quality of their work and should conduct certain quality assurance activities (unit testing, peer reviews, etc.), they should not be the final testers on a product. Not only are developers usually not motivated nor particularly competent as final product testers, the opportunity cost of pulling them off of development work is staggering if a company analyzes the real costs of doing so.
- ***The development engineers successfully lay the blame for quality/schedule/feature problems on software QA.*** The weak link is a test or QA team that is not able to advocate effectively for its own position - they get dumped on over and over again. In one major

company we know a debate is underway about how to fix exactly this problem. There is an excellent QA team that does system test but works under the VP of Engineering. Because it is part of engineering, the QA team tends to relieve the development teams of having to pass all of the entry criteria before acceptance for system test. Of course QA ends up being blamed when the ship date slips – this is very typical and easily solvable if the P&L manager establishes clear accountability for both development and QA functions, and establishes a quality management function to enforce policy.

- ***The QA team is not able to communicate product quality information to decision-makers (the P&L manager).*** The team may not have the experience required to decide when information is critical to the P&L manager. Or the team's information may be filtered through the current owner (usually a VP of Development or Engineering). The resulting information serves the VP but not the P&L manager.
- ***Ship dates are frequently delayed and the delays are surprises (at first) to everyone except to the developers and the testers.*** The testers did not try to, or were unsuccessful in, making the information available to the P&L manager.
- ***Product design and/or features are routinely changed causing schedule slips and expensive rework and retest before release.*** Major design or feature changes are accepted by management because the basic process discipline was not controlled from a quality perspective - e.g., no one enforced one of the early steps of requirements verification or design review and the impact on quality control activities was ignored in the decision process. This happens more often when there is not an adequate quality management function in place.

These problems all result because the P&L manager is not making an adequate investment in quality management. (He also is not willing to insist on accountability by his development group.) In many cases, the definition of "adequate" is not understood and quality management is under-funded. Since quality in software is treated as an engineering function that no one really wants to own, it is no wonder that software QA people are treated inadequately:

- Software test and/or QA engineering positions are entry-level positions used as training grounds for development. This perpetuates the weakness in these organizations since the best are routinely migrated to development. It is very difficult to mature an organization when all of its members are entry-level and intent on moving to development.
- Software test and/or QA engineers are treated like second-class citizens. They are not as good as developers because of a bias that says "he's not good enough to code so he tests" or "those who can write code; those who can't test."
- Software test and/or QA engineers are poorly paid relative to development engineers.
- There is little or no emphasis placed on a career path for software test and/or QA engineers. The test or QA engineer does not have nearly the same opportunity to rise in grade and pay as a development engineer.
- Budget decisions favor development. If both QA and development are asking for tool sets for their functions (and the company can't afford both) development usually wins.
- Management is willing to let test or QA suffer because development slipped their schedule.

All of these problems and indicators are caused by the lack of clear understanding and value placed by the P&L manager on the software quality functions. We have come to label this set of problems as cultural and management challenges facing the P&L manager.

### **Successful Software Quality Management**

Solving this problem is simple: the P&L manager needs to have a clear understanding of the quality requirements of his products, be willing to make appropriate strategic decisions about them, and then put in place a quality management function. Historically, this has meant funding an independent software quality management group that did not report to engineering and insisting on disciplined behavior in the whole process (using this group as a measurement and control mechanism). It may also have meant having an executive level VP, Director or Manager of Quality reporting directly to the P&L manager with adequate budget, experience and power to enforce quality disciplines and act as a gate for product release cycles.

A popular approach to this problem is integrating the quality functions into development teams with senior quality people and establishing a clear, appropriate process for control of quality during development. This can at times improve the ability to develop high quality product on time and within budget but it doesn't solve the problem of how to provide an objective, independent view of product quality to the P&L manager.

A strong P&L manager can have the quality function (usually just a test group) report to him directly. He can hire a VP of Quality to work directly for him and manage the test function. He can ensure that his development VP is someone who shares his view of the importance of the product quality management and the need for an independent quality function. Any of these approaches can be made to work.

In the end, the P&L manager must spend a significant amount of effort and dollars to develop a strong QA organization. I recently spoke with the CEO of a leading software company who three years ago had put QA directly under him. Unfortunately, the QA manager was not as strong as needed and a major release was shipped that had significant problems. Only after three years of working with this problem did the CEO finally understand the caliber of manager required. It took him another few months to find that person. Now they are in the rebuilding phase (the jury is still out on the success of this approach). That a P&L manager would make these decisions is actually unusual in our experience. Most continue to struggle with this problem and never really solve it.

In our view, for the P&L manager to succeed in the software business requires a quality management function (whether done internally or externally) with these characteristics:

- A clear definition and enforcement by the P&L manager of a quality policy.
- Independence at least within the organization and authority directly from the P&L manager.
- Stability and maturity of the team – meaning pay and promotion opportunities comparable to development; tenure of the team about the same as development; understanding of the business of developing successful software products; earned respect from the whole organization, etc.

- On-going investment in generic software testing and QA skills.
- On-going investment in tools and process improvement for the QA and test functions.
- An incentive structure that reinforces both effectiveness and efficiency in the QA and testing functions.

If a company can afford to spend its resources meeting these requirements it can and will maintain a powerful quality assurance function equal to the other elements required for product success. However, this is a set of investments that are often difficult for organizations to justify and they require a sustained interest on the part of the P&L manager. A viable alternative is to outsource some or all of the aspects of software quality management and/or software quality assurance or quality control to a third party specialist in this area.

There is an emerging model for solving this problem that is a consequence of natural industry evolution. This is outsourcing of some or all aspects of the software quality management function (quality management, quality assurance and/or quality control). This solution recognizes that, although the quality function done well is critical to success, it (or some aspects) may not be a strategic internal competency required for success. The generic methodology, process and tools of quality management, quality assurance, and test are a discipline in and of themselves. Companies must answer the question whether or not it is a strategically good investment for them to develop and maintain this functional expertise themselves - and it is not an inexpensive proposition.

### **The Evolution of Software Quality Management**

As with the rapid evolution of hardware platforms, software languages, software development tools and the process of defining and building software products, the business aspects of software quality are evolving. We have encountered at least five distinct models for organizing the software quality management function:

1. Developers doing their own QA;
2. Test or QA engineers integrated within the development teams;
3. A separate QA group belonging to the engineering manager or VP;
4. A separate QA group belonging to a VP other than the engineering VP;
5. A separate QA organization that reports directly to the senior P&L manager or a VP of Quality who reports to him.

The variety of specific solutions is not surprising given that the industry is still struggling to figure out this problem. Like the software business in general, each company seems to be intent on inventing its own model for software quality management. All of these models are based on the “do-it-yourself” approach, and are subject to the problems identified earlier in this paper.

Outsourcing of software QA activities is the next model to emerge and it provides another viable option to the P&L manager for solving his product quality and quality management problems. The historic view of QA outsourcing was one of low-cost, fast-turnaround supplements to internal testing efforts. A number of outsourcing companies thrived

by mainly providing compatibility testing of software against various hardware platforms and components. Typically, a client software company, would be running late on development and not have the in-house resources or equipment to do fast-turnaround compatibility testing. Although these test labs would have liked to build a more predictable and manageable business, they were (and still are) saddled with a large investment in a lab and computers. The business model of these test labs demands that they sell CPU cycles first and other types of projects second – they rarely get to the other types of QA activities and are not well equipped to succeed at them.

Software QA outsourcing historically has taken the form of contracting with independent test labs for specific test projects. This provides independence and objectivity but is typically done as a result of a company's QA manager attempting to solve a staffing shortfall problem rather than by a P&L manager trying to fix a basic quality management problem.

This early model of test outsourcing is rapidly evolving. Efforts by the major companies to improve their own quality processes (and, ultimately, quality) have accelerated the use of outsourcing in this decade. Our relationship with one of the leading PC manufacturers illustrates the evolution of thinking on quality management and the outsourcing of portions of it.

We first started talking with this customer as early as 1995. Over the next year they systematically investigated a number of labs and then began to employ them on small, non-critical projects that were not adequately staffed internally. Our first projects were testing localized versions of their software. After each of the early projects, reviews were held to identify how to improve the testing and communications processes on the next project. Thus, over time this client developed trained, trusted resources available to our client's test organization for overflow work. There was a clear plan to outsource some portion of the work and develop a set of trusted, long-term vendors.

In 1997, a conscious decision was made by our client not to grow their internal testing resources at the rate necessary to deal with an exploding workload. Instead, they formed an internal group with the sole function to manage the software test outsourcing activities. A key strategy was to encourage the best vendors to open local labs to improve the focus and communications in the relationship. There was clearly enough work at the client to support multiple labs if work was not sent outside the geographical area.

In early 1998, we opened a dedicated lab as a joint venture (with another competitor) near the customer's facilities. This lab marked a watershed for the test outsourcing industry in two critical ways. First, this is the first instance in the industry's short history that a local software-testing lab has been opened dedicated to working with a single customer at the invitation of that customer.<sup>5</sup> Secondly, this lab was entirely staffed with local people, many of whom had been employed by the customer as software QA engineers. The lab manager had been the customer's test center manager and brought with him a number of senior software test engineers.

A further evolution is in process already: the complete outsourcing of some or all aspects of the software quality management function. We are currently engaged with several clients that

---

<sup>5</sup> We are aware that a number of companies have contracted to put dedicated software test teams on a customer's site, but these companies have typically not been dedicated software testing companies nor have they put dedicated labs in place without specific long-term contracts.

have contracted with us to build and manage their entire software quality function. We hire their existing staff or new staff members as required, become an integral part of their organization (the team works on the client site), and report to the P&L manager directly or through his designated representative. Our QA manager is responsible to the P&L manager for insuring product quality and process quality within the defined budget. In fact, our QA manager is also the P&L manager for the specific software QA activity for the client. In all cases, we have a company-to-company business relationship directly with the P&L manager. In other words, we are solving the P&L manager's problem at the time that we solve the quality control problems of the engineering organization.

This model opens the door for the outsourced QA organization to be an influential participant in the client's internal development process and tool improvement initiatives. Not only do we conduct the actual testing function; we are providing quality assurance services to these clients. For instance, we are implementing defect tracking process and tools; configuration management process and tools; and planning and implementing other process improvement actions.

### **The Future of Software Quality Management**

What does the future hold? If the early success we've seen with customers is an indication, the logical next development in the management of the software quality function is the outsourcing of the entire QA function or some appropriate portion thereof.

A key advantage of this outsourcing model is that it can directly address the critical cultural and management problems identified in this paper. It can also result in improved quality and cost savings for the software company it serves.

These advantages come as a result of the unique characteristics of this outsourced QA team. First, by belonging to a company whose primary focus is conducting software QA activities many of the cultural problems discussed above are solved. In such an organization, the software QA engineer becomes a "first-class" citizen with all of the status and advantages the term implies. There is a well-defined career path with associated training and financial rewards. There is stability and maturity that can develop because the QA engineers are given a reason for staying with the organization and developing as first-rate professionals.

Secondly, by setting up the QA team as a profit and loss center with its own competent P&L manager (our QA manager), there ensues a profit motive to do both a better and more efficient job at providing software QA services to the customer. As dedicated and self-sacrificing as a top-notch internal QA team is, it is extremely difficult for a company to provide them with a financial incentive for doing a great job. QA is typically not a career path to senior management positions. QA salary levels are typically capped below those of development. Even when a company offers a bonus plan or stock options, the value is only indirectly tied to the actual effectiveness and efficiency of the QA team.

When a QA team is set up as its own P&L center it has a very tangible financial motivation for finding the most efficient ways to be most effective at its tasks. While an internal QA manager has little incentive to terminate a "temp" when the project is complete, a P&L manager with a bonus tied to financial results does have this incentive. When equipment is no longer really required to perform a testing task, the internal QA group typically hangs on to it for

some undefined future use. A P&L manager can't afford to keep problematic equipment as an expense.

Most importantly, a profit-motivated group, with an experienced management team, will find creative ways to both increase effectiveness (to make the customer happy) and improve the efficiency of the activities (to decrease costs). We've seen dozens of QA organizations waste thousands of dollars and hours of time attempting to automate the testing – only to fail. The failure resulted because the team didn't have the experience required to succeed and because there wasn't a serious enough consequence for failure. In an outsourced QA team, neither of these factors operates. Not only are the costs of failure reflected in the paychecks of the team, but also the relationship with their single customer is put at significant risk. A broken promise to automate testing can cause serious mistrust that could end in disaster for both the client company and the outsourced QA team.

The third critical notion is that the outsourced QA team would have a direct relationship with the P&L manager of their “parent” company (the customer that the QA team came from). This fact alone solves both of the critical problems for software P&L managers. The very act of making the QA team independent and directly responsible to the P&L manager (instead of an engineering or other VP) places the required strategic emphasis on software QA. In addition, the P&L manager now has an effective mechanism for monitoring the quality aspects of his products under development so that he can take decisive actions as required.

By having a direct relationship with the P&L manager, the QA team can also influence the overall software development process. The political power that comes with this relationship provides the opportunity to “push back” on development managers and teams who are shortcutting their own processes. This can't happen effectively when QA reports to the same VP as development. The QA team can also suggest improvements to the development process that will improve product quality and increase effectiveness – e.g., adding some programming hooks to support test automation or improving the product architecture standards to enhance testability and maintenance.

And lastly, the outsourcing of software QA can result in lowered overall costs for the client company. This comes in the form of improved quality and lower costs for customer support, interim fixes and releases, and better customer retention. Savings also occur in the actual software QA organization's costs to the client. We've already seen this in the case study discussed above. In this situation, the actual testing costs are 30-40% lower on a per-hour basis. A profit oriented QA team is simply more cost conscious than an internal team can be. In the new model of full QA function outsourcing, these costs can be lowered even more as effort is put into process improvement for the entire development cycle.

## **Conclusions**

Software company P&L managers have a new option for solving the software quality problem. Software quality management can be outsourced to a competent independent organization whose primary focus is on this area. This model provides an effective and efficient solution to the industry problem of obtaining higher quality in its products while managing costs and creating an appropriate cultural environment for the engineers who choose software QA as a career path.

## Using the Electronic Proceedings

Once again, PNSQC and ICSQ are proud to announce our electronic Proceedings on CD-ROM. On the CD-ROM, you will find most of the papers from the printed Proceedings, plus the slides from some of the presentations. We made every effort to see that the electronic and printed Proceedings match, but we were not able to get electronic copies of all 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](mailto:cdrom@pnsqc.org).

## Installing Acrobat Reader

The electronic Proceedings are in Adobe Acrobat format. The CD-ROM includes the free Acrobat Reader software for Macintosh, Microsoft Windows, and several of the most popular UNIX workstations.

If you do not currently have Acrobat Reader 3.01 installed, you can install it from the CD-ROM. The CD-ROM uses the industry standard ISO-9660 format. In the ACROBAT folder, find the appropriate folder for your system: Win16 (Windows 3.1), Win32 (Windows 95, 98, or NT), or UNIX. Follow the directions on the next page for your particular system.



## UNIX

Change directories to the `unix` directory on the CD-ROM, and run the shell script `./install`. The script will automatically detect your hardware and operating system. Answer the questions to install Acrobat Reader. If the script reports that your configuration is not supported, visit <http://www.adobe.com/acrobat> to see if you can download a version of Acrobat Reader for your system. The CD-ROM includes versions for the following systems: SunOS, Solaris, HP-UX, IBM AIX and SGI IRIX. Acrobat Reader without the Search tool is available for DEC Alpha and Linux. Read `instguid.txt` for information about installing in a network.

## Macintosh

You can download Acrobat Reader for the Macintosh from Adobe's web site: <http://www.adobe.com/acrobat>.

## Microsoft Windows 3.1, Windows 95, Windows 98, Windows NT

Run `SETUP.EXE` from the CD-ROM's root directory. The setup program creates Program Manager icons to access the Proceedings. If you want, you can copy the Proceedings to your hard drive. If you don't copy the files to your hard drive, the setup program will create links to the Proceedings on CD-ROM. If you do not already have a program that can read PDF files, the setup program asks if you want to install Acrobat Reader. If you decide not to, you can install Acrobat Reader at a later date by running `ACROBAT\WIN16\SETUP.EXE` for Windows 3.1 or `ACROBAT\WIN32\SETUP.EXE` for later versions of Windows.

## Using Acrobat Reader

After you have installed Acrobat Reader, you can use it to read and search the electronic Proceedings. On Windows, just select the 17th PNSQC icon to read the Proceedings. For the Macintosh or UNIX, start Acrobat Reader and open the file, PNSQC.PDF in the root directory of the CD-ROM. From there, you can read the electronic Proceedings, examine the table of contents to find a particular article, jump to the index to look up a topic, or use the search tool to find all the articles that include a keyword. The electronic Proceedings include a full-text index so the [Acrobat Search tool](#) is a fast and convenient way to find information on the CD-ROM.

You can print articles or slides for your own private use, but remember they are copyright to the original author. You cannot redistribute the Acrobat files or their printed copies. If you need extra copies of the printed or electronic Proceedings, please contact [Pacific Agenda](#). An order form appears on the [next page](#).

## **1999 Proceedings Order Form**

### **Pacific Northwest Software Quality Conference**

To order a copy of the 1999 Proceedings, please send a check in the amount of \$35.00 to:

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

Name\_\_\_\_\_

Affiliate\_\_\_\_\_

Mailing  
Address\_\_\_\_\_

City\_\_\_\_\_

State\_\_\_\_\_

Zip\_\_\_\_\_

Daytime  
phone\_\_\_\_\_